Tadeusz Kobus

Transactional Replication: Algorithms and Properties

Doctoral Dissertation

Submitted to the Council
of the Faculty of Computing Science
of Poznań University of Technology

Advisor: Paweł T. Wojciechowski, Ph. D., Dr. Habil.

Poznań · 2016

Tadeusz Kobus

Replikacja Transakcyjna: Algorytmy i Własności

Rozprawa doktorska

Przedłożono Radzie
Wydziału Informatyki
Politechniki Poznańskiej

Promotor: dr hab. inż. Paweł T. Wojciechowski

Poznań · 2016

A dissertation submitted in partial fulfilment of the requirements
for the degree of Doctor of Philosophy in Computing Science.

Tadeusz Kobus
Distributed Systems Research Group
Faculty of Computing Science
Institute of Computing Science
Poznań University of Technology
tadeusz.kobus@cs.put.edu.pl

Typeset by the author in LaTeX.

This dissertation and associated materials can be downloaded from:
http://www.cs.put.poznan.pl/tkobus/research/research.html

Institute of Computing Science
Poznań University of Technology
Piotrowo 2, 60–965 Poznań, Poland
http://www.cs.put.poznan.pl

The research presented in this dissertation was partially funded from National Science Centre (NCN) funds granted by decisions No. DEC-2011/01/N/ST6/06762 and No. DEC-2012/07/B/ST6/01230, from Foundation for Polish Science (FNP) granted by decision No. 103/UD/SKILLS/2014, and from the Polish Ministry of Science and Higher Education grant no. POIG.01.03.01-00-008/08. Experimental tests have been carried on the computing resources provided by Poznań Supercomputing and Networking Center (PSNC).

To my loving parents, Maria and Jacek.

Briefly, you can only find truth with logic
if you have already found truth without it.

G. K. Chesterton

Abstract

Replication is an established method to increase service availability and dependability. It means deployment of a service on multiple machines and coordination of their actions so that a consistent state is maintained across all the service replicas. In case of a (partial) system failure operational replicas continue to provide the service.

In this dissertation, we revisit State Machine Replication (SMR) and Deferred Update Replication (DUR), two basic strongly consistent replication schemes. We compare the schemes side-by-side and uncover their strong and weak sides.

We study the guarantees provided by SMR and DUR using our new correctness criteria, which we grouped into two families: $\diamond$-opacity and $\diamond$-linearizability. Our properties are not bound to SMR and DUR only. They allow us to formalize the behaviour of a wide variety of replicated schemes that do or do not support transactional semantics, respectively. By establishing a formal relationship between $\diamond$-opacity and $\diamond$-linearizability we can directly compare guarantees provided by SMR and DUR, even though only the latter provides true transactional support.

We also compare SMR and DUR experimentally across a wide range of workloads. Our results show that performance-wise, neither scheme is superior. Such a conclusion may come as a surprise since only the latter scheme is potentially scalable with an increasing number of processors. We discuss the characteristics of workloads that are favourable or troublesome for either scheme.

Finally, we combine SMR and DUR into a single, versatile replication scheme called Hybrid Transactional Replication (HTR). This way we bring together the best features of SMR and DUR. Not only HTR offers powerful transactional semantics (similarly to DUR) and is free of some limitations regarding the code of the replicated service itself (unlike both SMR and DUR), but it also works well across various workloads, often providing much better performance than either SMR or DUR. All these benefits are provided while maintaining strong correctness guarantees similar to those provided by DUR. By relying on a machine-learning-based approach, HTR can dynamically adopt to changing conditions.

Acknowledgements

I am truly grateful to my advisor Dr. Paweł T. Wojciechowski, who introduced me to the world of scientific research, offered his guidance along the way and helped me to pursue my goals.

I would like to thank Maciej Kokociński, my friend and research colleague, who collaborated with me on much of this work, for countless talks and insightful comments, teaching me how to become a better programmer and also for many good laughs. I also thank Prof. Jerzy Brzeziński, all the members of the Distributed Systems Research Group, and my fellow graduate students, Andrzej Stroiński, Dariusz Brzeziński, Dariusz Dwornikowski, Jan Kończak, Konrad Siek, Krzysztof Ciomek, Maciej Piernik, Mansureh Aghabeig, Paweł Kobyliński, Piotr Jakubowski, Szymon Francuzik, and Wojciech Wojciechowicz, for innumerable discussions on things related or not so much related to scientific research.

Last, but certainly not least, I wish to thank my family and friends. In particular, I am wholeheartedly thankful to my parents, Maria and Jacek, my siblings, Ewa and Stanisław, and my dearest companion Kalina, for their unconditional love and showing me what is most important in life. None of this work would have been possible without their support.

List of publications

Book chapters:

T. Kobus, M. Kokociński, and P. T. Wojciechowski, "Introduction to transactional replication," in *Transactional Memory. Foundations, Algorithms, Tools, and Applications* (R. Guerraoui and P. Romano, eds.), vol. 8913 of *Lecture Notes in Computer Science*, Springer, 2015

Journal papers:

P. T. Wojciechowski, T. Kobus, and M. Kokociński, "State-machine and deferred-update replication: Analysis and comparison," *IEEE Transactions on Parallel and Distributed Systems*, vol. PP, no. 99, 2016

T. Kobus, M. Kokociński, and P. T. Wojciechowski, "Relaxing real-time order in opacity and linearizability," *Elsevier Journal on Parallel and Distributed Computing*, vol. 100, 2017

T. Kobus, M. Kokociński, and P. T. Wojciechowski, "Hybrid transactional replication: State-machine and deferred-update replication combined," *In preparation for submission*. arXiv:1612.06302 [cs.DC]

Conference papers:

P. T. Wojciechowski, T. Kobus, and M. Kokociński, "Model-driven comparison of state-machine-based and deferred-update replication schemes," in *Proceedings of SRDS '12: the 31st IEEE International Symposium on Reliable Distributed Systems*, Oct. 2012

T. Kobus, M. Kokociński, and P. T. Wojciechowski, "Hybrid replication: State-machine-based and deferred-update replication schemes combined," in *Proceedings of ICDCS '13: the 33rd IEEE International Conference on Distributed Computing Systems*, July 2013

T. Kobus, M. Kokociński, and P. T. Wojciechowski, "The correctness criterion for deferred update replication," in *Program of TRANSACT '15: the 10th ACM SIGPLAN Workshop on Transactional Computing*, June 2015

Outside of the main scope of the research, the author participated in the following work.

Conference papers:

T. Kobus and P. T. Wojciechowski, "A 90% RESTful group communication service," in *Proceedings of DCDP '10: the 1st International Workshop on Decentralized Coordination of Distributed Processes*, June 2010

T. Kobus and P. T. Wojciechowski, "RESTGroups for resilient Web services," in *Proceedings of SOFSEM '12: the 38th International Conference on Current Trends in Theory and Practice of Computer Science: Software & Web Engineering Track*, Lecture Notes in Computer Science, Jan. 2012

M. Kokociński, T. Kobus, and P. T. Wojciechowski, "Make the leader work: Executive deferred update replication," in *Proceedings of SRDS '14: the 33rd IEEE International Symposium on Reliable Distributed Systems*, Oct. 2014

M. Kokociński, T. Kobus, and P. T. Wojciechowski, "Brief announcement: Eventually consistent linearizability," in *Proceedings of PODC '15: the 34th ACM Symposium on Principles of Distributed Computing*, July 2015

Patent applications:

P. T. Wojciechowski, T. Kobus, and M. Kokociński, "A fault-tolerant data processing computer system and method for implementing a distributed two-tier state machine." EPO patent application no. EP16461501.5, Jan 12. 2016

P. T. Wojciechowski, T. Kobus, and M. Kokociński, "A fault-tolerant data processing computer system and method for implementing a distributed two-tier state machine." USPTO patent application no. 14/995,211, Jan 14. 2016

Contents

1 Introduction

Along with the emergence of cloud computing, where services in the cloud must be accessed by a large number of users in parallel, there was an explosion of interest in various approaches to *distributed replication*. Distributed replication can improve service availability and reliability by storing data closer to the users and processing many client requests in parallel. In this approach, a service is deployed on several interconnected machines called replicas, each of which may fail independently. The replicas are coordinated, so that each one maintains a consistent state view despite any failures of communication links or crashes of some other replicas. Each replica can only access its local memory (or storage) and the coordination is ensured via a distributed memory (or storage) system that provides consistent access to replicated data.

1.1 Context

We consider a particular type of replication called *transactional replication*, which assumes that each client request is executed in isolation as an *atomic transaction*. It means that operations specified within the request (transaction) are executed in all-or-nothing semantics and the system is responsible for detecting and resolving conflicts between concurrent transactions that access shared data items. Also the transaction's code may include constructs such as *rollback* and *retry*, which allow the programmer to better control the execution flow of the transaction (the former revokes all changes performed by the transaction so far and the latter rolls back the transaction and restarts it, possibly when certain conditions are met).

Transactional replication of a service or an application is typically achieved by having the service/application built on top of a (distributed) transactional framework. By using the transactional framework, the programmer is relieved from having to manually manage concurrent accesses to shared data items us-

ing, e.g., locks or leases, i.e., mechanisms known to be difficult to use even for experienced programmers. A transactional framework hides all the complexity from the programmer and allows him to reason about transactions as if they are executed sequentially by a single, dependable server. In turn, providing transactional support can greatly ease development of highly available services.

The guarantee that a system gives an illusion of a single server that executes all requests sequentially is a very strong one. Originally it was the desired property of the traditional *SQL (Structured Query Language)* database systems and was formalized by the *(strict) serializability* criterion or simply referred to as the *strongly consistent* approach [14].

Serializability was typically achieved by processing all updating transactions on a single *master* replica and propagating updates to the other *slave* replicas. Usually all replicas contained a full copy of the database. Read-only transactions, which do not change the state of the system, were executed in parallel on any of the available slave replicas. A crash of the master replica required a pause in the computation, so that a new master is elected from amongst the operating replicas. This approach to replication was further enhanced for better performance and availability by allowing updating transactions to be executed concurrently on the master replica and then also on different replicas in parallel. Execution of every transaction was coordinated by one of the servers, which ensured correct execution of the transaction and its subsequent commit, despite of any machine or link failures. By delaying execution of some operations or rolling back and restarting some transactions, the coherence protocol (implemented by the replicated system) safeguarded transaction execution, so that no inconsistencies arise upon concurrent accesses to shared data items by different transactions. This approach is typically referred to as the *Deferred Update Replication (DUR)* scheme [15].

Alternatively, a database could be replicated using the *State Machine Replication (SMR)* approach [16] [17] [18]. In this scheme, identical copies of a database were run on multiple machines and every replica executed every request. This way the servers advanced their state in the same way, making failure of any of the replicas essentially invisible to clients. Naturally, all requests had to be deterministic and must have been delivered in the same order to all replicas, or otherwise the state of the replicas would diverge over time. Compared to DUR, SMR was much simpler. However, it also did not allow for parallel execution of requests and thus was not scalable (adding more servers, processors or processor cores did not improve the performance of the system). Note that although SMR does not provide transactional support, execution of a request by SMR can be regarded as a rudimentary transaction that is allowed to only commit.

In early 2000s, new *NoSQL (Non SQL, Not only SQL)* database systems started to emerge and supersede the traditional SQL ones. NoSQL databases trade strong consistency and ease of use in search of higher performance. Much higher throughput and better scalability of these systems (compared to SQL databases) allowed the globally accessible services running on the Internet to cope with increasing traffic (see, e.g., [19] [20] [21] [22]). NoSQL databases typically offer

no transactional support and provide only *weak (eventual) consistency* guarantees [23]. In turn, clients may encounter some anomalies in the results they observe. The anomalies can range from completely harmless (observing posts on a social networking site in a slightly different order than just seconds earlier), to more problematic ones (a client receives a confirmation of a purchase of a flight ticket and later it turns out that the ticket is invalid and a different flight needs to be booked). The observed erroneous behaviour can be further amplified by machine or link failures. Ensuring correct behaviour of the service means that programmers have to handle all the corner cases themselves. The lack of clearly laid out semantics of such systems and a high level of non-determinism, which is typical for distributed environment, result in high costs of service development and maintenance.

Clearly, the lack of strong consistency guarantees and transactional support is problematic for clients as well as for programmers. Owners of large services running on the Internet constantly look for optimizing costs and improving user experience. Hence, in recent years one can observe a renewed interest in strongly consistent solutions (see, e.g., [24] [25]). In this dissertation we focus on this type of systems.

Naturally, the strongly consistent data and service replication schemes, which we research, are heavily inspired by SQL database systems that also feature strongly consistent transactions running in a distributed environment. However, the inspiration also comes from the research on *Transactional Memory (TM)* [26], which applies the concept of transactions know from database systems as a concurrency control mechanism in concurrent programming (in a local environment). However, none of the existing solutions can be applied directly for service replication. One of the main reasons why it cannot be done concerns the characteristics of workloads typically encountered by modern replicated services. For example, the average request execution time in systems we consider is usually orders of magnitude shorter compared to database systems and orders of magnitude longer compared to TM systems [27]. Also some parts of logic of modern replicated services often cannot be easily expressed using SQL, the language of the traditional database systems. Instead, a more suitable approach is to rely on an interface akin to the one of TM systems, where a transaction is considered a high-level programming language construct, which can enclose arbitrary code and, in particular, may invoke complex methods or operations on shared objects (see, e.g., [28] [29]). In this work we consider the latter approach, which is the more general one.

Because of the differences in the offered semantics, the correctness criteria used in the context of database systems (such as *recoverability*, *avoiding cascading aborts* and *strictness* [30] or variations of the already mentioned serializability) cannot be used to formalize the behaviour of the modern strongly consistent replicated schemes. Unfortunately, the criteria known from the world of TM systems (such as various flavours of *opacity* [31] [32] [33], *TMS1* [34] or *TMS2* [34]) cannot be used either. It is because these criteria are designed to strictly adhere to the realm of a local environment, where certain guarantees concerting

the ordering of transactions (such as the *real-time* order guarantee) come naturally and are relatively inexpensive to provide at all times. On the other hand, in a distributed environment these guarantees are often relaxed for certain types of requests (such as read-only requests). Therefore, formalizing the semantics of replicated systems, such as those considered in this dissertation, requires defining new correctness criteria.

It seems then, that although the strongly consistent replicated schemes again start to gain popularity, the basis on which such solutions are established are not well understood yet and much more work is required in this field. For example, the existing literature lacks a thorough comparison of the basic replication schemes such as SMR and DUR, both in terms of semantics and performance. It comes as a surprise, because both SMR and DUR can be considered the base for a great number of more complex replication schemes (see e.g., [35] [36] [37] [38] [39] [40] among others). Only once we uncover the differences between these schemes and clearly define their strong and weak sides, can we propose new strongly consistent replication schemes that adhere to the current needs and fully exercise the capabilities of modern (parallel) computer system architectures.

1.2 Aims and Contributions

Given the motivations presented above, we formulate our main thesis as follows:

> *It is possible to develop a scheme for service and data replication that simultaneously offers rich transactional semantics, strong consistency guarantees and high performance across a wide range of workloads.*

Below we sum up the main contributions presented in this dissertation:

1. **Novel correctness criteria for strongly consistent replicated systems**. We define $\diamond$-*opacity* and $\diamond$-*linearizability*, two families of correctness criteria designed for strongly consistent replicated systems. We build our properties on opacity and linearizability, two well established correctness criteria defined for transactional systems and systems modelled as shared objects [31] [41]. By using the introduced properties, we can formalize the guarantees provided by a wide range of replication schemes which do or do not feature transactional semantics and implement various optimizations, such as execution of read-only requests by a single replica without any inter-process synchronization. We prove that all members of the $\diamond$-opacity and $\diamond$-linearizability families are safety properties (i.e., they are non-empty, prefix-closed, and limit-closed). We also establish a formal relationship between the two families. We show that when requests are executed as transactions in a $\diamond$-opaque system and the transactions are hidden from clients

(i.e. the intermediate results of a transaction execution is not available to the client during transaction execution but only once the transaction commits or aborts), then the system is $\diamond$-linearizable. In particular, we show the relationship between opacity and linearizability in their original definitions (to our best knowledge, it is the first result of this sort).

2. **A thorough comparison of SMR and DUR**. We conduct a comparison of SMR and DUR both in terms of semantics as well as performance. We prove the correctness guarantees of both systems and show that SMR satisfies *real-time linearizability* (a member of the $\diamond$-linearizability family) whereas DUR ensures *update-real-time opacity* (a member of the $\diamond$-opacity family). We prove that DUR satisfies *update-real-time linearizability* when transactions are hidden from clients. Thus we show that DUR's guarantees are strictly weaker than SMR's. We also consider *SMR with Locks (LSMR)*, a variant of SMR which allows read-only requests to be executed by a single replica without any inter-process synchronization. We highlight the consequences of introducing this optimization and prove that LSMR provides strictly weaker guarantees compared to SMR but stronger than DUR's. The results of experimental tests reveal the strong and weak sides of SMR and DUR across a wide range of workloads. The key result of the performance comparison of both schemes is that neither is superior in all cases. This finding is surprising, given that in the general case, only DUR can potentially scale with the increasing number of replicas (SMR executes all requests sequentially and LSMR, the optimized version of SMR, enables concurrent execution exclusively for read-only requests).
3. **A novel strongly consistent transactional replication scheme**. We propose a novel scheme called *Hybrid Transactional Replication (HTR)*, which combines SMR and DUR for better performance, scalability, and improved code expressiveness. We formally prove that HTR offers similar guarantees as DUR. As we demonstrate in evaluation tests, HTR performs well across a wide range of workloads. In particular, HTR is insensitive to workloads typically know to be problematic for either SMR or DUR (i.e., computation intensive workloads for SMR and workloads characterised by high contention levels for DUR). In some cases, HTR achieves up to 50% improvement in performance over HTR configured in a way that simulates DUR or LSMR (all updating transactions are executed in a way that resembles either transaction execution in DUR or request execution in LSMR). HTR's performance can be fine-tuned by defining a policy, which allows the system to alter the way transactions are handled and thus to adapt to changing conditions. We discuss several manual policy optimization techniques and we also propose a machine-learning-based approach, which supports the programmer in the task of creating a policy suitable for a given workload. By using the ML-based oracle, the system can automatically adapt to changing workloads.

1.3 Thesis Outline

The thesis is organized as follows. In Chapter 2 we revisit work that is related to our research. Then, in Chapter 3 we define the model of transactional replication and lay out basic assumptions about the replicated systems we consider. Next, in Chapter 4 we present the $\diamond$-opacity and $\diamond$-linearizability families of correctness properties for replicated systems. In Chapter 5 we discuss two basic approaches to service or data replication: SMR and DUR. We examine their characteristics, guarantees they offer and performance under various workloads. Then, in Chapter 6 we present HTR, a novel replication scheme that seamlessly merges SMR and DUR. Finally, we conclude in Chapter 7.

Proofs of all formal results presented in this dissertation can be found in the Appendix.

2
State of the Art

In this chapter we present work relevant to our research. We start with various service and data replication schemes. Then, we discuss correctness criteria defined for strongly consistent transactional and non-transactional replicated systems. Next, we focus on Distributed Transactional Memory systems. Finally, we cover work related to some specific aspects of our HTR scheme.

2.1 Replication Schemes

Over the years, multiple service and data replication techniques have emerged (see [15] for a survey). They differ in offered semantics, as well as performance under various workloads. We focus on replication schemes that offer strong consistency.

State Machine Replication [16] [17] [18] [42], which we describe in detail in Section 5.1, is the simplest and most commonly used non-transactional replication scheme. SMR uses a distributed agreement protocol to execute client requests on all processes (replicas) in the same order. For replica coordination, various fault-tolerant synchronization algorithms for totally ordering events were proposed (see, e.g., [43] [44] among others). In particular, *Total Order Broadcast (TOB)* has been used for request dissemination among replicas (see Section 3.2 for the specification of TOB, [45] for a survey of TOB algorithms and [15] for further references). Also implementations of TOB with optimistic delivery of messages are used to build systems based on state replication, see e.g., [35] [36] [46] [47]. Execution of read-only requests can be optimized in SMR, so that only updating requests require inter-process synchronization. The optimized version of SMR, which we call *SMR with Locks (LSMR)*, will be described in detail in Section 5.2.

Deferred Update Replication [15], described in detail in Section 5.3, is a basic replication scheme which features *transactional semantics* (explained in Section 3.5). DUR is based on a multi-primary-backup approach which, unlike

SMR and LSMR, allows multiple (updating) client requests to be executed concurrently. In DUR, a request is executed as an atomic transaction which runs in isolation and whose updates are propagated upon transaction commit to other replicas to guarantee persistence. DUR relies on an atomic commitment protocol to ensure consistency. Various flavors of DUR are implemented in several commercial database systems, including Ingres, MySQL Cluster and Oracle. These implementations use two-phase-commit [30] as the atomic commitment protocol. In this work, we consider DUR based on TOB [45]. This approach is advocated by several authors because of its non-blocking nature and predictable behaviour (see [48] [49] [50] among others). Most recently, it has been implemented in D2STM [51] and in our system called Paxos STM [5] [6] (characterised in Section 5.4). It has also been used as part of the coherence protocols of S-DUR [37] and RAM-DUR [38].

Despite the fact that SMR and DUR are very well established replication schemes, surprisingly, the literature lacked a direct comparison of both approaches. In [5] [2], we compared SMR and DUR both theoretically and practically and showed that neither scheme is superior.[1] We summarize this work and include the experimental results presented there in Sections 5.4 and 5.5.

Our novel approach called *Hybrid Transactional Replication*, which we describe and evaluate in Chapter 6, is also a strongly consistent replication scheme based on TOB. As discussed in detail in the chapter, HTR shares many similarities with both SMR and DUR by allowing transactions to be executed either in a pessimistic or optimistic mode, resembling request or transaction execution in SMR or DUR, respectively. Similarly to SMR and DUR, HTR fits the framework of *transactional replication (TR)* [1], which formalizes the interaction between clients and the replicated system. The programming model of TR corresponds to *Distributed Transactional Memory (DTM)*, as discussed below. As proved in Section 6.4, HTR offers guarantees on transaction execution, which are similar to those provided by DUR.

There are a number of optimistic replication protocols that, similarly to our HTR scheme, have their roots in DUR. For example, we can mention here *Postgres-R* [39], *PolyCert* [40] and *Executive DUR (E-DUR)* [10]. In Postgres-R, TOB is only used to broadcast the updates produced by a transaction. This optimization comes at the cost of an additional communication step, which is necessary to disseminate the decision regarding transaction commit or abort to other replicas. PolyCert (which we also discuss in Section 2.4.2) can switch between three TOB-based certification protocols, which differ in the way the readsets of updating transactions are handled. E-DUR streamlines transaction certification with the leader of the Paxos protocol. The differences between these systems lie not in the general approach to processes synchronization, but in the way the transaction certification is handled. In turn, the guarantees they offer can be formalized with the same correctness properties that we use for DUR. We show this for-

[1] Note that neither SMR nor LSMR offers transactional semantics, however, the execution of a request by SMR or LSMR can be regarded as a rudimentary transaction that is allowed to only commit.

malization in Sections 2.2.1, 4.3 and 5.3.3). Note that all the above mentioned protocols are aimed only at increasing the throughput of DUR and not at extending the transactional semantics of the base protocol, as in case of HTR (see also Section 2.4).

2.2 Correctness Properties for Replicated Systems

Now we focus on correctness properties that can be used to formalize the guarantees offered by replicated systems.

2.2.1 Correctness Properties for Transactional Systems

Several correctness criteria have been proposed to formalize the guarantees offered by transactional systems. Below we describe the most relevant ones. We show that none of them allows for relaxed time-ordering of transactions and at the same time provides sufficiently strong consistency guarantees that match those of real practical systems (such as those based on DUR). Next, we proceed to describe how our correctness criteria relate to others.

Serializability [14] specifies that all committed transactions are executed as if they were executed sequentially by a single process. *Strict serializability* [14] additionally requires that the real-time order of transaction execution is respected (i.e. the execution order of non-overlapping committed transactions is preserved, while the order of concurrent transactions is not specified). *Update serializability* [52] is very similar to serializability, but allows read-only transactions to observe different (but still legal) histories of the already committed transactions.

All three properties mentioned above regard only committed transactions and say nothing about live or aborted transactions. Therefore, they are not sufficient to describe behaviour of some transactional systems, especially when transactions may perform arbitrary operations and reading an inconsistent state may lead to erroneous behaviour (our model fits this class of transactional systems, see Chapter 3). Therefore, new correctness criteria emerged that formalize the behaviour of all transactions in the system, including the live ones. Some of the correctness criteria, such as *recoverability*, *avoiding cascading aborts* and *strictness* [30], specify the behaviour of read and write operations for both live and completed transactions. However, they say nothing about global ordering of transactions (unlike serializability and properties similar to it). This, in turn, limits their usefulness in the context of strongly consistent transactional systems. Therefore, our attention focuses on properties that (in most cases) maintain a global serialization for all transactions.

The following properties maintain a global serialization only for a subset of transactions.[2] *Extended update serializability* (formally defined in [53] as the

[2]Note, however, that the majority of correctness criteria discussed below were formulated with a local environment in mind, where communication between processes is relatively inexpensive.

no-update-conflict-misses condition) ensures update serializability for both committed and live transactions. Therefore, it features a global serialization for all the updating transactions (read-only transactions may observe a different serialization). *Virtual world consistency* [54] allows an aborted transaction to observe a different (but still legal) history; only committed transactions share a single view of past (committed) transactions.

Opacity [55] [31] [32], which was originally proposed by Guerraoui and Kapalka, features a global serialization of all transactions and extends strict serializability in a way that live transactions are always guaranteed to operate on a consistent state. In a sense, it is thus akin to extended update serializability, which adds similar guarantees to update serializability. *Rigorousness* [56], *TMS2* [34] and *DU-opacity* [33] offer even stronger guarantees. Rigorousness and TMS2 impose stronger requirements on the ordering of concurrent transactions. DU-opacity explicitly requires that no read operation ever reads from a commit-pending transaction which later aborts, unless the transaction which executes the read aborts itself. Moreover, all these three properties are only defined in a model that assumes read-write registers. *TMS1* [34], which was proposed to slightly relax opacity, allows live and aborted transactions to observe a different view of past transactions (committed transactions share a consistent view of the already committed ones). The possible histories are, however, restricted by a few conditions which enforce quite strong consistency (despite the lack of global serialization for all transactions). Similarly to strict-serializability, all of the above properties require that the real-time order of transaction execution is respected, thus they are not applicable to systems considered in this work (we elaborate on this in Section 4.1).

The $\diamond$-*opacity* family of properties, which is a contribution of this thesis and is defined in Section 4.3, relaxes real-time order requirements on transaction execution in opacity to a various degree (we consider opacity in its prefix-closed version, as in [31]). All members of $\diamond$-opacity maintain two crucial characteristics of opacity, namely a global serialization of all transactions and a guarantee for every transaction to always read a consistent state. As there are some analogies between the members of the $\diamond$-opacity family and other correctness properties, we now briefly outline each member, from the strongest to the weakest one. The strongest property in the $\diamond$-opacity family is *real-time opacity*, which is equivalent to opacity defined in [31]. *Commit-real-time opacity* allows live and aborted transactions to read stale (but still consistent) data. *Write-real-time opacity* further relaxes the real-time order guarantees on transactions that are known *a priori* to be read-only (i.e., before they commence execution). *Update-real-time* differs from write-real-time opacity by allowing all read-only transactions to not respect the real-time order (so also transactions, which happened not to perform any updating operations). *Program order opacity* requires real-time order only for transactions executed by the same process (the order of transactions executed by different processes is not specified). In this sense, program order opacity is similar to *virtual time opacity (VTO)* [54], but defined in the framework of the original

Therefore, they are not suited for use in a distributed environment.

definition of opacity and not using partially ordered sets of events. Moreover, unlike VTO, program order opacity (and other members of ⋄-opacity) is a prefix closed property (see Section 4.3). Finally, *arbitrary order opacity* makes no assumptions whatsoever on the relative ordering of transactions. In this respect, it is similar to serializability. However, contrary to serializability, arbitrary order opacity also ensures that live transactions always observe a consistent view of the system's state. Hence, arbitrary order opacity is stronger than serializability.

Note that program order opacity is stronger than *transactional causal consistency* [57] [58], because the latter respects the order of transaction execution on every process but does not require a single sequential view of all executed transactions. For the same reason, transactional causal consistency is incomparable with arbitrary order opacity, in which such a single view needs to be maintained but the order of transaction execution on any process does not need to be respected.

2.2.2 Correctness Properties for Non-transactional Systems

Linearizability, proposed by Herlihy and Wing in [41], is the most widely known correctness property for concurrent data structures and strongly consistent replicated systems that do not offer transactional semantics (see, e.g., [59] [60]). Unlike many other correctness criteria for (systems modelled as) shared objects, such as *PRAM consistency*, *cache consistency* and *slow consistency* [61], linearizability maintains a global serialization of all *operation executions* in a similar way to which opacity or strict serializability do so for transactions. Linearizability also requires that the real-time order on operation execution is respected at all times. This trait makes it unsuitable for distributed environments.

The ⋄-*linearizability* family of properties, which we define alongside ⋄-opacity, relaxes the real-time order guarantees on operation execution of linearizability in a similar way to which ⋄-opacity does so for transactions in opacity (see the definition and discussion of ⋄-linearizability in Section 4.4). *Real-time linearizability*, the strongest property of the family, is equivalent to the original definition of linearizability and can be used to formalize the guarantees provided by SMR (see Section 5.1.3). As we formally prove, a weaker member of ⋄-linearizability, namely *write-real-time linearizability*, can be used to formalize the guarantees provided by LSMR, an optimized version of SMR which allows read-only requests to be processed without inter-replica synchronization (see Section 5.2.3). Interestingly, *final-state program order linearizability* (an intermediary property used to define *program order linearizability*, another member of the ⋄-linearizability family) is equivalent to *sequential consistency* [62]. In a similar way to which program order opacity is stronger than transactional causal consistency, program order linearizability is stronger than *causal consistency* [61]. *Arbitrary order linearizability*, the weakest member of the ⋄-linearizability family, is incomparable with causal consistency for similar reasons why transactional causal consistency is incomparable with arbitrary order opacity.

In a recent paper [63], the authors used linearizability as a safety property for

a TM system, in which every invocation of a transaction is executed exactly once and transaction aborts are hidden from the programmer. In a somewhat similar fashion, we use $\diamond$-linearizability to formalize the behaviour of a replicated transactional system, in which the intermediate results of a transaction execution are invisible to the client (the client is notified only once a transaction commits or aborts, see details in Section 4.5). However, in our approach aborts are exposed to the processes, as we model transactional systems as *abortable objects*, defined as follows.

Objects which may return special responses *fail* or *pause* for any operation invocation when there is contention, appeared first in [64] under the name *obstruction-free objects*. The special response *fail* indicates that the operation was not applied and the process is free to invoke any other operation. On the other hand, *pause* means that the implementation is uncertain whether the operation had any effect on the object. The authors show that the objects that may return *fail* but not *pause*, cannot be implemented using only read/write registers. The work in [65] introduced the notions of *abortable objects* and *query-abortable objects*. In this approach, a special response *abort* always implies uncertainty whether an operation was applied or not. On the other hand, query-abortable objects provide a special *query* operation, which allows processes to determine their last operation that caused a state transition of the object. *Deterministic abortable objects* [66] feature only one special response, *abort*, which always indicates that the operation did not take an effect. As noted before, such objects cannot be implemented using only read/write registers. The authors in [66] investigate their computational power and the implicit hierarchy they form. The notion of deterministic abortable objects is now well-accepted by the community. Therefore, unless noted otherwise, we mean deterministic abortable objects when we say abortable objects.

The original definitions of obstruction-free, abortable and query-abortable objects require that the objects return special responses only when contention is encountered. This way these definitions prohibit trivial implementations, which, e.g., always return *abort* instead of a regular response (i.e., different than, e.g., *abort*). Among the most popular measures of contention are *step-contention* [64] and *interval-contention* [65]. Step contention of an execution fragment indicates the number of processes that take steps (e.g., perform operations on CPU) within the fragment. On the other hand, interval contention takes into account the number of concurrently processed high-level operations (some of which could be not using CPU at the moment). In this work, we place no restrictions on the conditions when objects may return the response *abort*. We do so for two main reasons. Firstly, we believe that this is an orthogonal problem, which is related to progress rather than safety. Secondly, we use an abortable object as a facade for a transactional system. Since transactional systems usually provide their own progress guarantees (e.g., *progressiveness*, *global progress* or *obstruction-freedom* [31]), they would translate into the properties of the facade object (in particular, an obstruction-free transactional memory would impose requirement on special responses of the facade object to occur only upon

encountering step-contention).

2.2.3 Safety and Liveness Properties

Some authors argue that it is useful to distinguish two classes of correctness properties: *liveness* and *safety* properties (see, e.g., [67] [68]). Although this distinction does not exhaust the whole spectrum of correctness properties (e.g., serializability is neither a safety nor a liveness property), such a distinction is useful, as it captures important and radically different facets of a computer system's correctness. Moreover, showing safety and liveness requires different proving techniques.

Informally, a liveness property ensures that *something good* eventually happens during system execution. On the other hand, a safety property guarantees that *nothing bad* ever does. This trait can be formalized by requiring that the property is non-empty, prefix-closed and limit-closed. In the dissertation we formally prove that all members of $\diamond$-opacity and $\diamond$-linearizability families are indeed safety properties.

2.3 Distributed Transactional Memory Systems

The model of replication considered in this dissertation closely corresponds to some Distributed Transactional Memory systems (briefly mentioned Section 2.1). DTM evolved as an extension of local (non-distributed) transactional memory [26] to distributed environment. In TM, transactions are used to synchronize accesses to shared data items and are meant as an alternative to lock-based synchronization mechanisms. TM also has been proposed as an efficient hardware-supported mechanism for implementing monitors [69].

Paxos STM is an object-based DTM system that we originally developed to evaluate DUR [5] (and compare with SMR) and later used as a testbed for the E-DUR and HTR schemes [6] [10] [4] (see also the test results in Sections 5.4 and 6.7). Paxos STM builds on JPaxos [70]–a highly optimized implementation of the Paxos algorithm [71].

Several other DTM systems were developed so far, e.g., Anaconda [72], Cluster-STM [73], DiSTM [74], Hyflow [75] and Hyflow2 [76]. Notably, our system was designed from the ground up as a fully distributed, fault-tolerant system, in which crashed replicas can recover. Unlike DiSTM, there is no central coordinator, which could become a bottleneck under high workload. The TOB-based transaction certification protocol implemented by Paxos STM simplifies the architecture, limits the number of communication steps and avoids deadlocks altogether (unlike the commit protocols in Anaconda or Hyflow/Hyflow2). The use of TOB also helps with graceful handling of replica crashes (which, e.g., are not considered in Cluster-STM). The closest design to ours is the one represented by D2STM [51], which also employs full replication and transaction certification

based on TOB. However, unlike Paxos STM, D2STM does not allow replicas to be recovered after crash nor transactions to contain *irrevocable operations*, i.e., operations whose side effects cannot be rolled back (such as local system calls).

2.4 Other Related Work

Now let us discuss work that is related to some aspects of our HTR scheme and HTR-enabled Paxos STM.

2.4.1 Transactional Semantics

HTR allows irrevocable operations in transactions executed in the SM mode (which guarantees a transaction commit, see Section 6.2). However, the code of these transactions have to be deterministic, because each SM transaction is executed by all replicas (independently). Notably, the system performance is not compromised since a single transaction in the SM mode can run in parallel with all transactions executed in the DU mode.

The problem of irrevocable operations has been researched in the context of non-distributed TM (see e.g., [77] [78] [79] [80] among other). These operations are typically either forbidden, postponed until commit, or switched into an *ad hoc* pessimistic mode [81]. Some solutions for starved transactions (i.e., transaction, which repeatedly abort) are relevant here, e.g., based on a global lock [82] or leases [83]. The former is not optimal as it impacts the capability of the system to process transactions concurrently. On the other hand, the latter solution does not guarantee abort-free execution and requires a transaction to be first executed fully optimistically at least once. More recently, Atomic RMI, a fully-pessimistic DTM system which provides support for irrevocable operations, has been presented in [84] and [85]. Unlike Paxos STM, in which transactions are local in scope and data is consistently replicated, Atomic RMI implements distributed transactions and does not replicate data across different machines.

In database systems, there exists work on allowing nondeterministic operations, so also irrevocable operations. In [86], a centralized preprocessor is used to split a transaction into a sequence of subtransactions that are guaranteed to commit. The next subtransaction to be executed is established after the previous one commits. This, however, requires one broadcast per subtransaction which significantly increases latency.

2.4.2 Protocol Switching

Since in HTR a transaction can be executed in two different modes, solutions which allow for protocol switching are relevant. For example, PolyCert [40] features three certification protocols that differ in the way the readsets of updating transactions are handled. Similarly, Morph-R [87] features three interchangeable

replication protocols (primary-backup, distributed locking based on two-phase commit, and TOB-based certification), which can be switched according to the current needs. Contrary to PolyCert and Morph-R, our approach aims at the ability to execute transactions in different modes with the mode chosen on per-transaction-run basis. Additionally, our system considers a much wider set of parameters and can be tuned by the programmer for the application-specific characteristics. Hyflow [88] allows various modes of accessing objects needed by a transaction: migrating them locally and caching (data flow) or invoking remote calls on them (control flow). StarTM [89] uses static code analysis to select between the execution satisfying *snapshot isolation (SI)* and serializability. Serializability is ensured at all times, since the system executes transactions in the SI mode only when it is guaranteed that no write-skew anomalies can occur.

In AKARA [90], a transaction may be executed either by all replicas as in SMR, or by one replica with updates propagated after transaction finishes execution, in a somewhat similar way to which it is done in DUR. In the latter case, execution can proceed either in an optimistic or in a pessimistic fashion, according to a schedule established prior to transaction execution using conflict classes. However, in both cases the protocol requires two broadcast messages for every transaction: a TOB message issued upon request submission, which establishes the final delivery order, and then, after a transaction finishes execution, a reliable broadcast message that carries the transaction's updates. On the other hand DUR and HTR require only one broadcast for every transaction. Unlike in HTR, in AKARA the execution mode is predetermined for every transaction and depends on the transaction type.

Approaches that combine locks and transactions are also relevant. In [91], Java monitors can dynamically switch between the lock-based and TM-based implementations. Similarly, adaptive locks [81] enable critical sections that are protected either by mutexes or executed as transactions. However, the above two approaches use a fixed policy. In our approach, the HTR oracles implement a switching policy that can adapt to changing conditions.

2.4.3 Machine Learning Techniques

The mechanisms implemented in *HybridML* (or *HybML* in short), our machine learning (ML) based oracle for HTR, are heavily inspired by some algorithms well known in the ML community. Most importantly, HybML implements a policy that is similar to the *epsilon-greedy strategy* for the *multi-armed bandit problem* (see [92] for the original definition of the problem, [93] for the proof of convergence, and [94] for the survey of the algorithms solving the problem). However, some crucial distinctions can be made between the original approach and ours. We discuss them in detail in Section 6.6.

A survey of self-tuning schemes for the algorithms and parameters used in various DTM systems can be found in [95]. A few ML-based mechanisms have been used in some of the transactional systems we discussed before. PolyCert [40] implements two ML approaches to select the optimal certification protocol.

The first is an offline approach based on regressor decision trees, whereas the second uses the *Upper Confidence Bounds* algorithm, typically used in the context of the multi-armed bandit problem. Because the used certification protocols behave differently under various workloads, in the latter approach the authors decided to discretize the workload state space using the size of readsets generated during execution of transactions. This differs from our approach, since in HybML we solve the multi-armed bandit problem independently for every class of transactions. The rough classification, which can be much finer than in PolyCert, is provided by the user. Morph-R [87] uses three different black-box offline learning techniques to build a prediction model used to determine the optimal replication schemes for the current workload, i.e., decision-trees, neural networks, and support vector machines. Such heavy-duty ML approaches are not suitable for our purposes because HTR selects an execution mode for each transaction run independently and not for the whole system once every several minutes, as it is usually the case in typical applications of ML techniques (see, e.g., [96] [97]). Hyflow [75] uses heuristics to switch between the data-flow and control-flow modes, but the authors do not provide details on the mechanisms used.

3
System model

In this chapter we describe the system model for transactional replication. We also provide the specification of Total Order Broadcast as it is the sole mechanism used for interprocess communication in all algorithms discussed in this dissertation.

3.1 Server Processes

A replicated service is implemented on top of a system consisting of a finite set $\mathcal{P} = \{p_1, p_2, ..., p_n\}$ of n *service processes* (or *processes* or *replicas*, in short) running on independent machines (nodes) connected via a network. We assume a *crash-recovery failure model* in which processes may crash independently and later on recover and rejoin computation. A process is said to be *up* if it correctly executes its program. Upon a *crash* event, a process fails by ceasing communication with any other processes and becomes a *down* process. It can rejoin distributed computation upon a *recovery* event, which requires executing a recovery procedure. A process is said to be *unstable* if it crashes and recovers infinitely many times. A process is *correct* if it is *eventually permanently up* (there is a time after which it never crashes). Otherwise it is *faulty*, i.e. unstable or *eventually permanently down* (there is a time when it crashes and later never recovers). No assumption is made on the relative computation speeds of the processes.

Each process has access to its own *volatile memory* and *stable storage*. Contrary to stable storage, the data stored in the volatile memory is lost upon crash. The combined content of the volatile and stable storage of a process p_i constitutes a *(local) state* S_i of p_i. $S = \{S_1, ..., S_N\}$ is a *replicated state* of the system.

3.2 Inter-processes Communication

Processes communicate solely by exchanging messages through a network. The network is formed by bidirectional channels called *fair-loss links* [98] maintained by every pair of processes. Messages may be lost by the links and no upper bound on message transmission is known. The failure pattern of the links is independent of the one of the processes. Fair-loss links are used to implement a higher-level abstraction called *perfect links*, which is free of many of the drawbacks of fair-loss links [99].

The above properties of processes and links imply that we deal with an *asynchronous system*. Furthermore, we assume availability of a *failure detector* Ω, which is the weakest failure detector capable of solving distributed consensus in the presence of failures [100]. Then, we can also implement the Total Order Broadcast mechanism [45], which is used by all algorithms discussed in this dissertation to disseminate messages to all processes.

TOB defines two primitives: *TO-Broadcast(m)* and *TO-Deliver(m)*. An application uses these primitives to broadcast and receive messages with the total order guarantee. Formally, the specification of TOB is given by the following four properties (we follow the specification of TOB from [101]):

- *Termination*: If a process p_i TO-Broadcasts a message m and then p_i does not crash, then p_i eventually TO-Delivers m;
- *Validity*: For any message m:
  1. every process p_i that TO-Delivers m, TO-Delivers m only if m was previously TO-Broadcast by some process p_j, and
  2. every process TO-Delivers m at most once;
- *Agreement*: If a process TO-Delivers a message m then every correct process eventually TO-Delivers m;
- *Total Order*: Let p_i and p_j be any two processes that TO-Deliver some message m. If p_i TO-Delivers some message m' before m then also p_j TO-Delivers m' before m.

The properties provide that indeed each process can broadcast messages and, if the sender does not crash for long enough, the messages will be eventually delivered in the same order by all correct processes.

3.3 Fault Tolerance

The replication robustness is influenced by the required availability of the service. Ideally, a service should be operational when all machines except one crash. However, this requirement is usually too strong since systems fulfilling it

cannot be implemented efficiently. It is because the replicas would have to extensively use stable storage in order to be able to recover in the event of failures. On the other hand, if majority of processes is up and running at any time, recovery of failed processes can be very efficient and does not require replicas to use stable storage during the normal (non-faulty) operation. All of the replication schemes discussed in this dissertation fall into the latter category.

3.4 Client Processes

In addition to service processes we consider an unspecified number of *client processes* (simply called *clients*). Clients submit *requests* to any of the service processes and await *responses*. A client may submit only one request at a time. If a client does not receive any response after submitting a request, it can choose a different replica and issue the request again. Such a situation can occur if a replica is down or a timeout was reached due to high communication latency.

Note that clients not necessarily are independent. It means that they can (directly or indirectly) communicate with each other outside the replicated service, exchange information about their interactions with the replicated service and make decisions (e.g., issue different kinds of requests) based on this information.

3.5 Requests and Transactions

We assume a simple communication interface. To communicate with a replicated service, a client sends a request message $r = \langle \texttt{Request}, id, prog, args \rangle$, where id is a unique value used to identify requests, $prog$ is a program to be executed by the replicated service (as a transaction, see below), and $args$ are the arguments necessary to execute $prog$. The request message is then handled by some service process. The service process replies to the client's request r by sending back a response message $\langle \texttt{Response}, id, res \rangle$, where res is the response computed by the replicated service.[1]

We think of a client request (more specifically the $prog$ field of a request) as a *transaction* to be executed in isolation and atomically by the replicated service. We assume that $prog$ can consist of arbitrary operations performed on shared objects (which are replicated by service processes) as well as some other (local) objects managed by the processes independently. We allow transactions to feature arbitrary code and assume no managed (sandboxed) environment, in which transactions are free to fail without impacting the local or replicated

[1] Some algorithms presented in this dissertation implement a slightly modified version of the interface: any request or response additionally features a $clock$ field which comes useful when a client changes replicas it issues requests to (see Sections 5.2, 5.3 and Chapter 6).

state. Therefore, it is very important that live transactions never observe an inconsistent state. Otherwise the transaction might cause damage, e.g., perform unexpected I/O operations, throw unhandled exceptions, or enter infinite loops [102] [103].

4
New Correctness Properties for Replication Schemes

This chapter introduces ⋄-*opacity* and ⋄-*linearizability*, two families of correctness properties suited to formalizing the guarantees of various replication schemes. We start by giving some intuition behind the new criteria. Next, we lay out basic notions, which are then used to formally define ⋄-opacity and ⋄-linearizability. Finally, we prove a formal relationship between the two families of properties.

Originally, we introduced ⋄-opacity in [7]. We further discussed it in [3], where we also described the ⋄-linearizability family of properties and established the relationship between the families. Our formal framework closely follows the one of opacity from [31]. We also borrow some definitions from [104].

4.1 Intuition Behind ⋄-Opacity and ⋄-Linearizability

It is often desirable that a replicated service provides a *real-time* guarantee on the execution of client requests. It means that if the execution of one request ends before another request starts its execution (according to, e.g., a wall-clock), the effects of the former are always visible to the latter. Providing this (intuitive) guarantee in a distributed environment is not easy and tends to be very costly. It is because respecting real-time order at all times requires inter-process synchronization on execution of every single request. Therefore, replicated services often relax the real-time requirement for at least some types of requests. E.g., read-only or aborted requests do not change the state of the system, so they do not need to be ordered w.r.t. concurrent update requests. This way the system performance can be greatly improved, especially when requests of those special types constitute the vast majority of all requests processed by the system.

To better understand the consequences of loosening real-time order for some types of requests, let us consider an example of DUR (see the detailed description of DUR in Section 5.3). In DUR, every request sent by a client to any of the replicas (server processes) is executed by the replica as an atomic transac-

tion. The execution is performed optimistically, without any synchronization with other replicas. Only after the execution of the transaction is finished, the resulting updates are broadcast to all replicas using a TOB [45], so that they can update their state accordingly. However, upon receipt of a message with a transaction's updates, the replicas do not update their state right away. In order to ensure that consistency is preserved across the system, all replicas (independently) execute a certification procedure that checks if the transaction read any stale data. If so, the transaction has to be rolled back and restarted. Since all updates are delivered in the same order (guaranteed by TOB), all replicas change their state in the same way. However, read-only transactions do not modify the local or replicated state in any way, so they do not require a distributed certification. Instead, only the replica that executed a read-only transaction certifies it to ensure that it has not read any inconsistent data. As we argue in Section 2.1, strong similarities in the way transactions are processed can be found between DUR and more complex replication schemes such as Postgres-R [39], PolyCert [40], HTR [6], E-DUR [10], and others. Therefore, the following reasoning as well as all the results presented in this dissertation are also applicable for these systems (see also Section 5.3 and Chapter 6).

Consider a DUR system, which consists of a few replicas (servers), where one of them is lagging behind, i.e., missing some updates compared to the up-to-date replicas (e.g., due to some messages being still in transit). Client c_1 interacts with an up-to-date replica, while client c_2 interacts with the lagging one. Suppose that c_1 issues an update request (an updating transaction) and receives feedback from its replica. The updates produced by this transaction do not reach c_2's replica because of the lag. If the two clients communicate with each other (directly or through some outside services), c_2 may notice it is missing some updates, or even worse, it may not notice, but still take actions, which depend on the operations issued by c_1. Indeed, if c_2 starts a new transaction on the lagging replica, c_2 will not be able to observe the updates it was already informed about. If the transaction undergoes certification, it will be aborted, because the updates it produced are broadcast using TOB and will be delivered by any replica after delivery of the updates of the transaction executed by the up-to-date replica (which are already delivered by some replicas). However, still up to some point in time (possibly until the commit attempt) the transaction handled by the lagging replica will be executed on stale data.[1] Also, if the transaction issued by c_2 is a query (a read-only transaction), the transaction may even commit (since no inter-replica synchronization is required for read-only transactions). This clearly stands in contrast with the *real-time order* requirement and shows that relaxed guarantees require more attention from the programmer so that all corner cases are handled correctly.[2]

The semantics of DUR and similar systems are not adequately formalized

[1] To prevent live transactions from reading stale data, the system would have to run a distributed consensus round before the start of each transaction.

[2] Note that the lack of the real-time guarantee in a replicated system is not problematic when we consider only independent clients, i.e., clients, whose decisions are never influenced by information provided by other clients through channels other than the replicated systems itself.

by any of the existing correctness criteria, as we argue in Section 2.2.1. Some properties that do not require that the real-time order of transaction execution is respected are too weak. In particular, serializability [14] provides no guarantees on live or aborted transactions, and update serializability [52] or extended update serializability [53] do not require all processes to witness all updates in the system in the same order. Other well established properties such as opacity [31] and TMS1 [34] are too strong, because they require that the real-time order on transaction execution is respected at all times for all types of transactions. Note that these criteria were proposed for TM systems (see e.g., [26] among others), which were meant as a concurrency control mechanism that can replace locks. Hence, the real-time order requirement is natural. Also, the real-time order is relatively easier to ensure in a local execution environment, in which TM functions.

⋄-*opacity*, which we define this chapter, is a family of closely related, opacity-based safety properties that relax in a systematic way the real-time order requirement on transaction execution. Roughly speaking, a system that satisfies any property from the ⋄-opacity family behaves as if all transactions (including the live and aborted ones) are executed sequentially. However, the requirement on the relative order in which transactions appear in the serialization depends on the considered property. In the extreme cases, either the real-time order on transaction execution has to be respected at all times (according to *real-time opacity*), or not at all (according to *arbitrary order opacity*). It means that real-time opacity, the strongest criterion from the ⋄-opacity family, is equivalent to the original definition of opacity in its prefix-closed version [31]. On the other hand, arbitrary order opacity resembles serializability, but is defined not only for committed transactions but also accounts for live and aborted ones. Currently, ⋄-opacity features four criteria in-between real-time opacity and arbitrary order opacity: *commit-real-time opacity*, *write-real-time opacity*, *update-real-time opacity* and *program order opacity*. We prove that DUR satisfies update-real-time opacity, which allows read-only and aborted transactions to operate on stale but consistent data, i.e., the data observed by read-only and aborted transactions does not necessarily reflect the most recent updates (see Section 5.3.3). Write-real-time opacity and commit-real-time opacity are intermediate properties that allow us to compare transactional replication schemes with some non-transactional replication schemes (explained below). Program order opacity is similar to virtual time opacity [54], but defined in the framework of the original definition of opacity (see also Section 2.2.1).

Alongside ⋄-opacity, we define a family of safety properties called ⋄-*linearizability*, based on linearizability [41], which are intended to formalize the guarantees offered by strongly consistent systems that hide transactions from clients. ⋄-linearizability can also be used for systems, which do not feature transactional semantics at all. In particular, the new properties are suitable for formalizing guarantees provided by various flavours of SMR. Depending on the used optimizations, SMR satisfies real-time linearizability or weaker properties such as write-real-time linearizability (see also Sections 5.1.3 and 5.2.3). Also, as we for-

mally prove, $\diamond$-linearizability preserves two important properties of linearizability: locality and non-blocking.

We also give a formal result on the relationship between $\diamond$-opacity and $\diamond$-linearizability. In order to establish a link between the two families we introduce a *gateway object*. We show that, roughly speaking, when transactions are hidden from clients, a $\diamond$-opaque replicated system is $\diamond$-linearizable. This result establishes a formal relationship between opacity and linearizability (in their original definitions) and also directly compares the guarantees provided by systems such as DUR and SMR. The definition of the gateway object is general enough so it can be used to compare other transactional and non-transactional properties.

4.2 Base Definitions

In order to reason about the guarantees provided by the replicated service, we model it as a set of *shared objects* $\mathcal{X} = \{X_1, X_2, ...\}$ accessible by processes from $\mathcal{P}$. Each shared object (or simply an *object*) has a unique identity and a type. Each type is defined by a *sequential specification* that consists of:

- a set Q of possible states for an object,
- an initial state $q_0 \in Q$,
- a set INV of operations that can be applied to an object,
- a set RES of possible responses an object can return, and
- a transition relation $\delta \subseteq Q \times INV \times RES \times Q$.

This specification describes how an object of a given type behaves, if the object is accessed one operation at a time. If $(q, op, res, q') \in \delta$, it means that applying operation op to an object in state q may move the object to state q' and the response res is returned to the process that invoked op. For simplicity, we assume that operation arguments are encoded in the operation itself.

We say that an operation op is *total*, if it is defined for every possible state of an object, i.e., if and only if for every $q \in Q$, there exists $(q, op, res, q') \in \delta$.

We say that an operation op is *updating* for a given state $q \in Q$, if and only if there exists $(q, op, res, q') \in \delta$, such that $q \neq q'$. We say that op is *read-only*, if and only if there does not exist a state q, for which op is updating. Otherwise, we say that op is *potentially updating*.

Let us consider an example. *Read/write registers* constitute an important class of shared objects. A read/write register features two simple operations: $read$, which returns the current integer value $v \in \mathbb{Z}$ of the register, and $write(i)$ for some $i \in \mathbb{Z}$, which sets the current value of the register to i and returns ok afterwards. Then, we define the sequential specification of read/write registers as $T_R = (\mathbb{Z}, 0, INV_R, RES_R, \delta_R)$, where $INV_R = \{read\} \cup \{write(i) : i \in \mathbb{Z}\}$, $RES_R = \{ok\} \cup \mathbb{Z}$, and $\delta_R = \{(i, read, i, i) : i \in \mathbb{Z}\} \cup \{(i, write(j), ok, j) : i, j \in \mathbb{Z}\}$. Both the $read$ and $write$ operations are total. Obviously, $read$ is a read-only

operation as it never modifies the state of the register, and $write$ is not ($write$ is potentially updating). However, $write$ is not always updating. When the register is in state i (its value is i), then $write(i)$ does not change its state. Therefore, $write(i)$ is not updating for state i.

Shared objects form a *hierarchy*, at the bottom of which are *base objects*. Base objects represent individual memory locations which can be read and written. Each higher level of the hierarchy contains shared objects of more abstract types. Implementations of objects at a given level may only use base objects and other lower-level shared objects.

Implementations of shared objects are defined by algorithms which describe the steps required to complete each operation. When a process invokes an operation on a shared object, it follows an appropriate algorithm. In each step, a process may either perform local computation or engage other object.

We assume that every shared object encapsulates its state, i.e., each object or local variable is only used by a single shared object implementation.

When a process p_i executes an operation op on object $X \in \mathcal{X}$, it invokes an event $X.inv_i(op)$ and expects a response event $X.resp_i(v)$. A pair of such events is called a *(completed) operation execution* and is denoted by $X.op \rightarrow_i v$. An invocation event, that is not followed by a response event, is called a *pending operation execution*. $X.op \rightarrow_i v$ is, respectively, read-only, potentially updating or updating, if op is read-only, potentially updating or updating.

We model the system execution as a (totally ordered) sequence of events called a *history*. Naturally, histories respect program order (events executed by the same process are ordered according to their execution order), and also causality between events across processes (if two events executed in the system are causally related, one will precede the other in the history according to the causality relation). Events that happen in parallel (in separate processes), and that are not causally dependent, can appear in a history in an arbitrary order.[3] For any history H, we denote by $H|p_i$ the restriction of H to events issued or received by process p_i. We denote by $H|X$ (where $X \in \mathcal{X}$) the restriction of H to operations executed on X and their appropriate responses, and by $H|S$ (where $S \subseteq \mathcal{X}$) the restriction of H to operations executed on objects from the set S and their appropriate responses.

A history which is not restricted to any particular object or a set of objects is called an *implementation history*. On the other hand, a *high-level history* is a history $H|S$ restricted to a set of objects $S = \{X_1, X_2, ...\}$, such that no object X_i is used by the implementation of some other object X_j. Unless stated otherwise, when we say a history we mean a high-level history.

An implementation history H is said to be *well-formed*, if for every process p_i and every shared object X, $(H|X)|p_i$ is a (finite or infinite) sequence of operation

[3] An alternative approach to model the system execution would be to employ *partially ordered sets* (or *posets*). This approach is equivalent to ours, because a poset can be represented by a set of totally ordered histories (and we always analyse all possible histories a system can produce). We argue that an approach based on totally ordered histories is more elegant, because it better matches the sequential specifications used for t-objects. We also want to stay close to, and remain compatible with, the original formal framework of opacity from [31].

executions, possibly ending with a pending operation execution. A high-level history H is well-formed, if for every process p_i, $H|p_i$ is a (finite or infinite) sequence of operation executions, possibly ending with a pending operation execution. We consider only well-formed histories.

4.3 The $\diamond$-Opacity Family of Properties

In this section, we define the $\diamond$-opacity family of safety properties. First we introduce some auxiliary definitions and then specify the properties. Finally, we provide a discussion on the properties' characteristics and show some examples.

4.3.1 Formal Definition

Let us start by distinguishing a subset of shared objects $\mathcal{Q} \subset \mathcal{X}$ called *t-objects* and defining a set $\mathcal{T} = \{T_1, T_2, ...\}$ of *transactions*.

The set $\mathcal{Q} = \{x_1, x_2, ...\}$ of t-objects consists of a special class of shared objects that cannot be accessed directly by processes. Instead, they have to be accessed within a context called a *transaction*, an abstract notion fully controlled by some process. Also, an operation on a t-object cannot return a value that belongs to a set $A = \{A_1, A_2, ...\}$ of special return values used by the system.

Interaction between processes and t-objects is managed by a single *transactional memory shared object (TM object)* M of the following interface:

- $M.texec(T_k, x.op) \rightarrow_i \{v, A_k\}$ which denotes that process p_i executes an operation op on a t-object x of some type $T = (Q, q_0, INV, RES, \delta)$ within a transaction T_k and as a result produces a return value $v \in RES$ or a special value $A_k \in A$;
- $M.tryC(T_k) \rightarrow_i \{A_k, C_k\}$ which denotes that process p_i attempts to commit a transaction T_k and returns the special values $A_k \in A$ or C_k;
- $M.tryA(T_k) \rightarrow_i A_k$ which denotes that process p_i aborts a transaction T_k and returns $A_k \in A$.

Each operation on a TM object (executed by some process p_i) can return a special value A_k, which indicates that the transaction T_k that executed this operation has aborted. The value C_k, returned by the operation $tryC(T_k)$, means that T_k has committed. For any t-object of type $T = (Q, q_0, INV, RES, \delta)$, $A_k \notin RES$ and $C_k \notin RES$. A response event with a return value A_k or C_k is called, respectively, an *abort event* or *commit event* (of transaction T_k). The commit or abort events of transaction T_k are always the last events for T_k.

Let H_I be an implementation history and M be a TM object. Then, we use the term *t-history* for a high-level history $H_I|M$. Let H be a t-history of some TM object M. We denote by $H|T_k$ the restriction of H to events concerning T_k, i.e. invocation events of operations $M.texec(T_k, x.op)$, $M.tryC(T_k)$, $M.tryA(T_k)$ and their corresponding response events (for any t-object x and operation op).

We say that a transaction T_k is in H, if $H|T_k$ is not empty. Let x be any t-object. We denote by $H|x$ the restriction of H to events concerning x, i.e., the invocation events of any operation $M.texec(T_k, x.op)$ and their corresponding response events, for any transaction T_k and operation op on x.

We say that a transaction T_k performs some action, when a given process executes this action as part of T_k. A transaction that only executes read-only operations in a given t-history H is called a *read-only transaction* in H. Otherwise, we say that it is an *updating transaction* in t-history H. In general, during an execution of a transaction, it is impossible to tell whether it will not perform any updating operations before it finishes its execution. However, we distinguish a special class of transactions called *declared read-only* (*DRO*), which are known *a priori* to be always read-only in any t-history (they are only allowed to execute read-only operations on t-objects). Then, for any such transaction T_k, we write $DRO(T_k) = true$. Every transaction T_k, for which $DRO(T_k) = true$, T_k is read-only, but the opposite is not necessarily true. A transaction that is *not* declared read-only is called a *potentially updating transaction*. Every updating transaction is potentially updating, and every read-only transaction that is not declared read-only is also potentially updating (no matter whether the read-only transaction is still live or already completed).[4]

We say that a transaction T_k is *committed* in a t-history H, if $H|T_k$ contains operation execution $M.tryC(T_k) \rightarrow_i C_k$ (for some process p_i). We say that the transaction T_k is *aborted* in H, if $H|T_k$ contains response event $resp_i(A_k)$ from any operation of T_k (for some process p_i). If an aborted transaction T_k contains an invocation of the operation $M.tryA(T_k)$, it is said to be *aborted on demand*. Otherwise, we say that T_k is *forcibly aborted* in H. A transaction T_k in H that is committed or aborted is called *completed*. A transaction that is not completed is called *live*. A transaction T_k is said to be *commit-pending* in H, if $H|T_k$ has a pending operation $M.tryC(T_k)$ (T_k invoked the operation $M.tryC(T_k)$, but has not received any response from this operation).

We say that a t-history H is *t-completed*, if every transaction T_k in H is completed. A *t-completion* of a t-history H is any (well-formed) t-completed t-history $\bar{H}$ such that:

1. H is a prefix of $\bar{H}$, and
2. for every transaction T_k in H, sub-history $\bar{H}|T_k$ is equal to one of the following histories:
   - $H|T_k$, when T_k is completed, or
   - $H|T_k \cdot \langle tryA(T_k) \rightarrow_i A_k \rangle$, for some process p_i, when T_k is live and there is no pending operation in $H|T_k$, or
   - $H|T_k \cdot \langle resp_i(A_k) \rangle$, when T_k is live and there is a pending operation in $H|T_k$ invoked by some process p_i, or

[4]The distinction between read-only and declared read-only (and between updating and potentially updating) transactions is relevant for some systems and is reflected in different ordering relations which we define later. Sometimes it is useful to treat a potentially updating transaction as an updating one, even though it does not produce any updates. Such a transaction resembles a *write* operation which is not always updating.

- $H|T_k \cdot \langle resp_i(C_k) \rangle$, when T_k is commit-pending for some process p_i.

Let T_i and T_j be any two transactions in some t-history H, where T_i is completed. We define the following order relations on transactions in H:

- *real-time order* $\prec^r_H$ — we say that $T_i \prec^r_H T_j$ (read as T_i *precedes* T_j), if the last event of T_i precedes the first event of T_j;
- *commit-real-time order* $\prec^c_H$ — we say that $T_i \prec^c_H T_j$, if (1) $T_i \prec^r_H T_j$, and (2a) both T_i and T_j are committed or (2b) both T_i and T_j are executed by the same process p_k;
- *write-real-time order* $\prec^w_H$ — we say that $T_i \prec^w_H T_j$, if (1) $T_i \prec^r_H T_j$, and (2a) both T_i and T_j are potentially updating and are committed or (2b) both T_i and T_j are executed by the same process p_k;
- *update-real-time order* $\prec^u_H$ — we say that $T_i \prec^u_H T_j$, if (1) $T_i \prec^r_H T_j$, and (2a) both T_i and T_j are updating and are committed or (2b) both T_i and T_j are executed by the same process p_k;
- *program order* $\prec^p_H$ — we say that $T_i \prec^p_H T_j$, if $T_i \prec^r_H T_j$ and both T_i and T_j are executed by the same process p_k;
- *arbitrary order* $\prec^a_H$ — equivalent to $\emptyset$. $T_i \prec^a_H T_j$ never holds true.

Note that (2a) and (2b) are not mutually exclusive.

Let H, H' be two t-histories. We say that H' respects the $\diamond$-order of H (where $\diamond$ is any of the order relations specified above), if and only if $\prec^\diamond_H \subseteq \prec^\diamond_{H'}$, which means that for any two transactions T_i and T_j in H, if $T_i \prec^\diamond_H T_j$ then $T_i \prec^\diamond_{H'} T_j$. For any t-history H the following holds: $\emptyset = \prec^a_H \subseteq \prec^p_H \subseteq \prec^u_H \subseteq \prec^w_H \subseteq \prec^c_H \subseteq \prec^r_H$.

We say that T_i and T_j are *concurrent* if neither $T_i \prec^r_H T_j$ nor $T_j \prec^r_H T_i$. We say that any t-history H is *t-sequential*, if H has no concurrent transactions.

Let S be any t-completed t-sequential t-history, such that every transaction in S, possibly except the last one, is committed. We say that S is *t-legal*, if for every t-object x, the subhistory $S|x = \langle texec(T_i, x.op_1) \rightarrow_k res_1, texec(T_j, x.op_2) \rightarrow_l res_2, ... \rangle$ (for any processes p_k, p_l, ..., and for any transactions T_i, T_j, ...) satisfies the sequential specification $(Q, q_0, INV, RES, \delta)$ of x, i.e., there exists a sequence of states $q_1, q_2, ...$ in Q, such that $(q_{n-1}, op_n, res_n, q_n) \in \delta$ for any $n > 0$.

Let S be any t-completed t-sequential t-history. We denote by $visible_S(T_k)$ the longest subsequence S' of S, such that for every transaction T_i in S', either (1) $i = k$, or (2) T_i is committed and T_i precedes T_k. We say that a transaction T_k in S is *legal* in S, if the t-history $visible_S(T_k)$ is t-legal.

We say that t-histories H and H' are *equivalent*, denoted $H \equiv H'$, if for every transaction T_k in $\mathcal{T}$, $H|T_k = H'|T_k$.

Definition 1. *A finite t-history H is* final-state $\diamond$-opaque, *if there exists a t-sequential t-history S equivalent to some t-completion of H, such that:*

1. *every transaction T_k in S is legal in S, and*
2. *S respects the $\diamond$-order of H.*

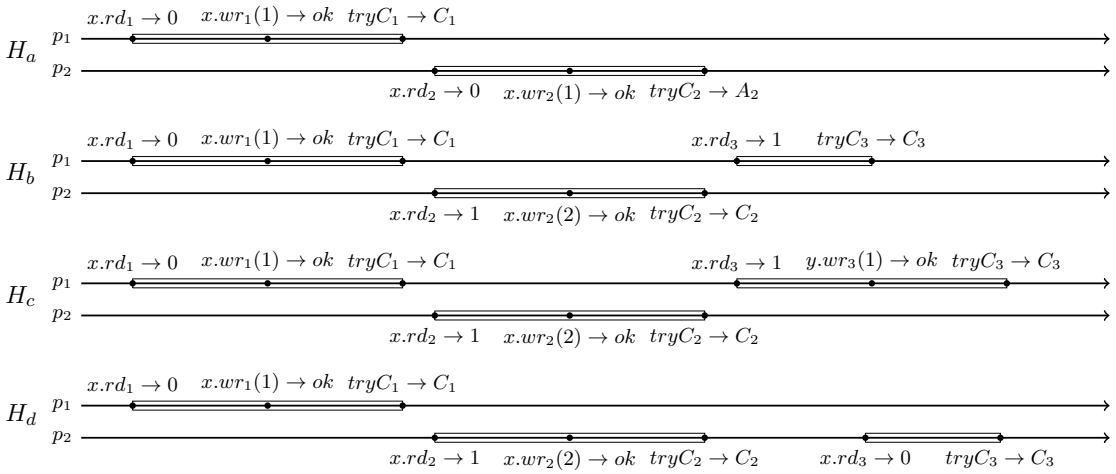


Figure 4.1: Example t-histories for two processes and up to three transactions. H_a is commit-real-time opaque, but not real-time opaque. H_b is update-real-time opaque, but not commit-real-time opaque. Additionally, H_b is write-real-time opaque, if and only if T_3 is known *a priori* to be read-only (the $DRO(T_3)$ predicate holds true in H_b). H_c is program order opaque, but not update-real-time opaque nor write-real-time opaque. H_d is arbitrary order opaque, but not program order opaque. We use simplified notation: explicit calls to the TM object M as well as process numbers are omitted, lower indices indicate the transaction number, rd is a read operation and wr is a write operation.

Definition 2. *A t-history H is* ⋄-opaque, *if every finite prefix of H is final-state ⋄-opaque.*

Definition 3. *A system (modelled as a TM object) is ⋄-opaque, if every history it produces is ⋄-opaque.*

In the above three definitions ⋄ can be either real-time, commit-real-time, write-real-time, update-real-time, program order, or arbitrary order. Therefore, we obtain a whole family of ⋄-opacity properties. Real-time opacity is equivalent to opacity. By substituting real-time with weaker ordering guarantees we obtain gradually weaker properties with arbitrary order opacity being the weakest one. Note that all these properties require a history to be legal.

4.3.2 Discussion

Real-time opacity, which is equivalent to the original definition of opacity in its prefix-closed version [31], requires that all transactions, regardless of their state of execution (live, aborted, commit-pending or committed) always observe a consistent and most recent view of the system. Commit-real-time opacity relaxes opacity, by restricting the real-time order to only committed transactions. It means that aborted transactions may observe stale, but still consistent data; no artefacts such as reads out of thin air are possible. Write-real-time opacity and update-real-time opacity additionally relax the real-time order requirement on

transactions that, respectively, are known *a priori* to be read-only (are declared read-only), or that have not performed any potentially updating operations (are read-only). Program order opacity ensures that transactions respect program order, i.e., the order of execution of all local transactions has to be respected across all processes. Finally, arbitrary order opacity imposes no requirements on the order of transactions' execution, as long as all transactions are legal.

Write-real time opacity is only suitable for systems that can distinguish between transactions that have not performed any potentially updating operations and transactions known *a priori* to be read only (the DRO predicate is true only for the latter ones). In such systems, the additional information about transactions can be either provided by the programmer, or can be deduced prior to a transaction execution from the transaction code. By manually marking some transactions as declared read-only, a programmer can decide whether a read-only transaction T_k may read stale data ($DRO(T_k)$ holds true) or has to respect the real-time order ($DRO(T_k)$ is false). We can make the following two simple observations, which follow directly from the definitions. Firstly, in a write real-time opaque t-history H, if there are no transactions for which the DRO predicate holds, H is also commit-real-time opaque. Secondly, in an update real-time opaque t-history H, if for all read-only transactions the predicate DRO holds, then H is also write real-time opaque.

Note that not every updating transaction modifies the state of any t-object. Consider a transaction T_k which first reads value i from a register x and then writes the same value i to x, or just happens to write i to x without reading x beforehand. Clearly, T_k is not a read-only transaction as it executes a write operation (which is a potentially updating operation). However, in this particular scenario, the write is not updating, because the state of x remains unchanged. A similar example can be formulated for transactions that execute operations on any objects featuring operations that are not always updating (see Section 4.2).

Our definition of update-real-time does not differentiate between transactions that performed potentially updating operations, which modified state of some t-objects, and those, which just happened not to change the value of the t-objects, as in the example above. Doing so would result in a correctness property that has no real practical application. None of the real-world transactional systems we are aware of treat differently these two kinds of transactions and neither does DUR (see 5.3). On the contrary, making such a distinction is necessary when considering sequences of operation executions, as in $\diamond$-linearizability (see Section 4.4).

Figure 4.1 illustrates the relationship between the members of the $\diamond$-opacity family by example. It depicts four t-histories (two variants of t-history H_b can be deduced depending on the value of $DRO(T_3)$). Each t-history represents a case when one property is satisfied while another, a stronger one, is not.

T-histories H_a and H_b represent our main motivation: enabling aborted and read-only transactions to read from a stale (but consistent) snapshot. Let us first consider H_a. Transactions T_1 and T_2 access the same t-object x. T_1 precedes T_2, however T_2 reads a stale value of x, and subsequently aborts. The only possible

serialization of H_a in which all transactions are legal is $\langle H_a|T_2 \cdot H_a|T_1\rangle$. This serialization does not respect the real-time order, as clearly $T_1 \prec^r_{H_a} T_2$. Therefore, H_a breaks (real-time) opacity. However, it satisfies commit-real-time opacity, because T_2 is aborted and therefore it may observe stale data.

In t-history H_b, transaction T_3, which does not perform any updating operations, is preceded by transaction T_2. However, T_3 does not observe the operation $x.wr_2(2)$ of T_2, as its operation $x.rd_3$ returns the value written by T_1. Therefore, H_b breaks real-time opacity. Moreover, it also breaks commit-real-time opacity. On the other hand, H_b satisfies write-real-time opacity when $DRO(T_3)$ holds, and update-real-time opacity when $DRO(T_3)$ does not hold true. Note that if T_3 would abort, then H_b would be commit-real-time opaque.

In t-history H_c, similarly to the previous example, T_3 does not obey the real-time order. This time, however, T_3 is an updating transaction which commits. Therefore, the history only satisfies program order opacity and not update-real-time opacity, nor any stronger property. Finally, in t-history H_d, even the program order is not preserved, as p_2's transaction T_3 does not observe the effects of another transaction (T_2) executed by p_2 earlier. This t-history, however, satisfies arbitrary order opacity, as transactions T_2 and T_3 can be reordered, yielding an equivalent legal execution $\langle T_3 \cdot T_1 \cdot T_2\rangle$. This trait makes arbitrary order opacity similar to serializability.

Now we show that ⋄-opacity is a safety property [67] [68], similarly to opacity. The full proofs of all formal results can be found in the Appendix.

Theorem 1. *⋄-opacity is a safety property.*

Proof sketch. ⋄-opacity is non-empty since a t-history $H = \langle\rangle$ is ⋄-opaque. Since ⋄-opacity of a t-history H is defined through final-state ⋄-opacity on all of its prefixes (which are finite), it is easy to show that ⋄-opacity is prefix and limit-closed. Therefore. ⋄-opacity is a safety property. □

Corollary 1. *A system (modelled as a TM object) is ⋄-opaque if, and only if every finite history it produces is final-state ⋄-opaque.*

Proof. The proof follows directly from Theorem 1. □

4.4 The ⋄-Linearizability Family of Properties

In this section, we introduce the ⋄-linearizability family of safety properties. We begin by giving a formal definition and then we discuss the characteristics of ⋄-linearizability.

4.4.1 Formal Definition

We provide the definition of $\diamond$-linearizability for systems where shared objects may be *abortable*. It means that any operation op invoked on an abortable shared object $X \in \mathcal{X}$ (for some operation execution o), may return a special value $\perp$, thus indicating that the execution of op failed and did not change X's state. More precisely, in the sequential specification of an abortable object, the transition $(q, op, \perp, q)$ is possible for any operation op and state q. In such case, we say that operation op *aborted*. Otherwise, i.e., when op returns some value $v \neq \perp$, we say that operation op *committed*. An operation execution $X.op \rightarrow_i v$ is, respectively, aborted or committed, if op aborted or committed.

A history which has no pending operation executions is called *completed*. A *completion* of a history H is any (well-formed) completed history $\bar{H}$ such that $\bar{H}$ consists of all completed operation executions from H and appropriate response events for a subset of pending operation executions in H (i.e., some pending operation executions in H appear completed in $\bar{H}$ or do not appear at all).[5]

Let o_i and o_j be any two operation executions in some history H, where o_i is a completed operation execution. We define the following order relations on operations in H:

- *real-time order* $<^r_H$ — we say that $o_i <^r_H o_j$ (read as o_i *precedes* o_j), if the response event of o_i precedes the invocation event of o_j;
- *commit-real-time order* $<^c_H$ — we say that $o_i <^c_H o_j$, if (1) $o_i <^r_H o_j$, and (2a) both o_i and o_j committed or (2b) both o_i and o_j are executed by the same process p_k;
- *write-real-time order* $<^w_H$ — we say that $o_i <^w_H o_j$, if (1) $o_i <^r_H o_j$, and (2a) both o_i and o_j are potentially updating and committed or (2b) both o_i and o_j are executed by the same process p_k;
- *program order* $<^p_H$ — we say that $o_i <^p_H o_j$, if $o_i <^r_H o_j$ and both o_i and o_j are executed by the same process p_k;
- *arbitrary order* $<^a_H$ — equivalent to $\emptyset$. Never $o_i <^a_H o_j$ holds true.

Note that (2a) and (2b) are not mutually exclusive.

Let H, H' be two histories. We say that H' respects the $\diamond$-order of H (where $\diamond$ is any of the order relations specified above), if and only if $<^\diamond_H \subseteq <^\diamond_{H'}$, which means that for any two operation executions o_i and o_j in H, if $o_i <^\diamond_H o_j$ then

[5] Note that the definition of completion and t-completion from Section 4.3 are quite different. t-completion is always a prefix of the original t-history, whereas a completion may lack some of the events of the original history. Such a formulation is necessary to account for non-abortable objects which have some non-total operations. Consider a history of a shared object implementing a blocking queue. Let us assume that the history contains a pending *dequeue* operation and no *enqueue* operations. Clearly, it is impossible to complete this history by adding some (legal) return event for the *dequeue* operation. Yet, it is still possible that a future arrival of an *enqueue* operation will allow the pending *dequeue* operation to complete. Therefore, the definition of completion has to be more admitting. Note also that we could use a more straightforward definition of completion, i.e., similar to the definition of t-completion. It is because we define $\diamond$-linearizability for abortable objects. However, in order to maintain compatibility with the original definition of linearizability, we opted not to do so.

$o_i <^{\diamond}_{H'} o_j$. For any history H the following holds: $\emptyset = <^a_H \subseteq <^p_H \subseteq <^w_H \subseteq <^c_H \subseteq <^r_H$.

We say that o_i and o_j are *concurrent*, if neither $o_i <^r_H o_j$ nor $o_j <^r_H o_i$. We say that any history H is *sequential*, if H has no concurrent operation executions.

Let S be any completed sequential history. We say that S is *legal*, if for every object X, the subhistory $S|X = \langle X.op_1 \rightarrow_k v_1, X.op_2 \rightarrow_l v_2, ...\rangle$ (for any processes p_k, p_l, ...) satisfies the sequential specification $(Q, q_0, INV, RES, \delta)$ of X, such that there exists a sequence $W = \langle q_0, q_1, ...\rangle$ of states in Q and for any $n > 0$, $(q_{n-1}, op_n, v_n, q_n) \in \delta$. We call any such sequence a *witness history* of S. For any operation execution $o_n = X.op_n \rightarrow_k v_n$ in S, we say that o_n is updating in S according to W, if $q_{n-1} \neq q_n$.

We say that histories H and H' are equivalent, denoted $H \equiv H'$, if H' contains all of the events of H and vice versa.

We distinguish yet another order relation whose definition we give below. Let H be any history and S be a legal sequential history equivalent to some completion of H. Let W be some witness history of S. Let o_i and o_j be any two operation executions in H, where o_i is a completed operation execution. We define the following order relation on operations in H (according to the sequential history S and the witness history W):

- *update-real-time order* $<^u_H (S, W)$ — we say that $o_i <^u_H (S, W)\ o_j$, if (1) $o_i <^r_H o_j$, and (2a) both o_i and o_j are updating in S according to W and are committed or (2b) both o_i and o_j are executed by the same process p_k.[6]

Let H, H' be two histories. We say that H' respects the update-real-time order of H according to S and W, if and only if $<^u_H (S, W) \subseteq <^u_{H'} (S, W)$, which means that for any two operation executions o_i and o_j in H, if $o_i <^u_H (S, W)\ o_j$ then $o_i <^u_{H'} (S, W)\ o_j$. When $H' = S$, we say that S respects the update-real-time order of H, if and only if there exists W, a witness history of S, such that $<^u_H (S, W) \subseteq <^u_S (S, W)$. In such case, we simplify the notation and write $<^u_H \subseteq <^u_S$. For any history H and any legal history S equivalent to some completion of H (with any witness history W) the following holds: $\emptyset = <^a_H \subseteq <^p_H \subseteq <^u_H (S, W) \subseteq <^w_H \subseteq <^c_H \subseteq <^r_H$.

Now, let us provide the definition of $\diamond$-linearizability.

Definition 4. *A finite history H is* final-state $\diamond$-linearizable*, if there exists a sequential history S, such that:*

1. *S is equivalent to $\bar{H}$, a completion of H,*
2. *S is legal, and*
3. *S respects the $\diamond$-order of $\bar{H}$.*

Definition 5. *A history H is* $\diamond$-linearizable*, if every finite prefix of H is final-state $\diamond$-linearizable.*

[6] Unlike other orders defined here, update-real-time order can only be considered in the context of some execution, because for different sequential histories S (equivalent to some completion of some history H), some operations in H may or may not be updating (depending on the previous values of the objects on which the operations are performed). Hence, update-real-time order is defined according to some witness history.

Definition 6. *A system (modelled as a set of shared objects) is $\diamond$-linearizable, if every history it produces is $\diamond$-linearizable.*

In the above two definitions $\diamond$ can be either real-time, commit-real-time, write-real-time, update-real-time, program order, or arbitrary order.

Note that, we require that S respects the $\diamond$-order of $\bar{H}$ and not of H. It is because S, which is equivalent to $\bar{H}$, may lack some of the operations that are pending in H (by the definition of completion of H).

4.4.2 Discussion

$\diamond$-linearizability relaxes the operation ordering guarantees of the original definition of linearizability [41] in a similar way to which $\diamond$-opacity does it for transaction ordering guarantees of the original definition of opacity [31]. Note that we consider abortable shared objects as well as the ordinary ones.[7] Not only this way we can encompass a wider array of systems (shared object implementations), but it also allows us to use $\diamond$-linearizability to describe the behaviour of transactional systems from the point of view of external clients (see Section 4.5).

The difference between write-real-time- and update-real-time linearizability is to some extent similar to the difference between write-real-time- and update-real-time opacity. Write-real-time linearizability relaxes the real-time order requirement for read-only operations, i.e., operations that never change the state of the object, such as a *read* operation on a register. Update-real-time linearizability relaxes real-time also for operations that in a given execution did not change the state of the object, i.e., a non-blocking pop operation performed on an empty stack.

Now let us highlight the subtle difference in the analogy between write/update-real-time opacity and write/update-real-time linearizability. Consider an example in Figure 4.2. X is a shared object implementing an initially empty stack on which two processes execute $push$ and pop operations ($push$ is always updating and pop is a potentially updating operation that immediately returns, if the stack is empty). These operations are executed directly on X in history H. In t-history H_t, processes operate on a stack t-object x in an analogous way as in H, but all operations are executed within separate transactions. Trivially, H is not write-real-time linearizable. If it were, the first $X.pop$ would have to return 5 instead of $null$, because it was issued after $X.push(5)$ committed and write-real-time linearizability requires that committed and updating or potentially updating operations respect the real-time order of operation execution. However, H is update-real-time linearizable: H is completed and there exists a legal sequential history $S = \langle X.pop \rightarrow null, X.push(5) \rightarrow ok, X.pop \rightarrow 5\rangle$ which is equivalent to H and respects the update-real-time order of H ($X.pop \rightarrow null$ does not modify the state of X in this particular execution, so can be moved in S before $X.push(5) \rightarrow ok$).

[7] Trivially, in the latter case commit-real-time linearizability is equivalent to real-time linearizability. As we show later, surprisingly both properties are equivalent also for abortable shared objects.

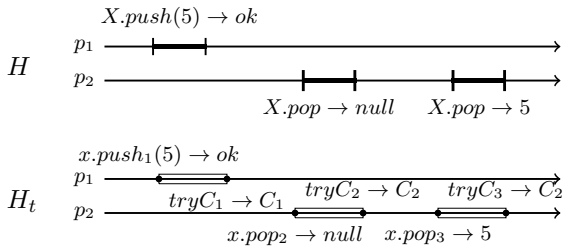


Figure 4.2: History H is not write-real-time linearizable, but is update-real-time linearizable. t-history H_t is neither write- nor update-real-time opaque.

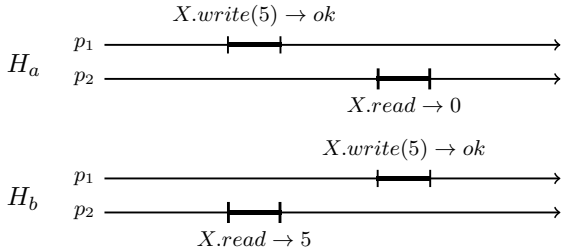


Figure 4.3: History H_a is update-real-time linearizable, whereas H_b is not.

On the other hand, H_t is neither update-real-time opaque nor write-real-time opaque. Even though transaction T_2 does not modify the state of any t-object (including x), we still treat T_2 as an updating transaction. This is because update-real-time opacity does not differentiate between transactions which performed potentially updating operations that modified the state of some t-objects, and those, which happened not to change the value of the t-objects (see Section 4.3.2). Therefore, H_t is not write-real-time opaque either, as this is a stronger property.

Now let us focus on the way ⋄-linearizability is formalized. Naturally, our property is based on linearizability [41]. However, unlike the original definition, ⋄-linearizability is defined indirectly, through final-state ⋄-linearizability, i.e., in a similar way to which opacity/⋄-opacity are defined through final-state opacity/⋄-opacity. This is because final-state ⋄-linearizability is not prefix-closed, if we consider any order relation weaker than real-time order.

To better understand why introducing the intermediate step in the definition of ⋄-linearizability is necessary, let us consider two example histories H_a and H_b in Figure 4.3. Our main motivation to relax real-time order was to enable executions such as the one in history H_a, where one process (p_1) modifies the state of some object and then, long after the operation committed, a lagging process (p_2) observes a stale value of this object by performing a read-only operation. Naturally, H_a is final-state update-real-time linearizable, so is H_b, where one operation ($X.read \rightarrow 5$) *observes* the future state of some object. Clearly, H_b represents an execution that should not be allowed. The difference between H_a and H_b lies in the fact that every prefix of H_a is update-real-time linearizable, and there exists a prefix of H_b, which is not ($H'_b = \langle X.read \rightarrow 5 \rangle$). Hence, we require

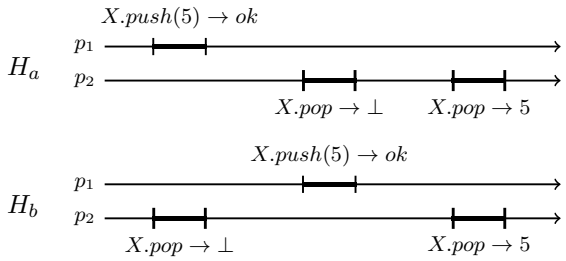


Figure 4.4: Both histories H_a and H_b are commit-real-time linearizable but also real-time linearizable. (The return value $\perp$ means that the operation aborted.)

that every finite prefix of a $\diamond$-linearizable history is final-state $\diamond$-linearizable.

Next we show that $\diamond$-linearizability, as $\diamond$-opacity, is a safety property.

Theorem 2. *$\diamond$-linearizability is a safety property.*

Proof. The proof is analogous to the proof of Theorem 1. □

As proven in [104], linearizability in its original definition is a safety property only for objects which are finitely non-deterministic (thus also for deterministic objects). For objects which are infinitely non-deterministic, linearizability is not limit-closed and thus it is not a safety property. On the other hand, $\diamond$-linearizability (thus also real-time linearizability) is more general as it is trivially limit-closed also for the latter ones. Hence real-time linearizability is equivalent to the original definition of linearizability, but only when we consider objects which are finitely non-deterministic.

Corollary 2. *A system (modelled as a set of shared objects) is $\diamond$-linearizable, if and only if every finite history it produces is final-state $\diamond$-linearizable.*

Proof. The proof follows directly from Theorem 2. □

$\diamond$-linearizability preserves two important properties of linearizability: *locality* and *non-blocking* [41]. The former requires that a system is $\diamond$-linearizable, if and only if every shared object managed by the system is $\diamond$-linearizable. The latter requires that every finite $\diamond$-linearizable history has an extension that is also $\diamond$-linearizable.

Theorem 3. *$\diamond$-linearizability is non-blocking and satisfies locality.*

Proof sketch. The proof is inspired by the proof in [41] of a similar theorem but regarding linearizability in its original definition. □

Now, let us show an interesting result that highlights the relationship between commit-real-time- and real-time linearizability. Consider histories H_a and H_b from Figure 4.4. In both histories, $X.push(5) \rightarrow ok$ pushes a value onto the stack X, whereas $X.pop \rightarrow \perp$ intends to remove a value from X but aborts. Clearly, the order of execution of these two operations does not matter, both histories are commit-real-time and real-time linearizable. It would not be so,

if $X.pop$ had not aborted. However, an aborted operation does not change the state of the object. As the special value $\perp$ carries no information on the current state of the object or why the operation failed, we can prove the following result.

Theorem 4. *Commit-real-time linearizability is equivalent to real-time linearizability.*

Proof sketch. We start by showing that in order to prove that commit-real-time linearizability is equivalent to real-time linearizability, it suffices to show that every final-state commit-real-time linearizable history is final-state real-time linearizable and vice versa.

Since, $<^c_H \subseteq <^r_H$, by the definition of final-state ⋄-linearizability, every final-state real-time linearizable history is trivially also final-state commit-real-time linearizable. The difficult part of the proof concerns showing that any final-state commit-real-time linearizable history H is also final-state real-time linearizable. We do so in two major steps.

First we take some completion $\bar{H}$ of H and then we create a directed graph G such that operation executions from $\bar{H}$ form vertices and there is an edge in G for any two preceding (in real-time) operation executions in $\bar{H}$. We then take S, a commit-real-time linearizable sequential history equivalent to $\bar{H}$, strip all aborted operations from S and use the resulting history S' to supplement G with additional edges reflecting the precedence order of operation executions in S'. We then show that the resulting graph G' is acyclic.

Since G' is acyclic and includes an edge for every pair of preceding operation executions in $\bar{H}$, history $S_{G'}$ obtained by performing a topological sort of G' is a sequential history which maintains this precedence. By construction of $S_{G'}$ it is easy to show that $S_{G'}$ is equivalent to $\bar{H}$.

We now move to the second major part of the proof and show that the constructed history $S_{G'}$ is legal. We do so in three steps. First, we consider a history S'', which is constructed by removing from $S_{G'}$ all aborted operation executions, and show that it is equivalent to S'. Then, we use a lemma (Lemma 3 in the Appendix) to prove that S' is legal. The lemma states that given two sequential histories S_1 and S_2 which only differ in that S_2 features some aborted operation o_a and S_1 does not, S_1 is legal if and only if S_2 is legal. In consequence S'' is also legal. Finally, we use the lemma again to prove that $S_{G'}$ is legal.

Since $S_{G'}$ is a sequential history equivalent to $\bar{H}$, $S_{G'}$ is legal, and $S_{G'}$ respects the real-time order of $\bar{H}$, therefore H is final-state real-time linearizable which concludes the proof. □

Note that an aborted operation is substantially different from an aborted transaction as one can witness the results of partial transaction execution (e.g., a return value of a TM operation o executed prior to the transaction's abort). Hence, commit-real-time opacity and real-time opacity are not equivalent.

4.5 Relationship Between ⋄-Opacity and ⋄-Linearizability

In order to show the relationship between ⋄-opacity and ⋄-linearizability, we introduce a *gateway shared object*, whose sole purpose is to simply execute a transactional program as a single operation thus hiding the results of the intermediate steps of transactional execution from the client.

A gateway shared object G is an abortable shared object implemented using a TM object M, as shown in Algorithm 1. The interface of G consists of a single operation $G.perform(prog_k) \rightarrow_i \{v_k, \perp\}$ where $prog_k$ is a program executed as a single transaction T_k by process p_i. $prog_k$ consists of steps that either perform some local computation, operate on t-objects managed by M, or control the flow of the transaction. Upon invocation of $perform(prog_k)$, a process p_i performs steps of $prog_k$ one by one starting from the first one. The effects of step execution are recorded in a special variable called $context$ (line 2). It stores values of temporary variables, state of the program execution (e.g., which step is next), and any other information required to execute the program (as defined by $prog_k$ itself). All the operations on t-objects, which form a part of some step, are executed by p_i through object M (we assume that each step contains at most one such operation). More precisely, for every call of operation op on a t-object x in $prog_k$, p_i invokes $M.texec(T_k, x.op)$ (line 7), where T_k is a transaction spawned to execute $prog_k$. Since the execution of the step was substituted by an operation on M, its execution is simulated to update the $context$ variable as if the step was executed directly (line 9). To this end we use the return value produced by $M.texec(T_k, x.op)$. The first execution of the $texec$ operation marks the start of transaction T_k. If $texec$ returns A_k, the execution of the rest of $prog_k$ is cancelled, T_k is aborted and $\perp$ becomes the return value for the $G.perform(prog_k)$ call (line 8). Transaction T_k is aborted on demand, if the execution of $prog_k$ produces a step containing an $abort$ command (line 12). The execution of the last operation specified within $prog_k$ is followed by the invocation of $M.tryC(T_k)$ (line 14), which attempts to commit the current transaction. If this operation succeeds and returns C_k, then $G.perform(prog_k)$ returns $context$ (line 16); otherwise, $\perp$ is returned (line 15).

The sequential specification $(Q, q_0, INV, RES, \delta)$ of G is given as follows:

- Q is a set of all possible combined states of all t-objects in M,
- q_0 is the combined initial state of all t-objects in M,
- INV is the set containing all operations $perform(prog_k)$, where $prog_k$ is any (correct) transactional program that eventually terminates,
- RES is the set of all possible results obtained by executing any $prog_k$,
- δ is a transition relation such that $\delta \subseteq Q \times INV \times RES \times Q$.

$(q, op, res, q') \in \delta$ if either $q = q'$, $op = perform(prog_k)$ and $res = \perp$, or $op = perform(prog_k)$ and $res = v_k$ ($v_k \neq \perp$), which is obtained as a result of execution of $prog_k$ on t-objects managed by M in state q. Then, q' is the state of (possibly)

Algorithm 1 Implementation of a gateway shared object G for process p_i

```
1: function PERFORM(text prog_k)
2:     context ← null
3:     while true do
4:         step ← FETCHNEXTSTEP(prog_k, context)
5:         if step is local computation then context ← EXECUTE(step, context)
6:         if step involves an operation op on a t-object x then
7:             v ← M.texec(T_k, x.op)
8:             if v = A_k then return ⊥
9:             context ← SIMULATESTEPEXECUTION(step, context, v)
10:        if step is an abort command then
11:            M.tryA(T_k)
12:            return ⊥
13:        if step is null (there are no more steps in prog_k) then
14:            v ← M.tryC(T_k)
15:            if v = A_k then return ⊥
16:            return context
```

modified t-objects in M.

We allow *context* to contain any data obtained from the execution of $prog_k$. In particular, it may reflect the whole interaction between G and M during the execution of transaction T_k (if the result of every transactional operation is stored in a separate local variable held within context). However, if for any reason T_k aborts, no results of partial execution of $prog_k$ (and thus T_k), are returned to p_i. This way the sequential specification of G is never compromised, in case T_k does not commit.

Note that by allowing a broad definition of $context$, we aimed at providing a definition of G, which is as general as possible. This approach allowed us to avoid the need for defining a programming language and embedding the whole reasoning in its framework as in [32].

Theorem 5. *Let M be a TM object and let G be a gateway shared object of M. If M is ⋄-opaque then G is ⋄-linearizable.*

Proof sketch. In order to show that G is ⋄-linearizable, we need to prove that any finite history produced by G is final-state ⋄-linearizable (by Corollary 2). Therefore, we assume some finite implementation history H, and show how to construct a sequential history S_G of events on G that is equivalent to $\bar{H}_G$, a completion of $H_G = H|G$. Next, we prove that S_G is legal. Finally, we show that S_G respects the ⋄-order of $\bar{H}_G$.

In order to obtain history S_G we first construct a completion $\bar{H}_G$ of H_G. Then we take S_M, a t-sequential t-history equivalent to a completion $\bar{H}_M$ of $H_M = H|M$ and rearrange the operation executions in $\bar{H}_G$ so that their order corresponds to the order of transactions in S_M. We say that S_G is *induced* by S_M.

To prove the legality of S_G we start by considering T_l, the last committed transaction in S_M (from assumptions we know that all transactions in S_M are legal, thus also the last one is legal). We use T_l to obtain a t-sequential t-history $vis_l = visible_{S_M}(T_l)$ which consists of all committed transactions in S_M. Then we show by a contradiction that vis_l is equal to $\alpha|M$, where $\alpha = \langle steps_H(o_1) \cdot steps_H(o_2) \cdot ...\rangle|M$ is a sequence of events such that $steps_H(o_k)$ returns all events

executed in (the implementation history) H as a part of a committed operation execution of operation $G.perform(prog_k)$.

Since for any t-object x, $vis_l|x$ satisfies the sequential specification of x, so does $\alpha|M$. Note that $\alpha|M$ is legal because any $prog_k$ executed through G operates in isolation and may interact with other processes only through t-objects (G allows only local computation besides operations on M). Now to prove that S_G is legal it suffices to show that inserting aborted operation executions to $\alpha|M$ does not break the legality of the history. This can be proved using the specification of G.

Finally, by construction of S_G it is easy to show that the matching order relations between transactions in S_M and operation executions in S_G are maintained. □

Theorem 5 allows us to reason about the properties of a TM system, when transactions are invisible to processes using the system. This result is particularly important when we consider a system built with several TM implementations, each one used by clients as a simple (non-transactional) shared object.

Note that an implication opposite to the one from Theorem 5 does not necessarily hold. The reason for this stems from the results presented in [105] and in [34]. In the former, the authors prove that when none of the intermediate results of a transaction execution are available after the transaction aborts, TMS1 is necessary and sufficient for *observational refinement* (TMS1 allows live and aborted transactions to observe a different view of past transactions, while committed transactions share a consistent view of the already committed ones, see also Section 2.2.1). It means that a programmer cannot distinguish between an execution of a transaction on a TM object that satisfies TMS1 and an execution of the same transaction on an abstract TM object that would execute all transactions sequentially. In the latter paper, the authors show that TMS1 is incomparable with opacity, i.e., TMS1 admits some executions that opacity does not and *vice versa*. We believe that a notion of a gateway shared object can be considered as an alternative to observational refinement for comparing guarantees offered by TM systems as seen by external clients.

Corollary 3. *Let M be a TM object and let G be a gateway shared object of M. If M is commit-real-time opaque, then G is real-time linearizable.*

Proof. The proof follows directly from Theorems 4 and 5. □

This result shows that a TM system which allows transactions to read stale data is still real-time linearizable if it eventually aborts every such transaction.

5
Basic Replication Schemes

In this chapter we discuss two basic approaches to service (or data) replication: *active* and *passive replication*. In active replication, which we demonstrate using the *State Machine Replication* scheme based on TOB, a request is executed by each replica independently. On the other hand, in passive replication, demonstrated using the *Deferred Update Replication* scheme, which is also based on TOB, a request is executed only by a single replica and then the updates produced during request execution are propagated to other service processes, so they can update their state accordingly.

SMR and DUR represent two widely different ways of replicating a service (or data) and they function as a base for more complex replication schemes, as discussed in Section 2.1. Both schemes have their advantages and disadvantages, which we explore in this chapter. Also, they offer different guarantees, as we formally prove.

We present SMR in two variants: basic and optimized called *SMR with Locks (LSMR)*. Compared to SMR, LSMR allows read-only requests to be executed concurrently and without inter-process synchronization. Although this optimization seems relatively straightforward, it is rarely found in real-life implementations. We discuss it to show how allowing read-only requests to be processed without inter-process synchronization impacts the guarantees offered by LSMR compared to SMR.

We follow the description of algorithms from [1]. Our proofs of correctness for SMR, LSMR and DUR, which are given in this chapter, can be found in [106] and [3]. The comparison of the schemes summarizes the work presented in [5] and [2]. The presented results of the experimental evaluation have been featured in the latter paper.

5.1 State Machine Replication

In this section we discuss the SMR scheme in its basic version. We also formally prove the guarantees provided by SMR.

5.1.1 Specification

In SMR [16] [17] [18] a service is fully replicated by every process. All replicas start from the same initial state and advance by processing all requests in the same order. Naturally, execution of each request has to be deterministic. Otherwise, the consistency among replicas could not be preserved as the replicas might advance differently. The crucial element of this scheme is the protocol used for dissemination of requests among replicas. The required semantics is provided by the TOB protocol [49] (see Section 3.2).

In our pseudocode, which is given in Algorithm 2, we assume that each request is handled by a separate thread. Each client request consists of three elements: id, a unique identification number, $prog$, which specifies the operations to be executed and $args$, which holds the arguments needed for the program execution. Prior to execution, a request is broadcast to all replicas using TO-BROADCAST (line 3). Only then each replica executes the request independently (line 6). After the request is executed, the thread that originally received the request returns the response to the client (line 4).

Some additional data structures holding a history of clients' requests have to be maintained to provide fault-tolerance in case of the loss of request/response messages. For brevity we omit them in the pseudocode.

5.1.2 Characteristics

SMR does not offer transactional semantics since it does not support constructs such as *rollback* and *retry*, which can be used to control the execution flow. However, the execution of a request by SMR can be regarded as a rudimentary transaction that is allowed to only commit.

Algorithm 2 State Machine Replication for process p_i

Thread q **on request** r **from client** c (executed on one replica)
1: response $res_q \leftarrow \perp$
2: **upon** INIT
3: TO-BROADCAST r *// blocking*
4: return $(r.id, res_q)$ to client c

The main thread of SMR (executed on all replicas)
5: **upon** TO-DELIVER (request r)
6: response $res \leftarrow$ execute $r.prog$ with $r.args$
7: **if** request with $r.id$ handled locally by thread q **then**
8: $res_q \leftarrow res$

Naturally, the program specified by a request needs to be deterministic. Otherwise replicas would diverge. This requirement can often be a very limiting one because even simple operations, such as generating a log entry with a timestamp, must be handled carefully.

Although SMR does not allow for any degree of parallelism in request execution, it performs surprisingly well (see the performance analysis of SMR in Section 5.4). It is because execution of a request happens directly on objects in memory (unlike in DUR, in which access to shared objects happens through the complicated transactional machinery that adds a substantial overhead to a request execution, see Section 5.3).

5.1.3 Correctness

In order to reason about correctness of SMR, typically it is modelled as a shared object which exports a set of operations and has a well defined *sequential specification*. Many authors state that SMR satisfies linearizability (see, e.g., [17], [107]). However, none of the papers we are aware of feature a formal proof of SMR's correctness. Therefore, we present a formal result on SMR's correctness and show that it satisfies real-time linearizability. The full proof of the theorem can be found in Appendix B.1.

Theorem 6. *State Machine Replication satisfies real-time linearizability.*

Proof sketch. In order to prove that SMR satisfies real-time linearizability, we have to show that for every finite history H produced by SMR, there exists a sequential legal history S equivalent to some completion $\bar{H}$ of H, such that S respects the real-time order of $\bar{H}$.

Constructing the sequential history S is easy, because we know that replicas execute all requests sequentially and in the order consistent with the order of message delivery established by TOB. Naturally then S also respects the real-time order of $\bar{H}$ and is trivially legal. □

5.2 State Machine Replication with Locks

In SMR in its basic version, every request is executed by all processes. Doing so may seem wasteful, especially for requests which do not change the state of the system, i.e., for requests that are *a priori* known to be read-only. Therefore, now we show SMR with Locks (LSMR), which allows read-only requests to be executed only by one replica.

5.2.1 Specification

In LSMR, whose pseudocode is given in Algorithm 3, every request is marked either as a read-only or as a potentially updating. In the first case, a request can

Algorithm 3 State Machine Replication with Locks for process p_i

```
1: integer LC ← 0

Thread q on request r from client c (executed on one replica)
2: response res_q ← ⊥
3: upon INIT
4:     if r.prog is read-only then
5:         wait until LC ≥ r.clock
6:         r-lock { res_q ← execute r.prog with r.args }        // readers-writer lock, read-mode
7:     else
8:         TO-BROADCAST r                                       // blocking
9:     return (r.id, LC, res_q) to client c

The main thread of LSMR (executed on all replicas)
10: upon TO-DELIVER (request r)
11:     w-lock { LC ← LC + 1                                     // readers-writer lock, write-mode
12:             response res ← execute r.prog with r.args }
13:     if request with r.id handled locally by thread q then
14:         res_q ← res
```

be executed by only one replica, as doing so will not result in any changes to the local or replicated state. On the other hand, potentially updating requests are executed as in SMR, i.e., first they are broadcast using TOB to all replicas and then executed by all replicas independently.

Note that allowing a read-only request issued by a client to be executed by only one replica without inter-process synchronization may result in a situation where the client does not observe the changes performed by the previous requests he issued. Such a situation may occur when a client first issues a potentially updating request r_1 to some replica p_i, receives a response and subsequently issues a read-only request r_2 to a replica $p_j \neq p_i$, which slightly lags behind others (p_j misses some updates compared to the up-to-date replicas, e.g., due to some messages being still in transit). In particular, p_j might have not yet delivered (through TOB) the updating request r_1, which has been already processed by p_i, and whose corresponding response message has been returned to the client. As a result, the execution of r_2 on p_j does not observe the changes performed by r_1, i.e., a request issued previously by the same client. To mitigate such situations, we introduce a simple mechanism based on logical clocks. To this end, LSMR maintains a special global variable LC initialized to 0 (line 1). LC represents the logical clock, which is incremented (line 11) every time an updating request is executed (line 12). LC allows the process to track whether its state is recent enough to execute the client's request. It is done by comparing the value of LC with the value of the $clock$ field of the client request (line 5). The check is not necessary in case of an updating request because disseminating it through TOB guarantees that it always executes on the most recent state.[1] After a request is executed, the current value of LC is returned to the client in the response message (line 9). This way, the client may use this value in the subsequent request he issues.

Note that execution of read-only and updating requests is guarded by a lock

[1] For this very reason, LC does not need to be maintained in SMR.

(lines 6 and 11–12). This way we guarantee, that the state of the system does not change during execution of a read-only request (execution of potentially updating requests is serialized, because requests delivered using TOB are processed as non-preemptable events). Since read-only requests do not change the state of the service, multiple read-only requests can be allowed to execute concurrently. To this end LSMR uses a readers-writer lock (in the pseudocode *r-lock* represents the lock in the readers mode while *w-lock* represents the lock in the writer mode, lines 6 and 11–12).

5.2.2 Characteristics

Naturally, optimizing the execution of read-only requests may greatly improve the performance of SMR, especially when such requests constitute the vast majority of requests processed by a service (which represents a typical case, see, e.g., [27]). Note that the information about the type of request (read-only or updating) is not always easily available prior to request execution. It is one of the reasons why LSMR implementations are not common in practice [108]. Also, LSMR is significantly more complex than SMR and still offers only limited scalability (all updating requests need to be executed sequentially by all replicas as in SMR). Also, as we show below, allowing read-only requests to be executed without any inter-process synchronization weakens the provided guarantees.

5.2.3 Correctness

We can show that LSMR does not satisfy real-time linearizability using a similar example as in Section 5.2.1, but featuring two (independent) clients issuing requests r_1 and r_2. The full proofs of the following theorems are available in Appendix B.2.

Theorem 7. *State Machine Replication with Locks does not satisfy real-time linearizability.*

Proof sketch. We show that LSMR does not satisfy real-time linearizability by giving a counter example involving three processes and two different requests. An updating request r_1 is broadcast using TOB to all three replicas but is first delivered and executed only by processes p_1 and p_2 (typically TOB requires that a majority of processes needs to receive a message before it can be delivered, see Section 3.3). Before r_1 reaches p_3 but after both p_1 and p_2 finish execution of r_1, p_3 executes locally a read-only request r_2 (issued by a client who did not issue r_1). Naturally r_2 needs to be serialized before r_1 for the serialization to be legal. However, such serialization does not respect the real-time order of the execution and thus LSMR does not satisfy real-time linearizability. □

Corollary 4. *State Machine Replication with Locks does not satisfy commit-time linearizability.*

Proof. The proof follows directly from Theorem 7 and Theorem 4 (commit-real-time linearizability is equivalent to real-time linearizability). □

As we now show below, LSMR satisfies write-real-time linearizability, which is slightly weaker than real-time linearizability.

Theorem 8. *State Machine Replication with Locks satisfies write-real-time linearizability.*

Proof sketch. In order to prove that LSMR satisfies write-real-time linearizability, we have to show that for every finite history H produced by LSMR, there exists a sequential legal history S equivalent to some completion $\bar{H}$ of H, such that S respects the write-real-time order of $\bar{H}$.

The crucial part of the proof concerns a proper construction of S. We do so in the following way. Let us start with an empty sequential history S'. Now we add to S' all updating operation executions from $\bar{H}$. However, we do so according to the order of the (unique) numbers associated with each updating operation execution. The number associated with each updating operation execution o_k of some request r_k is the value of the LC variable from the time when the operation execution is being processed by the replica (i.e., during execution of r_k). We show that this number is the same for each replica and that it unambiguously identifies the operation execution (thanks to the use of TOB as the sole mechanism used to disseminate messages among replicas and the fact that LC is incremented prior to execution of every potentially updating request). Then, one by one, we add to S' all read-only operation executions from $\bar{H}$. For every such operation execution o_k of some (read-only) request r_k, we insert it in S' immediately after a potentially updating operation execution o_l of some request r_l, such that the value set to LC (on the replica that executes r_k) just prior to execution of r_l is equal to the value of LC during execution of r_k. It means that r_l is the last potentially updating request processed by the replica prior to execution of r_k (note that r_l might not exist if r_k is executed on the initial state of the replica). Now we consider independently each (continuous) subhistory of S' which consists only of read-only operation executions (i.e., roughly speaking, we consider the periods of time in between execution of updating requests). For each such subhistory, when necessary, we rearrange the read-only operation executions so that their order respects the order in which these operation executions appear in subhistory $\bar{H}|p_i$, for every process p_i. Then $S = S'$.

By construction of S, it is easy to show that S respects the write-real-time order of H. Then we show how to construct the witness history W for S. Doing so is straightforward because replicas execute all potentially updating requests sequentially and in the order of message deliveries. □

The consequences of having weaker guarantees in LSMR compared to SMR can be illustrated using a similar scenario to the one described in Section 4.1. When clients can communicate with each other through channels outside the replicated service, the following situation can occur: a client c_1 connected to a lagging replica can receive from it a response, which is not consistent with the

more up-to-date information c_1 received from some other client c_2 (c_2 is connected to a different, more up-to-date replica). This problem can be mitigated by having the clients exchange the values of the logical clock when communicating outside of LSMR. More precisely, when client c_1 with the logical clock value v_1 receives a message from client c_2 outside of LSMR and the message carries v_2, i.e., the current value of the logical clock of c_2, then c_1 changes v_1 to $max(v_1, v_2)$ and uses the new value in the subsequent request he issues to LSMR. However, in principle using this technique can be difficult, because clients can communicate indirectly, through several other services.

5.3 Deferred Update Replication

In this section, we focus on the DUR protocol. We follow the description of DUR in its simplest form, as in [1] (see also [7], [3] and [4]). We then formally prove DUR's guarantees. Finally, we discuss an important optimization of DUR, which can greatly improve its performance.

5.3.1 Specification

Unlike SMR and LSMR, DUR [15] is a *multi-primary-backup* approach, in which any request (so also an updating request) is executed only by a single replica and all replicas can execute different requests concurrently (also in separate threads on multiple processor cores). After a request is executed, the replica that executed it issues the resulting modifications to other replicas so they can update their state accordingly. In this case, we have to resolve any conflicts between concurrently processed requests that access the same data items in a non-trivial manner (at least one of the requests modifies the data item). This is where transactions come into play. Then, each request is executed as an atomic transaction which, from the client's point of view, runs sequentially with respect to any other concurrent transactions in the system.

We focus on DUR in its basic version, in which all data items (shared objects) managed by the replicated service are fully replicated on each replica. During the execution phase, a transaction (spawned to execute a client request) operates in isolation on local copies of shared objects. The transaction's execution phase is followed by the committing phase where the processes synchronize and certify transactions. Transaction certification means checking if a (committing) transaction does not conflict with any other concurrent transactions (i.e., the transaction had not read any shared objects modified by a concurrent but already committed transaction). It is the only moment in a transaction's lifetime that requires replica synchronization. Upon successful certification, replicas update their state. Otherwise the transaction is rolled back and restarted. As mentioned in Section 2.1, DUR can use various protocols for transaction certification, but here we discuss

DUR relying on TOB, which avoids blocking and limits the number of costly network communication steps [49].

In order to ensure that a live transaction always executes on a consistent state, we perform partial transaction certifications during transaction execution. However, these procedures are done only locally and do not involve any inter-replica synchronization. Also note that for simplicity, we assume that each shared object (replicated on every replica) can only be read or written to.

Consider the pseudocode for DUR which is given in Algorithm 4. Clients interact with the replicated service running on DUR in a somewhat similar way as in SMR and LSMR. However, now we require that $prog$ (submitted as part of the request) at some point calls the COMMIT procedure thus indicating that transaction execution is finished. As in case of LSMR, we require that there is an additional element passed along with every client request (the $clock$ variable, line 20) and every client response (the current value of the LC variable, line 22, see below).

Each DUR process maintains two global variables. The first one, LC, represents the logical clock which is incremented every time a process applies updates of a transaction (line 52). As in LSMR, LC allows the process to track whether its state is recent enough to execute the client's request (line 20). Additionally, LC is used to mark the start and the end of the transaction execution (lines 25 and 53). The transaction's start and end timestamps, stored in the *transaction descriptor* (line 18), allow us to reason about the precedence order between transactions. Let H be some execution of DUR, T_i and T_j some transactions in H and t_i and t_j be their transaction descriptors, respectively. Then, $t_i.end \leq t_j.start$ holds true only if $T_i \prec^r_H T_j$; otherwise T_i and T_j are concurrent.[2] The second variable, Log, is a set used to store the transaction descriptors of committed transactions. Maintaining this set is necessary to perform transaction certification.

DUR detects conflicts among transactions by checking whether a transaction T_k that is being certified read any stale data (shared objects modified by a concurrent but already committed transaction).[3] To this end, DUR traces accesses to shared objects independently for each transaction. The identifiers of objects that were read and the modified objects themselves are stored in private, per transaction, memory spaces. On every read, an object's identifier is added to the $readset$ (line 30). Similarly, on every write a pair of the object's identifier and the corresponding object is recorded in the $updates$ set (line 36). Then, the CERTIFY function compares the given $readset$ against the $updates$ sets of all committed transactions in Log which are concurrent with T_k. If it finds any non-empty intersection of the sets, T_k is aborted and forced to retry; otherwise, it can proceed. Note that a check against conflicts is performed upon every read operation (line 31). This way T_k is guaranteed to always read from a consistent snapshot.

When a transaction's code calls COMMIT (line 37), the committing phase is

[2]Moreover, if both T_i and T_j are committed updating transactions and $t_i.end > t_j.start$, then T_i and T_j must not be in conflict (as otherwise T_j would be aborted).

[3]Note that two concurrent transactions which modify the same data item are not in conflict, unless the transaction that tries to commit later read any data modified by the transaction that commits first.

initiated. If T_k is a read-only transaction (T_k did not modify any objects), it can commit straight away without performing any further conflict checks or process synchronization (line 39). All possible conflicts would have been detected earlier, upon read operations (line 31). If T_k is an updating transaction, it is first certified locally (line 42). This step is not mandatory, but allows the process to detect conflicts earlier, and thus sometimes avoids costly network communication. Next, the transaction's descriptor containing $readset$ and $updates$ is broadcast to all processes through TOB using the TO-BROADCAST primitive (line 44). The message is delivered in the main thread, where the final certification takes place (line 51). This procedure ultimately determines the fate of the transaction, i.e., whether to commit or to abort it. Upon successful certification, the processes apply T_k's updates and commit T_k (lines 52–55). Otherwise, T_k is rolled back and reexecuted by the same process.

To manage the control flow of a transaction, the programmer can use two additional procedures: ROLLBACK and RETRY. ROLLBACK (line 47) stops the execution of a transaction and revokes all the changes it performed so far. RETRY (line 45) forces a transaction to rollback and restart (the $outcome_q$ variable is initialized with $failure$, thus the condition in line 27 is evaluated to $true$).

For clarity, we make several simplifications. Firstly, we use a single global (reentrant) lock to synchronize operations on LC (lines 25, 52, 53), Log (lines 10 and 54) and the accesses to transactional objects (lines 34 and 55). Secondly, we allow Log to grow indefinitely. Log can easily be kept small by garbage collecting information about the already committed transactions that ended before the oldest live transaction started its execution in the system. Thirdly, we use the same certification procedure for both the certification test performed upon every read operation (line 31) and the certification test that happens after a transaction descriptor is delivered to the main thread (line 51). In practice, doing so would be very inefficient, because upon every read operation we check for the conflicts against all the concurrent transactions (line 10), thus performing much of the same work again and again. However, these repeated actions can be easily avoided by associating the accessed shared objects with a version number equal to the value of LC at the time the objects were most recently modified.

5.3.2 Characteristics

In DUR requests are executed as independent transactions which operate in isolation on a consistent snapshot. A programmer can use the *rollback* and *retry* constructs to better manage the control flow of the transaction's execution. These constructs can be used to suspend execution of a transaction until a certain condition is met.

Note that a DUR transaction cannot execute *irrevocable operations*, i.e., operations whose side effects cannot be rolled back (such as local system calls). It is because, a transaction in DUR can be aborted at any point during execution or upon certification and then restarted.

A transaction may be aborted multiple times before it eventually commits.

Algorithm 4 Deferred Update Replication for process p_i

```
1: integer LC ← 0
2: set Log ← ∅
3: function GETOBJECT(txDescriptor t, objectId oid)
4:     if (oid, obj) ∈ t.updates then
5:         value ← obj
6:     else
7:         value ← retrieve object oid
8:     return value
9: function CERTIFY(integer start, set readset)
10:    lock { L ← {t ∈ Log : t.end > start} }
11:    for all t ∈ L do
12:        writeset ← {oid : ∃(oid, obj) ∈ t.updates}
13:        if readset ∩ writeset ≠ ∅ then
14:            return failure
15:    return success
```

Thread q **on request** r **from client** c (executed on one replica)

```
16: enum outcome_q ← failure                    // type: enum { success, failure }
17: response res_q ← null
18: txDescriptor t ← null                       // type: record (id, start, end, readset, updates)
19: upon INIT
20:     wait until LC ≥ r.clock
21:     TRANSACTION()
22:     return (r.id, LC, res_q) to client c
23: procedure TRANSACTION
24:     t ← (a new unique id, 0, 0, ∅, ∅)
25:     lock { t.start ← LC }
26:     res_q ← execute r.prog with r.args
27:     if outcome_q = failure then
28:         TRANSACTION()
29: function READ(objectId oid)
30:     t.readset ← t.readset ∪ {oid}
31:     lock { if CERTIFY(t.start, {oid}) = failure then
32:                 RETRY()
33:             else
34:                 return GETOBJECT(t, oid) }
35: procedure WRITE(objectId oid, object obj)
36:     t.updates ← {(oid', obj') ∈ t.updates : oid' ≠ oid} ∪ {(oid, obj)}
37: procedure COMMIT
38:     stop executing r.prog
39:     if t.updates = ∅ then
40:         outcome_q = success
41:         return
42:     if CERTIFY(t.start, t.readset) = failure then
43:         return
44:     TO-BROADCAST t                              // blocking
45: procedure RETRY
46:     stop executing r.prog
47: procedure ROLLBACK
48:     stop executing r.prog
49:     outcome_q ← success
```

The main thread of DUR (executed on all replicas)

```
50: upon TO-DELIVER (txDescriptor t)
51:     if CERTIFY(t.start, t.readset) = success then
52:         lock { LC ← LC + 1
53:                t.end ← LC
54:                Log ← Log ∪ {t}
55:                apply t.updates }
56:         if transaction with t.id executed locally by thread q then
57:             outcome_q ← success
```

The number of aborts depends on the level of *contention*, i.e., the number of other, concurrent updating transactions which access the same data. When contention is low, transactions sporadically need to be reexecuted. Then the overall cost of transaction reexecution is much lower than the cost of scheduling transactions so they never run into conflicts, as in a pessimistic (lock-based) approach to transaction execution. On the other hand, when contention is high, a lot of resources such as CPU or network can be wasted. Note that in an extreme case, one transaction's commit causes aborts of all concurrently executing transactions thus forcing them to restart. Then, execution of all transaction's is effectively sequential (for detailed analysis see [2]).

Although DUR executes requests concurrently, transaction certification and the subsequent process of applying updates (in case of successful certification) happens sequentially. Therefore the scalability of DUR is limited by the capability of the main threads of replicas to process updates. However, our experience shows that often there are other factors that limit the performance (and scalability) of DUR, e.g., insufficient network bandwidth (see Section 5.4).

5.3.3 Correctness

Now we formally prove DUR's guarantees. We use DUR as specified in Algorithm 4, but we consider only t-histories of DUR, i.e., histories limited to events that are related to operations on t-objects and controlling the flow of transactions such as *commit* and *abort* events. In this sense, we treat the implementation of DUR as some TM object M, and reason about t-histories $H|M$, in a similar way to which we do it in Section 4.5. As we argue in Section 2.1, strong similarities in the way transactions are processed can be found between DUR and more complex replication schemes such as Postgres-R [39], E-DUR [10] and PolyCert [40]. Therefore, the results presented below are also applicable for these systems. The full proofs of the following theorems are available in Appendix B.3.

Theorem 9. *Deferred Update Replication does not satisfy write-real-time opacity.*

Proof sketch. The proof follows a counter example similar to the one presented in Section 4.1. □

Corollary 5. *Deferred Update Replication does not satisfy real-time opacity.*

Proof. The proof follows directly from Theorem 9 and definitions of write-real-time opacity and real-time opacity (real-time opacity is strictly stronger than write-real-time opacity). □

Theorem 10. *Deferred Update Replication satisfies update-real-time opacity.*

Proof sketch. In order to prove that DUR satisfies update-real-time opacity, we have to show that (by Corollary 1) for every finite t-history H produced by DUR, there exists a t-sequential t-history S equivalent to $\bar{H}$ (some completion of H),

such that S respects the update-real-time order of H and every transaction T_k in S is legal in S.

The crucial part of the proof concerns a proper construction of S. We do so in the following way. Let us start with an empty t-sequential t-history S'. Now we add to S' t-histories $\bar{H}|T_k$ of all committed updating transactions T_k from $\bar{H}$. However, we do so according to the order of the (unique) numbers associated with each such transaction T_k (with transaction descriptor t_k). The number associated with T_k is the value of $t_k.end$. This value is equal to the value of the LC variable of the replica that processes the updates of T_k upon committing T_k. We show that this number is the same for each replica and that it unambiguously identifies the operation execution (thanks to the use of TOB as the sole mechanism used to disseminate messages among replicas and the fact that LC is incremented upon commit of every updating transaction). Then, one transaction at a time, we add to S' all operations of all aborted and read-only transactions from $\bar{H}$. For every such transaction T_k (with transaction descriptor t_k), we insert $\bar{H}|T_k$ in S' immediately after operations of a committed updating transaction T_l (with transaction descriptor t_l), such that $t_k.start = t_l.end$. It means that T_l is the last committed updating transaction processed by the replica prior to execution of T_k (note that T_l might not exist if T_k is executed on the initial state of the replica). Now we consider independently each (continuous) sub-t-history of S' which consists only of operations of read-only and aborted transactions (i.e., roughly speaking, we consider the periods of time in between commits of updating transactions). For each such sub-t-history, when necessary, we rearrange the operations of the read-only and aborted transactions so that their order respects the order in which these operation executions appear in sub-t-history $\bar{H}|p_i$, for every process p_i. Because we always consider all operations of every transaction together, S' is t-sequential. Then $S = S'$.

By construction of S, it is easy to show that S respects the update-real-time order of H. Then we show by a contradiction that there does not exist a transaction in S that is not t-legal, hence every transaction in S is legal. □

Now we can easily show how DUR can be used to build an update-real-time linearizable system.

Corollary 6. *Let G be a gateway shared object implemented using Deferred Update Replication. Then, G satisfies update-real-time linearizability.*

Proof. The proof follows directly from Theorem 5 and Theorem 10. □

5.3.4 The Multiversioning Optimization

Multiversioning [109] in an important optimization technique which allows a system that implements this optimization to store multiple versions of transactional objects in a way that is transparent to the programmer. Object versions are immutable, thus they can be accessed concurrently without any synchronization. A transaction uses only one version of any transactional object. The object

version is always chosen so that effectively a transaction operates on a consistent snapshot of the local state. As a result, read-only transactions never conflict (unlike concurrent updating transactions that access the same transactional objects). Moreover, because read-only transactions are guaranteed to commit, $readset$ does not have to be maintained for the *declared read-only* transactions, i.e., transactions which are known *a priori* not to perform any updating operations. Therefore the multiversioning optimization can greatly improve the overall performance and scalability of the transactional system when workloads are dominated by read-only transactions [27].

In Algorithm 5, we show the pseudocode for *MvDUR*, i.e., DUR with the multiversioning optimization. Compared to DUR, MvDUR no longer stores information about committed transactions in Log. Instead, MvDUR maintains multiple *object versions* obj for each object identifier oid. Each object version is paired with a corresponding *version number* ver. When a transaction commits, the system atomically creates new versions of all objects modified by the transaction (lines 54–55), all having the same version number assigned, which is equal to the current value of the logical clock LC.

Compared to DUR, there is also a new function GETVERSION, which takes two arguments oid and $notNewerThan$ and retrieves a version obj of an object identified with oid that is the most recent among all those object versions that have a version number lower than or equal $notNewerThan$ (lines 2–4). It can be used to read from a consistent snapshot of the system and return the most recent object version that existed in the system up to a given moment in time. This way all reads which are performed by a transaction are consistent and no conflict checks are necessary. Hence, as mentioned before, read-only transactions are guaranteed to always commit and declared read-only transactions do not need to record their accesses in the $readset$ (line 33).

The transaction certification phase in MvDUR is different and much more efficient than in DUR. Instead of checking a transaction's $readset$ against the $updates$ sets of (possibly many) committed concurrent transactions, for each object read by the committing transaction the certification procedure simply compares the committing transaction's $start$ timestamp with the version numbers of objects stored by the replica. If the most recent version of a read object has a version number which is greater than the transaction's $start$ timestamp, then a conflict exists–i.e., a new version was created after the transaction had already started execution. Note that the version number of an object in MvDUR directly corresponds to the value of the end field of a transaction descriptor of an already committed updating transaction in DUR.

The committing phase in MvDUR is similar to DUR's one. Both algorithms differ in the way each replica applies transaction updates. In MvDUR, replicas update their state by adding new object versions (lines 54–55). For this, we have to use locks since these operations must be done atomically. However, in practice MvDUR can be implemented in a way that avoids using locks altogether (by, e.g., exploiting the visibility rules of the memory model).

In our pseudocode, no object version is ever removed from the system. How-

Algorithm 5 Deferred Update Replication with Multiversioning for process p_i

```
1: integer LC ← 0
2: function GETVERSION(objectId oid, integer notNewerThan)
3:     lock { return (obj, ver) such that obj is a version of object oid whose version number ver
4:                                 is the highest available such that ver ≤ notNewerThan }
5: function GETOBJECT(txDescriptor t, objectId oid)
6:     if (oid, obj) ∈ t.updates then
7:         value ← obj
8:     else
9:         (obj, ver) ← GETVERSION(oid, t.start)
10:        value ← obj
11:    return value
12: function CERTIFY(integer start, set readset)
13:    for all oid ∈ readset do
14:        (obj, ver) ← GETVERSION(oid, ∞)
15:        if ver > start then
16:            return failure
17:    return success
```

Thread q **on request** r **from client** c (executed on one replica)

```
18: enum outcome_q ← failure                          // type: enum { success, failure }
19: response res_q ← null
20: txDescriptor t ← null                             // type: record (id, start, end, readset, updates)
21: upon INIT
22:     wait until LC ≥ r.clock
23:     TRANSACTION()
24:     return (r.id, LC, res_q) to client c
25: procedure TRANSACTION
26:     t ← (a new unique id, 0, 0, ∅, ∅)
27:     lock { t.start ← LC }
28:     res_q ← execute r.prog with r.args
29:     if outcome_q = failure then
30:         TRANSACTION()
31: function READ(objectId oid)
32:     obj ← GETOBJECT(t, oid)
33:     if r.prog is not declared-read-only then
34:         t.readset ← t.readset ∪ {oid}
35:     return obj
36: procedure WRITE(objectId oid, object obj)
37:     t.updates ← {(oid', obj') ∈ t.updates : oid' ≠ oid} ∪ {(oid, obj)}
38: procedure COMMIT
39:     stop executing r.prog
40:     if t.updates = ∅ then
41:         outcome_q = success
42:         return
43:     if CERTIFY(t.start, t.readset) = failure then
44:         return
45:     TO-BROADCAST t                                                    // blocking
46: procedure RETRY
47:     stop executing r.prog
48: procedure ROLLBACK
49:     stop executing r.prog
50:     outcome_q ← success
```

The main thread of DUR (executed on all replicas)

```
51: upon TO-DELIVER (txDescriptor t)
52:     if CERTIFY(t.start, t.readset) = success then
53:         lock { LC ← LC + 1
54:                for all (oid, obj) ∈ t.updates
55:                    add obj as a new version of object oid with version number LC }
56:         if transaction with t.id executed locally by thread q then
57:             outcome_q ← success
```

ever, a simple garbage collection mechanism can be proposed, as follows. Let us consider a replica r, and let t be the transaction descriptor of the oldest live transaction in r ($t.start$ is equal to the the lowest value among all descriptors of live transactions in r). Let d be the set of all object versions in r whose version numbers are lower than or equal $t.start$. Then, for each shared object identified with oid, all its versions in d excluding the most recent version of oid, can be safely dropped.

Note that some conflicts between updating transactions are detected earlier in DUR than in MvDUR. Imagine a scenario in which a replica receives a message regarding a transaction T_i and subsequently commits it. After that, a live transaction T_j executed by the replica attempts to read an object modified by T_i. In DUR, T_j would be immediately aborted because the certification function (invoked in line 31) would return $failure$. On the other hand, in MvDUR T_j would be free to execute without interruptions until it invokes the COMMIT, RETRY, or ROLLBACK procedure. However, by using objects whose version numbers are strictly lower than the numbers assigned to objects updated by T_i, T_j always executes on a consistent snapshot. Naturally, one could implement a simple early conflict detection mechanism, which aborts as soon as possible all updating transactions deemed to fail the certification test once these transaction attempt to commit (line 43).

It is easy to see why MvDUR satisfies the same correctness guarantees as DUR, once we understand that the conflict condition checked during transaction certification is the same as in DUR, although formulated slightly differently, i.e., using version numbers. It means that given the same set of the already committed transactions and a committing transaction T, DUR and MvDUR decide in the same way whether to commit or abort T. Then, the only difference between DUR and MvDUR lies in that MvDUR commits read-only transactions which would be aborted in DUR. However, in terms of the provided guarantees, update-real-time opacity, i.e., the property satisfied by DUR, equally treats (committed) read-only and aborted (read-only or updating) transactions. Hence, MvDUR also satisfies update-real-time opacity.

5.4 Evaluation

In this section, we present the results of experimental evaluation of SMR and DUR under different workload types and varying contention levels, obtained using popular microbenchmarks: *Hashtable* and *Bank*. For each benchmark, we developed a non-replicated service (*SeqHashtable* and *SeqBank*) executing requests sequentially on one machine, and a replicated, fault-tolerant counterpart, where the program code and data structures (hashtable and bank accounts) were fully replicated on all nodes.

5.4.1 Software and Environment

For tests of SMR and DUR we used JPaxos and Paxos STM (see Section 2.3 for relevant references). JPaxos implements the basic SMR (and not LSMR) scheme. It means that JPaxos treats read-only and read-write requests in the same way and every request is first broadcast and then executed independently by every replica. Paxos STM, our optimistic distributed transactional memory system, the DUR scheme and thus can execute requests (transactions) in parallel on many machines and threads running on multicore processors. Paxos STM employs the multiversioning optimization so it essentially implements the optimized version of DUR, which we called MvDUR in the previous section (see Section 5.3.4). Additionally, for better performance, Paxos STM implements early conflict detection and avoids using locks altogether. Both systems use an implementation of TOB based on Paxos [71]. The TOB protocol is optimized using batching and pipelining, which bring performance benefits (see e.g., [110]). *Batching* means broadcasting a batch of messages (if available) by only one protocol instance. *Pipelining* allows the protocol leader to initiate several instances of the protocol in parallel (as in [71]).

Both JPaxos and Paxos STM allow replicas to crash and later recover and seamlessly rejoin. Notably, nonvolatile storage is scarcely used during regular (non-faulty) system operation. During recovery, a recovering replica can obtain the current state from other live replicas (if at least a majority of replicas is operational all the time). In this evaluation, we compared the performance of the two systems during regular (non-faulty) execution. But in all our experiments, we ran both systems with the recovery protocol enabled, so they were fully fault-tolerant.

We ran tests in a cluster of 20 nodes connected via 1Gb Ethernet network. Each node had 28-core Intel E5-2697 v3 2.60GHz processor, 64GB RAM, and used Scientific Linux CERN 6.7 with Java HotSpot 1.8.0.

The TOB protocol used by JPaxos and Paxos STM was configured to have at most two concurrent instances of consensus and the batch capacity 64KB. We experimentally established an optimal number of threads in Paxos STM to be 160 for Hashtable and 280 for Bank (these values were used in all our tests). A high number of threads (far exceeding the number of physical cores) was necessary to fully exercise the hardware due to threads blocking on I/O (network) operations. To reduce the overhead caused by client-server communication, the clients ran on replicas. The number of clients in JPaxos and Paxos STM was constant per replica, and equal the number of threads in Paxos STM.

5.4.2 Benchmarks

We tested in total 12 different workloads using two microbenchmarks, *Hashmap* and *Bank*.

The Hashtable benchmark features a hashtable of size $h = 10000$, storing pairs of key and value, and accessed using the *get*, *put*, and *remove* operations. A run of this benchmark consists of a load of requests which are issued to the

Request type →	RO requests	RW requests
a) Hashtable Default	100 get	8 get + 2 put/remove
b) Hashtable Prolonged	100 get + 1 ms	8 get + 2 put/remove + 1 ms
c) Hashtable High-Contention	100 get	40 get + 10 put/remove
d) Bank	250k read	2 read + 2 write

Figure 5.1: Number of operations per benchmark configuration.

hashtable. We consider two types of requests (transactions): a *read-only (RO)* request executes a series of *get* operations on a randomly chosen set of keys whereas a *read-write (RW)* request executes a series of *get* operations, followed by a series of update operations (either *put* or *remove*). The Hashtable is pre-populated with $\frac{h}{2}$ random integer values from a defined range, thus giving the saturation of 50%. The saturation level is preserved all the time: if a randomly chosen key points at an empty element, a new value is inserted using *put*; otherwise, the element is removed using *remove*.

Execution times of hashtable operations are negligible compared to network latency and other factors that overshadow the computational cost of operations executed within a request. In order to simulate computation-heavy workloads (requests performing I/O operations, extensive data processing, cryptographic computation, etc.) request execution time can be prolonged by the *active wait* performed after finishing all operations specified by the request type. In order to simulate optimal utilization of CPU, the active wait can be performed only by a given number of concurrent threads (e.g., equal the number of processor cores).

We used three Hashtable configurations: Default, Prolonged, and High-Contention, which represent various *workload types*, modelled by varying the number and length of operations in RO and RW transactions (see Figure 5.1). In all cases, each RO request scans a vast amount of data using 100 *get* operations. In contrary, RW requests have fewer operations (10 in Default and Prolonged and 50 in High-Contention), where 20% are update operations. In Prolonged Hashtable, each RO and RW transaction is prolonged by 1 ms, which simulates a computation-heavy workload.

The Bank benchmark features a replicated array of 250k bank accounts. A run of this benchmark consists of a load of RW and RO requests accessing the accounts. A RW request transfers money between two accounts, by executing two *get* and two *put* operations on the replicated array. A RO request computes a balance, by reading all accounts and summing up the funds.

For each benchmark, we examined three *test scenarios*, obtained by the following mix of RW and RO requests in the test load: 10%, 50%, and 90% of RW requests. RO requests were known *a priori*. In Paxos STM, different test scenarios allow us to simulate variable *contention* in the access to data shared by concurrent transactions.

For all benchmarks, we present *throughput*–the number of completed requests (committed transactions) per second. In Paxos STM-based service, concurrent transactions may conflict and abort, so we also present the *abort rate*, the percent-

age of transactions aborted due to conflicts out of a total number of transaction executions. The abort rate gives a useful insight into the level of contention. We also measured data transfer in Mb/s to witness network congestion, as it is a limiting factor in many tests.

Note that given a large enough number of concurrent requests, in both SMR and DUR request execution happens in parallel with message dissemination. However, typical workloads are never able to fully exercise at the same time the available processing power and the TOB protocol. Therefore, for every benchmark configuration and tested system we tried to asses the intuitive predominant characteristics of the workload: execution versus TOB dominance. In case of execution dominance the processing power is the limiting factor: the rate of execution of requests' code is insufficient and thus the TOB protocol is not utilized in 100%. On the other hand, in case of TOB dominance, CPUs often wait for long periods of time until messages are disseminated. Only once the messages are delivered, the executions of currently processed requests can be completed, thus freeing the CPUs to process subsequent requests. The precise definitions of execution or TOB dominance can be found in [2].

5.4.3 Benchmark Results

In Figure 5.1 we present the test results of JPaxos and Paxos STM. The corresponding performance of the non-replicated (and thus fault-prone) sequential counterparts is given Figure 5.2. To better show the contention levels achieved by Paxos STM in the tests, we present in Figure 5.3 the average number of conflicts per update transaction (k_{rw}) for different workloads and cluster configurations. Below we discuss the test results in detail.

Default Hashtable

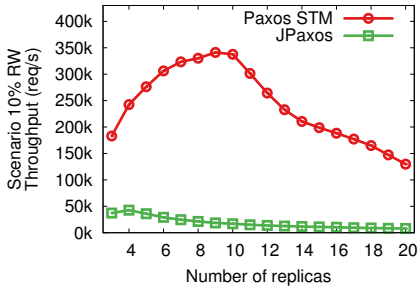
Default Hashtable replicated using JPaxos executes all requests sequentially on each node. Thus, it cannot scale with an increasing number of nodes. No requests are ever reexecuted, since they never conflict. However, the performance of JPaxos is not uniform across the whole range of cluster sizes (see Figure 5.1a). In all three test scenarios, JPaxos achieves the pick throughput when the network is nearly saturated (see the network congestion results in Figure 5.1a). Later, because the network is saturated, the throughput decreases with an increasing number of nodes, and so the time of broadcasting messages to a higher number of replicas also grows. On the other hand, before the network gets saturated, the increase in throughput in the low range of cluster sizes can only be attributed to gradually better utilization of the TOB protocol.

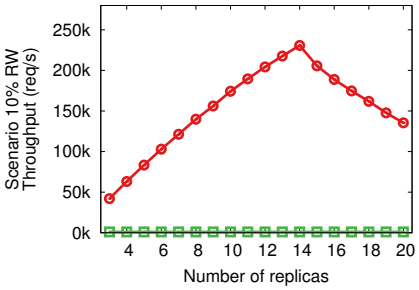
The results in Figure 5.1a show that the execution time in Default Hashtable under JPaxos is *not* dominant, since otherwise the throughput would be constant. Thus, this benchmark is TOB dominated, and processors often wait for requests to arrive.

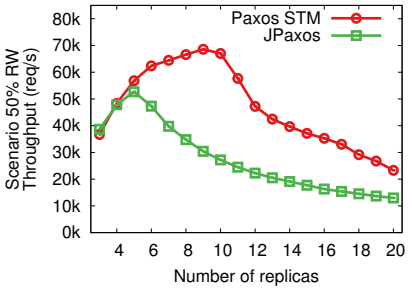
JPaxos achieves the highest performance in the 90% RW test scenario that features the highest percentage of short RW requests, hence we infer that the

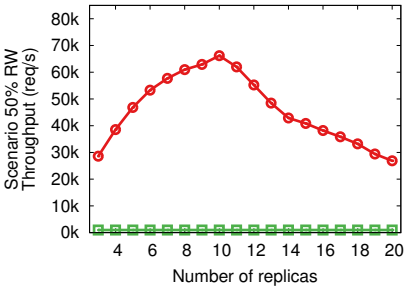
a) Hashmap Default

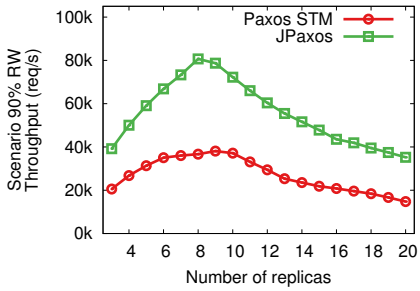
b) Hashmap Prolonged

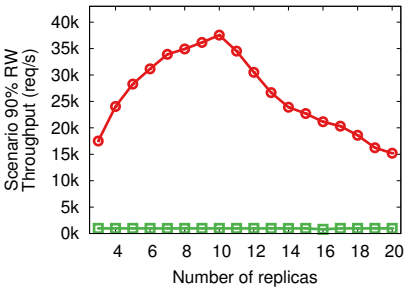


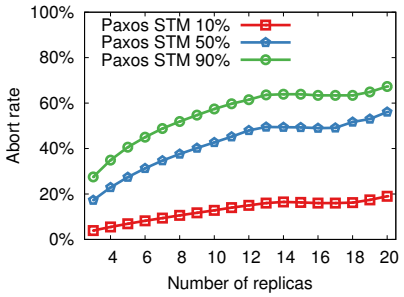


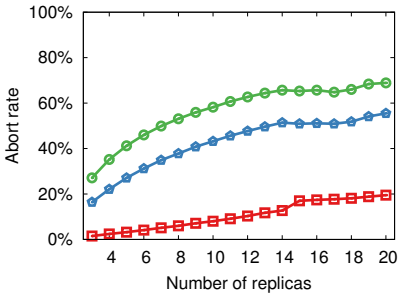


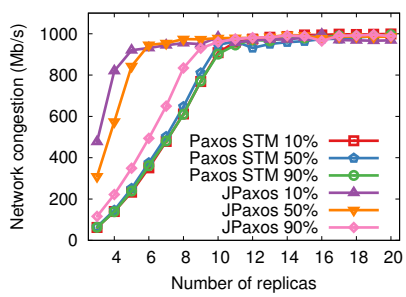


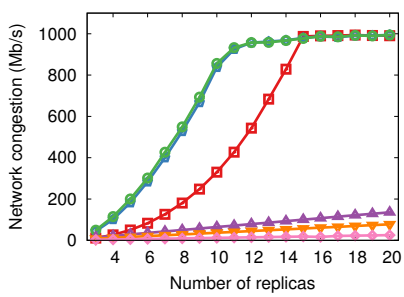












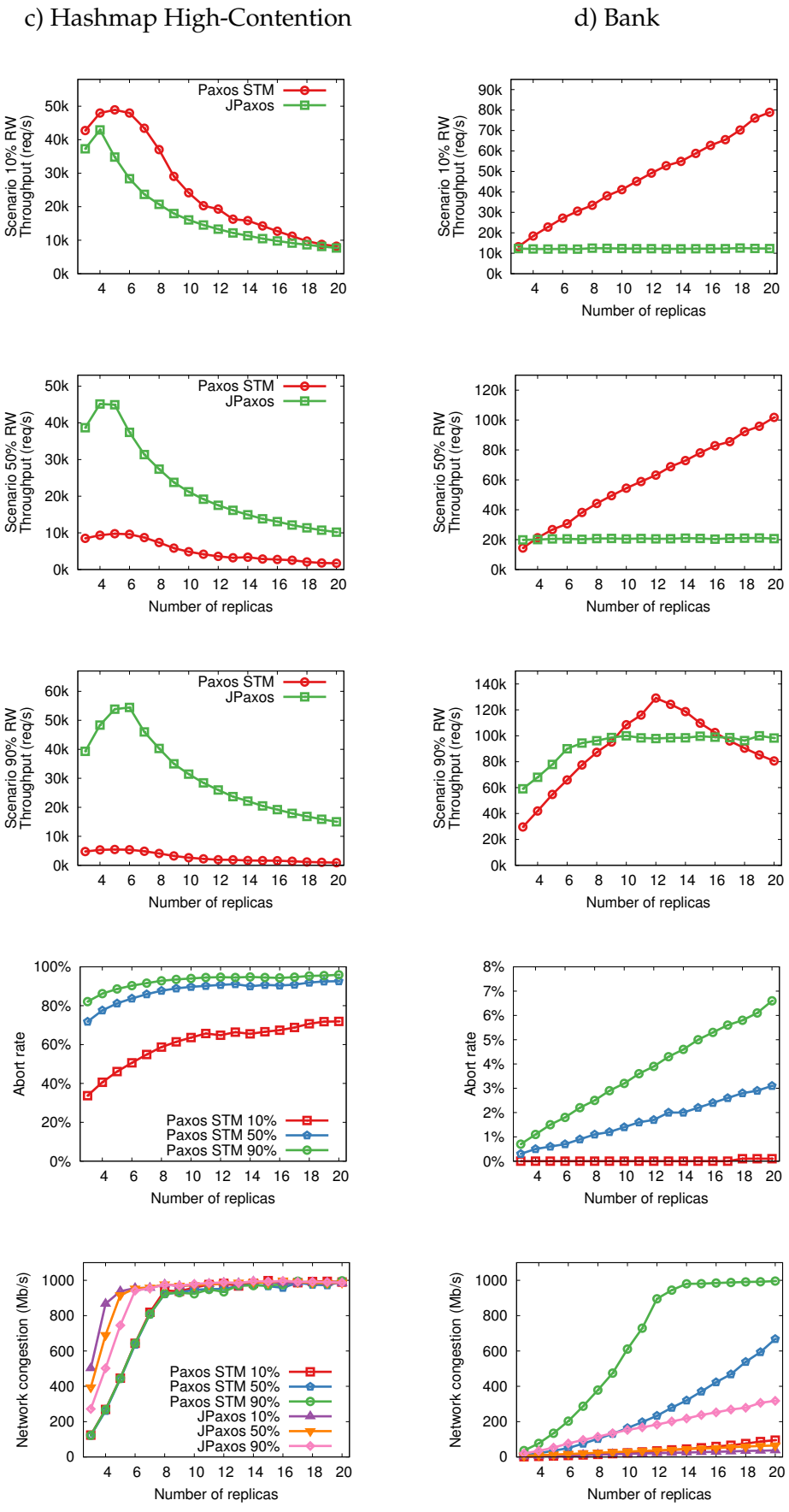


Figure 5.1: Benchmark results for 10%, 50%, and 90% of RW requests (or transactions). Paxos STM implements the DUR scheme whereas JPaxos implements the SMR scheme.

Test scenario →	10% RW	50% RW	90% RW
a) SeqHashtable Default	657785	1002342	2270151
b) SeqHashtable Prolonged	994	996	998
c) SeqHashtable High-Contention	468966	528255	650705
d) SeqBank	40477	73174	208853

Figure 5.2: Throughput of sequential services (req/s).

Cluster size →	3	6	9	11	14	17	20
a) Hashtable Default							
10% RW	0.41	0.89	1.33	1.63	1.98	1.90	2.35
50% RW	0.42	0.91	1.34	1.65	1.95	1.93	2.56
90% RW	0.42	0.91	1.34	1.65	1.97	1.92	2.29
b) Hashtable Prolonged							
10% RW	0.15	0.43	0.76	1.00	1.45	2.15	2.42
50% RW	0.39	0.91	1.38	1.68	2.12	2.07	2.49
90% RW	0.41	0.95	1.41	1.72	2.13	2.05	2.46
c) Hashtable High-Contention							
10% RW	5.08	10.24	15.91	19.15	18.99	22.05	25.59
50% RW	5.09	10.27	16.02	18.41	18.00	19.74	25.03
90% RW	5.10	10.34	15.98	19.09	20.26	19.85	25.99
d) Bank							
10% RW	0.00	0.00	0.00	0.00	0.00	0.00	0.01
50% RW	0.01	0.01	0.02	0.03	0.04	0.05	0.06
90% RW	0.01	0.02	0.03	0.04	0.05	0.07	0.08

Figure 5.3: The number of conflicts per update transaction (k_{rw}) in tests of Paxos STM.

TOB protocol was best utilized in this test compared to other test scenarios, i.e., a larger number of requests was delivered per protocol instance.

Conversely to JPaxos, the Default Hashtable benchmark replicated under Paxos STM gives better performance for a higher percentage of RO requests. This is not surprising since RO requests do not require agreement coordination, hence the costly TOB operation. The plots obtained through experimental evaluation of Paxos STM have a similar shape as for JPaxos, i.e., the throughput increases until the network is saturated, and later steadily drops as abort rate raises (see abort rate in Figure 5.1a). The explanation of this behaviour is the same as it was in case of Default Hashtable replicated with JPaxos. The abort rate is low for the 10% RW test scenario, moderate for the 50% RW test scenario, and high (ranging from 27% up to 67%) when 90% of transactions are updating. When Paxos STM had the highest throughput, then for all test scenarios on average every RW transaction was conflicting and had to be reexecuted at least once before it could commit, as $k_{rw} \approx 1.3$ (see Figure 5.3). For three replicas, on average every second RW transaction was executed ($k_{rw} \approx 0.4$), while for 20 replicas on average every RW transaction was reexecuted at least twice ($k_{rw} \approx 2.3$). A high number of conflicts means wasted resources and lower overall throughput, which was more than two times worse compared to JPaxos in the same scenario (in the 10% and 50% RW scenarios, Paxos STM always performed better than JPaxos).

One can see that Paxos STM in this workload underutilizes the TOB protocol for a small number of replicas until the network becomes saturates. Clients, who are collocated with the replicas, are not able to produce enough requests, so the number of update transactions (only those require agreement coordination using TOB) is not large enough to fully exercise the capability of the TOB protocol.

In all evaluation tests, the non-replicated Default SeqHashtable surpasses the performance of JPaxos- and Paxos STM-based replicated counterparts (see Figure 5.2a). But this immense throughput is achieved at the cost of no fault tolerance and scaling capability. The throughput values among test scenarios vary since the time of executing RO and RW requests is different. In the best 10% RW scenario, Paxos STM-based Default Hashtable reaches at most about half of the performance of its non-replicated counterpart.

Prolonged Hashtable

In contrast to Default Hashtable, where the TOB time was dominant, the Prolonged Hashtable benchmark aims at mimicking a computation-heavy workload with the execution time dominance. For this, we have used exactly the same set of operations in a transaction as in Default Hashtable, but the execution of each request is prolonged by 1 ms.

Prolonged Hashtable, when replicated using JPaxos, has a stunningly uniform throughput of about 965 req/s, regardless of the number of nodes (see Figure 5.1b). This indicates that the lengthy execution time of each request entirely covers up the cost of replica coordination. Since all requests are processed

sequentially, the parallel architecture does not bring any gain in throughput. Moreover, the lengths of RO and RW transactions are similar, thus despite the execution time dominance, there is no difference in performance between the 10%, 50%, and 90% RW scenarios.

In contrast, Prolonged Hashtable replicated using Paxos STM shows excellent scaling. In the 10% RW scenario, the throughput increases with the number of nodes almost linearly, up to the cluster with 14 replicas, when the network becomes saturated. In other scenarios, Paxos STM scales pretty well up to 10 nodes. Then, the network gets saturated and the performance drops. The overall throughput is significantly higher than it is for JPaxos-based Prolonged Hashtable, even for just three nodes. This is credited to Paxos STM's ability of executing transactions in parallel, thus taking the advantage of the multi-core hardware. Note that the abort rate is nearly the same as in the Default Hashtable benchmark.

The non-replicated Prolonged SeqHashtable has the throughput in the range of 994–998 req/s, depending on the scenario (see Figure 5.2b), which is very similar to the one of JPaxos-based replicated counterpart, where all requests are also processed sequentially. This is as expected since the Prolonged Hashtable benchmark replicated using JPaxos has the execution time dominant workload, thus broadcasting a message using TOB takes a relatively small amount of time compared to the duration of request execution. In contrast to Default SeqHashtable, the non-replicated Prolonged SeqHashtable does not even come close to the performance of Paxos STM-based replicated counterpart that was superior in all test scenarios and delivered a much higher throughput.

High-Contention Hashtable

The High-Contention Hashtable benchmark aims at testing a replicated system in a high contention environment. Thus, compared to Default Hashtable's configuration, the number of read and update operations in RW requests grew 5 times: to 40 *get* and 10 *put*/*remove* operations. Note that the execution time of a RW transaction is longer in High-Contention Hashtable than in Default Hashtable. However, it is still much shorter than the time of TOB, which is dominant in this benchmark both for JPaxos and Paxos STM.

The High-Contention and Default Hashtables, which were replicated using JPaxos, have a very similar performance (see Figure 5.1c). The slight difference in the overall throughput stems from the larger size of RW requests in the High-Contention Hashtable benchmark. Therefore, the maximum throughput which is achieved by JPaxos-based High-Contention Hashtable is just a few percent lower compared to the maximum throughput of JPaxos-based Default Hashtable across all scenarios.

The High-Contention Hashtable benchmark, which is implemented using Paxos STM, suffers from a very high contention level (a large number of concurrent transactions try to access the same data). The increased level of contention (compared to contention in Default and Prolonged Hashtables) causes a larger number of transactions to be aborted due to conflicts and reexecuted, which di-

minishes the overall throughput. The abort rate under Paxos STM, starts from 34% in the 10% RW test run on three nodes, and reaches striking 96% in the 90% RW test run on 20 nodes (see abort rate in Figure 5.1c). Therefore, on average, every RW transaction is reexecuted due to conflicts around 5 times in the former case, and 26 times in the latter case, before it finally commits (see Figure 5.3). Compared to JPaxos, Paxos STM-based High-Contention Hashtable performs better only in the 10% RW scenario. In other cases, JPaxos greatly outperforms Paxos STM, which is especially visible in the 90% RW scenario, where on average High-Contention Hashtable performs 10 times better under JPaxos than under Paxos STM which suffers from a very high abort rate.

The non-replicated High-Contention SeqHashtable provides visibly worse throughput than Default SeqHashtable (see Figure 5.2a,c). The lower performance can be attributed to the higher execution time of RW requests. However, the throughput variation across test scenarios is smaller. This is because the execution times of RO and RW requests are approximately the same. Therefore, since the overall throughput is lower, the difference among test scenarios is less noticeable as the number of RW requests increases. However, in all test scenarios, the performance of the non-replicated High-Contention SeqHashtable service trumps the performance of its fault-tolerant counterparts, and the throughput is a few times larger than the best throughput achieved by the implementations using JPaxos and Paxos STM.

Bank Benchmark

Contrary to the Hashtable benchmarks, the number of operations executed as the result of RO and RW requests are very different in the Bank benchmark: A RO request reads all 250k elements of the array, which represent bank accounts. On the other hand, a RW request is very short–it just reads two randomly chosen elements of the array and subsequently modifies them. In effect, the cost of executing a RO transaction is much higher compared to the cost of a RW transaction. However, a huge number of operations executed per each RO request does not lead to large TOB messages in JPaxos, as a RO request only contains a single command to read the bank accounts and calculate a total of funds. Therefore, the processing CPUs have become the bottleneck in this benchmark, not the computer network.

In the 10% and 50% RW test scenarios, the throughput of Bank replicated using JPaxos is constant across cluster sizes nodes (see Figure 5.1d), and it is clearly restricted by the CPU time of processing requests. In case of the 90% RW scenario, the situation is slightly different. For a small number of nodes (3-7), JPaxos-based Bank experiences the TOB time dominance, while for a larger cluster size, the execution time is dominant. In the range of 3-7 nodes, the throughput grows linearly, as in case of other TOB time dominant tests. This behaviour is attributed to a relatively low number of submitted requests, which make JPaxos underutilize the TOB protocol. However, on the contrary to other TOB time dominated benchmarks, when the throughput reaches its peak value, it stays constant instead of deteriorating with an increasing number of replicas. This is

because the peak performance is not bounded in this test by the network bandwidth, but rather by the available processing power. In fact, the network is far from being saturated for every cluster size. Hence, for a high number of nodes, the throughput of JPaxos is only limited by the time of processing requests.

Bank implemented using Paxos STM also experiences the TOB time dominance only for the 90% RW scenario. As in the most of other tests, the system improves the performance up to the point when the network becomes saturated. Since the contention levels are relatively small for the 90% RW scenario, the degradation of performance for a higher number of replicas is relatively benign. In the 10% and 50% RW scenarios, JPaxos-based Bank is execution time dominant and the system scales almost linearly across the whole range of cluster sizes. This behaviour can be attributed to the ability of Paxos STM to use the underlying parallel multicore architecture well and process many requests in parallel using a large number of available processor cores. These good results were heavily influenced by the optimizations of the TOB protocol, namely, batching and pipelining. Without these optimizations, the TOB protocol would perform much worse, given a very small size of agreement coordination messages that carry transaction read-sets and updates (each message has only 76 bytes), extremely short RW transaction execution times, and a large number of RW transactions performed concurrently.

Note that the performance results that we got for Bank are quite different from the results obtained by running the Hashtable benchmarks. In all variants of Hashtable, test scenarios with a larger number of RO requests generated a higher throughput for all cluster sizes. Now, the opposite is true–the best results are obtained for the 90% of RW requests. This behaviour can be easily explained if we recall that it takes significantly more time to execute a RO request than a RW request.

The non-replicated SeqBank outperforms its replicated counterparts for the 90% RW scenario, with the throughput reaching 208k req/s (see Figure 5.2d). However, in both 10% and 50% RW scenarios, SeqBank performs better than JPaxos-based Bank for all cluster sizes, but worse than Paxos STM-based Bank for larger cluster sizes. E.g., in the cluster with 20 nodes, in the 10% RW scenario, SeqBank has half the throughput of Paxos STM-based Bank, and in the 50% RW scenario, it reaches 3/4 of the throughput that is achieved by Paxos STM-based Bank. Note that the implementation of SeqBank does not incur any overhead that is characteristic for the replicated counterparts. This fact, together with a very short time of RW requests compared to RO requests, enabled SeqBank to perform five times better when the number of RW requests grew from 10% to 90%.

5.4.4 Evaluation Summary

The results of the evaluation indicate that neither SMR (JPaxos) nor DUR with the multiversioning optimization (Paxos STM) is superior in all cases. This outcome may be surprising given that only DUR can scale with increasing number

of replicas. The performance of both replication schemes heavily depends on the characteristics of the workload.

JPaxos executes all requests sequentially, thus if the execution time is dominant, it performs poorly compared to Paxos STM which can process transactions in parallel and thus scale (see Figure 5.1b). Paxos STM behaves well especially in test scenarios involving many RO transactions, but suffers under high contention (evidenced by abort rate). Then, JPaxos is clearly better since it delivers predictable and stable performance (see Figure 5.1c for 50% and 90% RW). However, the overhead caused by TOB, and by the network that often gets saturated, severely reduce the performance of both systems. TOB dominance can overshadow the gain of parallelism in Paxos STM and, in consequence, reduce scalability. If at least RO requests dominate TOB, then efficiency could be kept constant by decreasing the number of conflicts, e.g., by contention management.

The optimizations of TOB give Paxos STM considerable performance boost. This is especially visible in Bank (see Figure 5.1d) where replicas can get a lot of transactions ready to commit at the same time, so the TOB protocol with batching can broadcast them all at once.

When TOB time is dominant (Figure 5.1a,c), JPaxos does not perform uniformly, and the pick throughput is when the network is nearly saturated (see the network congestion plot). Before that, JPaxos' throughput grows with the number of replicas. This is because with replicas are collocated clients that can now produce more requests, but still their quantity and capacity are not enough to effectively utilize the TOB protocol. (The network saturation occurs later in Bank where message sizes are small.) Once the network becomes saturated, the throughput is deteriorating as the cluster size increases, since threads (CPUs) are exhausted by TOBs and execute less requests.

Not surprisingly JPaxos always yields lower performance than the non-replicated implementations of our benchmarks (see Figures 5.1 and 5.2). Although in both cases all requests are executed sequentially, in JPaxos a request additionally needs to be broadcast to all replicas, what can take a substantial amount of time and processing power. More interestingly, Paxos STM and sequential services gave variable results. E.g., Paxos STM was the clear winner for Prolonged Hashtable (see Figure 5.1b and 5.2b) and for Bank with 10% and 50% of RW requests (see Figure 5.1d and 5.2d). This is credited to Paxos STM's ability of executing transactions in parallel, thus being able to fully utilize the multi-core hardware. In other cases, the sequential, non-replicated services demonstrated higher throughput. However, they are vulnerable to system failures. In contrast, replicated services can tolerate failures of machines and communication links, thus ensuring service reliability and availability.

5.5 Comparison

As we showed in the previous section, in many cases SMR proves to be highly efficient although it allows no parallelism (or limited parallelism in case of LSMR). In fact, when a workload is not CPU intensive, it performs much better than DUR. Also SMR is relatively easy to implement, because most of the complexity is hidden behind TOB. A major drawback of SMR is that it requires a replicated service to be deterministic. Otherwise consistency could not be preserved.

Contrary to SMR (and partially to LSMR), in DUR parallelism is supported for read-only as well as updating transactions by default–each transaction is executed by a single replica in a separate thread and in isolation. This way DUR takes better advantage over modern multicore hardware. However, the performance of DUR is limited for workloads generating high contention. It is because in such conditions transactions may be aborted numerous times before eventually committing. Aborting live transactions as soon as they are known to be in conflict with a transaction that had just recently committed may help but only to some degree.

LSMR and DUR requires no synchronization (no communication step) among replicas for read-only transactions as they do not change the local or replicated state. This way read-only requests are handled by LSMR and DUR much more efficiently compared to SMR. Additionally, in DUR read-only transactions can be provided with abort-free execution guarantee by introducing the multiversioning scheme (LSMR does not provide transactional semantics and execution of requests cannot be aborted altogether).

Usually there is a significant difference in the size of network messages communicated between replicas in SMR/LSMR and DUR. In DUR, the broadcast messages contain transaction descriptors with readsets and updates sets. The size of these messages can be significant even for a medium sized transaction. Large messages cause strain on the TOB mechanism and increase transaction certification overhead. On the other hand, in SMR usually the requests consist only of an identifier of a method to be executed and data required for its execution; these messages are often as small as 100B.

The DUR scheme supports concurrency on multicore architectures. Concurrent programming is error-prone but atomic transactions greatly help to write correct programs. Firstly, operations defined within a transaction appear as a single logical operation whose results are seen entirely or not at all. Secondly, concurrent execution of transactions is deadlock-free which guarantees progress. Moreover, the *rollback* and *retry* constructs enhance expressiveness. However, as mentioned earlier, irrevocable operations are not permitted since at any moment a transaction may be forced to abort and restart due to conflicts with other transactions.

All three SMR, LSMR and DUR offer different correctness guarantees. SMR offers the strongest guarantees, i.e., real-time linearizability. LSMR offers slightly weaker guarantees for transactions marked as read-only. DUR guarantees up-

date-real-time opacity, which corresponds to update-real-time linearizability. Hence, it offers the weakest guarantees of the three schemes.

6 Hybrid Transactional Replication

In this chapter, we define Hybrid Transactional Replication (HTR), a novel transactional replication scheme that seamlessly merges DUR and SMR. First, we discuss the transaction oracle–the key new component of our algorithm. Next, we explain the HTR algorithm by presenting its pseudocode and giving the proof of correctness. Then, we discuss the strengths of HTR and present two approaches to creating an oracle: a manual, tailored for a given workload, and an automatic, based on machine learning. Finally, we evaluate the performance of our approach under various workloads.

Originally, we introduced HTR in [6]. We proposed the oracle based on machine learning in [4]. There we also presented the formal proof of correctness of HTR.

6.1 Transaction Oracle

Our aim was to seamlessly merge the SMR and DUR schemes, so that requests (transactions) can be executed in either scheme depending on the desired performance considerations and execution guarantees (e.g., support for irrevocable operations). *Transaction oracle* (or *oracle*, in short) is a mechanism that for a given transaction's run is able to assess the best execution mode: either the *SM mode*, which resembles request execution using SMR, or the *DU mode*, which is analogous to executing a request using DUR. The oracle may rely on hints declared by the programmer as well as on dynamically collected *statistics*, i.e., data regarding various aspects of system's performance, such as:

- duration of various phases of transaction processing, e.g., execution time of a request's (transaction's) code, TOB latency, and duration of transaction certification,
- abort rate, i.e., the ratio of aborted transaction runs to all execution at-

tempts,

- sizes of exchanged messages, readsets and updates sets,
- system load, i.e., a measure of utilization of system resources such as CPU and memory,
- delays introduced by garbage collector,
- saturation of the network.

Declared read-only transactions, i.e., transactions known *a priori* to be read-only, are always executed in the DU mode since they do not alter the local or replicated state and thus do not require distributed certification. Hence, decisions made by the oracle only regard updating transactions.

Since the hardware and the workload can vary between the replicas the system can use different oracles at different nodes and independently change them at runtime when desired. For brevity, in the description of the algorithm we abstract away the details of the oracle implementation and treat it as a black box with only two functions: FEED($data$), used to update the oracle with data collected over the last transaction's run, regardless of the outcome, and QUERY($request$), used to decide in which mode a new transaction is to be executed).

The problem of creating a well-performing oracle is non-trivial and depends on the expected type of workload. In Section 6.5 we discuss a handful of tips on how to build an oracle that matches the expected workload. Then, in Section 6.6, we also show an oracle which uses machine learning techniques to automatically adjust its policy to changes in the workload.

6.2 Specification

Below we describe the HTR algorithm, whose pseudocode is given in Algorithm 6. HTR is essentially DUR (Algorithm 4), extended with the SMR scheme (Algorithm 2) and the UPDATEORACLESTATISTICS procedure (line 16) that feeds the oracle with the statistics collected in a particular run of a transaction before the transaction is committed, rolled back, or retried.

Note that HTR features two sets of functions/procedures facilitating execution of a transaction (i.e., performing *read* and *write* operations on shared objects) and managing the control flow of the transaction (i.e., procedures used to commit, rollback or retry the transaction). One set of functions/procedures is used by transactions executed in the DU mode (lines 35–56) and one is used by transactions executed in the SM mode (lines 81–98).

When a transaction is about to be executed, the oracle is queried to determine the execution mode for this particular transaction's run (line 26). When the DU mode is chosen (line 27), a transaction, called a *DU transaction*, is executed and certified exactly as in DUR. It means that it is executed locally (line 30) and once the execution is finished and the COMMIT procedure invoked (line 43), the in-

formation about the transaction is broadcast using TOB (line 50) to all replicas so that the transaction can undergo the final certification and possibly commit (depending on the outcome of the final certification, line 62). On the other hand, when the SM mode is chosen (line 31), the request is first broadcast using TOB (line 32) and then executed on all replicas as an *SM transaction* (lines 77–80). The execution of SM transactions happens in the same thread which is responsible for certifying DU transactions and applying the updates they produced. It means that at most one SM transaction can execute at a time and its execution does not interleave with handling of commit of DU transactions. However, the algorithm does not prevent concurrent execution of an SM transaction and multiple DU transactions; only the certification test and the state update operations of these DU transactions may be delayed until the SM transaction is completed.

Since the execution of an SM transaction is never interrupted by receipt of a transaction descriptor of a DU transaction, no SM transaction is ever aborted. It means that an SM transaction does not need to be certified and can commit straight away (lines 88–91). For the same reason, reading a shared object does not involve checking for conflicts (line 82) and reading the current value of LC (line 78) does not have to be guarded by a lock.

Naturally, an SM transaction has to be deterministic, so that the state of the system is kept consistent across replicas.

Note that regardless of the used execution mode and the fate of the transaction (i.e., whether the transaction is committed, aborted or rolled back), the statistics gathered during the transaction's run are always fed to the oracle after the run is completed (lines 51 and 72).

Because the pseudocode of HTR is based on the pseudocode we provided for DUR (Algorithm 4), there are similar simplifications in both pseudocodes: we use a single global (reentrant) lock to synchronize operations on LC (lines 29, 63, 64, 88, 89), Log (lines 10, 65, 90), and the accesses to transactional objects (lines 40, 66, 91), we allow Log to grow indefinitely and we use the same certification procedure for both the certification test performed upon every read operation for DU transactions (line 37) and the certification test that happens after a transaction descriptor of a DU transaction is delivered to the main thread (line 62). The limitations introduced by these simplifications can be mitigated in a similar manner as in DUR.

6.3 Characteristics

Below we present the advantages of the HTR algorithm compared to the exclusive use of the schemes discussed in Chapter 5. We also discuss the potential performance benefits that will be evaluated experimentally in Section 6.7.

Algorithm 6 Hybrid Transactional Replication for process p_i

```
1: integer LC ← 0
2: set Log ← ∅
3: function GETOBJECT(txDescriptor t, objectId oid)
4:     if (oid, obj) ∈ t.updates then
5:         value ← obj
6:     else
7:         value ← retrieve object oid
8:     return value
9: function CERTIFY(integer start, set readset)
10:    lock { L ← {t ∈ Log : t.end > start} }
11:    for all t ∈ L do
12:        writeset ← {oid : ∃(oid, obj) ∈ t.updates}
13:        if readset ∩ writeset ≠ ∅ then
14:            return failure
15:    return success
16: procedure UPDATEORACLESTATISTICS(txDescriptor t)
17:    TransactionOracle.FEED(t.stats)

Thread q on request r from client c (executed on one replica)
18: enum outcome_q ← failure                    // type: enum { success, failure }
19: response res_q ← null
20: txDescriptor t_DU ← null                    // type: record (id, start, end, readset, updates, stats)
21: upon INIT
22:    wait until LC ≥ r.clock
23:    TRANSACTION()
24:    return (r.id, LC, res_q) to client c
25: procedure TRANSACTION
26:    mode ← TransactionOracle.QUERY(r)
27:    if mode = DUmode then
28:        t_DU ← (a new unique id, 0, 0, ∅, ∅, ∅)
29:        lock { t_DU.start ← LC }
30:        res_q ← execute r.prog with r.args
31:    else                                     // mode = SMmode
32:        TO-BROADCAST r                       // blocking
33:    if outcome_q = failure then
34:        TRANSACTION()
35: function READ(objectId oid)
36:    t_DU.readset ← t_DU.readset ∪ {oid}
37:    lock { if CERTIFY(t_DU.start, {oid}) = failure then
38:                RETRY()
39:            else
40:                return GETOBJECT(t_DU, oid) }
41: procedure WRITE(objectId oid, object obj)
42:    t_DU.updates ← {(oid', obj') ∈ t_DU.updates : oid' ≠ oid} ∪ {(oid, obj)}
43: procedure COMMIT
44:    stop executing r.prog
45:    if t_DU.updates = ∅ then
46:        outcome_q = success
47:        return
48:    if CERTIFY(t_DU.start, t_DU.readset) = failure then
49:        return
50:    TO-BROADCAST t_DU                        // blocking
51:    UPDATEORACLESTATISTICS(t_DU)
```

52: **procedure** RETRY
53: stop executing $r.prog$
54: **procedure** ROLLBACK
55: stop executing $r.prog$
56: $outcome_q \leftarrow success$

The main thread of HTR (executed on all replicas)
57: enum $outcome \leftarrow null$ *// type: enum { success, failure }*
58: response $res \leftarrow null$
59: request $r \leftarrow null$
60: txDescriptor $t_{SM} \leftarrow null$
61: **upon** TO-DELIVER (txDescriptor t_{DU})
62: **if** CERTIFY($t_{DU}.start, t_{DU}.readset$) $= success$ **then**
63: **lock** { $LC \leftarrow LC + 1$
64: $t_{DU}.end \leftarrow LC$
65: $Log \leftarrow Log \cup \{t_{DU}\}$
66: apply $t_{DU}.updates$ }
67: **if** transaction with $t_{DU}.id$ executed locally by thread q **then**
68: $outcome_q \leftarrow success$
69: **upon** TO-DELIVER (request r_q)
70: $r \leftarrow r_q$
71: TRANSACTION()
72: UPDATEORACLESTATISTICS(t_{SM})
73: **if** request with $r.id$ handled locally by thread q **then**
74: $outcome_q \leftarrow outcome$
75: $res_q \leftarrow res$
76: **procedure** TRANSACTION
77: $t_{SM} \leftarrow$ (a deterministic unique id based on $r.id, 0, 0, \emptyset, \emptyset, \emptyset$)
78: $t_{SM}.start \leftarrow LC$
79: $res \leftarrow null$
80: $res \leftarrow$ execute $r.prog$ with $r.args$
81: **function** READ(objectId oid, object obj)
82: **return** GETOBJECT(t_{SM}, oid)
83: **procedure** WRITE(objectId oid, object obj)
84: $t_{SM}.updates \leftarrow \{(oid', obj') \in t_{SM}.updates : oid' \neq oid\} \cup \{(oid,\ obj)\}$
85: **procedure** COMMIT
86: stop executing $r.prog$
87: **if** $t_{SM}.updates \neq \emptyset$ **then**
88: **lock** { $LC \leftarrow LC + 1$
89: $t_{SM}.end \leftarrow LC$
90: $Log \leftarrow Log \cup \{t_{SM}\}$
91: apply $t_{SM}.updates$ }
92: $outcome \leftarrow success$
93: **procedure** RETRY
94: stop executing $r.prog$
95: $outcome \leftarrow failure$
96: **procedure** ROLLBACK
97: stop executing $r.prog$
98: $outcome \leftarrow success$

6.3.1 Expressiveness

Implementing services using the original SMR replication scheme is straightforward since it does not involve any changes to the service code. However, the programmer does not have any constructs to express control-flow other than the execution of a request in its entirety. In our HTR replication scheme, the programmer can use expressive transactional primitives ROLLBACK and RETRY to withdraw any changes made by transactions and to retry transactions (possibly in a different replication mode). In this sense, these constructs are analogous to DUR's, but they are also applicable for transactions executed in the pessimistic SM mode. Upon retry, the SM transaction is not immediately reexecuted on each node. Instead, the control-flow returns to the thread which is responsible for handling the original request. The oracle is then queried again, to determine in which mode the transaction should be reexecuted. Similarly, reexecution of DU transactions is also controlled by the oracle.

Constructs such as RETRY can be used to suspend execution of a request until a certain condition is met. Note that in SMR doing so is not advisable since it would effectively block the whole system. It is because in SMR all requests are executed serially in the order they are received. On the contrary, when RETRY is called from within an SM transaction, the HTR algorithm rolls back the transaction and allows it to be restarted when the condition is met.

6.3.2 Irrevocable Operations

In DUR, transactions may be aborted and afterwards restarted due to conflicts with other older transactions. Thus, they are forbidden to perform *irrevocable operations*, i.e., operations whose side effects cannot be rolled back (such as local system calls). *Irrevocable (or inevitable) transactions* are transactions that contain irrevocable operations. Support for such transactions is problematic and has been subject of extensive research in the context of non-distributed TM (see Section 2.4.1). However, the proposed methods and algorithms are not directly transferable to distributed TM systems where problems caused by distribution, partial failures, and communication must also be considered. Below we explain how the HTR algorithm deals with irrevocable transactions.

In the HTR algorithm, irrevocable transactions are executed exclusively in the SM mode, thus guaranteeing abort-free execution, which is necessary for correctness. It also means that only one irrevocable transaction is executed at a time. However, our scheme does not prevent DU transactions to be executed in parallel–only certification and the subsequent process of applying updates of DU transactions (in case of successful certification) must be serialized with execution of SM transactions. Since an SM transaction runs on every replica, we only consider deterministic irrevocable transactions. Non-deterministic transactions would require acquisition of a global lock or a token to be executed exclusively on a single replica. Alternatively, some partially centralized approaches could be employed, as in [86]. However, they introduce additional communication steps, increase latency, and may force concurrent transactions to wait a

significant amount of time to commit.

We forbid the ROLLBACK and RETRY primitives in irrevocable transactions (as in [80] and other TM systems) since they may leave the system in an inconsistent state.[1]

6.3.3 Performance

As mentioned in Section 5.2, it is not straightforward to optimize the original SMR scheme to handle read-only requests in parallel with other (read-only or updating) requests. However, in the HTR algorithm, read-only transactions are executed only by one replica, in parallel with any updating transactions–there is no need for synchronization among replicas to handle the read-only transactions.

HTR can benefit from the multiversioning optimization in the same way as can the DUR scheme (see Section 5.3.4). In HTR extended with this optimization read-only transactions are guaranteed abort-free execution thus boosting HTR's performance for workloads dominated by read-only requests. The implementation of HTR which we use in our tests implements the multiversioning optimization (see Section 6.7).

Unless an updating transaction is irrevocable (thus executed in the SM mode) or non-deterministic (thus executed in the DU mode), it can be handled by HTR in either mode for increased performance. The choice is made by the HTR oracle that constantly gathers statistics during system execution and can dynamically adapt to the changing workload (which may vary between the replicas). In Section 6.5, we discuss the tuning of the oracle and in Section 6.6 we introduce an oracle, which relies on machine learning techniques for dynamic adaptation to changing conditions.

6.4 Correctness

Below we give formal results on the correctness of HTR. The reference safety property we aim for is update-real-time opacity which we introduced in Section 4.3 and used to prove correctness of DUR (see Section 5.3.3). Roughly speaking, update-real-time opacity is satisfied, if for every execution of an algorithm (represented by some history H) it is possible to construct a sequential history S such that:

1. H is equivalent to S, i.e., H and S contain the same set of transactions, all read and write operations return the same values and the matching transactions commit with the same outcome,

[1] Interestingly, Atomic RMI [85], a fully pessimistic distributed (but not replicated) TM system, allows nondeterministic irrevocable operations to be performed inside transactions.

2. every transaction in S is legal, i.e. the values of shared objects read by the transaction are not produced out of thin air but match the specification of the shared objects, and
3. S respects the real-time order for committed updating transactions in H, i.e., for any two committed updating transactions T_i and T_j, if T_i ended before T_j started then T_i appears before T_j in S.

However, it is impossible to directly prove that HTR satisfies update-real-time opacity due to a slight model mismatch, as we now explain. Recall that in HTR, every time a request is executed in the SM mode, multiple identical transactions are executed across the whole system (the transactions operate on the same state and produce the same updates). In the formalization of update-real-time opacity (which is identical to the formalization of the original definition of opacity by Guerraoui and Kapalka), every such transaction is treated independently. Therefore, unless such an SM transaction did not perform any modifications or rollback on demand, it is impossible to construct such a sequential history S, in which every transaction is legal.[2] However, we can show that execution of multiple SM transactions regarding the same client request is equivalent to an execution of a single transaction (on some replica) followed by dissemination of updates to all processes, as in case of a DU transaction. Therefore, we propose a mapping called *SMreduce*, which allows us to reason about the correctness of HTR. Roughly speaking, under the SMreduce mapping of some history of HTR, for any group of SM transactions regarding the same request r, such that the processes that executed the transactions applied the updates produced by the transactions, we allow only the first transaction of the group in the history to commit; other transactions appear aborted in the transformed history. The detailed definition of SMreduce, together with formal proof of correctness can be found in Appendix B.4.

Before we prove that HTR satisfies update-real-time opacity under the SMreduce mapping, we first show that HTR does not satisfy a slightly stronger property, *write-real-time opacity*, and thus also does not guarantee *real-time opacity* (which is equivalent to the original definition of opacity [31], as shown in Section 4.3).

Theorem 11. *Hybrid Transactional Replication does not satisfy write-real-time opacity.*

Proof. Trivially, every t-history of DUR is also a valid t-history of HTR, because transactions in DUR are handled exactly in the same way as DU transactions in HTR. Since DUR does not satisfy write-real-time opacity [3], neither does HTR. □

Corollary 7. *Hybrid Transactional Replication does not satisfy real-time opacity.*

[2] As a counter example consider an execution of HTR featuring a single client request which is executed as an SM transaction on every replica: a transaction first reads 0 from a transactional object x and subsequently increments the value of x, i.e., writes 1 to x. Even for 2 replicas, it is impossible to construct a legal sequential history featuring all the SM transactions.

Proof. The proof follows directly from Theorem 11 and definitions of write-real-time opacity and real-time opacity (real-time opacity is strictly stronger than write-real-time opacity). □

Theorem 12. *Under the SMreduce mapping, Hybrid Transactional Replication satisfies update-real-time opacity.*

Proof sketch. In order to prove that HTR satisfies update-real-time opacity, we have to show that (by Corollary 1) for every finite t-history H produced by HTR, under SMreduce there exists a t-sequential t-history S equivalent to $\bar{H}$ (some completion of H), such that S respects the update-real-time order of H and every transaction T_k in S is legal in S.

The proof is somewhat similar to the proof of correctness of DUR. The crucial part of the proof concerns a proper construction of S. We do so in the following way. Let us start with an empty t-sequential t-history S'. Now we add to S' t-histories $\bar{H}|T_k$ of all committed updating transactions T_k from $\bar{H}$. However, we do so according to the order of the (unique) numbers associated with each such transaction T_k (with transaction descriptor t_k). The number associated with T_k is the value of $t_k.end$. This value is equal to the value of the LC variable of the replica that processes the updates of T_k upon committing T_k. We show that this number is the same for each replica and that it unambiguously identifies the operation execution (thanks to the use of TOB as the sole mechanism used to disseminate messages among replicas, the fact that LC is incremented upon commit of every updating transaction and the SMreduce mapping). Then, one transaction at a time, we add to S' all operations of all aborted and read-only transactions from $\bar{H}$. Note that we include here all the SM transactions which are committed in the original history but aborted under the SMreduce mapping in $\bar{H}$. For every such transaction T_k (with transaction descriptor t_k), we insert $\bar{H}|T_k$ in S' immediately after operations of a committed updating transaction T_l (with transaction descriptor t_l), such that $t_k.start = t_l.end$. It means that T_l is the last committed updating transaction processed by the replica prior to execution of T_k (note that T_l might not exist if T_k is executed on the initial state of the replica). Now we consider independently each (continuous) sub-t-history of S' which consists only of operations of read-only and aborted transactions (i.e., roughly speaking, we consider the periods of time in between commits of updating transactions). For each such sub-t-history, when necessary, we rearrange the operations of the read-only and aborted transactions so that their order respects the order in which these operation executions appear in sub-t-history $\bar{H}|p_i$, for every process p_i. Because we always consider all operations of every transaction together, S' is t-sequential. Then $S = S'$.

By construction of S, it is easy to show that S respects the update-real-time order of H. Then we show by a contradiction that there does not exist a transaction in S that is not t-legal, hence every transaction in S is legal. □

6.5 Tuning the Oracle

As pointed out in [27], DTM workloads that are commonly considered are usually highly diversified in regard to the execution times and to the number of objects accessed by each transaction (this is also reflected in our benchmark tests in Section 6.7). However, the execution times of the majority of transactions are way under 1 ms. Therefore, the mechanisms that add to transaction execution time have to be lightweight or otherwise the benefits of having two execution modes will be overshadowed by the costs of maintaining an oracle.

In the HTR algorithm, the oracle is defined by only two methods that have to be provided by the programmer. Combined with multiple parameters collected by the system at runtime, the oracle allows for a flexible solution that can be tuned for a particular application. Our experience with HTR-enabled Paxos STM and multiple benchmarks shows that there are the two most important factors that should be considered when implementing an oracle:

- Keeping abort rate low. A high abort rate means that many transactions executed in the DU mode are rolled back (multiple times) before they finally commit. This undesirable behaviour can be prevented by executing some (or all) of them in the SM mode. The SM mode can also be chosen for transactions consisting of operations that are known to generate a lot of conflicts, such as resizing a hashtable. On the contrary, the DU mode is good for transactions that do not cause high contention, so can be executed in parallel thus taking advantage of modern multicore hardware.
- Choosing the SM mode for transactions that are known to generate large messages when executed optimistically in the DU mode. Large messages increase network congestion and put strain on the TOB mechanism, thus decreasing its performance. The execution of an SM transaction usually only requires broadcasting the name of the method to be invoked; such messages are often shorter than 100B.

Note also that since SM transactions are guaranteed to commit, they do not require certification, which eliminates the certification overhead. This overhead (in the DU mode) is proportional to the size of transactions' readsets and updates sets.

In [6] we evaluated HTR-enabled Paxos STM using manually devised oracles that were designed to fit the expected workload. The oracles delivered good performance, even though the oracles' policies were very simple: they either limited the abort rate, had transaction execution modes predefined for each transaction type or simply executed in the SM mode transactions which were known *a priori* to cause high contention.

Naturally, the more complex the application, the more difficult designing an oracle which works well. Moreover, manually defined oracles have limited capability to adjust to changing workloads. Therefore we decided to create mech-

anisms that aid the programmer in devising oracles that can adopt to varying conditions.

6.6 Machine-Learning-based Oracle

Before we describe our machine learning (ML) based approach to creating oracles, let us first reflect on the constraints of our environment and the requirements that we set.

6.6.1 Requirements and Assumptions

Determining the optimal execution mode for each transaction run (in a certain state of the system) can be considered a classification problem. Solving such problems is often accomplished by employing offline machine learning techniques such as decision trees, nearest neighbours or neural networks [111]. However, it seems that resorting to such (computation-heavy) mechanisms in our case is not most advantageous because of the high volatility of the environment which we consider. Our system scarcely uses stable storage (whose performance is typically the limiting factor in database and distributed storage systems) and thus Paxos STM's performance is sensitive even to small changes in the CPU load. In turn, the changes could be caused by variance in one or many aspects of the workload such as sizes of received requests, shared object access patterns, request execution times, number of clients, contention levels, etc. Therefore, we opted for reinforcement learning techniques, i.e., approaches which learn by observing the *rewards* on the already made decisions.

Naturally, the primary limitation for the automated oracle is that the mechanism it relies on cannot incur a noticeable overhead on transaction processing. Otherwise, any gains resulting from choosing an optimal execution mode would be overshadowed by the time required for training the oracle or querying it. It means that we had to resort to lightweight ML techniques that are neither CPU nor memory intensive (see below). Also, the ML mechanism must work well in a multithreaded environment. This can be tricky because each query to the oracle is followed by a feedback on transaction execution passed to the ML mechanism. Note that the statistics gathered on a particular transaction run heavily depend on the overall load of the system, therefore calculating the reward (used by the ML mechanism to learn) is not straightforward.

Ideally, before a transaction is executed, the oracle should know what objects the transaction will access and approximately how long the execution will take. This is typically done in, e.g., SQL query optimizers featured in most of the database engines. Unfortunately, obtaining such information in our case is very difficult. It is because in our system transactions may contain arbitrary code and are specified in Java, a rich programming language, which enables complex constructs. One could try static code analysis as in [112], but this approach tends

to be expensive and not that accurate in the general case. However, it is reasonable to assume that not every request (transaction) arriving in the system is completely different from any of the already executed ones. Therefore, transactions can be clustered based on some easily obtainable information (e.g., content of the arguments passed alongside transaction's code), statistics on past executions that aborted due to conflicts or simple hints given by the programmer. The latter could range from, e.g., a qualitative level of contention generated by the transaction (*low*, *medium*, *high*), to the number of objects accessed by the transaction compared to other transactions, or to as straightforward as a unique number which identifies a given class of transactions (as in our system, see below).

6.6.2 Approach inspired by Multi-armed Bandit Problem

The ML-based oracle called *HybridML* (or *HybML* in short), which we propose, relies on a rough classification provided by the programmer. As mentioned above, the classification may involve various elements but we investigate the simplest one, in which similar transactions have the same number associated with them. We say that transactions with the same number form a *class*. For example, a class can be formed out of transactions which perform money transfer operations between pairs of accounts. Such transactions are inherently similar despite moving funds between different pairs of accounts. The similarities regard, e.g., shared objects access pattern, CPU utilization, broadcast message sizes, etc.[3]

HybML is inspired by and closely resembles the *epsilon-greedy strategy* for solving the *multi-armed bandit problem* (see [92] [93] for the problem and [94] for algorithms). In the multi-armed bandit problem there is a number of slot machines which, when played, return a random reward from a fixed but unknown probability distribution specific to that machine. The goal is to maximize the sum of rewards in a sequence of plays. In the epsilon-greedy strategy, in any given play with some small probability ϵ a random slot machine is chosen. In the majority of plays, however, the chosen machine is the one that has been performing best in the previous rounds. Varying the value of ϵ enables balancing of exploration and exploitation.

Roughly speaking, in HybML we use a slightly modified version of the epsilon-greedy strategy to solve the two-armed bandit problem for each class independently (with DU and SM modes corresponding to the two slot machines). Firstly, HybML determines whether to optimize the network or CPU usage. In the former case, HybML aims at choosing an execution mode in which broadcast messages are smaller. Otherwise, HybML decides on an execution mode in which the transaction can execute and commit more quickly.

The exact way in which HybML works is a bit more complicated. When a new transaction is about to start, HybML first checks what was the preferred execution mode for the given class of transactions. Then, depending on the most

[3] In case of more complicated transactions, which feature loops or multiple if-then statements, there might be bigger differences in the mentioned characteristics. Then, however, the programmer may easily provide slightly finer classification of the transactions.

prevalent mode, it randomly chooses the execution mode with probabilities ϵ_{DU} or ϵ_{SM} (below we explain the reason for managing two values of ϵ instead of just one). Otherwise, HybML tries to optimize either network or CPU usage, depending on which is the observed bottleneck under a given workload. HybML always first ensures that network is not saturated, because saturating a network always results in degradation of performance (see Section 5.4, [5] and [2]). To this end, HybML compares the values of moving averages, which store message sizes for either execution mode, and chooses a mode which corresponds to smaller messages. If, on the other hand, CPU is the limiting factor, HybML relies on moving medians, which store the duration of transaction execution and commit (see also the discussion in Section 6.6.3 for the reasons on using moving medians instead of moving averages in case of optimizing CPU usage). Since unlike SM transactions, DU transactions can abort, HybML stores additional moving averages and medians to account for aborted DU transactions. This way, by knowing abort rate (measured independently for each class and accounting separately for conflicts detected before and after the network communication phase), HybML can estimate the overall cost of executing and committing a DU transaction (in terms of both network traffic and execution time).

Note that the average cost of a single attempt to execute (and hopefully commit) a transaction in the DU mode is smaller compared to the cost of executing a transaction in the SM mode. It is because a DU transaction can abort due to a conflict and an SM transaction is guaranteed to commit. When a DU transaction aborts, no costly state update is performed and sometimes, if the conflict is detected before performing the broadcast operation, no resources are wasted on network communication. Therefore, in order to guarantee fair exploration, the probability with which the DU mode is chosen should be higher than the probability with which the SM mode is chosen. This observation led us to use ϵ_{DU} and ϵ_{SM} instead of a single value ϵ. Currently $\epsilon_{DU} = 0.01$ and $\epsilon_{SM} = 0.1$, which could be interpreted as follows: due to a higher resource cost of choosing the SM mode over the DU mode, the latter is chosen 10 times more frequently. As shown in Section 6.7, the system works very well with these values, but by using abort rate, these values can be easily set to reflect the *true* cost of an execution attempt.

There are few substantial differences between the definition of the original problem of multi-armed bandit problem and our case. Firstly, in the original problem the probability distributions of rewards in slot machines do not change and thus the strategy must account for all previous plays. HybML must be able to adjust to changing environment (e.g., workload) and thus it relies on moving medians. Most importantly, however, we treat choosing an optimal execution mode for any class independently, i.e., as a separate instance of the multi-armed bandit problem. In reality the decisions made by HybML for different classes of transactions are (indirectly) inter-dependent. It is because the reward returned after a transaction commits or aborts does not reflect solely the accuracy of the decision made by HybML, but it also entails the current load of the system. The load of the system naturally depends on all transactions running concur-

rently and thus indirectly on the decisions made by HybML for transactions of different classes. Note that if we were to reflect the inter-dependency between decisions made for different classes (in the form of a context as in the contextual multi-armed bandit problem [113]), the scheme would get extremely complicated and in practice it would never converge.

6.6.3 Implementation Details

Although the idea behind HybML seems simple, implementing it in a way that it works reliably was far from easy. It is mainly because of the characteristics of workloads we consider in conjunction with quirks of JVM that we had to deal with, provided that Paxos STM is written in Java.

The biggest challenge we faced was to accurately measure the duration of transaction execution. In particular, we were interested in obtaining faithful measurements on the time spent by the main thread of HTR on handling SM and DU transactions. When network is not saturated, the main thread becomes the bottleneck because it serializes execution of SM transactions with certification of DU transactions and is also responsible for applying transaction updates to the local state.[4] The CPU times we measure (using the `ThreadMXBean` interface) are in orders of microseconds, which means that we can expect a large error. The instability of measurements is further amplified by the way Java threads are handled by JVM. In JVM, Java threads do not correspond directly to the low-level threads of the operating system (OS) and thus the same low-level OS thread which, e.g., executes transactions, can be also responsible for performing other tasks for the JVM such as garbage collecting unused objects every once in a while.[5] As a result, we often observed measurements that were up to 3 orders of magnitude higher than the typical ones. As we were unable to obtain consistent averages using relatively small windows (necessary to quickly adopt to changing conditions), we resorted to moving medians, which are less sensitive to outliers.

6.7 Evaluation

In this section, we present the results of the empirical study of the HTR scheme. To this end we compare the performance of HTR using HybML with the performance of HTR running with the DU or SM oracles, which execute all updating requests in either the DU mode or the SM mode. As we explained in Section 6.6,

[4] Note that parallelising operations in this thread does not necessarily result in better performance. It is because even for large transactions the cost of transaction certification or applying updates is comparable to the time required for completing a memory barrier, action which is necessary in order to preserve consistency (in the pseudocode, the memory barrier happens prior acquiring the lock and once it is released, lines 63, 66, 88, 91; the core of Paxos STM uses no locks but relies on memory barriers triggered by accessing volatile variables).

[5] Our testing environment does not allow us to use low-level JNI code for enabling thread affinity in Java.

HybML optimizes the usage of network or CPU, depending on which is the current bottleneck. Since avoiding network saturation is relatively simple, because it entails choosing the execution mode which results in smaller messages being broadcast, we focus on the more challenging scenario in which the processing power of the CPUs is the limiting factor. Under this scenario, we have to consider a much wider set of variables such as transaction execution and commit times, contention levels, shared object access patterns, etc.

6.7.1 Software and Environment

We conducted tests using HTR-enabled Paxos STM, our fault-tolerant object based DTM system written in Java, which we already featured in Section 5.4. We run Paxos STM in a cluster of 20 nodes connected via 16Gb Ethernet over Infiniband. Each node had 28-core Intel E5-2697 v3 2.60GHz processor 64GB RAM and was running Scientific Linux CERN 6.7 with Java HotSpot 1.8.0.

6.7.2 Benchmarks

In order to test the HTR scheme, we extended the hashtable microbenchmark, which we used in Section 5.4. The benchmark features a hashtable of size h, storing pairs of key and value accessed using the *get*, *put*, and *remove* operations. A run of this benchmark consists of a load of requests (transactions) which are issued to the hashtable, each consisting of a series of *get* operations on a randomly chosen keys and then a series of update operations (either *put* or *remove*). Initially, the hashtable is prepopulated with $\frac{h}{2}$ random integer values from a defined range, thus giving the saturation of 50%. This saturation level is always preserved: if a randomly chosen key points at an empty element, a new value is inserted; otherwise, the element is removed.

In the current implementation of the benchmark, we can adjust several parameters for each class of transactions independently and at run-time. The parameters include, among others, the number of read and write operations, the subrange of the hashmap from which the keys are chosen, access pattern (random keys or a continuous range of keys) and the duration of the additional sleep operation, which is invoked during transaction execution in order to simulate a computation heavy workload. By varying these parameters and the ratio of concurrently executing transactions of different classes, we can generate diverse workloads, which differ in CPU and network usage and are characterised by changing contention levels.

We consider three test scenarios: *Simple*, *Complex* and *Complex-Live*. We use the first two scenarios to evaluate the throughput (measured in requests per second) and scalability of our system. The latter scenario is essentially the Complex scenario, whose parameters are changed several times throughout the test. We use the Complex-Live scenario to demonstrate the ability of HybML to adjust to changing conditions at run-time (we show the throughput of HTR in the function of time). For each scenario we define from 2 up to 11 classes of transactions,

		Scenario	Parameter	T_0	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	T_9	T_{10}
		Simple	probability	90	10									
			reads	2500	300									
			updates	0	5									
			range	600k	600k									
		Complex	probability	90	1	1	1	1	1	1	1	1	1	1
			reads	2500	200	200	200	200	200	200	200	200	200	200
			updates	0	5	5	5	5	5	5	5	5	5	5
			range	10.24M	5.12M	2.5M	1.28M	640k	320k	160k	80k	40k	20k	10k
			offset	0	Distinct part of Hashtable for each T_1–T_{10}									
Complex-Live	a	(0-200 s)		As in Complex										
	b	(200-400 s)	reads	2500	400 (2x as in Complex)									
			updates	0	10 (2x as in Complex)									
	c	(400-600 s)	sleep	0	Extra 0.1ms sleep in every transaction									
	d	(600-800 s)	range	10.24M	Half the range from Complex for each T_1–T_{10}									
	e	(800-1000 s)		As in Complex										

Figure 6.1: Benchmark parameters for different test scenarios. In the *Complex-Live b-d* scenarios we change some parameters compared to the *Complex* scenario.

whose parameters are summarized in Figure 6.1). We have chosen the parameters so that one can observe the strong and weak aspects of HTR running with either the DU or SM oracle. This way we can demonstrate HybML's ability to adapt to different conditions. In order to utilize the processing power of the system across different cluster configurations, we increase the number of requests concurrently submitted to the system with the increasing number of replicas.

6.7.3 Benchmark Results

In Figures 6.2, 6.3 and 6.4 we present the test results of HTR. Below we discuss the test results in detail.

The Simple Scenario

In this scenario, for which the test results are given in Figure 6.2a, there are only two classes of transactions (T_0 and T_1), which operate on a hashmap of size h=600k. T_0 transactions (i.e., transactions, which belong to the T_0 class) are read-only and execute 2500 read operations in each run. T_1 transactions perform 300 read and 5 updating operations. The ratio between transactions T_0 and T_1 is 90:10.

In this scenario the throughput of HTR running with the SM oracle remains constant across all cluster configurations. It is because the execution of all T_1 transactions needs to be serialized in the main thread of HTR, which quickly becomes the bottleneck. The throughput of 150k tps (transactions per second), achieved already for 4 nodes, indicates the limit on the number of transactions that the system can handle in any given moment. Also, the transactions executed in the SM mode never abort, thus the abort rate is zero.

In the case of HTR running with the DU oracle, with the increasing number of replicas, the throughput first increases, then, after reaching maximum for 5 replicas, slowly diminishes. The initial scaling of performance can be attributed to increasing processing power that comes with a higher number of replicas taking part in the computation. In the 5 node configuration, the peak performance of HTR running with the DU oracle is achieved. As in case of the SM oracle, the main thread of HTR becomes saturated and cannot process any more messages which carry state updates. Naturally, with the increasing number of concurrently executed transactions, one can observe the raising number of transactions aborted due to conflicts. Therefore, adding more replicas results in diminishing performance. The abort rate of almost 40% in the 20 node cluster configuration means that every updating transaction is on average executed 7.5 times before it eventually commits.

The HybML oracle takes advantage of the scaling capabilities of DUR for smaller cluster configurations–the performance yielded by HybML is on par with the performance of the DU oracle, because HybML always chooses the DU mode for all updating transactions. From 10 nodes upwards, the performance of the DU oracle drops below the performance of the SM oracle. The 10-12 node configurations are problematic for the HybML oracle, because the relative difference in performance of DU and SM modes is modest, and thus it is difficult for the oracle to make an optimal decision. This is why we can observe that HybML still chooses the DU mode for the T_1 transactions, instead of SM. Then, however, the differences start to increase, and HybML begins to favour the SM mode over the DU mode for the T_1 transactions (see the third diagram in Figure 6.2a, which shows the dominant execution mode for each class in HybML; different colours signify the relative ratio between executions in the DU and the SM modes). Eventually (from the 15 node configuration upwards), HybML always chooses the SM mode thus yielding the same performance as the SM oracle.

The Complex Scenario

In the Complex scenario, for which the evaluation results are given in Figure 6.2b, there is only one class of read-only transactions (T_0, the same as in the Simple scenario) and 10 classes of updating transactions (T_1-T_{10}). Each updating transaction has the same likelihood of being chosen and each performs 200 read and 5 update operations. However, for each class we assign a disjoint subrange of the hashmap. It means that each (updating) transaction from a given class can only conflict with transactions, which belong to the same class. Because the sizes of subranges are different for every class of transactions, the contention levels for each class will greatly differ: they would be lowest for transaction T_1 (whose range encompasses 5.12M keys) and highest for T_{10} (whose range encompasses just 10k keys). The exact values of the subranges are given in Figure 6.1).

As in case of the Simple scenario, in the Complex scenario the DU oracle first yields better performance than the SM oracle. The highest throughput of about 200k tps is achieved by the DU oracle for the 8 nodes configuration, while

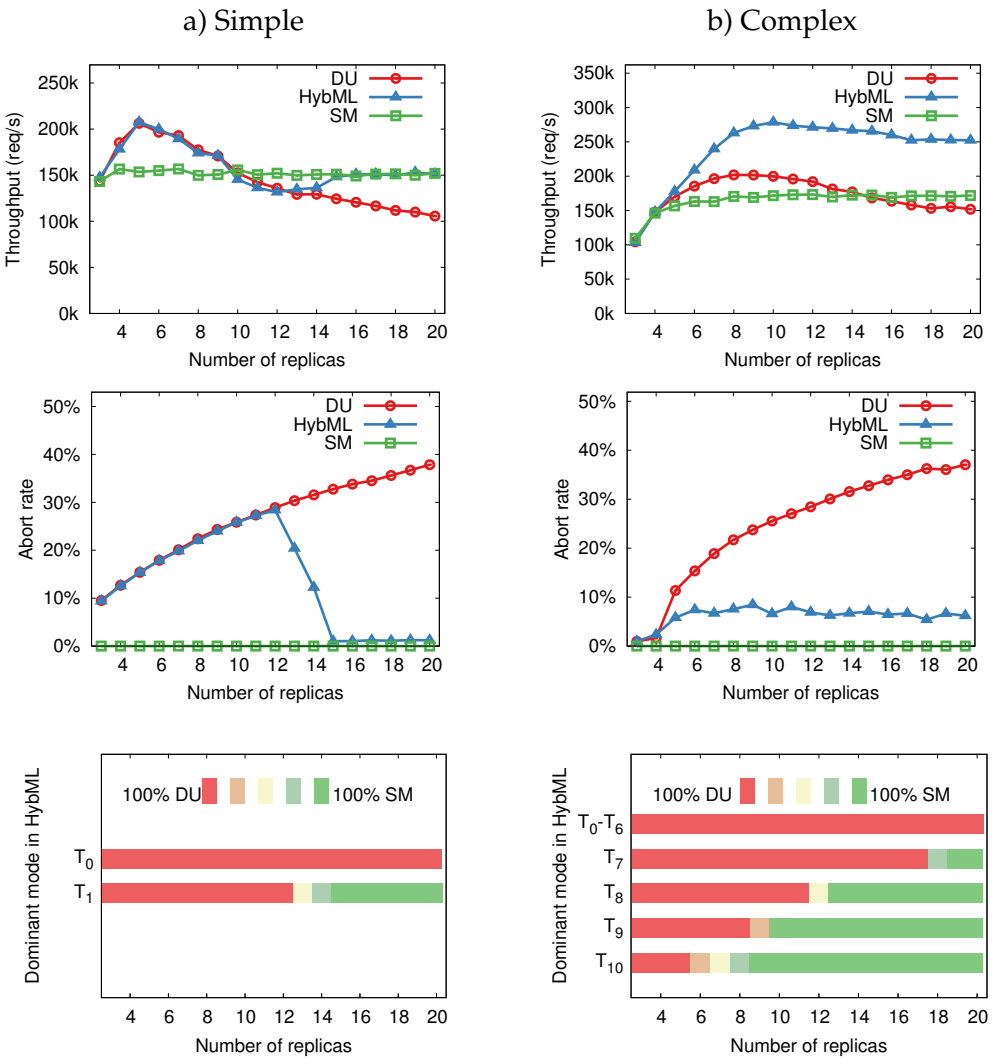


Figure 6.2: The performance of HTR with different oracles across various cluster configurations.

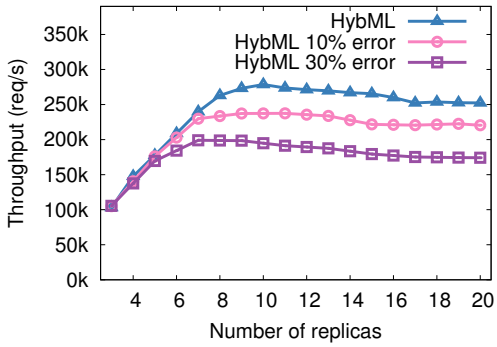


Figure 6.3: The performance of HybML in the Complex scenario, when HybML is provided with an inaccurate classification of transactions.

the performance of the SM oracle levels at about 160k tps. Note that for the 3-5 nodes configuration, the SM oracle performance scales. It is because in the Complex scenario the updating transactions are shorter than in the Simple scenario. Therefore, in order to saturate the main thread of HTR, more concurrently submitted requests are needed (the thread becomes saturated in the 5 nodes configuration).

For larger cluster configurations, the performance of the DU oracle degrades due to the rising number of conflicts. As a result, the performance of the DU oracle drops below the performance of the SM oracle for the 15 nodes configuration.

Note that the abort rate levels, which we can observe for the DU oracle, are very similar to the ones we saw in the Simple scenario. However, there are significant differences between the abort rates measured for each transaction class independently. For instance, for the 20 node configuration, the abort rate is about 4% for T_1 and over 99% for T_{10} (in the latter case a transaction is on average aborted 150 times before it eventually commits).

In this scenario, HybML demonstrates its ability to adjust to the workload and achieves performance that is up to 40% higher than the DU oracle's and up to 75% higher than the SM oracle's. This impressive improvement in performance justifies our ML-based approach. The plot show that HybML maintains a relatively low abort rate of about 7-8% across all cluster configurations. One can see that the higher number of concurrently executed transactions, the higher percentage of transactions is executed by HybML in the SM mode thus keeping contention levels low. Naturally, HybML chooses the SM mode first for the T_{10} transactions, for which the contention level is the highest. Then, gradually, HybML chooses the SM mode also for transactions, which belong to classes T_9, T_8 and also T_7. For other transactions the cost of execution in the DU mode is still lower than the cost of execution in the SM mode, and thus HybML always chooses for these transactions the DU mode, regardless of the cluster configuration.

In order to check the consequences of providing HybML with an inaccurate classification of transactions, we purposefully marked some percentage of updating transactions with a random number corresponding to some other class. The results of this experiment are given in Figure 6.3. Naturally, as a baseline we used the performance of HybML from the previous test. Understandably, with 10% or 30% of incorrectly marked transactions (*HybML 10% error* and *HybML 30% error* in the Figure), HybML still performs better than either the DU or SM oracles but not as fast as previously (the performance for the 10% and 30% mistake scenarios peaked at 240k tps and 200k tps, respectively). This result indicates that HybML gracefully handles even quite significant errors in the classification provided by the programmer.

The Complex-Live scenario

In the Complex-Live scenario we demonstrate the ability of HybML to adapt in real-time to changing conditions. To this end we consider the system consisting

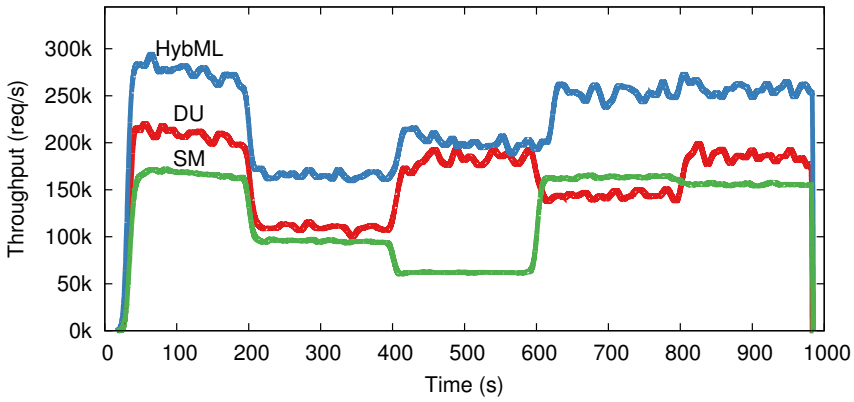


Figure 6.4: The Complex-Live scenario: the performance of HTR in the function of time.

of 9 nodes and a workload identical with the one from the Complex scenario, which we then change several times during a 1000 seconds run. A plot showing the throughput of HTR with different oracles is given in Figure 6.4. One can see that in all cases the HybML oracle gives better performance than either the DU or SM oracles and almost instantly reacts to changes of the workload.

During the first 200 seconds the observed performance matches the results from the Complex scenario. The throughput fluctuates a little bit because of the garbage collector, which periodically removes unused objects from memory.[6] Towards the 200th second the throughput slightly decreases as garbage collecting becomes regular.

In the 200th second we change the parameters of the benchmark, so now each updating transaction performs twice the number of read and updating operations as before (see *Complex-Live b* scenario in Figure 6.1). The performance of the system decreases, because such a change results in longer transaction execution times and larger messages. HybML performs 70% better than the SM oracle and over 50% better than the DU oracle.

Between the 400th and 600th second, the benchmark parameters are the same as in the Complex scenario but the execution of each updating transaction is prolonged with 0.1 ms sleep thus simulating a computation heavy workload. Naturally, such workload is troublesome for the SM oracle, because all updating transactions are executed sequentially. The additional 0.1 ms sleep is handled well by the system when transactions are executed in the DU mode, because transaction execute in parallel. HybML achieves about 15% better performance than the DU oracle, as it allows about 75% of the T_{10} transactions (which are

[6]Executing a transaction in the DU mode results in more noticeable overhead due to the garbage collector. Hence, one can observe bigger fluctuations in throughput for the DU and HybML oracles compared to the SM oracle.

most likely to be aborted due to conflicts) to be executed in the SM mode, thus reducing the abort rate and saving on transaction reexecutions.

The change to the benchmark parameters in the 600th second involves reducing by half the size of the hashmap subrange for each class. This way the updating transactions, which perform the same number of read and updating operations as in the Complex scenario, are much more likely to abort due to conflicts. This change is reflected by a steep decrease in performance of the DU oracle. On the other hand, the performance of the SM oracle is almost the same as in the Complex scenario, because execution of all updating transactions takes the same amount of work as in the Complex scenario. Smaller subranges impact favourably only data locality. This phenomenon is reflected by a slightly better performance. Stunningly, the performance of HybML is almost the same as in the Complex scenario: HybML automatically started to execute a higher percentage of updating transactions in the SM mode thus keeping the abort rate low. The achieved throughput is over 65% better than with the DU oracle and over 50% than with the SM oracle.

The last 200 seconds of the test is performed with the parameters from the Complex scenario. HybML quickly relearns the workload and starts to perform as in the first 200s of the test.

6.7.4 Evaluation Summary

We tested the HTR scheme with three oracles: DU, SM and HybML. The DU and SM oracles execute all updating transactions either in the DU or SM mode. Therefore, a system using these oracles resembles an implementation of DUR (see Section 5.3) and an implementation of the optimized version of SMR, which allows read-only requests to be executed in parallel (LSMR, see Section 5.2). Unlike the DU and SM oracles, the HybML oracle mixes transaction execution modes to achieve better performance and scalability (as evidenced by Figure 6.2). Our tests show that HybML provides performance that is at least as good as with either DU or SM (when the difference in performance between the system running with the DU or SM oracles is large enough) and often exceeds it by up to 50-70% across a wide range of cluster configurations and types of workload. HybML avoids the pitfalls of either SMR and DUR and handles very well the workloads that are notoriously problematic for either replication scheme (i.e., computation intensive workloads in SMR and workloads characterised by high contention levels in DUR).

We also demonstrated HybML's ability to quickly adjust to changing conditions. The automatic adaptation to a new workload type happens smoothly and almost instantly, without even temporary degradation of performance, compared to the performance under stable conditions (see Figure 6.4). All the benefits of the HTR scheme running with the HybML oracle require only minimal input from the programmer, which involves providing a rough classification of transactions submitted to the system. Slight inaccuracies in the classification do not heavily impact the performance achieved by HTR running with HybML.

7
Conclusions

In this dissertation we revisited State Machine Replication and Deferred Update Replication, two well established schemes for service and data replication. Our findings uncovered the differences in semantics offered by the schemes as well as in their performance under diverse workloads. However, our research not only broadened the understanding of SMR and DUR, but it also led us to several contributions in the field of distributed replication.

The main goal of this dissertation, as stated in Section 1.2, was to propose a new replication scheme that simultaneously offers rich transactional semantics, strong consistency guarantees and provides high performance across a wide range of workloads, thus bringing the best features of both SMR and DUR. We believe that the goal has been achieved. In support of this claim we summarize below the contributions of this dissertation.

Firstly, in Chapter 4, we defined the $\diamond$-opacity and $\diamond$-linearizability families of correctness properties, which can be used to formalize the guarantees offered by replicated transactional systems and replicated systems modelled as shared objects. Our properties relax in a systematic way the real-time order requirement of opacity and linearizability, two well established correctness criteria used in the context of transactional memory systems and shared objects. The formal result on the relationship between members of the $\diamond$-opacity and $\diamond$-linearizability families allows us to compare the guarantees provided by the replicated systems which do or do not provide transactional support. In particular, we showed that when in a replicated $\diamond$-opaque transactional system transactions are hidden from clients, the system is $\diamond$-linearizable. Also our result directly compares opacity and linearizability in their original definitions.

Secondly, in Chapter 5, we directly compared the guarantees provided by SMR and DUR using our new criteria. We also demonstrated the consequences of introducing an important optimization of SMR which involves executing read-only requests without any interprocess-synchronization.

Our comparison of semantics of SMR and DUR was complementary to the experimental evaluation of both schemes (presented also in Chapter 5), which

showed that performance-wise neither scheme is superior in general. This result is quite surprising, provided that only DUR is potentially scalable. Of course, this result is valid, since we consider a wide spectrum of possible workload types.

Finally, in Chapter 6, we combined SMR and DUR into a single replication scheme called Hybrid Transactional Replication. This way we brought together the best features of SMR and DUR. HTR offers powerful transactional semantics similarly to DUR. Also HTR neither prohibits nondeterministic nor irrevocable operations (in requests) as does SMR and DUR, respectively. HTR can dynamically adopt to a changing workload by executing requests either in a way that resembles the execution of a transaction in DUR or a request in SMR. The performance of HTR, which, as we show, is at least as good as SMR's or DUR's can be fine-tuned by specifying an application-specific policy. The policies are not necessarily static. We showed that they can rely on a machine learning mechanism to enable automatic adaptation to changing conditions. Our ML-based approach quickly adapts to changes in the workload and allows HTR to achieve significant performance advantage over SMR and DUR. As we formally proved, HTR offers similar guarantees to those of DUR.

In the future, we plan to further investigate transactional replication schemes. We would like to research the eventually consistent ones, because strong consistency is not a crucial requirement in many applications. Moreover, depending on the considered notion of *eventual consistency*, the eventually consistent systems can provide different guarantees, which are neither well understood, nor adequately formalized.

Streszczenie

Wraz z rosnącą popularnością przetwarzania w chmurze (ang. *cloud computing*), gdzie usługi (ang. *services*) działające w chmurze muszą obsługiwać ogromną liczbę użytkowników w tym samym czasie, nastąpił gwałtowny wzrost zainteresowania różnymi podejściami do *rozproszonej replikacji* (ang. *distributed replication*). Rozproszona replikacja poprawia dostępność (ang. *availability*) i niezawodność (ang. *reliability*) usługi poprzez przechowywanie danych bliżej klientów i przetwarzaniu wielu żądań klientów równolegle. W tym podejściu usługa uruchomiona jest na wielu połączonych ze sobą serwerach zwanych *replikami* (ang. *replicas*). Działania replik są koordynowane, tak by każda z replik utrzymywała spójny obraz stanu systemu rozproszonego, pomimo awarii łączy komunikacyjnych lub poszczególnych replik. Każda replika ma dostęp do lokalnej pamięci (ang. *local memory*), zaś synchronizacja replik odbywa się poprzez *pamięć współdzieloną* lub *współdzielony magazyn danych* (ang. *distributed memory*, *distributed storage*), tj. wysokopoziomową abstrakcję oferującą spójny dostęp do replikowanych danych.

Kontekst

W tej dysertacji rozważamy szczególny rodzaj replikacji zwany *replikacją transakcyjną* (ang. *transactional replication*), w której każde żądanie zgłoszone przez klienta wykonywane jest jako *atomowa transakcja* (ang. *atomic transaction*). Oznacza to, ze sekwencja operacji zdefiniowanych w żądaniu (transakcji) będzie wykonana przez zreplikowany system w semantyce *wszystko-albo-nic* (ang. *all-or-nothing semantics*). Ponadto, system odpowiedzialny jest za wykrywanie i rozwiązywanie konfliktów pomiędzy współbieżnie wykonującymi się transakcjami, które wykonują operacje na tych samych współdzielonych danych. Kod transakcji może korzystać z dodatkowych konstrukcji składniowych takich jak *rollback* czy *retry*, które pozwalają programiście na lepszą kontrolę przepływu sterowania w transakcji (*rollback* wycofuje wszystkie zmiany dokonane przez

transakcję, *retry* wycofuje transakcję i ponawia jej wykonanie od razu lub gdy spełnione są określone warunki dodatkowo zdefiniowane przez programistę).

Replikację transakcyjną usługi najłatwiej osiągnąć poprzez zbudowanie tejże usługi w oparciu o *(rozproszoną) transakcyjną platformę programistyczną* (ang. *distributed transactional programming framework*). Dzięki takiemu podejściu, programista nie musi ręcznie synchronizować współbieżnych dostępów do danych współdzielonych w oparciu o np. mechanizm *zamków* (ang. *locks*) czy *monitorów* (ang. *monitors*), których użycie stwarza problemy nawet doświadczonym programistom. Transakcyjna platforma programistyczna ukrywa całą tę złożoność przed programistą i pozwala mu wnioskować o przepływie sterowania w usłudze tak, jak gdyby wszystkie transakcje wykonywane byłyby sekwencyjnie na jednym, niezawodnym serwerze. Dzięki zapewnieniu wsparcia dla przetwarzania transakcyjnego można więc znacznie ułatwić projektowanie i rozwijanie wysoko dostępnych usług.

Iluzja jaką daje transakcyjna platforma programistyczna, polegająca na tym, że z punktu widzenia programisty transakcje wykonują się tak, jakby (logicznie) wykonywały się sekwencyjnie, wiąże się z silnymi gwarancjami, jakie ta platforma musi dawać. Gwarancje te, typowe dla tradycyjnych baz danych SQL, oryginalnie zostały sformalizowane w postaci własności poprawności (ang. *correctness property*) zwanej *(strict) serializability* (pl. *(ścisła) uszeregowalność*). Często też podejście do budowy systemów oferujących wspomniane gwarancje określane jest mianem *podejścia silnie spójnego* (ang. *the strongly consistent approach*) [14].

Uszeregowalność zazwyczaj osiągana była poprzez wykonanie wszystkich *modyfikujących transakcji* (ang. *updating transactions*) przez wyróżnioną replikę zwaną *mistrzem* (ang. *master*) i propagowanie modyfikacji wytworzonych wskutek wykonania transakcji do reszty replik zwanych *służącymi* (ang. *slaves*). Zazwyczaj każda z replik przechowywała pełną kopię bazy danych. *Transakcje niemodyfikujące* (ang. *read-only transactions*), które nie zmieniają stanu usługi, były wykonywane współbieżnie na replikach-sługach. Awaria repliki-mistrza wymagała zawieszenia przetwarzania do momentu wyłonienia nowej repliki-mistrza spośród działających replik-służących. To podejście do replikacji zostało usprawnione poprzez umożliwienie wykonania modyfikujących transakcji współbieżnie na replice-mistrzu, a później także i na różnych replikach równolegle. Wykonanie każdej transakcji koordynowane było przez jedną z replik, dzięki czemu możliwe było zapewnienie poprawności wykonania transakcji i jej późniejszego zatwierdzenia nawet w przypadku wystąpienia awarii serwerów czy też łączy komunikacyjnych. Poprzez opóźnianie wykonania niektórych operacji, bądź też wycofanie i ponowne wykonanie niektórych transakcji, protokół spójności (implementowany przez replikowany system) nie dopuszczał do wystąpienia niespójności przy współbieżnym dostępnie do współdzielonych danych. To podejście zwane jest zazwyczaj *replikacją z opóźnioną aktualizacją* (ang. *Deferred Update Replication, DUR*) [15].

Alternatywnie, baza danych mogła być replikowana przy użyciu podejścia zwanego *replikacją maszyny stanowej* (ang. *State Machine Replication, SMR*) [16]

[17] [18]. W tym podejściu identyczne kopie bazy danych uruchomione były na kilku maszynach i każda replika wykonywała wszystkie żądania przesyłane przez klientów. W ten sposób każda z replik modyfikowała swój stan w ten sam sposób, czyniąc awarie serwerów niewidocznymi dla klientów. Oczywiście wszystkie żądania musiały być deterministyczne, a także musiały być dostarczane w tej samej kolejności do wszystkich replik. W przeciwnym wypadku stan replik rozbiegłby się z czasem. W porównaniu do DUR, SMR jest dużo prostszym podejściem. Poważną wadą SMR jest jednak to, że nie pozwala na współbieżne wykonanie żądań, a więc nie może się skalować (wydajność systemu nie poprawia się wraz ze zwiększaniem liczby replik, procesorów lub rdzeni procesora). Jakkolwiek SMR nie oferuje wsparcia dla przetwarzania transakcyjnego, wykonanie pojedynczego żądania w SMR można traktować jak wykonanie prostej transakcji, która ma gwarancję zatwierdzenia.

Na początku tego milenium, pojawiły się nowe systemy baz danych, zwane bazami lub magazynami danych *NoSQL*, które powoli zaczęły zastępować tradycyjne bazy danych SQL w niektórych zastosowaniach. Magazyny NoSQL cechuje znacznie wyższa wydajność w porównaniu z tradycyjnymi bazami danych SQL, ale odbywa się to kosztem osłabionych gwarancji spójności, co utrudnia programowanie. Dużo wyższa wydajność i skalowalność tego typu systemów pozwoliła globalnie dostępnym usługom działającym w Internecie na radzenie sobie ze stale rosnącym ruchem (patrz np. [19] [20] [21] [22]).

Magazyny NoSQL zazwyczaj nie oferują wsparcia dla przetwarzania transakcyjnego i są tylko *słabo (ostatecznie) spójne* (ang. *weakly (eventually) consistent*). Oznacza to, ze klienci od czasu do czasu mogą zauważyć, że usługa działa niezgodnie z przewidzianą przez usługodawcę logiką. Zakres obserwowalnych anomalii jest szeroki: od zupełnie niegroźnych (np. kolejność wpisów na stronie portalu społecznościowego jest nieco inna niż chwilę wcześniej) do tych całkiem poważnych (klient odbiera potwierdzenie rezerwacji biletu lotniczego, po czym po pewnym czasie okazuje się, że jednak miejsce w samolocie jest już zajęte i konieczna jest rezerwacja innego lotu). Obserwowane niepożądane zachowanie usługi (niezgodne z domyślną logiką usługi) może być dalej amplifikowane przez awarie serwerów i łączy komunikacyjnych. W efekcie zapewnienie poprawnej realizacji żądań klientów wymaga od programistów więcej uwagi i implementacji dodatkowej obsługi wszystkich szczególnych przypadków. Brak jasno określonej semantyki jaką oferują magazyny NoSQL i wysoki stopień niedeterminizmu, który jest charakterystyczny dla środowiska rozproszonego, przekłada się na wysokie koszty tworzenia i utrzymywania usług zbudowanych w oparciu o takie systemy.

Jasnym jest zatem, że brak silnych gwarancji spójności oraz wsparcia dla przetwarzania transakcyjnego jest problematyczny nie tylko dla klientów, ale także i programistów. Kierujący firmami, do których należą duże usługi działające w Internecie, cały czas poszukują nowych sposobów optymalizacji kosztów działania usługi oraz poprawy jakości obsługi użytkowników (ang. *user experience*). Stąd też w ostatnich latach można obserwować wzrost zainteresowania silnie spójnymi rozwiązaniami (patrz np. [24] [25]). W tej dysertacji skupiamy

się właśnie na tego typu systemach.

W naturalny sposób silnie spójne schematy replikacji danych i usług, które badamy, czerpią z rozwiązań znanych z systemów baz danych SQL, bowiem one także są systemami rozproszonymi oferującymi semantykę transakcyjną. Drugim źródłem inspiracji są badania nad *pamięcią transakcyjną* (ang. *Transactional Memory, TM*) [26], tj. podejściem, które wykorzystuje ideę transakcji znaną z systemów bazodanowych jako mechanizm kontroli współbieżności w środowisku lokalnym (na poziomie języka programowania). Niestety żadne z istniejących rozwiązań nie może być bezpośrednio wykorzystane do rozważanych przez nas celów. Jedną z głównych przyczyn, dla których jest to niemożliwe, jest charakterystyka obciążeń (ang. *workloads*) typowa dla współczesnych replikowanych usług. Na przykład, średni czas wykonania transakcji w rozważanych przez nas systemach jest często rząd wielkości krótszy niż w w systemach bazodanowych i rząd wielkości dłuższy niż w systemach pamięci transakcyjnej [27]. Także logika współczesnych replikowanych usług (lub fragment tejże logiki) często nie może być łatwo wyrażona przy pomocy SQL, tj. języka tradycyjnych baz danych. Często lepszym podejściem jest wykorzystanie interfejsu przypominającego interfejs systemów pamięci transakcyjnych, gdzie transakcja traktowana jest jako wysokopoziomowa konstrukcja języka programowania, w ramach której można definiować dowolny kod, a w szczególności taki, który wywołuje złożone metody czy operacje na obiektach współdzielonych (patrz np. [28] [29]). W naszej pracy skupiamy się na takim właśnie podejściu, gdyż jest ono bardziej ogólne.

Z uwagi na różnice w oferowanych interfejsach, własności poprawności używane w kontekście systemów baz danych (takie jak *recoverability*, *avoiding cascading aborts*, *strictness* [30] czy warianty wspomnianej już własności uszeregowalności) nie mogą być stosowane do formalizacji gwarancji oferowanych przez współczesne silnie spójne schematy replikacji. Niestety nie mają zastosowania w naszej pracy także własności znane ze świata systemów pamięci transakcyjnych (takie jak różne warianty *opacity* [31] [32] [33], *TMS1* [34] czy *TMS2* [34]). Jest tak, ponieważ własności te zostały zaprojektowane w kontekście lokalnego środowiska, gdzie pewne gwarancje dotyczące uszeregowania transakcji (takie jak uwzględnienie ograniczeń *czasu rzeczywistego*, ang. *real-time*) są naturalne i relatywnie niedrogie do zapewnienia. Natomiast w środowisku rozproszonym wspomniane gwarancje muszą często być osłabione w odniesieniu do niektórych typów żądań (transakcji), np. żądań niemodyfikujących (ang. *read-only*). Dlatego też formalizacja semantyki replikowanych systemów, które rozważane są w tej dysertacji, wymaga zdefiniowania nowych własności poprawności.

Można zatem stwierdzić, że pomimo na powrót rosnącej popularności silnie spójnych schematów replikacji, podstawy teoretyczne tego typu rozwiązań nie są jeszcze dobrze poznane i potrzeba więcej badań w tej dziedzinie. Na przykład brak w istniejącej literaturze szczegółowego porównania podstawowych schematów replikacji takich jak SMR i DUR, zarówno pod kątem semantyki jak i wydajności. Jest to zaskakujące, ponieważ zarówno SMR jak i DUR stanowią podstawę wielu innych, bardziej skomplikowanych schematów replikacji (patrz

np. [35] [36] [37] [38] [39] [40]). Dopiero gdy odkryjemy różnice między tymi podejściami i jasno określimy ich silne i słabe strony, będziemy mogli zaproponować nowe schematy replikacji, które będą odpowiadać aktualnym potrzebom oraz w pełni wykorzystywać możliwości współczesnych, wysoce równoległych architektur systemów komputerowych.

Cele i kontrybucje

Biorąc pod uwagę powyższe motywacje, w następujący sposób formułujemy główną tezę dysertacji:

> *Jest możliwe stworzenie schematu replikacji usług i danych, który oferuje bogatą semantykę transakcyjną, daje silne gwarancje spójności oraz cechuje go wysoka wydajność dla różnych typów obciążeń.*

Poniżej krótko podsumowujemy kontrybucje ujęte w dysertacji:

1. **Nowe własności poprawności dla silnie spójnych replikowanych systemów**. Definiujemy $\diamond$-*opacity* i $\diamond$-*linearizability*, dwie rodziny własności poprawności zaprojektowane dla silnie spójnych replikowanych systemów. Nasze własności wywodzą się z *opacity* (pl. *nieprzezroczystości*) i *linearizability* (pl. *liniowości*)–dwóch dobrze znanych własności poprawności zdefiniowanych dla systemów transakcyjnych i systemów modelowanych jako obiekty współdzielone [31] [41]. Dzięki zaproponowanym nowym własnościom, możemy sformalizować gwarancje oferowane przez różne schematy replikacji, które oferują (lub nie) semantykę transakcyjną oraz implementują różnego rodzaju optymalizacje, takie jak np. wykonanie żądań niemodyfikujących przez pojedynczą replikę, bez dodatkowej synchronizacji między replikami. Dowodzimy, że wszystkie własności z rodzin $\diamond$-opacity i $\diamond$-linearizability są własnościami bezpieczeństwa (tzn. są niepuste, prefiksowo-domknięte i domknięte, ang. *non-empty*, *prefix-closed*, *limit-closed*). Określamy również formalny związek pomiędzy zaproponowanymi rodzinami własności. Pokazujemy, że gdy żądania są wykonane w systemie gwarantującym $\diamond$-opacity i transakcje są niewidoczne dla klientów (tzn. klienci nie widzą pośrednich wyników wykonania transakcji i powiadamiani są o wyniku przetwarzania tylko przy zatwierdzaniu lub wycofywaniu transakcji), wtedy system gwarantuje $\diamond$-linearizability. W szczególności pokazujemy związek pomiędzy opacity i linearizability w ich oryginalnych definicjach (wg. naszej najlepszej wiedzy, jest to pierwszy rezultat tego typu).
2. **Szczegółowe porównanie SMR i DUR**. Porównujemy SMR i DUR zarówno pod względem oferowanej semantyki jak i wydajności. Formalnie dowodzimy poprawność obu podejść i pokazujemy, że SMR gwarantuje *real-time linearizability* (własność z rodziny $\diamond$-linearizability), podczas gdy

DUR gwarantuje *update-real-time opacity* (własność z rodziny $\diamond$-opacity). Dowodzimy także, że DUR gwarantuje *update-real-time linearizability*, gdy transakcje są ukryte przed klientami. Tym samym pokazujemy, że gwarancje oferowane przez DUR są ściśle słabsze niż gwarancje oferowane przez SMR. Rozważamy również *SMR with Locks (LSMR)*, tj. SMR implementujące optymalizację polegającą na tym, że niemodyfikujące żądania są wykonywane tylko przez pojedynczą replikę. Określamy dokładnie jakie skutki dla oferowanych gwarancji ma wprowadzenie tej optymalizacji i formalnie pokazujemy, że LSMR oferuje gwarancje ściśle słabsze niż LSMR, ale ściśle silniejsze niż DUR. Wyniki przeprowadzonej przez nas ewaluacji eksperymentalnej, pokazują silne i słabe strony SMR i DUR w przypadku różnego rodzaju obciążeń. Głównym wnioskiem płynącym z porównania wydajności obu podejść jest to, że żadne podejście nie jest ściśle lepsze od drugiego w ogólnym przypadku. Ten rezultat może zaskakiwać, ponieważ jedynie w przypadku DUR wydajność może potencjalnie rosnąć wraz ze zwiększającą się liczbą replik biorących udział w przetwarzaniu (SMR wykonuje wszystkie żądania sekwencyjnie, natomiast LSMR, tj. zoptymalizowany wariant SMR, pozwala na równoległe przetwarzanie żądań tylko w przypadku żądań niemodyfikujących).

3. **Nowy, silnie spójny schemat replikacji transakcyjnej**. Proponujemy nowy schemat replikacji zwany *hybrydową replikacją transakcyjną* (ang. *Hybrid Transactional Replication, HTR*). HTR łączy SMR i DUR dla lepszej wydajności, skalowalności i bogatszej semantyki. Formalnie dowodzimy, że HTR oferuje gwarancje podobne do DUR. Jak pokazujemy w testach ewaluacyjnych, HTR działa dobrze przy różnego rodzaju obciążeniach. W szczególności, HTR dobrze radzi sobie z obciążeniami, o których wiadomo, że są problematyczne dla SMR czy DUR (np. obciążenia charakteryzujące się długimi czasami wykonań żądań w przypadku SMR i obciążenia, w których występuje wysokie współzawodnictwo w dostępie do tych samych danych w przypadku DUR). W niektórych przypadkach, HTR pozwala na osiągnięcie nawet 50% lepszej wydajności niż w przypadku uruchomienia HTR symulującego działanie DUR lub LSMR (wszystkie żądania modyfikujące są wykonywane w sposób, który przypomina wykonanie transakcji w DUR lub żądań w LSMR). HTR pozwala na definiowanie polityk, dzięki którym możliwe jest dostosowanie działania systemu do spodziewanego obciążenia i tym samym adaptacja do zmieniających się warunków brzegowych. Przedstawiamy szereg technik przydatnych przy tworzeniu polityk, a także proponujemy politykę wykorzystującą mechanizmy uczenia maszynowego, która ułatwia pracę programiście i pozwala na automatyczne dostosowywanie działania systemu do zmieniających się warunków.

Powyższym kontrybucjom, które też opisujemy nieco bardziej szczegółowo w dalszej części streszczenia, poświęcone są Rozdziały 4, 5 i 6 dysertacji. W pozostałych rozdziałach dysertacji przedstawiamy kontekst i tezę dysertacji (Roz-

dział 1), przegląd literatury związanej z tematyką pracy (Rozdział 2), definiujemy model rozważanych systemów (Rozdział 3). Podsumowujemy dysertację w Rozdziale 7.

Nowe własności poprawności dla systemów zreplikowanych

Gwarancja wykonania żądań klientów z uwzględnieniem ograniczeń *czasu rzeczywistego* (ang. *real-time*) jest często pożądaną cechą systemów rozproszonych. Gwarancja czasu rzeczywistego oznacza, że gdy wykonanie jednego żądania kończy się zanim rozpocznie się wykonanie innego żądania (porównując np. czas zegarowy obu zdarzeń), efekty wykonania pierwszego żądania są zawsze widoczne dla wykonania drugiego żądania. Zapewnienie takiej (intuicyjnej) gwarancji w środowisku rozproszonym nie jest proste i często okazuje się bardzo kosztowne. Jest tak dlatego, ponieważ uwzględnienie ograniczeń czasu rzeczywistego wymaga synchronizacji między procesami (serwerami) w przypadku wykonania każdego żądania. Dlatego też replikowane usługi często osłabiają ograniczenie czasu rzeczywistego w przypadku niektórych typów żądań, na przykład żądań *niemodyfikujących* (żądań, które nie wykonały żadnych operacji modyfikujących jak np. operacja zapisu, lub żądań *wycofanych*, ang. *aborted*, *rolled back*). Wykonanie tego typu żądań nie wpływa na stan usługi, dlatego też ich wykonanie nie musi uwzględniać ograniczeń czasu rzeczywistego względem reszty żądań, które zmieniają stan systemu. W ten sposób wydajność systemu może znacznie wzrosnąć, w szczególności, gdy żądania niemodyfikujące stanowią większość żądań przetwarzanych przez system.

Brak gwarancji uwzględnienia czasu rzeczywistego (dla chociażby niektórych typów żądań) z pozoru wydaje się dość błahy. Jednak, jak pokazujemy w Sekcji 4.1 na przykładzie DUR, brak owych gwarancji ma istotne konsekwencje. Muszą one być być brane pod uwagę przez programistę replikowanej usługi, gdy klienci usługi mogą komunikować się między sobą nie tylko poprzez zreplikowany system, ale także innymi kanałami (np. w szczególności poprzez zewnętrzne usługi).

Jak się okazuje, semantyka takich schematów replikacji jak DUR, nie jest właściwie opisana przez żadną z istniejących własności poprawności (patrz Sekcja 2.2.1). Niektóre własności, które nie uwzględniają ograniczeń czasu rzeczywistego są za słabe. Na przykład wspomniane już wcześniej *serializability* (pl. *uszeregowalność*) [14] definiuje ograniczenia tylko dla zatwierdzonych transakcji i nie określa gwarancji dla transakcji żywych lub wycofanych. *Update serializability* [52] czy *extended update serializability* [53] dopuszczają *obserwowanie* przez różne procesy różnej historii zatwierdzeń modyfikujących transakcji w systemie. Inne znane własności takie jak *opacity* (pl. *nieprzezroczystość*) [31] czy *TMS1* [34] są zbyt silne, ponieważ wymagają by ograniczenie czasu rzeczywistego było zawsze przestrzegane (dla wszystkich typów transakcji). Warto zauważyć, że własności takie jak opacity czy TMS1 były zaproponowane dla systemów *pa-*

mięci transakcyjnych (ang. *transactional memory, TM systems*), tj. mechanizmów synchronizacji współbieżnego dostępu do danych mającego stanowić alternatywę dla zamków, patrz np. [26]. Dlatego też ograniczenie czasu rzeczywistego jest w tym wypadku naturalne. Jakkolwiek z uwagi na to, że systemy pamięci transakcyjnej funkcjonują w środowisku lokalnym a nie rozproszonym, ograniczenie czasu rzeczywistego jest relatywnie proste do zagwarantowania.

$\diamond$-*opacity*, które definiujemy w Sekcji 4.3, jest rodziną blisko powiązanych ze sobą własności poprawności opartych na opacity. Własności te osłabiają w systematyczny sposób ograniczenie czasu rzeczywistego dotyczące uszeregowania transakcji w opacity. Ogólnie mówiąc, system który gwarantuje którąkolwiek własność z rodziny $\diamond$-opacity, zachowuje się tak, jak gdyby wszystkie transakcje (a więc również żywe i wycofane transakcje) były wykonywane sekwencyjnie. Wymagania dotyczące uszeregowania transakcji, które to uszeregowanie obserwuje klient, zależą od rozważanej własności. W skrajnych przypadkach ograniczenie czasu rzeczywistego musi być zawsze przestrzegane (zgodnie z *real-time opacity*) albo nigdy nie musi być brane pod uwagę (zgodnie z *arbitrary order opacity*). Oznacza to, że najsilniejsza własność z rodziny $\diamond$-opacity jest tożsama z oryginalną definicją opacity w jej prefiksowo-zamkniętej definicji [31]. Z drugiej strony, arbitrary order opacity przypomina serializability, ale jest zdefiniowane nie tylko dla zatwierdzonych transakcji, ale także dla transakcji żywych i wycofanych. Obecnie $\diamond$-opacity definiuje jeszcze cztery inne własności, słabsze od real-time opacity ale silniejsze niż arbitrary order opacity: *commit-real-time opacity*, *write-real-time opacity*, *update-real-time opacity* i *program order opacity*. Formalnie dowodzimy, że DUR spełnia update-real-time opacity. Własność ta pozwala transakcjom niemodyfikującym oraz transakcjom wycofanym na działanie na stanie systemu, który nie jest najświeższy, ale jest nadal spójny (patrz Sekcja 5.3.3). Write-real-time opacity i commit-real-time opacity są pośrednimi własnościami, dzięki którym możemy porównywać gwarancje oferowane przez transakcyjne i nietransakcyjne schematy replikacji (patrz niżej). Program order opacity jest własnością podobną do virtual time opacity [54], ale zdefiniowaną w oparciu o zbiór pojęć podstawowych wykorzystanych w oryginalnej definicji opacity (patrz także Sekcja 2.2.1).

Wraz z $\diamond$-opacity, definiujemy rodzinę własności zwaną $\diamond$-*linearizability* i bazującą na *linearizability* (pl. *liniowość*) [41], czyli dobrze znanej własności zwykle wykorzystywanej do formalizacji semantyki współbieżnych struktur danych. Własności z rodziny $\diamond$-linearizability pozwalają określić gwarancje oferowane przez silnie spójne systemy, w których przetwarzanie transakcyjne następuje w sposób *niewidoczny* dla klientów (klienci nigdy nie widzą pośrednich wyników wykonania transakcji). Własności te mogą być również używane do zdefiniowania gwarancji systemów, które nie oferują semantyki transakcyjnej. W szczególności, nowe własności pozwalają opisać gwarancje różnych wariantów SMR. W zależności od implementowanej optymalizacji, SMR spełnia real-time linearizability lub też słabsze własności takie jak *write-real-time linearizability* (patrz Sekcje 5.1.3 i 5.2.3). Ponadto, jak dowodzimy, $\diamond$-linearizability zachowuje dwie istotne własności linearizability: *lokalność* (ang. *locality*) i *nieblokowanie* (ang. *non-*

blocking).

W Sekcji 4.5 przedstawiamy formalny rezultat dotyczący relacji pomiędzy $\diamond$-opacity i $\diamond$-linearizability. Ogólnie mówiąc, pokazujemy, że jeśli transakcje są niewidoczne dla klientów, replikowany system spełniający $\diamond$-opacity, spełnia również $\diamond$-linearizability. Rezultat ten ustanawia formalny związek między opacity i linearizability (w ich oryginalnych definicjach) i pozwala bezpośrednio porównać gwarancje systemów takich jak DUR i SMR. Ponadto *obiekt-brama* (ang. *gateway object*), który zdefiniowaliśmy w celu ustanowienia związku między dwiema rodzinami zaproponowanych własności, jest na tyle ogólny, że może być używany do porównania innych (transakcyjnych lub nietransakcyjnych) własności poprawności.

Porównanie podstawowych schematów replikacji

Dwoma podstawowymi silnie spójnymi schematami replikacji są wspomniane już wcześniej podejścia zwane *replikacją maszyny stanowej* (ang. *State Machine Replication, SMR*) [16] [17] [18] i *replikacją z opóźnioną aktualizacją* (ang. *Deferred Update Replication, DUR*) [15]. Oba podejścia można implementować przy użyciu różnych *rozproszonych protokołów uzgadniania* (ang. *distributed agreement protocols*). My skupiamy się na implementacjach zbudowanych w oparciu o protokół *rozgłaszania z globalnym uporządkowaniem* (ang. *Total Order Broadcast, TOB* [45]).

Fundamentalna różnica pomiędzy SMR i DUR polega na innej kolejności synchronizacji serwerów (replik), na których uruchomiona jest replikowana usługa, oraz wykonania żądań. W SMR repliki komunikują się przed wykonaniem żądania poprzez rozgłoszenie żądania przy użyciu TOB. Następnie każda z replik niezależnie wykonuje żądanie, tym samym uaktualniając swój stan w podobny sposób (o ile wykonanie żądania jest deterministyczne). Natomiast w DUR, jak już wspomnieliśmy wcześniej, żądanie jest wykonywane optymistycznie jako atomowa transakcja na pojedynczej replice. Dopiero po zakończeniu wykonania transakcji, modyfikacje utworzone w trakcie wykonania są rozgłaszane przy pomocy TOB do wszystkich replik, tak by uaktualniły one swój stan (ale tylko wtedy, gdy procedura certyfikacji transakcji zakończy się powodzeniem).

Różnice w sposobie wykonania żądań w SMR i DUR skutkują znacznymi rozbieżnościami w wydajności obu podejść wobec różnego rodzaju obciążeń. Na przykład SMR jest bardzo wrażliwe na obciążenia charakteryzujące się dużą liczbą żądań wymagających długiego czasu wykonania. Jest to zrozumiałe, ponieważ SMR wykonuje wszystkie żądania sekwencyjnie. DUR natomiast radzi sobie z tego typu obciążeniami bardzo dobrze. Dzięki temu, że w DUR każde żądanie wykonywane jest (jako transakcja) na pojedynczym serwerze, wykonanie żądań można zrównoleglić przetwarzając je na nowoczesnych wielordzeniowych procesorach i wielu replikach jednocześnie. Z drugiej strony DUR źle ra-

dzi sobie z obciążeniami, w których można obserwować wysoki stopień współzawodnictwa w dostępie do tych samych danych (ang. *high contention levels*). Do takiej sytuacji dochodzi na przykład wtedy, gdy wiele transakcji w tym samym momencie chce modyfikować te same dane. W takim wypadku wiele transakcji musi być wycofanych i ponowionych, co odbija się negatywnie na wydajności i skalowalności DUR. Ponieważ w SMR wszystkie żądania wykonywane są sekwencyjnie, to czy żądania często odwołują się do tych samych danych nie wpływa zasadniczo na wydajność SMR.

Głównym wnioskiem płynącym z badań ewaluacyjnych SMR i DUR, których wyniki przedstawiamy i omawiamy w Sekcji 5.4, jest to, że żadne z podejść nie jest ściśle lepsze od drugiego. Jest to wniosek zaskakujący, ponieważ w przeciwieństwie do DUR, SMR nie może skalować się wraz z rosnącą liczbą replik (a więc także coraz większą dostępną mocą obliczeniową). Dla danego schematu replikacji, obciążenia i rozmiaru klastra, niemal zawsze można jednoznacznie wskazać czynnik decydujący o ograniczonej przepustowości systemu, tzn. relatywnie długie czasy wykonania żądań (które dominują czasy synchronizacji replik) lub długie czasy synchronizacji replik (które dominują czasy wykonania żądań). Niestety, jak pokazujemy, ów dominujący czynnik bywa różny nie tylko dla różnego rodzaju obciążeń, ale czasami zmienia się wraz ze zmianą konfiguracji klastra (np. włączeniem do przetwarzania dodatkowych replik). Tak więc wybór sposobu replikacji usługi powinien być uzależniony nie tylko od spodziewanego obciążenia, ale i rozmiaru klastra.

W kwestii semantyki oba podejścia również znacznie się różnią. DUR oferuje semantykę transakcyjną, a zatem kod żądania (wykonywanego jako transakcja) może korzystać z takich konstrukcji składniowych jak *rollback* czy *retry*, które dają programiście kontrolę nad przepływem sterowania w transakcji. Ponieważ transakcje w DUR mogą zostać w dowolnym momencie przerwane i powtórzone (ze względu na wykrywane przez system konflikty przy współbieżnym dostępie do danych), kod takiej transakcji nie może zawierać tzw. *operacji niewycofywalnych* (ang. *irrevocable operations*), a więc takich, których efektów nie będzie można cofnąć, jak np. w przypadku wywołań systemowych czy operacji wejścia/wyjścia. Kod transakcji może natomiast zawierać niedeterministyczne operacje, ponieważ wykonanie transakcji odbywa w sposób optymistyczny na pojedynczym serwerze. Nie jest tak w przypadku żądań wykonywanych przez SMR. Ponieważ każde żądanie jest wykonywane niezależnie przez każdą replikę, kod żądania musi być deterministyczny. W przeciwnym wypadku stan replik uległby rozbiegnięciu.

Gwarancje oferowane przez SMR i DUR można sformalizować przy pomocy własności własności należących do rodzin $\diamond$-linearizability i $\diamond$-opacity, tj. *real-time linearizability* i *update-real-time linearizability* (patrz Sekcje 5.1.3 i 5.3.3). Znając relację pomiędzy rodzinami własności, możemy pokazać, że DUR oferuje ściśle słabsze gwarancje niż SMR. W Sekcji 5.2.3 badamy gwarancje LSMR, a więc SMR, który implementuje optymalizację polegającą na tym, że żądania, o których z góry wiadomo, że nie zmienią stanu usługi, można wykonywać tylko na pojedynczej replice, bez dodatkowej synchronizacji z innymi replikami. Jak po-

kazujemy, LSMR gwarantuje *write-real-time linearizability*, a więc oferuje gwarancje ściśle silniejsze niż DUR ale ściśle słabsze niż SMR. Widać zatem, że wybór sposobu replikacji usługi nie tylko powinien uwzględniać kwestie wydajności, ale także oczekiwanych gwarancji na wykonanie żądań (znacznie różniących się między SMR i DUR).

Nowy silnie spójny transakcyjny schemat replikacji

W dysertacji opisujemy zaproponowane przez nas nowe podejście do replikacji zwane *hybrydową replikacją transakcyjną* (ang. *Hybrid Transactional Replication, HTR*). W HTR każde żądanie jest wykonywane jako transakcja w jednym z dwóch trybów: *deferred update (DU)* lub *state machine (SM)*. W pierwszym przypadku wykonanie transakcji odbywa się dokładnie tak jak w DUR. Tym samym, HTR wykonujący wszystkie żądania jako transakcje w trybie DU jest tożsamy z DUR. W przypadku wykonania transakcji w trybie SM, najpierw transakcja jest rozgłaszana przy pomocy TOB do wszystkich replik i wtedy każda z replik niezależnie wykonuje transakcję. Tak więc wykonanie transakcji w trybie SM jest bardzo podobne do wykonania żądania w SMR, ale z dodatkowym wsparciem dla przetwarzania transakcyjnego (kod transakcji SM może zawierać konstrukcje składniowe takie jak *rollback* i *retry*). HTR pozwala na współbieżne wykonywanie wielu transakcji w trybie DU i jednej transakcji w trybie SM. W HTR wykonanie transakcji SM jest serializowane w jednym wątku z certyfikacją transakcji wykonanych w trybie DU i ewentualnym późniejszym uaktualnianiem stanu repliki (gdy procedura certyfikacji transakcji zakończy się sukcesem).

Semantyka obu trybów wykonania transakcji jest podobna do semantyki wykonania transakcji w DUR i żądań w SMR. Transakcje wykonywane w trybie DU nie mogą zawierać operacji niewycofywalnych, ale mogą wykonywać niedeterministyczny kod. Natomiast transakcje wykonywane w trybie SM muszą być deterministyczne, ale mogą zawierać operacje niewycofywalne, gdyż takie transakcje mają gwarancję zatwierdzenia. W efekcie, HTR oferuje semantykę bogatszą niż DUR i SMR.

Jak formalnie dowodzimy w Sekcji 6.4, HTR spełnia update-real-time opacity, a więc oferuje te same gwarancje spójności co DUR. Dodatkowo, jeśli wszystkie transakcje wykonywane w HTR są wykonywane w trybie SM, to HTR spełnia real-time opacity, własność analogiczną do tej, którą gwarantuje SMR.

Wybór trybu wykonania transakcji nie musi być wyłącznie uzależniony od tego, jakie gwarancje oczekiwane są w odniesieniu do wykonania transakcji. Na przykład transakcje, które w trybie DU wielokrotnie musiałyby być wycofywane z uwagi na wykrycie konfliktów przy współbieżnym dostępie do danych (np. operacja zmieniania rozmiaru tablicy z haszowaniem), często lepiej jest wykonać w trybie SM. Wybór tego trybu wykonania transakcji daje pewność, że zatwierdzenie transakcji powiedzie się. W trybie SM warto także wykonywać transakcje, które dokonują wielu modyfikacji. W przypadku wykonania trans-

akcji w trybie SM zmiany są tworzone niezależnie przez wszystkie repliki i nie muszą być transmitowane przez sieć (tak jakby to było konieczne w przypadku wykonania transakcji w trybie DU). Natomiast tryb DU sprawdza się lepiej w przypadku transakcji, które sporadycznie odwołują się do tych samych danych i których wykonanie trwa stosunkowo długo. Wtedy takie transakcje można wykonywać w pełni równolegle.

Decyzja na temat tego, w którym trybie wykonać transakcję jest podejmowana przed każdym wykonaniem transakcji przez tzw. *wyrocznię* (ang. *oracle*). Naturalnie transakcja, o której z góry wiadomo, że nie zmodyfikuje stanu systemu jest wykonywana w trybie DU przez pojedynczą replikę. W celu optymalizacji wykorzystania zasobów, wyrocznia podejmuje decyzję dotyczącą trybu wykonania potencjalnie modyfikującej transakcji w oparciu o aktualny stan systemu, rodzaj wykonywanej transakcji oraz zdefiniowaną wcześniej politykę. W szczególności wyrocznia bierze pod uwagę obciążenie serwerów, stopień nasycenia sieci, średnie czasy wykonania i zatwierdzania transakcji oraz stopień współzawodnictwa transakcji w dostępie do tych samych danych. Polityka, którą dostarcza programista, powinna odpowiadać oczekiwanemu obciążeniu i może wykorzystywać mechanizmy uczenia maszynowego (ang. *machine learning*), dzięki czemu system może automatycznie dostosowywać się do zmieniającego się obciążenia. Przykładową politykę korzystającą z mechanizmów uczenia maszynowego przedstawiamy i ewaluujemy w rozprawie (patrz Sekcje 6.6 i 6.7). Zaprezentowane rozwiązanie inspirowane jest dobrze znanymi w środowisku uczenia maszynowego algorytmami rozwiązującymi *problem wielorękiego bandyty* (ang. *multi-armed bandit problem*) [92] [94].

Wyniki testów jasno demonstrują korzyści płynące z łączenia dwóch podejść do replikacji w HTR. Nasz system radzi sobie dobrze z różnego rodzaju obciążeniami, w tym z obciążeniami, o których wiadomo, że będą problematyczne dla DUR czy SMR. Wydajność HTR jest co najmniej tak dobra jak wydajność DUR czy SMR z osobna. Przy niektórych typach obciążeń wydajność oferowana przez HTR jest nawet 50% lepsza niż w przypadku wykonania wszystkich modyfikujących transakcji w trybie DU czy SM (co odpowiada wykonaniu transakcji w DUR czy wykonaniu żądań w LSMR, a więc zoptymalizowanej wersji SMR). Przeprowadzone przez nas testy pokazują również, że wyrocznia wykorzystująca mechanizmy uczenia maszynowego niemal natychmiast reaguje na zmiany obciążenia i działa bardzo dobrze bez względu na rozmiar klastra, na którym uruchomiona jest zreplikowana usługa.

Podsumowanie

Główna teza dysertacji dotyczyła zaproponowania nowego silnie spójnego schematu replikacji o semantyce transakcyjnej, który oferowałby wysoką wydajność dla różnych typów obciążeń. W celu dowiedzenia tezy dysertacji wykonaliśmy badania, które podsumowujemy poniżej.

Po pierwsze zaproponowaliśmy $\diamond$-opacity i $\diamond$-linearizability, dwie nowe rodziny własności poprawności bazujące na dobrze znanych własnościach poprawności wykorzystywanych w kontekście systemów transakcyjnych i systemów modelowanych jako obiekty współdzielone. Zaproponowane własności systematyzują klasy różnych systemów replikacji, umożliwiają formalną analizę oferowanych przez nich gwarancji, a także ich porównanie na płaszczyźnie semantyki. W szczególności, dzięki zaproponowanym w dysertacji nowym własnościom i udowodnionej formalnej relacji między nimi, możliwe jest porównanie gwarancji oferowanych przez silnie spójne systemy, które oferują bądź nie semantykę transakcyjną, a także możliwe jest porównanie bazowych własności (tj. opacity i linearizability) w ich oryginalnych definicjach.

Po drugie dokonaliśmy dogłębnego porównania SMR i DUR, dwóch podstawowych silnie spójnych schematów replikacji. W tym celu, korzystając z wcześniej zdefiniowanych własności, formalnie udowodniliśmy oferowane przez te podejścia gwarancje. Następnie nakreśliliśmy różnice w semantyce obu podejść a także porównaliśmy SMR i DUR eksperymentalnie. Co ciekawe, żadne z podejść nie okazało się ściśle lepsze od drugiego. Wybór schematu replikacji powinien zatem zależeć nie tylko od oczekiwanych gwarancji odnośnie wykonywanych żądań, ale także od spodziewanego rodzaju obciążenia.

Po trzecie zaproponowaliśmy nowe, silnie spójne podejście do replikacji zwane hybrydową replikacją transakcyjną, które łączy w jeden schemat SMR i DUR. HTR zachowuje bogatą semantykę transakcyjną DUR i rozszerza ją o wsparcie dla operacji niewycofywalnych. Jak formalnie dowodzimy, HTR oferuje te same gwarancje na wykonanie transakcji jak DUR (a w przypadku wykonania wszystkich żądań w trybie SM oferowane gwarancje są nawet silniejsze). Ponadto, HTR cechuje bardzo dobra wydajność w szerokim zakresie obciążeń, w tym obciążeń będących problematycznymi dla SMR czy DUR. Co więcej, możliwość wykonania części transakcji w HTR w trybie DU a innych w trybie SM pozwala niekiedy na poprawę przepustowości i skalowalności systemu nawet o kilkadziesiąt procent. Wykorzystanie mechanizmów uczenia maszynowego w HTR pozwoliło na automatyczne i niemal natychmiastowe dostosowywanie się systemu do zmieniającego się obciążenia. Tym samym w opinii autora główna teza dysertacji została udowodniona.

Bibliography

[1] T. Kobus, M. Kokociński, and P. T. Wojciechowski, "Introduction to transactional replication," in *Transactional Memory. Foundations, Algorithms, Tools, and Applications* (R. Guerraoui and P. Romano, eds.), vol. 8913 of *Lecture Notes in Computer Science*, Springer, 2015.

[2] P. T. Wojciechowski, T. Kobus, and M. Kokociński, "State-machine and deferred-update replication: Analysis and comparison," *IEEE Transactions on Parallel and Distributed Systems*, vol. PP, no. 99, 2016.

[3] T. Kobus, M. Kokociński, and P. T. Wojciechowski, "Relaxing real-time order in opacity and linearizability," *Elsevier Journal on Parallel and Distributed Computing*, vol. 100, 2017.

[4] T. Kobus, M. Kokociński, and P. T. Wojciechowski, "Hybrid transactional replication: State-machine and deferred-update replication combined," *In preparation for submission*. arXiv:1612.06302 [cs.DC].

[5] P. T. Wojciechowski, T. Kobus, and M. Kokociński, "Model-driven comparison of state-machine-based and deferred-update replication schemes," in *Proceedings of SRDS '12: the 31st IEEE International Symposium on Reliable Distributed Systems*, Oct. 2012.

[6] T. Kobus, M. Kokociński, and P. T. Wojciechowski, "Hybrid replication: State-machine-based and deferred-update replication schemes combined," in *Proceedings of ICDCS '13: the 33rd IEEE International Conference on Distributed Computing Systems*, July 2013.

[7] T. Kobus, M. Kokociński, and P. T. Wojciechowski, "The correctness criterion for deferred update replication," in *Program of TRANSACT '15: the 10th ACM SIGPLAN Workshop on Transactional Computing*, June 2015.

[8] T. Kobus and P. T. Wojciechowski, "A 90% RESTful group communication service," in *Proceedings of DCDP '10: the 1st International Workshop on Decentralized Coordination of Distributed Processes*, June 2010.

[9] T. Kobus and P. T. Wojciechowski, "RESTGroups for resilient Web services," in *Proceedings of SOFSEM '12: the 38th International Conference on Current Trends in Theory and Practice of Computer Science: Software & Web Engineering Track*, Lecture Notes in Computer Science, Jan. 2012.

[10] M. Kokociński, T. Kobus, and P. T. Wojciechowski, "Make the leader work: Executive deferred update replication," in *Proceedings of SRDS '14: the 33rd IEEE International Symposium on Reliable Distributed Systems*, Oct. 2014.

[11] M. Kokociński, T. Kobus, and P. T. Wojciechowski, "Brief announcement: Eventually consistent linearizability," in *Proceedings of PODC '15: the 34th ACM Symposium on Principles of Distributed Computing*, July 2015.

[12] P. T. Wojciechowski, T. Kobus, and M. Kokociński, "A fault-tolerant data processing computer system and method for implementing a distributed two-tier state machine." EPO patent application no. EP16461501.5, Jan 12. 2016.

[13] P. T. Wojciechowski, T. Kobus, and M. Kokociński, "A fault-tolerant data processing computer system and method for implementing a distributed two-tier state machine." USPTO patent application no. 14/995,211, Jan 14. 2016.

[14] C. H. Papadimitriou, "The serializability of concurrent database updates," *Journal of the ACM*, vol. 26, no. 4, 1979.

[15] B. Charron-Bost, F. Pedone, and A. Schiper, eds., *Replication - Theory and Practice*, vol. 5959 of *Lecture Notes in Computer Science*. 2010.

[16] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM (CACM)*, vol. 21, July 1978.

[17] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: a tutorial," *ACM Computing Surveys (CSUR)*, vol. 22, Dec. 1990.

[18] F. B. Schneider, "Synchronization in distributed programs," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 4, Apr. 1982.

[19] D. Terry, M. Theimer, K. Petersen, A. Demers, M. Spreitzer, and C. Hauser, "Managing update conflicts in Bayou, a weakly connected replicated storage system," in *Proceeding of SOSP '95: the 15th ACM Symposium on Operating Systems Principles*, Dec. 1995.

[20] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," *SIGOPS Operating Systems Review*, vol. 41, Oct. 2007.

[21] A. Lakshman and P. Malik, "Cassandra: A decentralized structured storage system," *SIGOPS Operating Systems Review*, vol. 44, Apr. 2010.

[22] D. Abadi, "Consistency tradeoffs in modern distributed database system design: CAP is only part of the story," *Computer*, vol. 45, Feb. 2012.

[23] P. Bailis and A. Ghodsi, "Eventual consistency today: Limitations, extensions, and beyond," *Queue*, vol. 11, Mar. 2013.

[24] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford, "Spanner: Google's globally-distributed database," in *Proceedings of OSDI '12: the 10th USENIX Conference on Operating Systems Design and Implementation*, Oct. 2012.

[25] J. Shute, R. Vingralek, B. Samwel, B. Handy, C. Whipkey, E. Rollins, M. Oancea, K. Littlefield, D. Menestrina, S. Ellner, J. Cieslewicz, I. Rae, T. Stancescu, and H. Apte, "F1: A distributed SQL database that scales," *Proceeding of Very Large Data Base (VLDB) Endowment*, vol. 6, Aug. 2013.

[26] M. Herlihy and J. E. B. Moss, "Transactional memory: architectural support for lock-free data structures," in *Proceedings of ISCA '93: the 20th International Symposium on Computer Architecture*, June 1993.

[27] P. Romano, N. Carvalho, and L. Rodrigues, "Towards distributed software transactional memory systems," in *Proceedings of LADIS '08: the 2nd Workshop on Large-Scale Distributed Systems and Middleware*, Sept. 2008.

[28] J. Cachopo and A. Rito-Silva, "Versioned boxes as the basis for memory transactions," in *Proceedings of OOPSLA '05: Workshop on Synchronization and Concurrency in Object-Oriented Languages (SCOOL)*, Oct. 2005.

[29] G. Korl, N. Shavit, and P. Felber, "Noninvasive concurrency with Java STM," in *Proceedings of MULTIPROG '10: the 3rd Workshop on Programmability Issues for Multi-Core Computers*, Jan. 2010.

[30] P. A., Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency control and recovery in database systems*. Addison-Wesley, 1987.

[31] R. Guerraoui and M. Kapalka, *Principles of Transactional Memory*. Synthesis Lectures on Distributed Computing Theory, 2010.

[32] H. Attiya, A. Gotsman, S. Hans, and N. Rinetzky, "A programming language perspective on transactional memory consistency," in *Proceedings of PODC '13: the 32nd ACM Symposium on Principles of Distributed Computing*, June 2013.

[33] H. Attiya, S. Hans, P. Kuznetsov, and S. Ravi, "Safety of deferred update in transactional memory," in *Proceedings of ICDCS '13: the 33rd IEEE International Conference on Distributed Computing Systems*, July 2013.

[34] S. Doherty, L. Groves, V. Luchangco, and M. Moir, "Towards formally specifying and verifying transactional memory," *Formal Aspects of Computing*, vol. 25, no. 5, 2013.

[35] B. Kemme, F. Pedone, G. Alonso, A. Schiper, and M. Wiesmann, "Using optimistic atomic broadcast in transaction processing systems," *IEEE Transactions on Knowledge and Data Engineering*, vol. 15, July 2003.

[36] R. Palmieri, F. Quaglia, and P. Romano, "AGGRO: Boosting STM Replication via Aggressively Optimistic Transaction Processing," in *Proceedings of NCA 2010: the 9th IEEE International Symposium on Network Computing and Applications*, Feb. 2010.

[37] D. Sciascia, F. Pedone, and F. Junqueira, "Scalable deferred update replication," in *Proceedings of DSN '12: the 42nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, June 2012.

[38] D. Sciascia and F. Pedone, "RAM-DUR: In-Memory Deferred Update Replication.," in *Proceedings of SRDS '12: the 31st IEEE International Symposium on Reliable Distributed Systems*, Oct. 2012.

[39] B. Kemme and G. Alonso, "Don't be lazy, be consistent: Postgres-R, a new way to implement database replication," in *Proceedings of VLDB 2000: the 26th International Conference on Very Large Data Bases*, Sept. 2000.

[40] M. Couceiro, P. Romano, and L. Rodrigues, "PolyCert: Polymorphic self-optimizing replication for in-memory transactional grids," in *Proceedings of Middleware '11: the 12th ACM/IFIP/USENIX International Conference on Middleware*, Dec. 2011.

[41] M. P. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 12, no. 3, 1990.

[42] F. B. Schneider, *Replication management using the state-machine approach*. ACM Press/Addison-Wesley, 1993.

[43] L. Lamport, "The implementation of reliable distributed multiprocess systems," *Computer Networks*, vol. 2, 1978.

[44] L. Lamport, "Using time instead of timeout for fault-tolerant distributed systems," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 6, Apr. 1984.

[45] X. Défago, A. Schiper, and P. Urbán, "Total order broadcast and multicast algorithms: Taxonomy and survey," *ACM Computing Surveys*, vol. 36, no. 4, 2004.

[46] R. Palmieri, F. Quaglia, and P. Romano, "OSARE: Opportunistic Speculation in Actively REplicated transactional systems," in *Proceedings of SRDS*

'11: the 30th IEEE International Symposium on Reliable Distributed Systems, Oct 2011.

[47] S. Hirve, R. Palmieri, and B. Ravindran, "HiperTM: High performance, fault-tolerant transactional memory," in *Proceedings of ICDCN'14: the 15th International Conference on Distributed Computing and Networking*, Jan. 2014.

[48] F. Pedone, R. Guerraoui, and A. Schiper, "Exploiting atomic broadcast in replicated databases," in *Proceedings of Euro-Par '98: the 4th International Conference on Parallel Processing*, Sept. 1998.

[49] F. Pedone, R. Guerraoui, and André, "The database state machine approach," *Distributed and Parallel Databases*, vol. 14, no. 1, 2003.

[50] D. Agrawal, G. Alonso, A. E. Abbadi, and I. Stanoi, "Exploiting atomic broadcast in replicated databases (extended abstract)," in *Proceedings of Euro-Par '97: the 3rd International Conference on Parallel Processing*, Aug. 1997.

[51] M. Couceiro, P. Romano, N. Carvalho, and L. Rodrigues, "D2STM: Dependable distributed software transactional memory," in *Proceedings of PRDC '09: the 15th IEEE Pacific Rim International Symposium on Dependable Computing*, Nov. 2009.

[52] R. Hansdah and L. Patnaik, "Update serializability in locking," in *Proceedings of ICDT '86: the 1st International Conference on Database Theory*, Sept. 1986.

[53] A. Adya, *Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions*. Ph.D., MIT, 1999. Also as Technical Report MIT/LCS/TR-786.

[54] D. Imbs, J. R. G. De Mendivil Moreno, and M. Raynal, "On the consistency conditions of transactional memories," Research Report PI 1917, Inria, 2008.

[55] R. Guerraoui and M. Kapalka, "On the correctness of transactional memory," in *Proceedings of PPoPP '08: the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Feb. 2008.

[56] Y. Breitbart, D. Georgakopoulos, M. Rusinkiewicz, and A. Silberschatz, "On rigorous transaction scheduling," *IEEE Transactions on Software Engineering*, vol. 17, no. 9, 1991.

[57] M. Raynal, G. Thia-Kime, and M. Ahamad, "From serializable to causal transactions (abstract)," in *Proceedings of PODC '95: the 15th ACM Symposium on Principles of Distributed Computing*, Aug. 1996.

[58] M. Raynal, G. Thia-Kime, and M. Ahamad, "From serializable to causal transactions for collaborative applications," in *Proceedings of EUROMICRO '97: the 23rd Conference on New Frontiers of Information Technology*, Sept. 1997.

[59] C. E. Bezerra, F. Pedone, and R. V. Renesse, "Scalable state-machine replication," *Proceedings of DSN '14: the 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2014.

[60] L. L. Hoang, C. E. B. Bezerra, and F. Pedone, "Dynamic scalable state machine replication," in *Proceedings of DSN '16: the 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2016.

[61] D. Mosberger, "Memory consistency models," *SIGOPS Operating Systems Review*, vol. 27, no. 1, 1993.

[62] L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs," *IEEE Transactions on Computers*, vol. C-28, no. 9, 1979.

[63] T. Crain, D. Imbs, and M. Raynal, "Towards a universal construction for transaction-based multiprocess programs," *Theoretical Computer Science*, vol. 496, 2013.

[64] H. Attiya, R. Guerraoui, and P. Kouznetsov, "Computing with reads and writes in the absence of step contention," in *Proceedings of DISC '05: the 19th International Conference on Distributed Computing*, Sept. 2005.

[65] M. K. Aguilera, S. Frolund, V. Hadzilacos, S. L. Horn, and S. Toueg, "Abortable and query-abortable objects and their efficient implementation," in *Proceedings of PODC '07: the 26th ACM Symposium on Principles of Distributed Computing*, Aug. 2007.

[66] V. Hadzilacos and S. Toueg, "On deterministic abortable objects," in *Proceedings of PODC '13: the 32nd ACM Symposium on Principles of Distributed Computing*, July 2013.

[67] B. Alpern and F. B. Schneider, "Defining liveness," *Information Processing Letters*, vol. 21, no. 4, 1985.

[68] N. A. Lynch, *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., 1996.

[69] R. M. Yoo, S. Viswanathan, V. Deshpande, C. J. Hughes, and S. Aundhe, "Early experience on transactional execution of Java programs using Intel Transactional Synchronization Extensions," in *Program of TRANSACT '14: the 9th ACM SIGPLAN Workshop on Transactional Computing*, Mar. 2014.

[70] J. Kończak, N. Santos, T. Żurkowski, P. T. Wojciechowski, and A. Schiper, "JPaxos: State machine replication based on the Paxos protocol," Tech.

Rep. EPFL-REPORT-167765, Faculté Informatique et Communications, EPFL, July 2011.

[71] L. Lamport, "The part-time parliament," *ACM Transactions on Computer Systems (TOCS)*, vol. 16, no. 2, 1998.

[72] C. Kotselidis, M. Lujan, M. Ansari, K. Malakasis, B. Kahn, C. Kirkham, and I. Watson, "Clustering JVMs with software transactional memory support," in *Proceedings of IPDPS '10: the 24th IEEE International Parallel and Distributed Processing Symposium*, Apr. 2010.

[73] R. L. Bocchino, V. S. Adve, and B. L. Chamberlain, "Software transactional memory for large scale clusters," in *Proceedings of PPoPP '08: the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, Feb. 2008.

[74] C. Kotselidis, M. Ansari, K. Jarvis, M. Luján, C. C. Kirkham, and I. Watson, "DiSTM: A software transactional memory framework for clusters," in *Proceedings of ICPP '08: the 37th IEEE International Conference on Parallel Processing*, Sept. 2008.

[75] M. M. Saad and B. Ravindran, "HyFlow: A high performance distributed transactional memory framework," in *Proceedings of HPDC '11: the 20th International Symposium on High Performance Distributed Computing*, June 2011.

[76] A. Turcu, B. Ravindran, and R. Palmieri, "HyFlow2: A high performance distributed transactional memory framework in scala," in *Proceedings of PPPJ'13: the 10th International Conference on Principles and Practices of Programming on JAVA platform: virtual machines, languages, and tools*, Sept. 2013.

[77] C. Blundell, J. Devietti, E. C. Lewis, and M. M. K. Martin, "Making the fast case common and the uncommon case simple in unbounded transactional memory," in *Proceedings of ISCA '07: the 34th International Symposium on Computer Architecture*, June 2007.

[78] M. Olszewski, J. Cutler, and J. G. Steffan, "JudoSTM: A dynamic binary-rewriting approach to software transactional memory," in *Proceedings of PACT '07: the 16th International Conference on Parallel Architectures and Compilation Techniques*, Sept. 2007.

[79] M. F. Spear, M. Michael, and M. L. Scott, "Inevitability mechanisms for software transactional memory," in *Program of TRANSACT '08: the 3rd Workshop on Transactional Computing*, Feb. 2008.

[80] A. Welc, B. Saha, and A.-R. Adl-Tabatabai, "Irrevocable transactions and their applications," in *Proceedings of SPAA '08: the 20th ACM Symposium on Parallelism in Algorithms and Architectures*, June 2008.

[81] T. Usui, R. Behrends, J. Evans, and Y. Smaragdakis, "Adaptive locks: Combining transactions and locks for efficient concurrency," *Journal of Parallel Distributed Computing*, vol. 70, no. 10, 2010.

[82] H. T. Kung and J. T. Robinson, "On optimistic methods for concurrency control," *ACM Transactions on Database Systems (TODS)*, vol. 6, June 1981.

[83] N. Carvalho, P. Romano, and L. Rodrigues, "Asynchronous lease-based replication of software transactional memory," in *Proceedings of Middleware '10: the 12th ACM/IFIP/USENIX International Middleware Conference*, Dec. 2010.

[84] K. Siek and P. T. Wojciechowski, "Atomic RMI: A distributed transactional memory framework," *International Journal of Parallel Programming*, vol. 44, June 2015.

[85] P. T. Wojciechowski and K. Siek, "Atomic RMI 2: Distributed transactions for Java," in *Proceedings of AGERE '16: the 6th International Workshop on Programming Based on Actors, Agents, and Decentralized Control*, 2016.

[86] A. Thomson and D. J. Abadi, "The case for determinism in database systems," *Proceeding of Very Large Data Base (VLDB) Endowment*, vol. 3, Sept. 2010.

[87] M. Couceiro, P. Ruivo, P. Romano, and L. Rodrigues, "Chasing the optimum in replicated in-memory transactional platforms via protocol adaptation," in *Proceedings of DSN'13: the 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, June 2013.

[88] M. M. Saad and B. Ravindran, "Supporting STM in distributed systems: Mechanisms and a Java framework," in *Program of TRANSACT '11: the 6rd Workshop on Transactional Computing*, June 2011.

[89] R. J. Dias, D. Distefano, J. C. Seco, and J. Lourenço, "Verification of snapshot isolation in transactional memory Java programs," in *Proceedings of ECOOP '12: the 26th European Conference on Object-Oriented Programming*, June 2012.

[90] A. Correia, J. Pereira, and R. Oliveira, "AKARA: A flexible clustering protocol for demanding transactional workloads," in *Proceedings of On the Move to Meaningful Internet Systems: OTM 2008*, Nov. 2008.

[91] A. Welc, A. L. Hosking, and S. Jagannathan, "Transparently reconciling transactions with locking for Java synchronization," in *Proceedings of ECOOP '06: the 20th European Conference on Object-Oriented Programming*, July 2006.

[92] H. Robbins, "Some aspects of the sequential design of experiments," *Bulletin of the American Mathematical Society*, vol. 58, no. 5, 1952.

[93] T. Lai and H. Robbins, "Asymptotically efficient adaptive allocation rules," *Advances in Applied Mathematics*, vol. 6, Mar. 1985.

[94] V. Kuleshov and D. Precup, "Algorithms for multi-armed bandit problems," *Computing Research Repository*, vol. abs/1402.6028, 2014.

[95] M. Couceiro, D. Didona, L. Rodrigues, and P. Romano, "Self-tuning in distributed transactional memory," in *Transactional Memory. Foundations, Algorithms, Tools, and Applications*, vol. 8913 of *Lecture Notes in Computer Science*, 2015.

[96] D. Rughetti, P. D. Sanzo, B. Ciciani, and F. Quaglia, "Machine learning-based self-adjusting concurrency in software transactional memory systems," in *Proceedings of MASCOTS '12: the 20th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, 2012.

[97] M. Castro, L. F. W. Góes, C. P. Ribeiro, M. Cole, M. Cintra, and J. F. Méhaut, "A machine learning-based approach for thread mapping on transactional memory applications," in *Proceedings of HiPC '11: the 18th International Conference on High Performance Computing*, Dec 2011.

[98] M. K. Aguilera, C. Wei, and S. Toueg, "Failure detection and consensus in the crash-recovery model," in *Proceedings of DISC '98: the 12th International Symposium on Distributed Computing*, Sept. 1998.

[99] R. Guerraoui and L. Rodrigues, *Introduction to Reliable Distributed Programming*. Springer-Verlag New York, Inc., 2006.

[100] T. D. Chandra, V. Hadzilacos, and S. Toueg, "The weakest failure detector for solving consensus," *Journal of the ACM (JACM)*, vol. 43, July 1996.

[101] R. Boichat, P. Dutta, S. Frølund, and R. Guerraoui, "Deconstructing Paxos," *SIGACT News*, vol. 34, no. 1, 2003.

[102] M. F. Spear, V. J. Marathe, W. N. Scherer, and M. L. Scott, "Conflict detection and validation strategies for software transactional memory," in *Proceedings of DISC '06: the 20th International Symposium on Distributed Computing*, Sept. 2006.

[103] D. Dice, O. Shalev, and N. Shavit, "Transactional locking II," in *Proceedings of DISC '06: the 20th International Symposium on Distributed Computing*, Sept. 2006.

[104] R. Guerraoui and E. Ruppert, "Linearizability is not always a safety property," in *Proceedings of NETYS '14: the 4th International Conference on Networked Systems*, May 2014.

[105] H. Attiya, A. Gotsman, S. Hans, and N. Rinetzky, "Safety of live transactions in transactional memory: TMS is necessary and sufficient," in *Proceedings of DISC '14: the 28th International Symposium on Distributed Computing*, May 2014.

[106] T. Kobus, M. Kokociński, and P. T. Wojciechowski, "State machine replication: variants and properties," Tech. Rep. RA-7/16, Poznań Univ. of Technology, May 2016.

[107] P. J. Marandi, C. E. Bezerra, and F. Pedone, "Rethinking state-machine replication for parallelism," in *Proceedings of ICDCS '14: the 34th IEEE International Conference on Distributed Systems*, June 2014.

[108] R. Van Renesse and D. Altinbuken, "Paxos made moderately complex," *ACM Computing Surveys*, vol. 47, Feb. 2015.

[109] P. A. Bernstein and N. Goodman, "Multiversion concurrency control—theory and algorithms," *ACM Transactions on Database Systems (TODS)*, vol. 8, Dec. 1983.

[110] N. Santos and A. Schiper, "Optimizing Paxos with batching and pipelining," *Theor. Comput. Sci.*, vol. 496, 2013.

[111] T. J. Hastie, R. J. Tibshirani, and J. H. Friedman, *The elements of statistical learning: data mining, inference, and prediction*. Springer series in statistics, Springer, 2009.

[112] K. Siek and P. T. Wojciechowski, "A formal design of a tool for static analysis of upper bounds on object calls in Java," in *Proceedings of FMICS '12: the 17th International Workshop on Methods for Industrial Critical Systems*, 2012.

[113] C.-C. Wang, S. R. Kulkarni, and H. V. Poor, "Bandit problems with side observations.," *IEEE Transactions on Automatic Control*, vol. 50, no. 3, 2005.

A

Proofs Regarding Traits of ⋄-Opacity and ⋄-Linearizability

In this chapter we provide the detailed proofs for all the formal results related to the ⋄-opacity and ⋄-linearizability families of properties.

A.1 ⋄-Opacity is a Safety Property

Now we prove that ⋄-opacity is a safety property [67] [68]. The use of order relations other than real-time does not influence opacity's prefix-closeness and limit-closeness. Therefore, the proof of the following theorem is identical as for the original definition of opacity [31].

Theorem 1. *⋄-opacity is a safety property.*

Proof. In order to prove that ⋄-opacity is a safety property, it is necessary to show that it is non-empty, prefix-closed and limit-closed.

Trivially an empty t-history $H_\perp = \langle\rangle$ is ⋄-opaque. Therefore, ⋄-opacity is non-empty.

By the definition of ⋄-opacity, if a t-history H is ⋄-opaque, then every H_k, a finite prefix of H, is final-state ⋄-opaque. Since for every H_k, all its prefixes (H_i, $i < k$) are final-state ⋄-opaque, H_k is also ⋄-opaque. Therefore, every finite prefix of H is ⋄-opaque, which means that ⋄-opacity is prefix-closed.

Now, let us consider an infinite sequence $H_0, H_1, H_2, ...$ of finite t-histories, such that each H_k is a prefix of H_{k+1}, and H_k is ⋄-opaque. Let H be a t-history which is the limit of this sequence. Since each H_k is finite and ⋄-opaque, H_k is final-state ⋄-opaque. Therefore by definition of ⋄-opacity H is ⋄-opaque. Thus, ⋄-opacity is limit-closed.

To conclude, we have shown that ⋄-opacity is non-empty, prefix-closed and limit-closed. Therefore it is a safety property. □

Corollary 1. *A system (modelled as a TM object) is ⋄-opaque if, and only if every finite history it produces is final-state ⋄-opaque.*

Proof. The proof follows directly from Theorem 1. □

A.2 ⋄-Linearizability is a Safety Property

Theorem 2. *⋄-linearizability is a safety property.*

Proof. The proof is analogous to the proof of Theorem 1. □

Corollary 2. *A system (modelled as a set of shared objects) is ⋄-linearizable, if and only if every finite history it produces is final-state ⋄-linearizable.*

Proof. The proof follows directly from Theorem 2. □

A.3 ⋄-Linearizability is Non-blocking and Satisfies Locality

The proof of Theorem 3, which we prove using the following two lemmas, is inspired by the proof in [41].

Lemma 1. *Let $X \in \mathcal{X}$ be a shared object with a sequential specification $(Q, q_0, INV, RES, \delta)$. Let $op \in INV$ be a total operation on X. If a finite ⋄-linearizable history H contains the invocation event $X.inv_l(op)$ of some pending operation execution o_k (for some process p_l), then there exists a response $v \in RES$, such that the history H' obtained by appending the response event $X.resp_l(v)$ to H is ⋄-linearizable.*

Proof. Since H is a finite ⋄-linearizable history, H is also final-state ⋄-linearizable. Let $\bar{H}$ be some completion of H such that there exists S, a sequential history equivalent to $\bar{H}$ such that S is legal and S respects the ⋄-order of $\bar{H}$ (i.e., $<^{\diamond}_{\bar{H}} \subseteq <^{\diamond}_{S}$). We have two cases to consider. If S contains $X.inv_l(op)$, then S also contains $X.resp_l(v)$, as S is complete. Therefore S is equivalent to $\bar{H}$ and $\bar{H}$ is ⋄-linearizable.

Now consider a case when S does not contain $X.inv_l(op)$. Because $X.inv_l(op)$ is an invocation of a total operation, a response to this operation is defined for every state of X. Therefore, there exists a response $v \in RES$ such that $S' = S \cdot \langle X.inv_l(op), X.resp_l(v)\rangle$ is legal. Let $\bar{H}'$ be a completion of H constructed in the same way as $\bar{H}$ but without removing $X.inv_l(op)$ and by appending $X.resp_l(v)$ at the end of the history. Then S' is equivalent to $\bar{H}'$.

Now we have to show that S' satisfies the ⋄-order of $\bar{H}'$. We give the proof by a contradiction. Assume that S' does not satisfy the ⋄-order of $\bar{H}'$. Then there exist two operation executions o_i and o_j, $i \neq j$, such that $o_i <^{\diamond}_{\bar{H}'} o_j$ and $o_i \not<^{\diamond}_{S'} o_j$. Since $o_i <^{\diamond}_{\bar{H}'} o_j$, we know that $<^{\diamond}_{\bar{H}'} \neq <^{a}_{\bar{H}'}$ ($<^{a}_{\bar{H}'} = \emptyset$), and that $o_i <^{r}_{\bar{H}'} o_j$

and $o_i \not<^r_{S'} o_j$ (every order relation other than arbitrary order assumes real-time precedence). Note that, since $<^{\diamond}_{\bar{H}} \subseteq <^{\diamond}_{S}$ and $S' = S \cdot \langle X.inv_l(op), X.resp_l(v) \rangle$, then $<^{\diamond}_{\bar{H}} \subseteq <^{\diamond}_{S'}$. Since $o_i <^{\diamond}_{\bar{H}'} o_j$, then $o_i <^{\diamond}_{S'} o_j$.

Now we show that either $o_i = o_k$ or $o_j = o_k$, which means that either o_i or o_j appears in $\bar{H}'$ but not in $\bar{H}$. Assume otherwise, i.e., o_i and o_j are some two operation executions in $\bar{H}$. From our assumption, $o_i <^{\diamond}_{\bar{H}'} o_j$. Therefore, also $o_i <^{\diamond}_{\bar{H}} o_j$. Because S respects the $\diamond$-order of $\bar{H}$, $o_i <^{\diamond}_{S} o_j$, Because S' is constructed by appending two evens to S, also $o_i <^{\diamond}_{S'} o_j$. However, we assumed that $o_i \not<^{\diamond}_{S'} o_j$. Hence, either $o_i = o_k$ or $o_j = o_k$.

In the former case, $o_k <^r_{\bar{H}'} o_j$. By construction of $\bar{H}'$ we know also that the response event of o_k is the last event in $\bar{H}'$ and therefore no operation executions can succeed o_k in $\bar{H}'$, a contradiction. In the latter case, we know that $o_i \not<^r_{S'} o_k$ (from the assumption). Therefore, $o_k <^r_{S'} o_i$ because S' is totally ordered. But it is impossible, since o_k is the last operation execution in S', a contradiction. Therefore the assumption was false and S' respects the $\diamond$-order of $\bar{H}'$, what concludes the proof. □

Lemma 2. *A history H is $\diamond$-linearizable if and only if for every shared object $X \in \mathcal{X}$, $H|X$ is $\diamond$-linearizable.*

Proof. First we show that if H is $\diamond$-linearizable then, for every shared object $X \in \mathcal{X}$, $H|X$ is $\diamond$-linearizable. From the assumption we know that H is $\diamond$-linearizable and thus every finite prefix H' of H is final-state $\diamond$-linearizable. Let $\bar{H}'$ be a completion of H' such that there exists a legal sequential history S' equivalent to $\bar{H}'$, which respects the $\diamond$-order of $\bar{H}'$. Since S' is legal, for each X, $S'|X$ is legal. Trivially, for every X, $\bar{H}'|X$ is a completion of $H'|X$. Therefore, for each X, $S'|X$ is equivalent to $\bar{H}'|X$, $S'|X$ satisfies the $\diamond$-order of X, $H'|X$ is final-state $\diamond$-linearizable. Obviously, $H'|X$ is a prefix of $H|X$. Since, every finite prefix of $H|X$ is final-state $\diamond$-linearizable, by definition, $H|X$ is $\diamond$-linearizable.

Now we show that if H is a history such that for every shared object $X \in \mathcal{X}$, $H|X$ is $\diamond$-linearizable then H is $\diamond$-linearizable. Let H' be any finite prefix of H. Now let us consider any X, and let $H'_X = H'|X$. Then, H'_X is final-state $\diamond$-linearizable. Let $\bar{H}'_X$ be a completion of H'_X such that S'_X is a legal sequential history equivalent to $\bar{H}'_X$ and $<^{\diamond}_{S'_X}$ be an order relation in S'_X which respects the $\diamond$-order of $\bar{H}'_X$ ($<^{\diamond}_{\bar{H}'_X} \subseteq <^{\diamond}_{S'_X}$).

Let $\bar{H}'$ be a completion of H' which contains all events of every $\bar{H}'_X$ such that $X \in \mathcal{X}$, and satisfies the order $<_{\bar{H}'}$ which is defined as follows. Let e_i and e_j be any two events in $\bar{H}'$:

- if $e_i, e_j \in H'$ and e_i precedes e_j in H' then e_i precedes e_j in $\bar{H}'$ ($e_i <_{\bar{H}'} e_j$),
- if $e_i \in H'$ and $e_j \notin H'$ then $e_i <_{\bar{H}'} e_j$,
- if $e_i, e_j \notin H'$, then the order of e_i and e_j in $\bar{H}'$ is arbitrary.

Let $\ll$ be a transitive closure of $\bigcup_X (<^r_{S'_X})$ and $<^{\diamond}_{\bar{H}'}$. Assuming that $\ll$ is a partial order, we can construct a sequential history S' which is equivalent to $\bar{H}'$ and respects $\ll$ (i.e., by performing a topological sort of $\ll$). Then S' is legal, because

for every X, $S'|X = S'_X$ ($<^r_{S'_X} \subseteq \ll$ and $<^r_{S'_X}$ is a total order) and we know that S'_X is legal. Also by construction, S' respects the $\diamond$-order of $\bar{H}'$ ($<^{\diamond}_{\bar{H}'} \subseteq \ll$). Therefore H' is final-state $\diamond$-linearizable. We now show that $\ll$ is indeed a partial order, i.e., no cycle exists in $\ll$. We give a proof by contradiction.

Assume that the cycle of minimal length in $\ll$ is $o_1 \ll o_2 \ll ... \ll o_n \ll o_1$. Then each edge in the cycle follows either from the relation $<^r_{S'_X}$ for some shared object X or from the relation $<^{\diamond}_{\bar{H}'}$. Suppose that all operation executions in the cycle pertain to the same object X. Since $<^r_{S'_X}$ is a total order, the cycle must contain two operation executions o_i and o_j such that $o_i <^r_{S'_X} o_j$ and $o_j <^{\diamond}_{\bar{H}'} o_i$. Because $o_j <^{\diamond}_{\bar{H}'} o_i$, we know that $<^{\diamond}_{\bar{H}'} \neq <^a_{\bar{H}'}$ ($<^a_{\bar{H}'} = \emptyset$), and $o_j <^r_{\bar{H}'} o_i$ (every order relation other than arbitrary order assumes real-time precedence). Note that by the construction of $\bar{H}'$ and the fact that by definition $\bar{H}'_X = \bar{H}'|X$, for any two operation executions o_k and o_l on the same object X, if $o_k <^r_{\bar{H}'} o_l$ then $o_k <^r_{\bar{H}'_X} o_l$. In turn $o_j <^r_{\bar{H}'_X} o_i$ and $o_j <^r_{S'_X} o_i$ (because $<^{\diamond}_{\bar{H}'_X} \subseteq <^{\diamond}_{S'_X}$). Since $<^r_{S'_X}$ is a total order, both $o_i <^r_{S'_X} o_j$ and $o_j <^r_{S'_X} o_i$ cannot be true, a contradiction. Therefore the cycle must contain operation executions on at least two different shared objects.

Let o_j be an operation execution on X. Let o_k be the first operation execution after o_j in the cycle, such that o_k is an operation execution on $Y \neq X$. Also let o_i be the first operation execution before o_j in the cycle, such that o_i is an operation execution on $Z \neq X$ (possibly $Y = Z$). Then $o_i \ll o_j \ll o_k$. Since o_i and o_j are on different objects, $o_i <^{\diamond}_{\bar{H}'} o_j$ (by the construction of the cycle, if an edge does not follow from the relation $<^r_{S'_X}$ then it has to follow from the relation $<^{\diamond}_{\bar{H}'}$). Analogically, $o_j <^{\diamond}_{\bar{H}'} o_k$. Therefore $o_i <^{\diamond}_{\bar{H}'} o_k$. If $o_i = o_k$, then $o_k <^{\diamond}_{\bar{H}'} o_j$ and $o_j <^{\diamond}_{\bar{H}'} o_k$, which is impossible. Therefore $o_i \neq o_k$. Since $o_i \neq o_k$, $o_i <^{\diamond}_{\bar{H}'} o_k$ and $<^{\diamond}_{\bar{H}'} \subseteq \ll$, there exists a shorter cycle $o_1 \ll o_2 \ll ... \ll o_i \ll o_k \ll ... \ll o_n \ll o_1$ which contradicts the assumption that the original cycle was the shortest one. Therefore, the assumption that $\ll$ is not a partial order was false, thus reassuring that H' is final-state $\diamond$-linearizable.

Since, every prefix of H is final-state $\diamond$-linearizable, H is $\diamond$-linearizable, what concludes the proof. □

Theorem 3. *$\diamond$-linearizability is non-blocking and satisfies locality.*

Proof. The proof follows directly from Lemma 1 and Lemma 2. □

A.4 Commit-real-time Linearizability is Equivalent to Real-time Linearizability

Lemma 3. *Let $S = \langle o_1, o_2, ... \rangle$ be a sequential history and $S' = \langle o_1, o_2, ..., o_k, o_a, o_{k+1}, ... \rangle$ be a sequential history constructed by inserting an aborted operation execution $o_a = X.op \rightarrow_r \perp$ to S (for some process p_r). S is legal if and only if S' is legal.*

Proof. First let us assume that S is legal. By definition of legality, $S|X$ satisfies the sequential specification of X. Now let o_i and o_j be two operation executions on X in S such that i is the largest possible number lower or equal k and o_j is the next operation execution after o_i in $S|X$ (j is greater or equal $k+1$; o_i and o_j may or may not exist). It means that, if o_i exists, it is the last operation execution on X in S before o_k, and o_j, if it exists, is the operation execution in $S|X$, which directly follows o_i.

Now let q_i be the state of X after execution of o_i (or the initial state of X if o_i does not exist). Naturally, q_i is also the state of X that o_j operates on. Since o_a is aborted, the transition $(q_i, op, \perp, q_i)$ belongs to the sequential specification of X (by definition of aborted operation). Therefore, adding o_a to $S|X$ in between o_i and o_j, and thus creating $S'|X$, does not break the sequential specification of X. In turn S' is also legal.

Now let us assume that S' is legal. By the definition of legality, $S'|X$ satisfies the sequential specification of X. Now let o_i be an operation execution which directly precedes o_a in $S'|X$ and let o_j be an operation execution which directly succeeds o_a in $S'|X$ (o_i and o_j may or may not exist). Additionally, let q_i be the state of X after the execution of o_i (or the initial state of X if o_i does not exist). Then the transition of o_a is equal $(q_i, op, \perp, q_i)$ (by definition of aborted operation). It means that the state that o_j operates on is equal to q_i. Therefore, removing o_a from $S'|X$ (and thus creating $S|X$) still satisfies the sequential specification of X. In turn, S is legal.

Since we proved implications in both directions, S is legal if and only if S' is legal. □

Theorem 4. *Commit-real-time linearizability is equivalent to real-time linearizability.*

Proof. In order to prove that commit-real-time linearizability is equivalent to real-time linearizability we have to show that every commit-real-time linearizable history is real-time linearizable and vice versa.

Actually, it suffices to show that every final-state commit-real-time linearizable history is final-state real-time linearizable and vice versa. Now we prove by a contradiction why it is so.

Assume that for any history H, if H is final-state commit-real-time linearizable then H is final-state real-time linearizable but (the contradiction part) it is not true that if H is commit-real-time linearizable then H is real-time linearizable. The latter implication is false if there exists a history H such that it is commit-real-time linearizable and it is not real-time linearizable. From the definition of commit-real-time linearizability, every history H', which is a prefix of H, is final-state commit-real-time linearizable. But now, using the first assumption, we know that every such H' is final-state real-time linearizable. Then, H is real-time linearizable, because every its prefix H' is final-state real-time linearizable (by definition). This contradicts the assumption that H is not real-time linearizable thus completing the proof. An analogical reasoning can be inferred to show that in order to prove that every real-time linearizable history H is commit-real-time linearizable it suffices to show that every final-state

real-time linearizable history is final-state commit-real-time linearizable.

Since, $<^c_H \subseteq <^r_H$, by the definition of final-state ⋄-linearizability, every final-state real-time linearizable history is trivially also final-state commit-real-time linearizable. Now, we proceed to prove that every final-state commit-real-time linearizable history is also final-state real-time linearizable.

Let H be any final-state commit-real-time linearizable history. We now show how to construct a sequential history which is equivalent to some completion of H, such that it is legal and respects the real-time order of that completion.

Part 1. Construction of the sequential history.

Let us denote by $DSG(\hat{H})$ a directed graph which represents partial order induced by operation execution precedence in some history $\hat{H}$. Operation executions constitute the vertices of the graph, which are connected by a directed edge if and only if the operation execution o_i represented by the first vertex precedes (in real-time) the operation execution o_j represented by the second vertex, i.e., $o_i <^r_{\hat{H}} o_j$. Let o_i and o_j be any two operation executions in $\hat{H}$ such that $o_i <^r_{\hat{H}} o_j$. Then, naturally, it is impossible that $o_j <^r_{\hat{H}} o_i$ ($<^r_{\hat{H}}$ is asymmetric). The precedence relation (the precedence in real-time) is transitive and asymmetric, therefore, we know that $DSG(\hat{H})$ is acyclic.

Since H is final-state commit-real-time linearizable, we know that there exists a sequential history S equivalent to some completion of H called $\bar{H}$, such that S is legal and S respects the commit-real-time order of $\bar{H}$.

Let us construct a history S' by stripping all aborted operation executions from S. This way S' is a sequential history consisting of only committed operation executions. Now let us construct a directed graph G' by extending the graph $G = DSG(\bar{H})$ in the following way. For any two operation executions o_i and o_j in S' such that $o_i <^r_{S'} o_j$, we add an edge from o_i to o_j to G'.

Claim 1. *G' is acyclic.*

Proof. Because G is acyclic we know that there are no two vertices u and v in G, such that u is reachable from v and v is reachable from u (there exists a path from u to v and from v to u). Let us call this the *no mutual reachability invariant*. The condition is necessary and sufficient for a graph to be acyclic. We have to show that G' also exhibits this property.

Because S respects the commit-real-time order of $\bar{H}$, we know that for any two committed operation executions o_i and o_j in S such that $o_i <^r_S o_j$, either $o_i <^r_{\bar{H}} o_j$, or o_i and o_j are concurrent in $\bar{H}$. Therefore, $o_j \not<^r_{\bar{H}} o_i$.

S' contains the same committed operation executions as S. Thus the above also holds for operation executions in S'. Let as assume that $S' = \langle o_1, o_2, ... \rangle$. Then, for any two operation executions o_i and o_j in S', such that $i < j$, o_i is not reachable from o_j in G. We call this the *path invariant*.

Let us consider the one-by-one insertion of edges between vertices induced by S'. We denote by G^k the modified graph G after k insertions. We start with $G^0 = G$, and assume that every G^k satisfies both, the no mutual reachability invariant, and the path invariant. The graph G' can be expressed as G^n, where n is the total number of edges induced by S'. Let us take an arbitrary edge from

o_i to o_j, where both o_i and o_j are in S', and $o_i <^r_{S'} o_j$ ($i < j$). We insert it into G^k. Now we show that the resulting G^{k+1} satisfies both invariants.

By the path invariant we know that o_i is not reachable from o_j in G^k. When an edge from o_i to o_j is inserted in G^k, then o_j becomes reachable from o_i in G^{k+1} (if it was not in G^k). However, the opposite is not true – o_i is still not reachable from o_j. Adding this edge also does not change anything in mutual reachability of other operation executions as we now show by contradiction. Assume o_p and o_q are not mutually reachable in G^k (either o_q is not reachable from o_p, or o_p is not reachable from o_q), but the new edge from o_i to o_j changes it in G^{k+1}. Without loss of generality let us assume that o_q is reachable from o_p in G^k, but that o_p becomes reachable from o_q only in G^{k+1} due to the insertion of a new edge. It means that there exists a directed path from o_q to o_p in G^{k+1} that includes the new edge from o_i to o_j. For that to be possible, o_i has to be reachable from o_q and o_p has to be reachable from o_j. This would mean that also o_q is reachable from o_j (since o_q is reachable from o_p), and in consequence also o_i is reachable from o_j. But we know that o_i is not reachable from o_j in G^{k+1}, a contradiction. Therefore, the no mutual reachability invariant holds.

Now let us assume that o_p and o_q belong to S', and that $p < q$. By the path invariant, o_p is not reachable from o_q in G^k. By the same reasoning as above, the insertion of an edge from o_i to o_j cannot make o_p reachable from o_q. Therefore, also the path invariant holds.

Since, every graph G^k satisfies the no mutual reachability invariant, G' is acyclic. □

Since G' is acyclic and includes an edge for every pair of preceding operation executions in $\bar{H}$, a topological sort of G' can only yield a serialization which maintains this precedence. Let $S_{G'}$ be a history obtained by performing a topological sort on G'. Then $S_{G'}$ is a sequential history that respects the real-time order of $\bar{H}$. Additionally, since G' contains all the operation executions of $\bar{H}$, so does $S_{G'}$. Thus, $S_{G'}$ is equivalent to $\bar{H}$.

Part 2. Proving the legality of $S_{G'}$.

We now show that the constructed history $S_{G'}$ is legal. We do so in three steps. First, we consider a history S'', which is constructed by removing from $S_{G'}$ all the aborted operation executions, and show that it is equivalent to S'. Then, we use Lemma 3 to prove that S' is legal. In consequence S'' is also legal. Finally, we use Lemma 3 again to prove that $S_{G'}$ is legal.

Claim 2. *S' and S'' are identical histories.*

Proof. First, we show that S' and S'' consist of the same operation executions. S' is constructed by removing all aborted operation executions from S. Since S is equivalent to $\bar{H}$ (a completion of H), S' contains all committed operation executions of $\bar{H}$. On the other hand, S'' is constructed by removing all aborted operation executions from $S_{G'}$, a history obtained by a topological sort of graph G'. G' is created by adding some edges to graph $G = DSG(\bar{H})$. Therefore, both

G and G' consist of all operation executions of $\bar{H}$. It means that S'', similarly to S', consists of all committed operation executions of $\bar{H}$.

Now we have to show that the order of operation executions in S' and S'' is the same. Since both histories are sequential, there exist total orders $<^r_{S'}$ and $<^r_{S''}$. We therefore have to prove that for any two (committed) operation executions o_i and o_j, $o_i <^r_{S''} o_j$ if $o_i <^r_{S'} o_j$.

We give a proof by a contradiction. Assume that $o_i <^r_{S'} o_j$ and $o_i \not<^r_{S''} o_j$. Since G' contains an edge from o_i to o_j, the result of a topological sort on G' can only yield a serialization in which o_i precedes o_j. Therefore, $o_i <^r_{S_{G'}} o_j$. S'' is constructed by removing aborted operation executions from $S_{G'}$, therefore it contains the same committed operation executions as $S_{G'}$ arranged in the same order. As S' contains only committed operation executions, both o_i and o_j are committed. Therefore, $o_i <^r_{S''} o_j$, a contradiction.

We have shown that S' and S'' consist of the same operation executions arranged in the same order. Therefore, S' and S'' are equal. □

From the assumptions we know that S is legal. Since S' is constructed by removing all aborted operations executions from S, then S' is legal as well (by Lemma 3). In consequence, S'' is also legal (from Claim 2). Since S'' is legal and S'' contains all the committed operations executions of $S_{G'}$ and none of its aborted operation executions, then also $S_{G'}$ is legal (by Lemma 3).

Since $S_{G'}$ is a sequential history equivalent to $\bar{H}$ (a completion of H), $S_{G'}$ is legal and $S_{G'}$ respects the real-time order of $\bar{H}$, H is final-state real-time linearizable. □

A.5 Relationship Between ⋄-Opacity and ⋄-Linearizability

Proposition 1. *Let H be some implementation history and let G be a gateway shared object of a TM object M. If an operation execution $G.perform(prog_k) \rightarrow_i v_k$ is completed (for some process p_i and transaction T_k) in $H|G$, then transaction T_k is t-completed in $H|M$.*

Proof. The execution of PERFORM($prog_k$) ends through calling the *return* statement in lines 8, 12, 15, or 16 of Algorithm 1. We now consider each case independently. In the first case, T_k is forcefully aborted during execution of operation $x.op$ (line 7). In the second case, T_k is aborted on demand as the result of execution of $M.tryA(T_k)$ in line 11. In the third and fourth cases, T_k is either aborted or committed, depending on the result of execution of $M.tryC(T_k)$ in line 14. Therefore, in either case T_k is completed in $H|M$. □

Proposition 2. *Let H be some implementation history and let G be a gateway shared object of a TM object M. If transaction T_k is commit-pending in $H|M$, then the function* PERFORM($prog_k$) *completed execution of all steps of $prog_k$.*

Proof. Transaction T_k becomes commit-pending after executing $M.tryC(T_k)$ in line 14. It means that the function FETCHNEXTSTEP in line 4 returned $\perp$, thus signaling that there are no more steps left to complete in $prog_k$. □

Proposition 3. *Let H be some implementation history, G be a gateway shared object of a TM object M and $prog_k$ be some algorithm executed by invoking an operation execution $o_k = G.perform(prog_k) \rightarrow_i v_k$ (for some process p_i). The state of any t-object $x \in \mathcal{Q}$ does not change if o_k aborts.*

Proof. Abort of o_k is indicated by the return value $\perp$. Function PERFORM returns $\perp$ in three cases:

1. when an operation on a t-object aborts (line 7),
2. when $prog_k$ requests an abort on demand (line 11),
3. when $prog_k$'s request to commit the transaction T_k ends with failure (line 14).

In all three cases M aborts T_k, thus revoking any changes to any t-objects performed by T_k. □

Proposition 4. *Let H be some implementation history, G be a gateway shared object of a TM object M and $prog_k$ be some algorithm executed by invoking an operation execution $o_k = G.perform(prog_k) \rightarrow_i v_k$ (for some process p_i). Then, for transaction $T_k \in H|M$ all events of T_k occur in H after the invocation event of o_k and before the response event of o_k, if such an event exists in H.*

Proof. Trivially, if T_k has no operations, i.e., $(H|M)|T_k$ is empty, then the proposition is satisfied. Let us consider the case where $(H|M)|T_k$ is not empty. The first event of transaction T_k is the invocation of the first operation $M.texec(T_k, x.op)$ (line 7) executed within the function PERFORM$(prog_k)$. Trivially, this event must appear after the invocation of o_k. Moreover, from Proposition 1, we know that T_k must be completed in $H|M$ before o_k can return. From the definition of a transaction, we know that T_k cannot execute any other operations after it completes. Therefore, all events of T_k always appear after invocation of o_k and before the response event of o_k, if such a response event exists in a given execution. □

Proposition 5. *Let H be some implementation history and let G be a gateway shared object of a TM object M. If there exists a (possibly pending) operation execution $o_k = G.perform(prog_k) \rightarrow_i v_k$ in H for some process p_i and $(H|M)|T_k$ is not empty then T_k is executed by p_i.*

Proof. A process p_i that invokes the function PERFORM$(prog_k)$, executes it without any help from other processes until the function returns. Since a transaction T_k can only exist through execution of $prog_k$, all operations of T_k on M are issued by p_i. □

Proposition 6. *Let H be some implementation history and let G be a gateway shared object of a TM object M. If there exists a committed operation execution $o_k = G.perform(prog_k) \rightarrow_i v_k$ in H for some process p_i, then transaction T_k is committed in $H|M$.*

Proof. Operation execution o_k is committed if it returns a value $v_k \neq \perp$. That only happens if operation $M.tryC(T_k)$ (line 14) return C_k (T_k is successfully committed). □

Proposition 7. *Let H be some implementation history and let G be a gateway shared object of a TM object M. If there exists a committed operation execution $o_k = G.perform(prog_k) \rightarrow_i v_k$in H for some process p_i such that o_k is potentially updating, then transaction T_k is committed and potentially updating in $H|M$.*

Proof. From Proposition 6 we know that T_k has to be committed since o_k is committed. Since o_k is potentially updating, there exist a state for which o_k is updating. It means that potentially o_k may change the state of G. In this case, T_k would perform some updating operations. It means that the program $prog_k$ which defines T_k may produce an execution in which T_k contains some updating operations. Therefore, T_k cannot be declared-read-only, and thus it is potentially updating in $H|M$. □

Now let us introduce some auxiliary definitions that are necessary for the following proposition and the proof of the theorem that comes after the proposition.

Let H be an implementation history and let G be a gateway shared object of M. If there exists a t-sequential t-history S_M equivalent to some t-completion of $H_M = H|M$, such that S_M respects the $\diamond$-order of H_M and every transaction in S_M is legal in S_M, then we can construct a history S_G of events on G in the following way.

Let us first construct a completion $\bar{H}_G$ of the history $H_G = H|G$ by completing every pending operation execution in H_G with an appropriate response, as follows. For every pending operation execution o_k of operation $G.perform(prog_k)$ in H_G, there are two cases possible:

1. if T_k is aborted in S_M, then the response event for o_k in $\bar{H}_G$ is equal $\perp$, otherwise
2. the response value of o_k in $\bar{H}_G$ is equal to the final value of $context$ from the execution of $prog_k$ in H (after p_i executes $M.tryC(T_k)$).

Naturally, $\bar{H}_G$ contains all the operation executions which are completed in H_G. Note that if o_k is completed in H_G, by Proposition 1, T_k is t-completed in H_M. On the other hand, if o_k is pending, then T_k may be t-completed or not. If T_k is live in H_M, it is either aborted (case 1) or committed (case 2) in S_M. However, T_k may also be t-completed in H_M but o_k may simply lack a response event in H_G (cases 1 and 2, depending on whether T_k is aborted or committed). In case 2 we know that T_k is either commit-pending or committed in H_M. Therefore, by Proposition 2 we know that p_i completed all steps of $prog_k$ and o_k can complete successfully and return the final value of $context$.

Now, let us construct a sequential completed history S_G equivalent to $\bar{H}_G$. Let S_G contain all operation executions of $\bar{H}_G$, but sequentially ordered according to the order of the corresponding transactions in S_M. An operation execu-

tion $o_k = G.perform(prog_k) \rightarrow_i v_k$ in $\bar{H}_G$ corresponds to transaction T_l in S_M, for some process p_i if $k = l$.

We say that S_G is *induced* by S_M.

For any transaction T_k let us denote by V_k the history $visible_{S_M}(T_k)$ in case T_k is committed. Otherwise, let V_k be equal the history $visible_{S_M}(T_k)$ with T_k omitted. Since, every transaction T_k in S_M is legal in S_M, the history $visible_{S_M}(T_k)$ is t-legal and so is V_k (by the definition of a t-legal history, each one of its prefixes is also t-legal). For any t-object x the subhistory $V_k|x$ satisfies the sequential specification of x. It means that there exists a sequence $w_{k,x} = \langle q_0, q_1, ...q_l \rangle$ of states in Q_x, where Q_x is the set of all possible states of x, and q_l is the last state that x takes in V_k. Let us denote by $state_{S_M}(T_k)$ the combined state that includes every last state of all t-objects in V_k according to the sequence $w_{k,x}$.

Let q_i be the combined state of all the t-objects in M after the execution of the i-th transaction in S_M, i.e. $q_i = state_{S_M}(T_k)$, where T_k is the i-th transaction in S_M. We say that the sequence $W = \langle q_0, q_1, ... \rangle$ is *state-induced* by S_M. It can be shown (as we demonstrate later) that every such sequence is a witness history of the induced history S_G (witness history is defined in Section 4.4).

Proposition 8. *Let H be some implementation history and let G be a gateway shared object of a TM object M. Let S_G be a sequential history equivalent to $\bar{H}_G$, a completion of $H_G = H|G$, such that S_G is induced by some history S_M, and let W_G be its witness history state-induced by S_M. If there exists a committed operation execution $o_k = G.perform(prog_k) \rightarrow_j v_k$ in $\bar{H}_G$ (for some process p_j), such that o_k is updating in S_G according to W_G, then transaction T_k is committed and is updating in $H|M$.*

Proof. From Proposition 6 we know that T_k has to be committed since o_k is committed. Since o_k is updating in S_G according to W_G, the states q_{i-1} and q_i in W_G, which correspond to o_k (o_k is the i-th operation execution in S_G) are different. Since the states q_{i-1}, q_i in W_G are the combined states of all the t-objects in M after the execution of transaction T_k, the state of some t-object x had to be changed by T_k (according to the sequence $w_{k,x}$). This can only happen if T_k executed a not read-only operation in S_M. This operation has to be also present in $H|M$. Therefore, T_k is updating in $H|M$. □

Theorem 5. *Let M be a TM object and let G be a gateway shared object of M. If M is ⋄-opaque then G is ⋄-linearizable.*

Proof. In order to show that G is ⋄-linearizable, we need to prove that any finite history produced by G is final-state ⋄-linearizable (by Corollary 2). Therefore, we assume some finite implementation history H, and show how to construct a sequential history S_G of events on G that is equivalent to $\bar{H}_G$, a completion of $H_G = H|G$. Next, we prove that S_G is legal. Finally, we show that S_G respects the ⋄-order of $\bar{H}_G$.

Part 1. Construction of a sequential history S_G that is equivalent to a completion of $H|G$.

Since M is ⋄-opaque, there exists a t-sequential t-history S_M such that S_M is equivalent to some t-completion of $H_M = H|M$, S_M respects the ⋄-order of H_M

and every transaction T_k in S_M is legal in S_M. Therefore, there exists a history S_G induced by S_M.

Part 2. Proof of legality of S_G.

From assumptions we know that every transaction in S_M is legal, thus also the last committed transaction T_l in S_M is legal. Therefore, $vis_l = visible_{S_M}(T_l)$ contains all the committed transactions in S_M (see the definition of $visible_S(T_k)$ in Section 4.3). Moreover, vis_l is t-legal. It means, that for any t-object x, $vis_l|x$ satisfies the sequential specification of x.

Let $steps_H(o_k)$, where H is any implementation history and o_k is a (possibly pending) operation execution of operation $G.perform(prog_k)$, be a function which returns all the events in H executed within the operation execution o_k (excluding the operations on G itself), which are related to the execution of $prog_k$. Therefore, $steps_H(o_k)$ contains all the steps of $prog_k$'s execution.

Let S'_G be a history obtained from S_G by removing all the aborted operation executions and assume S'_G is a sequence of operation executions $\langle o_1 \cdot o_2 \cdot ... \rangle$. We construct a sequence $\alpha = \langle steps_H(o_1) \cdot steps_H(o_2) \cdot ... \rangle$. Now we show that $\alpha|M = vis_l$ by a contradiction.

Assume that $\alpha|M \neq vis_l$. It means that there exists $o_k = G.perform(prog_k) \rightarrow_i v_k$ in H such that $steps_H(o_k)|M \neq S_M|T_k$. There are two cases to consider:

1. $\exists op \in H : op \in steps_H(o_k)|M \wedge op \notin S_M|T_k$. It means that op was defined in $prog_k$ and was executed by p_i through G and then M as part of transaction T_k (by definition of G). Therefore, $op \in S_M|T_k$, a contradiction.
2. $\exists op \in H : op \in S_M|T_k \wedge op \notin steps_H(o_k)|M$. It means that op was executed by p_i on M within transaction T_k. The only possibility that it happened is by specifying op as part of $prog_k$ (by definition of G). Therefore, $op \in steps_H(o_k)|M$, a contradiction.

Since both cases lead to a contradiction, the initial assumption was false and $\alpha|M = vis_l$.

Since for any t-object x, $vis_l|x$ satisfies the sequential specification of x, so does $\alpha|M$. Note that any $prog_k$ executed through G operates in isolation and may interact with other processes only through t-objects (G allows only local computation besides operations on M). However, all operations on t-objects induced by executing $G.perform(prog_k)$ in S'_G satisfy the sequential specifications of these t-objects. Therefore, the execution of algorithms in S'_G transitions t-objects in M from some initial state $q_0 \in Q$, through states $q_1, q_2, ...$ to some correct state $q' \in Q$, thus satisfying the sequential specification of G. Therefore S'_G is legal and $W'_G = \langle q_0, q_1 ... \rangle$ is its witness history.

By Proposition 3, aborted operation executions removed from S_G to produce S'_G do not change the state of G. Therefore, also S_G is legal. If we extend the sequence W'_G with a repeated state for every aborted operation execution, we obtain W_G, a witness history of S_G.

Note that W_G is state-induced by S_M, because for each $q_i \in W_G$, q_i is the combined state of all the t-objects in M after the execution of the i-th transaction in S_M (and thus also after the execution of i-th operation execution in S_G).

Part 3. Proof that S_G respects the $\diamond$-order of $\bar{H}_G$.

To show that S_G respects the $\diamond$-order of $\bar{H}_G$, we have to show that for any two operation executions o_i and o_j on G (o_i invokes $G.perform(prog_i)$, while o_j invokes $G.perform(prog_j)$) such that $o_i <^{\diamond}_{\bar{H}_G} o_j$, the relation $o_i <^{\diamond}_{S_G} o_j$ holds as well. In case of update-real-time, we have to show that there exists a witness history W_G, such that for all o_i and o_j such that $o_i <^{u}_{\bar{H}_G} (S_G, W_G) o_j$, the relation $o_i <^{u}_{S_G} (S_G, W_G) o_j$ holds (W_G was given in the previous part of the proof).

If $<^{\diamond}_{\bar{H}_G} = <^{a}_{\bar{H}_G}$ then there are no operation executions o_i and o_j, such that $o_i <^{a}_{\bar{H}_G} o_j$ (by definition $<^{a}_{\bar{H}_G}$ is equivalent to $\emptyset$).

Now we consider $<^{\diamond}_{\bar{H}_G} \neq <^{a}_{\bar{H}_G}$. Note that every order definition apart from arbitrary order assumes precedence in real-time between the two operation executions (besides making some additional requirements on the operation executions). Therefore, we now assume that $o_i <^{r}_{\bar{H}_G} o_j$. By definition of real-time order, it means that the response event of o_i precedes the invocation event of o_j in $\bar{H}_G$. By Proposition 4, all events of T_i precede all events of T_j in H. In particular, the last event of T_i precedes the first event of T_j in H. Therefore T_i completed in H_M before T_j started in H_M. It means that if $<^{\diamond}_{\bar{H}_G} = <^{r}_{\bar{H}_G}$ then $T_i \prec^{r}_{\bar{H}_G} T_j$. In case $<^{\diamond}_{\bar{H}_G} \neq <^{r}_{\bar{H}_G}$, additionally:

- if $<^{\diamond}_{\bar{H}_G} = <^{c}_{\bar{H}_G}$ then both o_i and o_j are committed or both are executed by the same process. It means that both T_i and T_j are committed or both are executed by the same process (by Propositions 5 and 6). Thus $T_i \prec^{c}_{H_M} T_j$.
- if $<^{\diamond}_{\bar{H}_G} = <^{w}_{\bar{H}_G}$ then both o_i and o_j are potentially updating and committed or both are executed by the same process. It means that both T_i and T_j are committed and are potentially updating or both are executed by the same process (by Propositions 5 and 7). Thus $T_i \prec^{w}_{H_M} T_j$.
- if $<^{\diamond}_{\bar{H}_G} = <^{u}_{\bar{H}_G} (S_G, W_G)$ then both o_i and o_j are updating in S_G according to W_G and are committed, or both are executed by the same process. It means that both T_i and T_j are committed and updating or both are executed by the same process (by Propositions 5 and 8). Thus $T_i \prec^{u}_{H_M} T_j$.
- if $<^{\diamond}_{\bar{H}_G} = <^{p}_{\bar{H}_G}$ then both o_i and o_j are executed by the same process. It means that both T_i and T_j are executed by the same process (by Proposition 5). Thus $T_i \prec^{p}_{H_M} T_j$.

Since there is no ambiguity between naming convention of orders in $\diamond$-opacity and $\diamond$-linearizability, we can simply write that $T_i \prec^{\diamond}_{H_M} T_j$.

Since S_M respects the $\diamond$-order of H_M then also $T_i \prec^{\diamond}_{S_M} T_j$. In turn, the response event of o_i precedes the invocation event of o_j in S_G, because the construction of S_G requires an order of operation executions in S_G that directly follows the order of transactions in S_M. Therefore, S_G respects the $\diamond$-order of $\bar{H}_G$ (we proved the case for $<^{a}_{\bar{H}_G}$ earlier). This way we conclude the proof. □

Corollary 3. *Let M be a TM object and let G be a gateway shared object of M. If M is commit-real-time opaque, then G is real-time linearizable.*

Proof. The proof follows directly from Theorems 4 and 5. □

B
Proofs of algorithm correctness

In this chapter we give the proofs of correctness for all replication schemes discussed in this dissertation.

B.1 Correctness of State Machine Replication

In order to reason about the correctness of a SMR/LSMR system (or the SMR/LSMR algorithm in short), we model it as a shared object, which exports a set of operations (i.e., it has a well defined sequential specification), and consider its history H. A client request issued to any replica is mapped into an operation invocation. Analogically, a client response is mapped into an operation response. A pair of request submission (by a client) and return of a response (to the client) we treat as a (completed) operation execution executed by the process that received the request. For easier comprehension, we sometimes refer to execution of an operation on SMR/LSMR as to execution of a request (submitted by a client). A request (operation) r is *read-only*, if $r.prog$ is read-only (see also Section 3.5). Otherwise r is potentially updating. Note that in SMR/LSMR all (or some) requests are executed by all processes but an operation execution of a request is always executed by one process (i.e., the process which received the client request).

In the following discussion by S_k^i, a (local) state of some process p_i, we understand the service state q_k of p_i, i.e., the combined state of all objects maintained by p_i and related to the replicated service itself, and the current values of additional variables maintained by p_i, such as LC. Note that in case of SMR, which does not feature additional variables, a state of a process is the service state of this process.

Let r be a request executed by some process p. Then, let $timestamp(p, r)$ be a function that returns a value (integer) corresponding to the number of messages delivered through TOB by p, i.e., if r is the k-th request delivered by p (in line 5)

then $timestamp(p, r) = k$. Later we show that for a request r, and for any two processes p_i and p_j that executed r, $timestamp(p_i, r) = timestamp(p_j, r)$. Therefore, later, when it is not ambiguous, we use simpler notation $timestamp(r)$.

Proposition 9. *Let both K_1 and K_2 be executions of any two requests by SMR (line 6) but K_1 and K_2 pertain to different operation executions (i.e., K_1 and K_2 pertaining to different requests). For any process p_i executing SMR, K_1 and K_2 never interleave.*

Proof. Every request is processed as a non-preemptable event (line 6). Therefore, the execution of K_1 and K_2 on any process never interleaves. □

Proposition 10. *Let p_i be a process executing SMR. Let r be a request delivered by p_i using TOB, let S be the state of p_i at the moment of delivery of r and S' be the state of p_i after execution of r. For every process p_j in state S, if p_j delivers r using TOB then execution of r by p_j transfers p_j to state S'.*

Proof. By Proposition 9, the state of p_k does not change throughout the execution of r.

Since SMR assumes that only a request with deterministic $prog$ can be executed, $r.prog$ must be deterministic. Both processes execute $r.prog$ with the same $r.args$ (line 6) and operate on the same state S which does not change throughout the execution of $r.prog$.

Thus both processes move to the same state, i.e., S'. □

Proposition 11. *Let $S(i) = (S_0^i, S_1^i, ...)$ be a sequence of states of a process p_i executing SMR, where S_0^i is the initial state (comprising of the initial state of all objects managed by SMR) and S_k^i is the state after the k-th request was delivered using TOB and processed by p_i. For every pair of processes p_i and p_j, either $S(i)$ is a prefix of $S(j)$, or $S(j)$ is a prefix of $S(i)$.*

Proof. We prove the proposition by a contradiction. Let us assume that $S(i)$ and $S(j)$ differ on position k (and k is the lowest number for which $S_k^i \neq S_k^j$), thus neither is a prefix of another.

If $k = 0$, then the initial state of p_i is different from the initial state of p_j. Since all processes start with same objects with the same initial values, that is a contradiction. Therefore $k > 0$, and the difference between S_k^i and S_k^j must stem from some later change to the state.

Since $S_{k-1}^i = S_{k-1}^j$, the receipt of the k-th request r_k (which is equal for both processes due to the properties of TOB) and processing it resulted in a different change of values of some (or all) objects managed by SMR. Since both processes are in the same state S_{k-1} and execute the same request, by Proposition 10, both processes move to the same state S_k, a contradiction. Therefore the assumption is false and either $S(i)$ is a prefix of $S(j)$, or $S(j)$ is a prefix of $S(i)$. □

Proposition 12. *Let H be a history of SMR. Let o be a completed operation execution in H of some request r. Then, for any two processes p_i and p_j which delivered r using TOB and subsequently executed r, $timestamp(p_i, r) = timestamp(p_j, r)$.*

Proof. State of any process p can change only as a result of execution of a request (line 6). By properties of TOB, both p_i and p_j must have delivered using TOB the same set of requests prior to delivery of r. Therefore $timestamp(p_i, r) = timestamp(p_j, r)$ (the $timestamp$ function maps a request to the number of messages so far delivered through TOB by the process). □

From now on, we can use simplified notation $timestamp(r)$.

Theorem 6. *State Machine Replication satisfies real-time linearizability.*

Proof. In order to prove that SMR satisfies real-time linearizability, we have to show that every finite history produced by SMR is final-state real-time linearizable (by Corollary 2). In other words, we have to show that for a finite history H produced by SMR, there exists a sequential legal history S equivalent to some completion $\bar{H}$ of H, such that S respects the real-time order of $\bar{H}$.

Part 1. Construction of a sequential history S that is equivalent to a completion of H.

Let us first construct a completion $\bar{H}$ of H. We start with $\bar{H} = H$. For every operation execution o resulting from an invocation of some request r, we complete o in $\bar{H}$, only if there exists a replica which finished the execution of line 6. Because of this requirement, we know that there exists an appropriate response that can be returned to the client. We use this value to complete o in $\bar{H}$. On the other hand, if there does not exist a replica which completed execution of line 6, we remove o from $\bar{H}$.

By Proposition 12, we know that the value of $timestamp(r)$ of some request r is the same across all processes that delivered r. Now we show that $timestamp$ can be used to uniquely identify all operation executions in $\bar{H}$. We conduct the proof by a contradiction. Let o_i and o_j be two (completed) operation executions in $\bar{H}$ which result from delivery of some requests r_i and r_j, respectively. Assume that $o_i \neq o_j$, and $timestamp(r_i) = timestamp(r_j)$. Since both operation executions are completed, both r_i and r_j had to be delivered by some processes. By properties of TOB, we know that there exists a process p that delivers both r_i and r_j. Without loss of generality, let us assume that p delivers r_i before r_j. By Proposition 9, we know that processing of r_i and r_j cannot interleave and the execution of $r_i.prog$ and $r_j.prog$ on p happens in the order of delivery of r_i and r_j. Naturally then by definition of the $timestamp$ function, $timestamp(r_i) \neq timestamp(r_j)$, a contradiction. Therefore, $timestamp(r)$ uniquely identifies request r.

Let $\mathcal{R}$ be a set comprising all operation executions in $\bar{H}$. Now let us construct the following function $update$. Let $update : \mathbb{N} \to \mathcal{R}$ be an inverse function to the function $timestamp$. In other words, $update$ maps $timestamp(r)$, i.e., the number of requests so far delivered through TOB by the process, to the operation execution pertaining to said request r.

Now we construct a sequential history S. Let $S = \langle update(1) \cdot update(2) \cdot ... \rangle$. This way S includes the operations of all operation executions in $\bar{H}$. The order of operation executions in S directly corresponds to the order in which every process executes requests (pertaining to said operation executions).

Part 2. Proof that S respects the real-time order of $\bar{H}$.

Let o_i and o_j be any two operation executions of some requests r_i and r_j such that $o_i <^r_{\bar{H}} o_j$ (the invocation event of o_j appears in $\bar{H}$ after the response event of o_i). By Proposition 17, we know that $timestamp(r_i) < timestamp(r_j)$. Then, by construction of S, o_i has to appear in S before o_j. Therefore, $o_i <^r_S o_j$. This way S respects the real-time order of $\bar{H}$ (trivially, for any operation execution o_k executed by process p_i in $\bar{H}$, o_k is executed by p_i in S).

Part 3. Proof that S is legal.

Let $(Q, q_0, INV, RES, \delta)$ be the sequential specification of the replicated service. In order to show that S is legal, we have to show that there exists a witness history $W = \langle q_0, q_1, ... \rangle$ of service states in Q such that for every k $(q_{k-1}, r_k, v_k, q_k) \in \delta$, where r_k is some request and v_k is a response obtained as a result of executing r_k.

Let o_k be any operation execution of some request r_k in S. Then we can make two observations about o_k. The first observation is that by Proposition 9 we know that throughout the execution of r_k the changes to the state of (any) process that executes r_k may result only from execution of r_k. It means that the response computed as a result of execution of r_k depends only on two things: (1) on the state of the process that executes r_k just prior to execution of r_k,[1] and (2) on r_k itself. Since by Proposition 11 all processes change their states in the same way, we can say that the response computed as a result of the execution of r_k depends on q_{k-1} and r_k itself.

The second observation regards $\bar{H}$ which is equivalent to S. By construction of $\bar{H}$, any operation execution o_k of a request r_k in $\bar{H}$ is either completed in H or there exists a process which finished the execution of r_k. Therefore there exists an appropriate response v_k to r_k. Also by Proposition 9 we know that throughout the execution of r_k, the state of the process that executed r_k does not change for reasons other than the execution of r_k itself. Therefore there exists a service state q_k which solely depends on the execution of r_k on state q_{k-1} (q_k is equal to the service state of the process that finished execution of r_k, line 6).

Now let us construct the witness history W. Let $W = \langle q_0 \rangle$, where q_0 is the initial service state of any process p_i (processes share the initial service state). Then, for any operation execution o_k in S of request r_k, let us append to W the service state q_k which is equal to the service state of any process that finished the execution of r_k.[2]

Because for all operation executions in S we have requests, matching responses, the sequence of the service states before and after requests' execution, and we know that all these requests were executed sequentially (by Proposition 9), then for every k such that r_k is a potentially updating request $(q_{k-1}, r_k, v_k, q_k) \in \delta$. Hence W is the witness history of S and S is legal. This way we conclude the proof that SMR satisfies real-time linearizability.

[1] Naturally there there are multiple such processes and each executes r_k independently.

[2] By Proposition 11 all processes change their states in the same way as a result of execution of requests. Hence also their service states change in the same way, because a service state is a part of a state of a process. Therefore in this reasoning we can consider the service state of any process after the execution of r_k.

□

B.2 Correctness of State Machine Replication with Locks

Theorem 7. *State Machine Replication with Locks does not satisfy real-time linearizability.*

Proof. We give a proof by a contradiction. Let us assume that LSMR satisfies real-time linearizability. It means that every history produced by LSMR is real-time linearizable. Now consider an example in Figure B.1, which shows a valid execution of LSMR implementing a simple banking system which processes two client requests: r_1 (handled by p_1) and r_2 (handled by p_3). Prior to the execution of r_1 and r_2, the balances of accounts A and B are equal $\$1000$.

Request r_1 is potentially updating (it realizes a transfer of $\$100$ between accounts A and B) and therefore it is first broadcast to all processes using TOB (line 8). Subsequently, p_1 and p_2 deliver r_1 (line 10) and execute it (line 12).[3] After the execution of r_1 the balance of account A is $\$900$ and the balance of account B is $\$1100$.

Request r_1 sent from p_1 to p_3 takes more time to reach its destination. Before the reception of the message (but after p_1 and p_2 finished the execution of r_1), p_3 receives and executes a read-only request r_2 which checks the balance of account A which is equal $\$1000$. Since r_2 is read-only, it can be executed without inter-process synchronization (line 6).

Naturally, r_1 precedes r_2 in real-time. However, the only legal serialization of r_1 and r_2 is $\langle r_2, r_1 \rangle$. It means that this execution is not real-time linearizable and thus LSMR does not satisfy real-time linearizability. □

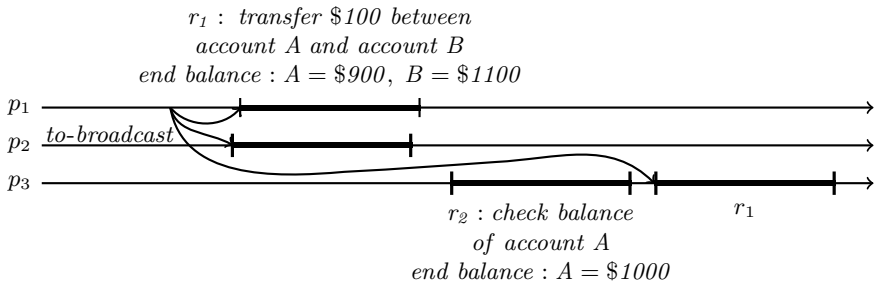


Figure B.1: A valid execution of LSMR which is not real-time linearizable: r_1 is a potentially updating request, whereas r_2 is a read-only request.

Corollary 8. *State Machine Replication with Locks does not satisfy commit-time linearizability.*

[3] Typically a message broadcast using TO-BROADCAST (e.g., based on the Paxos algorithm [62]) can be delivered once the majority of processes receive the message.

Proof. The proof follows directly from Theorem 7 and definitions of real-time linearizability and commit-real-time linearizability. □

Let r be a request executed by some process p. Then, let $timestamp(p, r)$ equal the value of LC during the execution of r on some process p. Later we show that for a potentially updating request r, and any two processes p_i and p_j who executed r, $timestamp(p_i, r) = timestamp(p_j, r)$. Therefore, later, when it is not ambiguous, we use a simpler notation $timestamp(r)$. We also use it for read-only requests (naturally, then there is only a single process p_i such that $timestamp(p_i, r) = timestamp(r)$).

Proposition 13. *Let k_a and k_b be such that:*

1. *k_a is an execution of a potentially updating request by LSMR,*
2. *k_b is an execution of a read-only request by LSMR.*

Let K_1 and K_2 be either k_a or k_b but K_1 and K_2 are not both k_b. Also let K_1 and K_2 pertain to different operation executions (i.e., to different requests). For any process p_i executing LSMR, K_1 and K_2 never interleave.

Proof. We have three cases to consider:

1. Both K_1 and K_2 pertain to two different potentially updating requests r_k and r_l. Since the execution of a potentially updating request is guarded with a readers-writer lock used in the writers mode (line 12), p_i cannot execute r_k and r_l concurrently.
2. K_1 pertains to a potentially updating request r_k and K_2 pertains to a read-only request r_l. Execution of the requests is protected with a readers-writer lock. However, the lock is used in different modes to execute r_k and r_l (lines 12 and 6). By properties of readers-writer lock, p_i cannot execute r_k and r_l concurrently.
3. K_1 pertains to a read-only request r_k and K_2 pertains to a potentially updating request r_l. Analogically as in case 2, p_i cannot execute r_k and r_l concurrently.

Therefore, the execution of K_1 and K_2 on any process never interleaves. □

Proposition 14. *Let p_i be a process executing LSMR. Let r be a potentially updating request delivered by p_i using TOB, let S be the state of p_i at the moment of delivery of r and S' be the state of p_i after the execution of r. For every process p_j in state S, if p_j delivers r using TOB then the execution of r by p_j yields state S' of p_j.*

Proof. By Proposition 13, the state of p_k does not change throughout the execution of r.

Since both p_i and p_j start the execution of r from the same state, the current values of their LC variables are equal. Then both processes increment LC and so both LC variables have still the same value.

Since LSMR assumes that only a request with deterministic $prog$ can be executed, $r.prog$ must be deterministic. Both processes execute $r.prog$ with the same

$r.args$ (line 12) and operate on the same state S which does not change throughout the execution of $r.prog$.

Thus both processes move to the same state, i.e., S'. □

Proposition 15. *Let $S(i) = (S_0^i, S_1^i, ...)$ be a sequence of states of a process p_i executing LSMR, where S_0^i is the initial state (comprising of the initial state of all objects managed by LSMR and $LC = 0$) and S_k^i is the state after the k-th (potentially updating) request was delivered using TOB and processed by p_i. For every pair of processes p_i and p_j, either $S(i)$ is a prefix of $S(j)$, or $S(j)$ is a prefix of $S(i)$.*

Proof. We prove the proposition by a contradiction. Let us assume that $S(i)$ and $S(j)$ differ on position k (and k is the lowest number for which $S_k^i \neq S_k^j$), thus neither is a prefix of another.

If $k = 0$, then the initial state of p_i is different from the initial state of p_j. Since all processes start with the same values of LC (line 1) and maintain the same objects with the same initial values, that is a contradiction. Therefore $k > 0$, and the difference between S_k^i and S_k^j must stem from some later change to the state.

Since $S_{k-1}^i = S_{k-1}^j$, the receipt of the k-th request r_k (which is equal for both processes due to the properties of TOB) and processing it resulted in a different change to LC at both processes or a different change of values of some (or all) objects managed by LSMR. Since both processes are in the same state S_{k-1} and execute the same request, by Proposition 14, both processes move to the same state S_k, a contradiction. Therefore the assumption is false and either $S(i)$ is a prefix of $S(j)$, or $S(j)$ is a prefix of $S(i)$. □

Proposition 16. *Let H be a history of LSMR. Let o be a completed potentially updating operation execution in H of some request r. Then, for any two processes p_i and p_j which delivered r using TOB and subsequently executed r, $timestamp(p_i, r) = timestamp(p_j, r)$.*

Proof. State of any process p can change only as a result of the execution of a potentially updating request (line 12). By properties of TOB, both p_i and p_j must have delivered using TOB the same set of potentially updating requests prior to the delivery of r. Therefore, by Proposition 15, both p_i and p_j are in the same state prior the execution of r. It means that the value of LC on both processes is the same. Then, both processes increment their LC variables (in line 11) and subsequently execute r. It means that $timestamp(p_i, r) = timestamp(p_j, r)$ (the $timestamp$ function maps a request to the value of LC during the execution of the request on a given process). □

From now on, we can use simplified notation $timestamp(r)$.

Proposition 17. *Let H be a history of LSMR. For any two completed potentially updating operation executions o_i and o_j (in H) of some requests r_i and r_j, if $o_i <_H^r o_j$ then $timestamp(r_i) < timestamp(r_j)$.*

Proof. Since both operation executions are completed and potentially updating, both r_i and r_j had to be delivered by some processes. By properties of TOB, we

know that there exists a process p that delivers both r_i and r_j. Since $o_i <^r_H o_j$, we know that o_i is completed and the invocation event of o_j appears in H after the response event of o_i. Therefore, r_i had to be broadcast using TOB before r_j was broadcast. It means that p first incremented LC as a result of delivery of r_i thus changing the value of LC to $timestamp(p, r_i)$ (the $timestamp$ function maps a request to the value of LC during the execution of the request on a given process). Later p incremented LC again as a result of delivery of r_j thus changing the value of LC to $timestamp(p, r_j)$. LC increases monotonically, therefore $timestamp(p, r_i) < timestamp(p, r_j)$. Then, by Proposition 16 and using a simpler notation, we can stipulate that $timestamp(r_i) \leq timestamp(r_j)$. □

Proposition 18. *Let H by a history of LSMR. For any two operation executions o_i and o_j (in H) of some requests r_i and r_j such that both r_i and r_j are executed by some process p_l, if $o_i <^r_H o_j$ then $timestamp(r_i) \leq timestamp(r_j)$.*

Proof. From the assumption that $o_i <^r_H o_j$ and both are executed by the same process p_l, we know that o_i is completed and the invocation event of o_j appears in H after the response event of o_i. Since the value of LC increases monotonically (line 11) and the $timestamp$ function maps a request to the value of LC during the execution of the request on a given process, $timestamp(p_l, r_i) \leq timestamp(p_l, r_j)$. Then, by Proposition 16 and using simpler a notation, we can stipulate that $timestamp(r_i) \leq timestamp(r_j)$. □

Theorem 8. *State Machine Replication with Locks satisfies write-real-time linearizability.*

Proof. In order to prove that LSMR satisfies write-real-time linearizability, we have to show that every finite history produced by LSMR is final-state write-real-time linearizable (by Corollary 2). In other words, we have to show that for a finite history H produced by LSMR, there exists a sequential legal history S equivalent to some completion $\bar{H}$ of H, such that S respects the write-real-time order of $\bar{H}$.

Part 1. Construction of a sequential history S that is equivalent to a completion of H.

Let us first construct a completion $\bar{H}$ of H. We start with $\bar{H} = H$. Next, we remove from $\bar{H}$ any pending read-only operation executions, as they could not possibly modify the local or replicated state. Now, for every potentially updating pending operation execution o resulting from an invocation of some request r, we complete o in $\bar{H}$, only if there exists a replica which released the lock after the execution of r (line 12). Because we require that the lock is released, we know that r was successfully finished and there exists an appropriate response that can be returned to the client. We use this value to complete o in $\bar{H}$. On the other hand, if there does not exist a replica which released the lock after executing r, we remove o from $\bar{H}$.

By Proposition 16 we know that the value of $timestamp(r)$ of some request r is the same across all processes. Now we show that $timestamp$ can be used to uniquely identify potentially updating operation executions in $\bar{H}$. We conduct

the proof by a contradiction. Let o_i and o_j be two (completed) potentially updating operation executions in $\bar{H}$ which result from delivery of some requests r_i and r_j, respectively. Assume that $o_i \neq o_j$, and $timestamp(r_i) = timestamp(r_j)$. Since both operation executions are completed, both r_i and r_j had to be delivered by some processes. By properties of TOB, we know that there exists a process p that delivers both r_i and r_j. Without loss of generality, let us assume that p delivers r_i before r_j. By Proposition 13, we know that processing of r_i and r_j cannot interleave and execution of $r_i.prog$ and $r_j.prog$ on p happens in the order of delivery of r_i and r_j. Every time p executes a request, p first increments LC (line 11). Therefore, in the course of the execution of r_i and r_j, p increments LC twice, $timestamp(r_i) \neq timestamp(r_j)$, a contradiction. Therefore, $timestamp(r)$ uniquely identifies request r.

Let $\mathcal{R}$ be a set comprising all operation executions in $\bar{H}$. Now let us construct the following function $update$. Let $update : \mathbb{N} \to \mathcal{R}$ be an inverse function to the function $timestamp$ but defined only for potentially updating operation executions. In other words, $update$ maps $timestamp(r)$, i.e., the value of LC when a potentially updating request r is executed, to the operation execution pertaining to said request r.

Now we construct a sequential history S. Let $S = \langle update(1) \cdot update(2) \cdot ... \rangle$. This way S includes the operations of all completed potentially updating operation executions in $\bar{H}$. Let us add the rest of operation executions from $\bar{H}$ to S in the following way. For every such an operation execution o_k of some request r_k (with $timestamp(r_k)$), find in S a completed potentially updating operation execution o_l of a request r_l, such that $timestamp(r_k) = timestamp(r_l)$, and insert o_k immediately after o_l's response in S. If there is no such an operation execution o_l ($timestamp(r_k) = 0$), then add o_k to the beginning of S. If there are multiple operation executions with the same timestamp value of LC, then insert them in the same place in S. Their relative order is irrelevant unless they are executed by the same process. In such a case, rearrange them in S according to the order in which they were executed by the process.

Part 2. Proof that S respects the write-real-time order of $\bar{H}$.

Let o_i and o_j be any two operation executions of some requests r_i and r_j such that $o_i <^w_{\bar{H}} o_j$. Then, $o_i <^r_{\bar{H}} o_j$ and

1. o_i and o_j are potentially updating and committed, or
2. o_i and o_j are executed by the same process.

In case 1, since $o_i <^r_{\bar{H}} o_j$ (the invocation event of o_j appears in $\bar{H}$ after the response event of o_i), by Proposition 17, we know that $timestamp(r_i) < timestamp(r_j)$. Then, by construction of S, o_i has to appear in S before o_j. Therefore, in this case $o_i <^w_S o_j$.

Now let us consider case 2. We have several subcases to consider:

1. o_i is a committed potentially updating operation execution and o_j is a read-only operation execution. Since we assume that $o_i <^r_{\bar{H}} o_j$ and o_i and o_j are executed by the same process, naturally $timestamp(r_i) \leq timestamp(r_j)$

(the value of LC, which corresponds to values of the $timestamp$ function, increases monotonically). By construction of S, o_j (which is a read-only operation execution) appears in S after a committed potentially updating operation execution o_k (of some request r_k), such that $timestamp(r_k) = timestamp(r_j)$. Therefore $o_k <^r_S o_j$ and $timestamp(r_i) \leq timestamp(r_k)$. If $timestamp(r_i) = timestamp(r_k)$, then $o_i = o_k$ and $o_i <^r_S o_j$. If $t_i.end < t_k.end$, by construction of S, $T_i <^r_S T_k$, and thus $T_i <^r_S T_j$.

2. o_i is a read-only operation execution and o_j is a committed potentially updating operation execution. By Proposition 18, $timestamp(r_i) \leq timestamp(r_j)$. However, since o_j is a committed potentially updating operation execution, $timestamp(r_i) < timestamp(r_j)$ (LC is always incremented prior to execution of a potentially updating request). Since o_i is a read-only operation execution, by construction of S, o_i appears in S after some committed potentially updating operation execution o_k (of some request r_k) such that $timestamp(r_k) = timestamp(r_i)$ and before some committed potentially updating operation execution o'_k (of request r'_k) such that $timestamp(r'_k) = timestamp(r_k) + 1$. o_k may exist or may not exist. We consider both cases:

   a) o_k exists. It means that $o_k <^r_S o_i <^r_S o'_k$. Since $timestamp(r_k) = timestamp(r_i)$ and $timestamp(r_k)+1 = timestamp(r'_k)$, $timestamp(r'_k) \leq timestamp(r_j)$. If $timestamp(r'_k) = timestamp(r_j)$, then $o'_k = o_j$ and $o_i <^r_S o_j$. If $timestamp(r'_k) < timestamp(r_j)$, by construction of S, $o'_k <^r_S o_j$, and thus $o_i <^r_S o_j$.

   b) o_k does not exist. It means that there is no committed potentially updating operation execution in S before o_i ($timestamp(r_i) = 0$). By construction of S, o_i is placed at the beginning of S, before any committed potentially updating operation execution. Therefore $o_i <^r_S o_j$.

3. Both o_i and o_j are read-only. From Proposition 18 we know that $timestamp(r_i) \leq timestamp(r_j)$. If $timestamp(r_i) = timestamp(r_j)$ (and both o_i and o_j are executed by the same process), then the construction of S explicitly requires that o_i and o_j are ordered in S according to the order in which they were executed by this process. On the other hand, if $timestamp(r_i) < timestamp(r_j)$ then by the construction of S:

   a) o_i and o_j appear in S after some committed potentially updating operation executions o'_i and o'_j (of requests r'_i and r'_j) such that $timestamp(r_i) = timestamp(r'_i)$and $timestamp(r_j) = timestamp(r'_j)$. It means that $timestamp(r'_i) < timestamp(r'_j)$, therefore o'_i appears in S before o'_j (by the construction of S). Moreover, between o'_i and o_i in S there is no other committed potentially updating operation execution, since, by the construction of S, o_i is inserted immediately after o'_i. In turn, the four operation executions appear in S in the following order: o'_i, o_i, o'_j, o_j. Thus $o_i <^r_S o_j$.

   b) If such o'_i does not exist ($timestamp(r_i) = 0$; there is no committed potentially updating operation execution in S before o_i), we know that

o'_j has to exist since $timestamp(r'_j) = timestamp(r_j) > timestamp(r_i) = 0$. Then, the three operation executions appear in S in the following order: o_i, o'_j, o_j. Thus also $o_i <^r_S o_j$.

This way S respects the write-real-time order of $\bar{H}$ (trivially, for any operation execution o_k executed by process p_i in $\bar{H}$, o_k is executed by p_i in S).

Part 3. Proof that S is legal.

Let $(Q, q_0, INV, RES, \delta)$ be the sequential specification of the replicated service.[4] In order to prove that S is legal, it is sufficient to show that there exists a witness history $W = \langle q_0, q_1, ... \rangle$ of service states in Q such that for every $k > 0$ $(q_{k-1}, r_k, v_k, q_k) \in \delta$, where r_k is some request and v_k is a response obtained as a result of executing r_k.

Let o_k be any operation execution of some request r_k in S. Then we can make two observations about o_k. The first observation is that by Proposition 13 we know that throughout the execution of r_k the changes to the state of (any) process that executes r_k may result only from the execution of r_k. It means that the response computed as a result of the execution of r_k depends only on two things: (1) on the state of the process that executes r_k just prior to the execution of r_k[5] and (2) on the r_k itself. Since by Proposition 15, all processes change their states in the same way and the state comprises some service state q_{k-1} and the current value of the LC variable, which is equal between the replicas, we can say that the response computed as a result of the execution of r_k depends on q_{k-1} and r_k itself. Note that if r_k is a read-only request, the state of the process that executes r_k does not change.

The second observation regards $\bar{H}$ which is equivalent to S. By construction of $\bar{H}$, if r_k is read-only then o_k must have been completed in H and so there is an appropriate response v_k to r_k. Moreover the service state of the process that executes r_k does not change throughout the execution of r_k, thus q_k (the service state of the process that executes r_k after execution of r_k is finished) is equal to q_{k-1}. On the other hand if r_k is potentially updating (and thus r_k was broadcast using TOB to be executed by all replicas) then o_k appears in $\bar{H}$ only if o_k is completed in H or there exists a process which released a lock after the execution of r_k. It means that if r_k is potentially updating then there also exists an appropriate response v_k to r_k. Also by Proposition 13, we know that throughout the execution of r_k, the state of the process that executed r_k does not change for reasons other than the execution of r_k itself. Therefore there exists a service state q_k which solely depends on the execution of r_k on state q_{k-1} (q_k is equal to the service state of the process that executed r_k and subsequently released the lock).

Now let us construct the witness history W. We do so in two steps. First we start from a witness history $W' = \langle q_0 \rangle$, where q_0 is the initial service state of any

[4] Note that any state S^i_k of some process p_i does not belong to Q. It is because by definition S^i_k consists of some service state $q_k \in Q$ as well as the current value of the LC variable of maintained by p_i.

[5] If r_k is potentially updating then there are multiple such processes and each one executes r_k independently.

process p_i (processes share the initial service state) and consider only potentially updating operation executions in S.

Let S' be a sequential history created by removing from S all read-only operation executions. Then, for any operation execution o_k of request r_k in S', let us append to W' the service state q_k which is equal to the service state of any process that executed r_k and subsequently released the lock.[6]

Because for all operation executions in S' we have requests, matching responses, the sequence of service states before and after requests' execution, and we know that all these requests were executed sequentially (by Proposition 13), for every k such that r_k is a potentially updating request $(q_{k-1}, r_k, v_k, q_k) \in \delta$.

Now we account for the read-only operation executions in S. We create a witness history W in the following way. Let W be an empty sequence. Let us remove the first service state from W' (q_0) and append it to W. Then, for any operation execution o_k in S of request r_k, starting from the first operation execution in S:

1. if o_k is a potentially updating operation execution then remove the first service state q_k from W' and append it to W.
2. if o_k is a read-only operation execution, append to W the duplicate of the last service state in W.

Naturally the order of service states in W' (which corresponds to the updating operation executions in S) is maintained in W. Hence also for every k such that r_k is a potentially updating request $(q_{k-1}, r_k, v_k, q_k) \in \delta$ still holds.

Let us consider any read-only operation execution o_k of some request r_k in S. We have two cases to consider:

1. There does not exist a potentially updating operation execution before o_k in S. It means that o_k operates on the initial state. This fact is reflected by W because for all read-only operation executions which appear prior to o_k in S there are copies of q_0 in W. Since o_k is read-only itself, the service state of the process that executes r_k just after the execution of r_k is finished, also equals q_0. This is reflected by W because q_0 was appended to W for o_k.
2. There exists a potentially updating operation execution before o_k in S. Let o_i be the last such an operation execution of some request r_i and let q_i be the service state of any process just after the execution of r_i. Since between o_i and o_k there are only read-only operation executions, by construction of W we know that for every such a read-only operation execution there is a copy of q_i after q_i in W. It means that r_k operates on q_i. Since o_k is read-only itself, the service state of the process that executes r_k just after the execution of r_k is finished, also equals q_i. This is reflected by W because q_i was appended to W for o_k.

[6] By Proposition 15 all processes change their states in the same way as a result of the execution of potentially updating requests. Hence also their service states change in the same way, because a service state is part of a state of a process. Therefore in this reasoning we can consider the service state of any process after the execution of r_k.

For all read-only operation executions in S' we have requests, matching responses (by the second observation), and the sequence of service states before and after requests' execution. Moreover, by the first observation we know that during the execution of any read-only request the state of the process does not change. It means that even though LSMR allows multiple read-only requests to be executed in parallel, the effects of their execution are as if all these requests were executed sequentially on the service state. Therefore for every k, such that r_k is a read-only request, $(q_{k-1}, r_k, v_k, q_k) \in \delta$. Hence W is the witness history of S and S is legal. This way we conclude the proof that LSMR satisfies write-real-time linearizability. □

B.3 Correctness of Deferred Update Replication

In this section we use the simplified notation which we have already used in Figures: 4.1, 4.2, 4.3 and 4.4.

Theorem 9. *Deferred Update Replication does not satisfy write-real-time opacity.*

Proof. We give a proof by a contradiction. Let us assume that DUR satisfies write-real-time opacity. It means that every history produced by DUR is write-real-time opaque. Now let us consider an example in Figure B.2, which shows a valid execution of DUR with two transactions T_1 (executed by p_1) and T_2 (executed by p_3) such that neither T_1 nor T_2 is declared-read-only, i.e. neither $DRO(T_1)$ nor $DRO(T_2)$ is true.

Transaction T_1 first reads the current value of a shared object x, next writes a new value to it and finally calls the COMMIT procedure (line 37) which corresponds to the $tryC_1$ event in the example. Since T_1 modified x, and there are no concurrent transactions with which T_1 may conflict (lines 39 and 42), transaction descriptor of T_1 is broadcast using TO-BROADCAST. Subsequently, p_1 and p_2 deliver the transaction descriptor of T_1 (line 50), successfully certify it and apply the updates stored in the transaction descriptor.[7]

The message with the transaction descriptor of T_1 sent from p_1 to p_3 takes more time to reach its destination. Before the receipt of the message (but after p_1 and p_2 commit T_1, i.e., apply the updates T_1 produced), p_3 executes transaction T_2 which only performs a read operation on x. Since T_2 did not perform any updating operations it can commit without inter-process synchronization (line 39).

Since both T_1 and T_2 are committed, T_1 precedes in real-time T_2 (i.e., T_1 ended before T_2 started) and T_2 is not declared read-only, by definition of write-real-time opacity, the history from the example is not write-real-time opaque. Therefore, the assumption was false and DUR does not satisfy write-real-time opacity. □

[7] Typically a message broadcast using TO-BROADCAST (e.g., based on the Paxos algorithm [62]) can be delivered once the majority of processes receive the message.

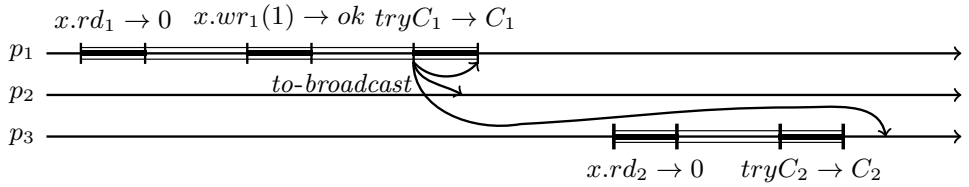


Figure B.2: A valid execution of DUR which is not write-real-time opaque. Neither $DRO(T_1)$ nor $DRO(T_2)$ is true.

Corollary 5. *Deferred Update Replication does not satisfy real-time opacity.*

Proof. The proof follows directly from Theorem 9 and definitions of write-real-time opacity and real-time opacity (real-time opacity is strictly stronger than write-real-time opacity). □

In the following propositions by state of some process p_i we understand the combined state of all t-objects maintained by p_i and the current values of LC and Log that p_i holds (but excluding statistics held in transaction descriptors, which do not count as part of the state).

Proposition 19. *Let $k_a \cdot k_b$ a part of DUR execution on a single process be such that k_a is a certification of a transaction whose transaction descriptor has been delivered using TOB (line 51) and k_b is modifying the local state afterwards (lines 52–55). Let K_1 and K_2 be both $k_a \cdot k_b$ but K_1 and K_2 pertain to different transactions. For any process p_i executing DUR, K_1 and K_2 never interleave, and changes to the state of p_i happen atomically only after k_b.*

Proof. The state of any p_i changes only if the value of LC, Log or any t-object changes (within k_b). This can happen only when p_i delivers through TOB a message, i.e., when p_i processes a transaction descriptor ($k_a \cdot k_b$, lines 51–55). p_i can process only one message at a time (these messages are processed as non-preemptable events). Therefore, both K_1 and K_2 happen atomically and sequentially to each other. □

Proposition 20. *Let p_i be a process executing DUR. Let t be a transaction descriptor of a transaction delivered by p_i using TOB and let S be the state of p_i at the moment of delivery. Let S' be the state of p_i after p_i certifies and (possibly) updates its state, Log_i be the log of p_i in state S', and t_i be the value of t such that $t_i \in Log_i$ in case of successful certification of the transaction. Then for every process p_j in state S, if p_j delivers t using TOB, then p_j moves to state S'.*

Proof. By Proposition 19, the state of p_k does not change throughout certification of a transaction and applying the updates produced by the transaction.

Since the certification procedure (line 9) is deterministic and the values of the Log variables are equal between processes (except for the statistics field which is not used by the procedure), the procedure yields the same result. If the outcome is negative, neither process changes its state (line 51). Otherwise, both processes increment LC to the same value (line 52), assign LC's current value to the end

field of the transaction descriptors (line 53, the value of LC could not change during processing of the transaction descriptor). Next, processes append the transaction descriptors to the Log (line 54) and then apply $t.updates$ (line 55). Therefore both processes move to the same state S'. □

Proposition 21. *Let $S(i) = (S_0^i, S_1^i, ...)$ be a sequence of states of a process p_i executing DUR, where S_0^i is the initial state (comprising of the initial state of t-objects, $LC = 0$ and $Log = \emptyset$) and S_k^i is the state after the k-th message was delivered using TOB and processed by p_i. For every pair of processes p_i and p_j either $S(i)$ is a prefix of $S(j)$ or $S(j)$ is a prefix of $S(i)$.*

Proof. We prove the proposition by a contradiction. Let us assume that $S(i)$ and $S(j)$ differ on position k (and k is the lowest number for which $S_k^i \neq S_k^j$), thus neither is a prefix of another.

If $k = 0$, then the initial state of p_i is different from the initial state of p_j. Since all processes start with the same values of LC, Log (lines 1–2) and maintain the same t-objects with the same initial values, that is a contradiction. Therefore $k > 0$, and the difference between S_k^i and S_k^j must stem from some later change to the state.

Since $S_{k-1}^i = S_{k-1}^j$, the receipt of the k-th message m_k (which is equal for both processes thanks to the use of TOB) and processing it must have resulted in a different change to LC, Log or values of some (or all) t-objects at both processes. Message m_k is a transaction descriptor (line 50). Both processes are in the same state S_{k-1} and process the same transaction descriptor. Therefore, by Proposition 20, both processes move to the same state S_k, a contradiction. It means that the assumption is false, therefore either $S(i)$ is a prefix of $S(j)$, or $S(j)$ is a prefix of $S(i)$. □

Proposition 22. *Let H by a t-history of DUR. Let T_k be an updating committed transaction in H such that t_k is the transaction descriptor of T_k. Then, any process replicates t_k (excluding the statistics field) in its Log as the value of LC on this process is set to $t_k.end$ (both actions happen atomically, i.e., within a lock statement).*

Proof. From Proposition 21 we know that all processes move through the same sequence of states as a result of delivering messages using TOB. Let S be the state of any (correct) process immediately before delivering and processing a message m such that processing of m results in applying updates produced by T_k to the local state (we know that T_k is an updating committed transaction, thus $t_k.updates \neq \emptyset$). By Proposition 20, upon delivery of m any process p_i updates its state in the same way and the new state includes a transaction descriptor t'_k which is identical with t_k and such that t'_k is in Log of p_i. The value of $t'_k.end$ is equal to the current value of LC on p_i because LC is first incremented and then its current value is assigned to $t'_i.end$ in the same lock statement (lines 52–55). This way we gather that t_k is stored in the Log of every process as the value of LC on this process is set to $t_k.end$. □

Proposition 23. *Let H be a t-history of DUR. For any two updating committed transactions $T_i, T_j \in H$ and their transaction descriptors t_i and t_j, if $T_i \prec^r_H T_j$ then $t_i.end < t_j.end$.*

Proof. From the assumption that $T_i \prec^r_H T_j$, we know that T_i is committed and the first event of T_j appears in H after the last event of T_i (the commit of T_i). It means that $tryC(T_j)$ was invoked after the commit of T_i. Since $tryC(T_j)$ is invoked after the commit of T_i, t_j was broadcast (using TOB) after t_i is delivered by the process that executed T_i. Hence, any process can deliver t_j only after t_i. Since LC increases monotonically (line 52) and its current value is assigned to the end field of a transaction descriptor (line 53), on every process $t_i.end < t_j.end$. □

Proposition 24. *Let H by a t-history of DUR. Let T_i and T_j be two transactions in H executed by some process p_l and let t_i and t_j be transaction descriptors of T_i and T_j, respectively. If $T_i \prec^r_H T_j$ then $t_i.start \leq t_j.start$.*

Proof. From the assumption that $T_i \prec^r_H T_j$ and both are executed by the same process p_l, we know that T_i is completed and the first event of T_j appears in H after the last event of T_i. Therefore p_l assigns the current value of LC to $t_i.start$ before it does so for $t_j.start$ (in line 25). The value of LC increases monotonically (line 52, the values of LC correspond to commits of updating transactions). Therefore $t_i.start \leq t_j.start$. □

Proposition 25. *Let H be a t-history of DUR and let $r = x.read \rightarrow v$ be a read operation on some t-object $x \in \mathcal{Q}$ performed by some transaction T_k in H. If T_k did not perform any write operations on x prior to r then either there exists a transaction T_i that performed $x.write(v) \rightarrow ok$ and committed before r returns, or (if there is no such transaction T_i) v is equal to the initial value of x.*

Proof. From the assumption that T_k did not perform any write operations on x prior to r, we know that the value of x is retrieved from the local state (line 7). The value of x is updated on the process that executes T_k only when a transaction descriptor t'_i of a committed updating transaction T'_i is delivered using TOB. Then $t'_i.updates$ are used to modify x in the local state of the process that executes T_k (line 55). Before commit, T'_i stores the modified values of t-objects in the $updates$ set of the transaction descriptor of T'_i. The only possibility that a new value of x is stored in the $updates$ set is upon write operation on x (line 36). Then $T_i = T'_i$ thus satisfying the Proposition.

On the other hand, if the value of x in the system was never updated (through line 55), the initial value of x is returned (line 7). □

Proposition 26. *Let H be a t-history of DUR and T_k (with a transaction descriptor t_k) be some transaction in H. If there exists a t-object $x \in \mathcal{Q}$, such that T_k performs a read operation $r = x.read \rightarrow v$ and T_k executed earlier at least one write operation on x, where $w = x.write(v') \rightarrow ok$ is the last such an operation before r, then $v = v'$.*

Proof. Upon the execution of w, if there were no prior write operations on x in T_k then a pair (x, v') is added to $t_k.updates$; otherwise, the current pair $(x, v")$ is substituted by (x, v') in $t_k.updates$ (line 36). Then v' would be returned upon the execution of r (line 34 and then 5), unless T_k aborts. This may only happen if T_k aborted due to a conflict with other transaction (line 31). However, then r would not return any value. Therefore $v = v'$ and indeed v' was assigned to x by the last write operation on x in T_k before r. □

Theorem 10. *Deferred Update Replication satisfies update-real-time opacity.*

Proof. In order to prove that DUR satisfies update-real-time opacity, we have to show that every finite t-history produced by DUR is final-state update-real-time opaque (by Corollary 1). In other words, we have to show that for every finite t-history H produced by DUR, there exists a t-sequential t-history S equivalent to some completion of H, such that S respects the update-real-time order of H and every transaction T_k in S is legal in S.

Part 1. Construction of a t-sequential t-history S that is equivalent to a completion of H.

Let us first construct a t-completion $\bar{H}$ of H. We start with $\bar{H} = H$. Next, for each live transaction T_k in H performed by process p_i, we append some event to $\bar{H}$ according to the following rules:

- if T_k is not commit-pending and the last event of T_k is an invocation of some operation, append $resp_i(A_k)$,
- if T_k is not commit-pending and the last event of T_k is a response event to some operation, append $\langle tryA(T_k) \rightarrow_i A_k \rangle$,
- if T_k is commit pending and its transaction descriptor t_k was delivered using TOB by some process p_j and p_j successfully certified T_k, then append $resp_i(C_k)$, otherwise append $resp_i(A_k)$.

Now we show that for each committed updating transaction T_k there exists a unique value which corresponds to this transaction. This value is equal to the value of the end field of T_k's transaction descriptor when the updates of T_k are applied (on any process). We show that it is true by a contradiction. Let T_i and T_j be two updating committed transactions with transaction descriptors t_i and t_j, respectively. Let us assume that $T_i \neq T_j$, $t_i.end = v$, $t_j.end = v'$, but $v = v'$. We also assume that t_i and t_j are broadcast (using TOB) to all processes in messages m_i and m_j, respectively (line 44). Since both transactions are updating committed, both m_i and m_j had to be delivered by some processes. By properties of TOB, we know that there exists a process p that delivers both m_i and m_j. Without loss of generality, let us assume that p delivers m_i before m_j. By Proposition 19, we know that processing of m_i and m_j cannot interleave and the updates of t_i and t_j on p are applied in the order of delivery of m_i and m_j. Every time p updates its state, p first increments LC (line 52) and then assigns its value to the end field of the currently processed transaction descriptor (line 53). Moreover, this occurs atomically. Therefore $v \neq v'$, a contradiction. Moreover, by Proposition 21, all processes deliver m_i while being in the same state

and then deliver m_j while also being in the same state. Therefore the values of LC (and matching end fields of transaction descriptors of updating committed transactions) are the same on every process when processing updates of T_i and T_j. Thus, the value of the end field of a transaction descriptor of a committed updating transaction uniquely identifies the transaction.

We can now construct the following function $update$. Let $update : \mathbb{N} \to \mathcal{T}$ be a function that maps the end field of a transaction descriptor of a committed updating transaction to the transaction. Let $S = \langle \bar{H}|update(1) \cdot \bar{H}|update(2) \cdot ... \rangle$. This way S includes the operations of all the committed updating transactions in H. Now, let us add the rest of transactions from $\bar{H}$ to S in the following way. For every such a transaction T_k with a transaction descriptor t_k, we find a committed updating transaction T_l (with transaction descriptor t_l) in S, such that $t_k.start = t_l.end$, and insert $\bar{H}|T_k$ immediately after T_l's operations in S. If there is no such transaction T_l ($t_k.start = 0$), then add $\bar{H}|T_k$ to the beginning of S. If there are multiple transactions with the same value of the $start$ timestamp (in the transaction's transaction descriptor), then insert them in the same place in S. Their relative order is irrelevant unless they are executed by the same process. In such a case, rearrange them in S according to the order in which they were executed by the process.

Part 2. Proof that S respects the update-real-time order of H.

Let T_i and T_j be any two transactions such that $T_i \prec^u_H T_j$ and let t_i and t_j be transaction descriptors of T_i and T_j, respectively. Then, $T_i \prec^r_H T_j$ and

1. T_i and T_j are updating and committed, or
2. T_i and T_j are executed by the same process.

In case 1, by Proposition 23 we know that $t_i.end < t_j.end$. Both $t_i.end$ and $t_j.end$ correspond to the values assigned to LC when t_i and t_j are processed (lines 52 and 53). Then, by the construction of S, T_i must appear in S before T_j. Therefore, $T_i \prec^r_S T_j$. Moreover, the construction requires that for any transaction $T_k \in H$, S includes all events of $H|T_k$. In turn both T_i and T_j are updating and committed in S. Therefore, in this case, $T_i \prec^u_S T_j$.

Now let us consider case 2. We have several subcases to consider:

1. T_i is a committed updating transaction and T_j is a read-only or an aborted transaction. Since $T_i \prec^r_H T_j$ and both T_i and T_j are executed by the same process, naturally $t_i.end \leq t_j.start$ (the value of LC, which is assigned to the $start$ and end fields of transaction descriptor, increases monotonically). By construction of S, T_j (which is a read-only or an aborted transaction) appears in S after a committed updating transaction T_k (with transaction descriptor t_k), such that $t_k.end = t_j.start$. Therefore $T_k \prec^r_S T_j$ and $t_i.end \leq t_k.end$. If $t_i.end = t_k.end$, then $T_i = T_k$ and $T_i \prec^r_S T_j$. If $t_i.end < t_k.end$, by construction of S, $T_i \prec^r_S T_k$, and thus $T_i \prec^r_S T_j$.
2. T_i is a read-only or an aborted transaction and T_j is a committed updating transaction. By Proposition 24, $t_i.start \leq t_j.start$. Since T_j is a committed updating transaction, $t_j.start < t_j.end$ (LC is always incremented prior to

assigning it to the end field of the transaction descriptor upon transaction commit). Therefore $t_i.start \leq t_j.start < t_j.end$, and thus $t_i.start < t_j.end$. Since T_i is a read-only or an aborted transaction, by construction of S, T_i appears in S after some committed updating transaction T_k (with transaction descriptor t_k) such that $t_k.end = t_i.start$ and before some committed updating transaction T'_k (with transaction descriptor t'_k) such that $t'_k.end = t_k.end + 1$. T_k may exist or may not exist. We consider both cases:

a) T_k exists. It means that $T_k \prec^r_S T_i \prec^r_S T'_k$. Since $t_i.start < t_j.end$ and $t_k.end = t_i.start$, $t_k.end < t_j.end$. Because $t_k.end + 1 = t'_k.end$, $t'_k.end \leq t_j.end$. If $t'_k.end = t_j.end$, then $T'_k = T_j$ and $T_i \prec^r_S T_j$. If $t'_k.end < t_j.end$, by construction of S, $T'_k \prec^r_S T_j$, and thus $T_i \prec^r_S T_j$.

b) T_k does not exist. It means that there is no committed updating transaction in S before T_i ($t_i.start = 0$). By construction of S, T_i is placed at the beginning of S, before any committed updating transaction. Therefore $T_i \prec^r_S T_j$.

3. Both T_i and T_j are read-only or aborted transactions. From Proposition 24 we know that $t_i.start \leq t_j.start$. If $t_i.start = t_j.start$ (and both T_i and T_j are executed by the same process), then the construction of S explicitly requires that T_i and T_j are ordered in S according to the order in which they were executed by this process. On the other hand, if $t_i.start < t_j.start$ then by the construction of S:

a) T_i and T_j appear in S after some committed updating transactions T'_i and T'_j with transaction descriptors t'_i and t'_j such that $t_i.start = t'_i.end$ and $t_j.start = t'_j.end$. It means that $t'_i.end < t'_j.end$, therefore T'_i appears in S before T'_j (by the construction of S). Moreover, between T'_i and T_i in S there is no other committed updating transaction, since, by the construction of S, T_i is inserted immediately after T'_i. In turn, the four transactions appear in S in the following order: T'_i, T_i, T'_j, T_j. Thus $T_i \prec^r_S T_j$.

b) If such T'_i does not exist ($t_i.start = 0$; there is no committed updating transaction in S before T_i), we know that T'_j has to exist since $t'_j.end = t_j.start > t_i.start = 0$. Then, the three transactions appear in S in the following order: T_i, T'_j, T_j. Thus also $T_i \prec^r_S T_j$.

This way S respects the update-real-time order of H (trivially, for any transaction T_k executed by process p_i in H, T_k is executed by p_i in S).

Part 3. Proof that every transaction T_j in S is legal in S.

We give the proof by contradiction. Let us assume that there exists a transaction T_j (with a transaction descriptor t_j and executed by some process p) such that T_j is the first transaction that is not legal in S. It means that there exists $x \in \mathcal{Q}$ such that $vis = visible_S(T_j)|x$ does not satisfy the sequential specification of x.

The only type of t-object considered in DUR are simple registers (see Section 4.2). Sequential specification of a register x is violated when a *read* opera-

tion $r = x.read \rightarrow v$ returns a value v that is different from a value most recently written to this register using the *write* operation, or its initial value if there was no such operation.

Therefore, vis does not satisfy the sequential specification of x, if there exists an operation $r = x.read \rightarrow v$ in T_j such that v is not the most recently written value to x in vis. Then, either $v' \neq v$ is the initial value of x or there exists an operation $w = x.write(v') \rightarrow ok$ in vis such that w is the most recent write operation on x in vis prior to r.

By definition of $visible_S(T_j)$, instead of considering t-history vis, we can simply operate on S while excluding from consideration any write operations performed by all aborted transactions in S.

Let us first assume that x was not modified prior to r, i.e., there is no write operation execution on x in S (and in vis) prior to r. Then, trivially, v has to be equal to the initial value of x (by Proposition 25), a contradiction.

Therefore, there exists a transaction T_i (with transaction descriptor t_i) which executes w. First, let us assume that $T_i = T_j$. Given that w is executed prior to r, from Proposition 26, $v = v'$, a contradiction. Therefore $T_i \neq T_j$.

Since we require that w is in vis, T_i must be a committed updating transaction and $T_i \prec^r_S T_j$.

Now we show that $t_i.end \leq t_j.start$. We have two cases to consider:

1. T_j is an aborted or read-only transaction in S. By construction of S, there exists a committed updating transaction T_l (with transaction descriptor t_l) such that $T_l \prec^r_S T_j$ and $t_l.end = t_j.start$. Because both T_i and T_l are committed updating transactions in S, either $T_l \prec^r_S T_i$, $T_i \prec^r_S T_l$ or $T_i = T_l$. By construction of S, between T_l and T_j there must be no committed updating transactions. If $T_l \prec^r_S T_i$ then T_i must appear after T_j in S. However, it is impossible since $T_i \prec^r_S T_j$, a contradiction. Then, either $T_i = T_l$ or $T_i \prec^r_S T_l$. In the first case, $t_i.end = t_l.end$. In the second case, $t_i.end < t_l.end$ (by Proposition 23). Since $t_l.end = t_j.start$, $t_i.end \leq t_j.start$.
2. T_j is a committed updating transaction in S. By Proposition 23, $t_i.end < t_j.end$. By Proposition 22, we know that t_i is stored in Log of p (process that executes T_j) by the time the value of LC on that process reaches $t_i.end$. Since T_j is a committed updating transaction, it has to pass the certification test (line 9). This test takes place as late as the commit of T_j (line 51). Since the commit sets the value of LC to $t_j.end$ (line 53), the certification takes place when $LC = t_j.end - 1$. Since $t_i.end < t_j.end$, $t_i.end \leq t_j.end - 1$. This means that t_i is already stored in the Log of p when certification happens. We know that $x \in t_j.readset$ and $(x, v') \in t_i.updates$, where v' is some value. If we had $t_i.end > t_j.start$, then the certification procedure would compare the T_j's readset against the T_i's updates and return $failure$, thus aborting T_j. But we know that T_j is committed. Therefore, $t_i.end \leq t_j.start$.

By Proposition 22 and the fact that $t_i.end \leq t_j.start$, we know that t_i is stored in Log of p (process that executes T_j) before T_j starts. It means that inside the

same lock statement, LC is incremented (line 52) and its value is assigned to $t_i.end$ (line 53), t_i is appended to Log (line 54) and $t_i.updates$ are applied to the local state (line 55). Therefore, the updates of T_i are applied to the local state of p before T_j starts.

Now, unless there is some transaction T_k (with transaction descriptor t_k), such that T_k modified x, $t_k.updates$ are applied to the local state of p after $t_i.updates$ are applied but before r returns, r would have to return v'. But it is impossible, because we assumed that r returns $v \neq v'$. Therefore we now consider such T_k. If $t_k.updates$ are indeed applied by p after $t_i.updates$ are, then $t_k.end > t_i.end$ (p increments LC each time p applies updates of some transaction, line 52). By construction of S, T_k would have to appear in S after T_i and before r returns. However, then w would not be the most recent write operation on x prior to r in S, a contradiction.

From the contradiction we know that the assumption that there exists such a transaction T_k is false. Therefore r has to return $v = v'$, thus concluding the proof. Therefore DUR guarantees update-real-time opacity. □

Corollary 6. *Let G be a gateway shared object implemented using Deferred Update Replication. Then, G satisfies update-real-time linearizability.*

Proof. The proof follows directly from Theorem 5 and Theorem 10. □

B.4 Correctness of Hybrid Transactional Replication

Below we only consider t-histories of HTR, i.e., histories limited to events that are related to operations on t-objects ($texec$ operations) and controlling the flow of transactions such as *commit* and *abort* events ($tryC$ and $tryA$ operations, respectively). In this sense, we treat the implementation of HTR as some TM object M, and reason about t-histories $H|M$.

Theorem 11. *Hybrid Transactional Replication does not satisfy write-real-time opacity.*

Proof. Trivially, every t-history of DUR is also a valid t-history of HTR since transactions in DUR are handled in exactly the same way as DU transactions in HTR. Since DUR does not satisfy write-real-time opacity (by Theorem 9), neither does HTR. □

Corollary 7. *Hybrid Transactional Replication does not satisfy real-time opacity.*

Proof. The proof follows directly from Theorem 11 and definitions of write-real-time opacity and real-time opacity (real-time opacity is strictly stronger than write-real-time opacity). □

In the following propositions by state of some process p_i we understand the combined state of all t-objects maintained by p_i and the current values of LC

and Log that p_i holds (but excluding statistics held in transaction descriptors, which do not count as part of the state).

The following proofs in many places are analogous to the proofs of the correctness of DUR (see Section 5.3.3), where we showed that DUR satisfies update-real-time opacity.

Proposition 27. *Let $k_a \cdot k_b$ and $k'_a \cdot k'_b$ be parts of HTR execution on a single process be such that:*

1. *k_a is the certification of a DU transaction whose transaction descriptor has been delivered using TOB (line 62) and k_b is modifying the local state afterwards (lines 63–66),*
2. *k'_a is an execution of an SM transaction (lines 77–80) and k'_b is modifying the local state afterwards (lines 88–91).*

Let K_1 be either $k_a \cdot k_b$ or $k'_a \cdot k'_b$, and K_2 also be either $k_a \cdot k_b$ or $k'_a \cdot k'_b$ but K_1 and K_2 pertain to different transactions. For any process p_i executing HTR, K_1 and K_2 never interleave, and changes to the state of p_i happen atomically only after k_b or k'_b.

Proof. The state of any p_i changes only if the value of LC, Log or any t-object changes (within k_b or k'_b, which are semantically identical and are guarded by the same lock). This can happen only when p_i delivers a message through TOB, i.e., either when p_i processes a transaction descriptor of a DU transaction ($k_a \cdot k_b$, lines 62–66) or when p_i processes a request (line 69) which then p_i executes as an SM transaction ($k'_a \cdot k_b$, lines 77–80 and 88–91). p_i can process only one message at a time (these messages are processed as non-preemptable events). Therefore, both K_1 and K_2 happen atomically and sequentially to each other. □

Proposition 28. *Let p_i be a process executing HTR. Let t be a transaction descriptor of a DU transaction delivered by p_i using TOB and let S be the state of p_i at the moment of delivery. Let S' be the state of p_i after p_i certifies and (possibly) updates its state, Log_i be the log of p_i in state S' and t_i be the value of t such that $t_i \in Log_i$ in case of successful certification of the transaction. Then for every process p_j in state S, if p_j delivers t using TOB, then p_j moves to state S'.*

Proof. By Proposition 27, the state of p_k does not change throughout certification of a DU transaction (whose transaction descriptor has been delivered) and applying the updates produced by the transaction.

Since the certification procedure (line 9) is deterministic and the values of the Log variables are equal between processes (except for the statistics field which is not used by the procedure), the procedure yields the same result. If the outcome is negative, neither process changes its state (line 62). Otherwise, both processes increment LC to the same value (line 63), assign LC's current value to the end field of the transaction descriptors (line 64, the value of LC could not change during processing of the transaction descriptor). Next, processes append the transaction descriptors to the Log (line 65) and then apply $t.updates$ (line 66). Therefore both processes move to the same state S'. □

Proposition 29. *Let p_i be a process executing HTR. Let r be a request delivered by p_i using TOB, let S be the state of p_i at the moment of delivery of r and S' be the state of p_i after the execution of r as an SM transaction T_k with transaction descriptor t_k. For every process p_j in state S, if p_j delivers r using TOB then the execution of r as an SM transaction T_l (with transaction descriptor t_l) by p_j yields state S' of p_j and $t_k = t_l$ (except for the statistics field).*

Proof. By Proposition 27, the state of p_k does not change throughout an execution of an SM transaction and applying the updates produced by the transaction.

The values of $t_k.id$ and $t_l.id$ are equal, since processes assign to the id field a value which deterministically depends on $r.id$ (line 77).

Since both p_i and p_j start the execution of r from the same state, the current values of their LC variables are equal. Hence, $t_k.start = t_l.start$ (line 78).

During the execution of an SM transaction nothing is ever added to $readset$ (line 82). Therefore $t_k.readset = t_l.readset = \emptyset$.

Since HTR assumes that only a request with deterministic $prog$ can be executed as an SM transaction, $r.prog$ must be deterministic. Both processes execute $r.prog$ with the same $r.args$ (line 80) and operate on the same state S which does not change throughout the execution of $r.prog$. Moreover, all updates produced by the transactions are stored in the $updates$ sets (line 84). Therefore $t_l.updates = t_k.updates$.

Also $t_k.end = t_l.end$. If T_k and T_l are read-only ($t_k.updates = t_l.updates = \emptyset$), the initial values of $t_k.end$ and $t_l.end$ do not change. Otherwise, both processes increment LC and assign its current value to the end fields (line 88, the value of LC could not change during the execution of SM transactions).

Because $t_k.id = t_l.id$, $t_k.start = t_l.start$, $t_k.readset = t_l.readset$, $t_k.updates = t_l.updates$, and $t_k.end = t_l.end$, we gather that $t_k = t_l$ (except for the statistics field). If both transactions are updating, p_i adds t_k to p_i's *Log* and p_j adds t_l to p_j's *Log*. Then, both processes apply all updates from the respective transaction descriptors. Thus both processes move to the same state, i.e., S'. □

Proposition 30. *Let $S(i) = (S_0^i, S_1^i, ...)$ be a sequence of states of a process p_i executing HTR, where S_0^i is the initial state (comprising the initial state of t-objects, $LC = 0$ and $Log = \emptyset$) and S_k^i is the state after the k-th message was delivered using TOB and processed by p_i. For every pair of processes p_i and p_j either $S(i)$ is a prefix of $S(j)$ or $S(j)$ is a prefix of $S(i)$.*

Proof. We prove the proposition by a contradiction. Let us assume that $S(i)$ and $S(j)$ differ on position k (and k is the lowest number for which $S_k^i \neq S_k^j$), thus neither is a prefix of another.

If $k = 0$, then the initial state of p_i is different from the initial state of p_j. Since all processes start with the same values of LC, *Log* (lines 1–2) and maintain the same t-objects with the same initial values, that is a contradiction. Therefore $k > 0$, and the difference between S_k^i and S_k^j must stem from some later change to the state.

Since $S_{k-1}^i = S_{k-1}^j$, the receipt of the k-th message m_k (which is equal for both processes thanks to the use of TOB) and processing it must have resulted

in a different change to LC, Log or values of some (or all) t-objects at both processes. We have two cases to consider:

1. Message m_k is a transaction descriptor of a DU transaction (line 61). Both processes are in the same state S_{k-1} and process the same transaction descriptor. Therefore, by Proposition 28, both processes move to the same state S_k, which is a contradiction.
2. Message m_k is a request to be executed as an SM transaction (line 69). Both processes are in the same state S_{k-1} and execute the same request as SM transactions. Therefore, by Proposition 29, both processes move to the same state S_k, which is a contradiction.

Since both cases yield a contradiction, the assumption is false. Therefore either $S(i)$ is a prefix of $S(j)$, or $S(j)$ is a prefix of $S(i)$.

□

We know that all communication between processes in HTR happens through TOB. It means that all processes deliver all messages in the same order. If the message is a request forwarded by some process to be executed as an SM transaction, then every process delivers this request while being in the same state (by Proposition 30). Then, all processes execute the request as different SM transactions but end up with transaction descriptors of the same exact value (except for the statistics field, by Proposition 29). Therefore, processes need not to disseminate the transaction descriptors after they complete the transaction execution (as in case of a DU transaction). Instead, processes may promptly apply the updates from the transaction descriptors to their state. It all means that executing a request as multiple SM transactions across the whole system is equivalent to the execution of the request only once and then distributing the resulting updates to all processes. Also, unless a request must be executed in the SM mode (because it performs some irrevocable operations), the client has no knowledge which execution mode was chosen to execute his request.

The way HTR handles SM transactions means that HTR does not exactly fit the model of (update-real-time) opacity which requires that updates produced by every committed transaction must be accounted for. Therefore, unless the SM transactions resulting from the execution of the same request did not perform any modifications or are rolled back on demand, it is impossible to construct a t-sequential t-history S in which every transaction is t-legal. However, since we proved that the execution of multiple SM transactions regarding the same request is equivalent to the execution of a single one, we can propose the following mapping of t-histories, which we call *SMreduce*. Roughly speaking, under the *SMreduce* mapping of some t-history of HTR, for any group of SM transactions regarding the same request r, such that the processes that executed the transactions applied the updates produced by the transactions, we only allow the first transaction of the group in the t-history to commit; other transactions appear aborted in the transformed t-history.

Now let us give a formal definition of the *SMreduce* mapping. Let H be a t-

history of HTR and let $SMmode$ be a predicate such that for any transaction T_k in H, $SMmode(T_k)$ is true if T_k was executed as an SM transaction in H. Otherwise $SMmode(T_k)$ is false. Then, let $H' = SMreduce(H)$ be a t-history constructed by changing H in the following way. For any event e in H such that:

- $e = resp_i(C_k)$ is a response event of an operation execution $M.tryC(T_k) \rightarrow_i C_k$ for some transaction T_k and process p_i, and
- $SMmode(T_k)$ is true (and r is the request whose execution resulted in T_k), and
- T_k is not the first completed transaction in H which resulted from the execution of r in the SM mode,

replace e in H' with $e' = resp_i(A_k)$. We say that H' is an *SMreduced* t-history of HTR.

Proposition 31. *Let H be an SMreduced t-history of HTR. Let T_k be an updating committed transaction in H such that t_k is the transaction descriptor of T_k. Then, any process stores t_k (excluding the statistics field) in its Log as the value of LC on this process is set to $t_k.end$ (both actions happen atomically, i.e., within a lock statement).*

Proof. From Proposition 30 we know that all processes move through the same sequence of states as a result of delivering messages using TOB. Let S be the state of any (correct) process immediately before delivering and processing a message m such that processing of m results in applying updates produced by T_k to the local state (we know that T_k is an updating committed transaction, thus $t_k.updates \neq \emptyset$). We have two cases to consider:

1. T_k is a DU transaction. Then, by Proposition 28, upon delivery of m any process p_i updates its state in the same way and the new state includes a transaction descriptor $t'_k = t_k$ such that t'_k is in Log of p_i. The value of $t'_k.end$ is equal to the current value of LC held by p_i because LC is first incremented and then its current value is assigned to $t'_i.end$ atomically within a single lock statement (lines 63–66).
2. T_k is an SM transaction. Then, upon delivery of m any process p_i executes the request received in m as an SM transaction T'_k (T'_k may or may not be equal T_k) with transaction descriptor t'_k. After T'_k finishes its execution, inside the same lock statement p_i increments the value of its LC, appends t'_k to its Log and applies the updates produced by T'_k (lines 88–91). By Proposition 29, $t'_k = t_k$ (except for the statistics field). Note that by definition of SMreduce, every transaction $T'_k \neq T_k$, which is executed as a result of receipt of m, is aborted (which means that updates of a committed SM transaction are in fact applied by every process only once).

This way for both cases we gather that t_k is stored in the Log of every process as the value of LC on this process is set to $t_k.end$. □

Proposition 32. *Let H be an SMreduced t-history of HTR. For any two updating committed transactions $T_i, T_j \in H$ and their transaction descriptors t_i and t_j, if $T_i \prec^r_H T_j$ then $t_i.end < t_j.end$.*

Proof. From the assumption that $T_i \prec^r_H T_j$, we know that T_i is committed and the first event of T_j appears in H after the last event of T_i (the commit of T_i). It means that $tryC(T_j)$ was invoked after the commit of T_i. Now we have four cases to consider:

1. T_i and T_j are DU transactions. Since $tryC(T_j)$ is invoked after the commit of T_i, t_j was broadcast (using TOB) after t_i is delivered by the process that executed T_i. Hence, any process can deliver t_j only after t_i. Since LC increases monotonically (line 63) and its current value is assigned to the end field of a transaction descriptor (line 64), on every process $t_i.end < t_j.end$.
2. T_i is a DU transaction and T_j is an SM transaction (whose execution resulted from delivery of request r using TOB). Since T_j is committed, by definition of SMreduce, T_j is the first SM transaction in H to complete and such that T_j's execution resulted from delivery of r. It means that there does not exist an SM transaction T'_j on the process that executes T_i, whose execution resulted from delivery of request r and which completed before T_j did. Since T_i commits before T_j, T_i has to commit before any transaction T''_j (whose execution also results from delivery of r) completes. By Proposition 27, T_i's certification and commit and T''_j's execution do not interleave. It means that T''_j must have started after T_i committed. Then T_i must have incremented LC before T''_j started (line 63) and so $t_i.end < t''_j.end$, where t''_j is the transaction descriptor of T''_j. By Proposition 29, $t''_j = t_j$. Therefore $t_i.end < t_j.end$.
3. T_i is an SM transaction (whose execution resulted from delivery of request r using TOB) and T_j is a DU transaction. Since $tryC(T_j)$ was invoked after the commit of T_i, t_j was broadcast (using TOB) later than r was delivered by the process that executes T_i. Therefore, this process can only deliver t_j after handling r and executing T_i. By properties of TOB, the process that executes T_j can also deliver t_j after delivery of r. Therefore, the process that executes T_j had to deliver r, execute an SM transaction T'_i (with transaction descriptor t'_i) and modify the local state afterwards but before T_j started (by Proposition 27). Since LC increases monotonically (line 88), $t'_i.end < t_j.end$. By Proposition 29, $t'_i = t_i$, thus $t'_i.end = t_i.end$. Therefore $t_i.end < t_j.end$.
4. T_i and T_j are SM transactions (whose execution resulted from delivery of requests r and r' using TOB, respectively). By properties of TOB, r and r' had to be delivered by any process in the same order. We now show that r must be delivered before r'. Let us assume the opposite. Then, the process that executes T_i delivers r' prior to the execution of T_i. As a result, the process executes an SM transaction T'_j which produces the same results as T_j (by Proposition 29). By Proposition 27, T'_j must complete before T_i starts. Then, $T'_j \prec^r_H T_i$. By definition of SMreduce, we know that T_j is the first SM transaction to complete in H, such that T_j's execution resulted from delivery of r'. Therefore T_j must have completed before T'_j. It means that

$T_j \prec^r_H T_i$, a contradiction. Therefore r must be delivered by TOB prior to r'. Since the process that executes T_j must have delivered r before delivering r', the process must have executed an SM transaction T'_i prior to T_j such that the execution of T'_i resulted from delivery of r. By Proposition 27, T'_i completes before T_j starts. Since LC increases monotonically (line 88), $t'_i.end < t_j.end$, where t'_i is the transaction descriptor of T'_i. By Proposition 29, $t'_i = t_i$, thus $t'_i.end = t_i.end$. Therefore $t_i.end < t_j.end$.

□

Proposition 33. *Let H by an SMreduced t-history of HTR. Let T_i and T_j be two transactions in H executed by some process p_l and let t_i and t_j be transaction descriptors of T_i and T_j, respectively. If $T_i \prec^r_H T_j$ then $t_i.start \leq t_j.start$.*

Proof. From the assumption that $T_i \prec^r_H T_j$ and both are executed by the same process p_l, we know that T_i is completed and the first event of T_j appears in H after the last event of T_i. Therefore p_l assigns the current value of LC to $t_i.start$ before it does so for $t_j.start$ (in line 29 and line 78, if T_i is a DU or SM transaction, respectively). The value of LC increases monotonically (lines 63 and 88, the values of LC correspond to commits of updating transactions). Therefore $t_i.start \leq t_j.start$. □

Proposition 34. *Let H be an SMreduced t-history of HTR and let $r = x.read \rightarrow v$ be a read operation on some t-object $x \in \mathcal{Q}$ performed by some transaction T_k in H. If T_k did not perform any write operations on x prior to r then either there exists a transaction T_i that performed $x.write(v) \rightarrow ok$ and committed before r returns, or (if there is no such transaction T_i) v is equal to the initial value of x.*

Proof. From the assumption that T_k did not perform any write operations on x prior to r, we know that the value of x is retrieved from the local state (line 7). The value of x is updated on the process that executes T_k only in two cases:

1. A transaction descriptor t'_i of a committed updating DU transaction T'_i is delivered using TOB. Then $t'_i.updates$ are used to modify x in the local state of the process that executes T_k (line 66). Before commit, T'_i stores the modified values of t-objects in the $updates$ set of the transaction descriptor of T'_i. The only possibility that a new value of x is stored in the $updates$ set is upon write operation on x (line 42). Then $T_i = T'_i$ thus satisfying the Proposition.
2. An updating SM transaction T'_i (whose execution resulted from delivery of a request r_i using TOB) modified x upon applying the updates it produced (line 91); T'_i is executed by the same process that executes T_k. Before that, during execution, T'_i stores the modified values of t-objects in the $updates$ sets of T_i's transaction descriptor. The only possibility that a new value of x is stored in the $updates$ set is upon write operation on x (line 84). Now, because H is SMreduced there are two cases to consider. In the first case T'_i is committed. Then $T_i = T'_i$ thus satisfying the Proposition. In the second case T'_i is aborted. However, from the definition of SMreduce, we know

that there exists a committed SM transaction T_i whose execution resulted from delivery of r_i, such that T_i committed before T'_i completed (and therefore also prior to r). The transaction descriptor of T_i is equivalent to the transaction descriptor of T'_i (except for the statistics, by Proposition 29). Then, when T_k performs r, the local state of the process that performed T'_k contains the updates produced by T_i, i.e., it contains also v as the current value of x.

On the other hand, if the value of x in the system was never updated (through line 66 or 91), the initial value of x is returned (line 7). □

Proposition 35. *Let H be an (SMreduced) t-history of HTR and T_k (with a transaction descriptor t_k) be some transaction in H. If there exists a t-object $x \in \mathcal{Q}$, such that T_k performs a read operation $r = x.read \rightarrow v$ and T_k executed earlier at least one write operation on x, where $w = x.write(v') \rightarrow ok$ is the last such an operation before r, then $v = v'$.*

Proof. Upon the execution of w, if there were no prior write operations on x in T_k then a pair (x, v') is added to $t_k.updates$; otherwise, the current pair $(x, v")$ is substituted by (x, v') in $t_k.updates$ (line 42 or line 84, if T_k is a DU or an SM transaction, respectively). Then v' would be returned upon execution of r (line 40 and then 5), unless T_k aborts. This may only happen if T_k is a DU transaction and T_k aborted due to a conflict with other transaction (line 37). However, then r would not return any value (the execution of the transaction would be stopped). Therefore $v = v'$ and indeed v' was assigned to x by the last write operation on x in T_k before r. □

Theorem 12. *Under the SMreduce mapping, Hybrid Transactional Replication satisfies update-real-time opacity.*

Proof. In order to prove that HTR satisfies update-real-time opacity under SMreduce, we have to show that every SMreduced finite t-history produced by HTR is final-state update-real-time opaque (by Corollary 1). In other words, we have to show that for every SMreduced finite t-history H produced by HTR, there exists a t-sequential t-history S equivalent to some completion of H, such that S respects the update-real-time order of H and every transaction T_k in S is legal in S.

Part 1. Construction of a t-sequential t-history S that is equivalent to a completion of H.

Let us first construct a t-completion $\bar{H}$ of H. We start with $\bar{H} = H$. Next, for each live transaction T_k in H performed by process p_i, we append some event to $\bar{H}$ according to the following rules:

- if T_k is not commit-pending and the last event of T_k is an invocation of some operation, append $resp_i(A_k)$,
- if T_k is not commit-pending and the last event of T_k is a response event to some operation, append $\langle tryA(T_k) \rightarrow_i A_k \rangle$,

- if T_k is commit pending and T_k is an SM transaction, then append $resp_i(A_k)$,
- if T_k is commit pending and T_k is a DU transaction with a transaction descriptor t_k, then if t_k was delivered by some process p_j using TOB and p_j successfully certified T_k, then append $resp_i(C_k)$. Otherwise append $resp_i(A_k)$.

Now we show that for each committed updating transaction T_k there exists a unique value which corresponds to this transaction. This value is equal to the value of the end field of T_k's transaction descriptor when the updates of T_k are applied (on any process), as we now prove by a contradiction. Let T_i and T_j be two updating committed transactions with transaction descriptors t_i and t_j, respectively. T_i and T_j result from delivery of some requests r_i and r_j (T_i and T_j may be DU or SM transactions). Let us assume that $T_i \neq T_j$, $t_i.end = v$, $t_j.end = v'$, but $v = v'$. If T_i is a DU transaction, then t_i is broadcast using TOB to all processes in a message m_i (line 50). If T_i is an SM transaction, then the request r_i is broadcast using TOB prior to the execution of T_i in a message m_i (line 32). Analogically, for T_j, message m_j contains either t_j or r_j. Since both transactions are updating committed, both m_i and m_j had to be delivered by some processes. By properties of TOB, we know that there exists a process p that delivers both m_i and m_j. Without loss of generality, let us assume that p delivers m_i before m_j. By Proposition 27, we know that processing of m_i and m_j cannot interleave (irrespective of the modes of the transactions) and the updates of t_i and t_j on p are applied in the order of delivery of m_i and m_j. Every time p updates its state, p first increments LC (line 63 or 88) and then assigns its value to the end field of the currently processed transaction descriptor (line 64 or 89). Therefore $v \neq v'$, a contradiction. Moreover, by Proposition 30, all processes deliver m_i while being in the same states and then deliver m_j while also being in the same states. Therefore the values of LC (and matching end fields of transaction descriptors of updating committed transactions) are the same on every process when processing updates of T_i and T_j. Thus, the value of the end field of a transaction descriptor of a committed updating transaction uniquely identifies the transaction.

We can now construct the following function $update$. Let $update : \mathbb{N} \to \mathcal{T}$ be a function that maps the end field of a transaction descriptor of a committed updating transaction to the transaction. Let $S = \langle \bar{H}|update(1) \cdot \bar{H}|update(2) \cdot ... \rangle$. This way S includes the operations of all the committed updating transactions in H. Now, let us add the rest of transactions from $\bar{H}$ to S in the following way. For every such a transaction T_k with a transaction descriptor t_k, find a committed updating transaction T_l (with transaction descriptor t_l) in S, such that $t_k.start = t_l.end$, and insert $\bar{H}|T_k$ immediately after T_l's operations in S. If there is no such transaction T_l ($t_k.start = 0$), then add $\bar{H}|T_k$ to the beginning of S. If there are multiple transactions with the same value of $start$ timestamp, then insert them in the same place in S. Their relative order is irrelevant unless they are executed by the same process. In such a case, rearrange them in S according to the order in which they were executed by the process.

Part 2. Proof that S respects the update-real-time order of H.

Let T_i and T_j be any two transactions such that $T_i \prec^u_H T_j$ and let t_i and t_j be transaction descriptors of T_i and T_j, respectively. Then, $T_i \prec^r_H T_j$ and

1. T_i and T_j are updating and committed, or
2. T_i and T_j are executed by the same process.

In case 1, by Proposition 32 we know that $t_i.end < t_j.end$. Both $t_i.end$ and $t_j.end$ correspond to the values assigned to LC when t_i and t_j are processed (lines 63 and 64). Then, by the construction of S, T_i must appear in S before T_j. Therefore, $T_i \prec^r_S T_j$. Moreover, the construction requires that for any transaction $T_k \in H$, S includes all events of $H|T_k$. In turn both T_i and T_j are updating and committed in S. Therefore, in this case, $T_i \prec^u_S T_j$.

Now let us consider case 2. We have several subcases to consider:

1. T_i is a committed updating transaction and T_j is a read-only or an aborted transaction. Since $T_i \prec^r_H T_j$ and both T_i and T_j are executed by the same process, naturally $t_i.end \leq t_j.start$ (the value of LC, which is assigned to the $start$ and end fields of transaction descriptor, increases monotonically). By construction of S, T_j (which is a read-only or an aborted transaction) appears in S after a committed updating transaction T_k (with transaction descriptor t_k), such that $t_k.end = t_j.start$. Therefore $T_k \prec^r_S T_j$ and $t_i.end \leq t_k.end$. If $t_i.end = t_k.end$, then $T_i = T_k$ and $T_i \prec^r_S T_j$. If $t_i.end < t_k.end$, by construction of S, $T_i \prec^r_S T_k$, and thus $T_i \prec^r_S T_j$.
2. T_i is a read-only or an aborted transaction and T_j is a committed updating transaction. By Proposition 33, $t_i.start \leq t_j.start$. Since T_j is a committed updating transaction, $t_j.start < t_j.end$ (LC is always incremented prior to assigning it to the end field of the transaction descriptor upon transaction commit). Therefore $t_i.start \leq t_j.start < t_j.end$, and thus $t_i.start < t_j.end$. Since T_i is a read-only or an aborted transaction, by construction of S, T_i appears in S after some committed updating transaction T_k (with transaction descriptor t_k) such that $t_k.end = t_i.start$ and before some committed updating transaction T'_k (with transaction descriptor t'_k) such that $t'_k.end = t_k.end + 1$. T_k may exist or may not exist. We consider both cases:
   a) T_k exists. It means that $T_k \prec^r_S T_i \prec^r_S T'_k$. Since $t_i.start < t_j.end$ and $t_k.end = t_i.start$, $t_k.end < t_j.end$. Because $t_k.end + 1 = t'_k.end$, $t'_k.end \leq t_j.end$. If $t'_k.end = t_j.end$, then $T'_k = T_j$ and $T_i \prec^r_S T_j$. If $t'_k.end < t_j.end$, by construction of S, $T'_k \prec^r_S T_j$, and thus $T_i \prec^r_S T_j$.
   b) T_k does not exist. It means that there is no committed updating transaction in S before T_i ($t_i.start = 0$). By construction of S, T_i is placed at the beginning of S, before any committed updating transaction. Therefore $T_i \prec^r_S T_j$.
3. Both T_i and T_j are read-only or aborted transactions. From Proposition 33 we know that $t_i.start \leq t_j.start$. If $t_i.start = t_j.start$ (and both T_i and T_j are executed by the same process), then the construction of S explicitly

requires that T_i and T_j are ordered in S according to the order in which they were executed by this process. On the other hand, if $t_i.start < t_j.start$ then by the construction of S:

a) T_i and T_j appear in S after some committed updating transactions T'_i and T'_j with transaction descriptors t'_i and t'_j such that $t_i.start = t'_i.end$ and $t_j.start = t'_j.end$. It means that $t'_i.end < t'_j.end$, therefore T'_i appears in S before T'_j (by the construction of S). Moreover, between T'_i and T_i in S there is no other committed updating transaction, since, by the construction of S, T_i is inserted immediately after T'_i. In turn, the four transactions appear in S in the following order: T'_i, T_i, T'_j, T_j. Thus $T_i \prec^r_S T_j$.

b) If such T'_i does not exist ($t_i.start = 0$; there is no committed updating transaction in S before T_i), we know that T'_j has to exist since $t'_j.end = t_j.start > t_i.start = 0$. Then, the three transactions appear in S in the following order: T_i, T'_j, T_j. Thus also $T_i \prec^r_S T_j$.

This way S respects the update-real-time order of H (trivially, for any transaction T_k executed by process p_i in H, T_k is executed by p_i in S).

Part 3. Proof that every transaction T_j in S is legal in S.

We give the proof by contradiction. Let us assume that there exists a transaction T_j (with a transaction descriptor t_j and executed by some process p) such that T_j is the first transaction that is not legal in S. It means that there exists $x \in \mathcal{Q}$ such that $vis = visible_S(T_j)|x$ does not satisfy the sequential specification of x.

The only type of t-object considered in HTR are simple registers. Sequential specification of a register x is violated when a *read* operation $r = x.read \rightarrow v$ returns a value v that is different a value most recently written to this register using the *write* operation, or its initial value if there was no such operation.

Therefore, vis does not satisfy the sequential specification of x, if there exists an operation $r = x.read \rightarrow v$ in T_j such that v is not the most recently written value to x in vis. Then, either $v' \neq v$ is the initial value of x or there exists an operation $w = x.write(v') \rightarrow ok$ in vis such that w is the most recent write operation on x in vis prior to r.

By definition of $visible_S(T_j)$, instead of considering t-history vis, we can simply operate on S while excluding from consideration any write operations performed by all aborted transactions in S.

Let us first assume that x was not modified prior to r, i.e., there is no write operation execution on x in S (and in vis) prior to r. Then, trivially, v has to be equal to the initial value of x (by Proposition 34), a contradiction.

Therefore, there exists a transaction T_i (with transaction descriptor t_i) which executes w. First, we assume that $T_i = T_j$. Given that w is executed prior to r, from Proposition 35, $v = v'$, a contradiction. Therefore $T_i \neq T_j$.

Since we require that w is in vis, T_i must be a committed updating transaction and $T_i \prec^r_S T_j$.

Now we show that $t_i.end \leq t_j.start$. We have two cases to consider:

1. T_j is an aborted or read-only transaction in S. By construction of S, there exists a committed updating transaction T_l (with transaction descriptor t_l) such that $T_l \prec^r_S T_j$ and $t_l.end = t_j.start$. Because both T_i and T_l are committed updating transactions in S, either $T_l \prec^r_S T_i$, $T_i \prec^r_S T_l$ or $T_i = T_l$. By construction of S, between T_l and T_j there must be no committed updating transactions. If $T_l \prec^r_S T_i$ then T_i must appear after T_j in S. However, it is impossible since $T_i \prec^r_S T_j$, a contradiction. Then, either $T_i = T_l$ or $T_i \prec^r_S T_l$. In the first case, $t_i.end = t_l.end$. In the second case, $t_i.end < t_l.end$ (by Proposition 32). Since $t_l.end = t_j.start$, $t_i.end \leq t_j.start$.
2. T_j is a committed updating transaction in S. By Proposition 32, $t_i.end < t_j.end$. Now we have additional two cases to consider:
   a) T_j is a DU transaction. By Proposition 31, we know that t_i is stored in Log of p (process that executes T_j) by the time the value of LC on that process reaches $t_i.end$. Since T_j is a committed updating transaction, it has to pass the certification test (line 9). This test takes place as late as the commit of T_j (line 62). Since the commit sets the value of LC to $t_j.end$ (line 64), the certification takes place when $LC = t_j.end - 1$. Since $t_i.end < t_j.end$, then $t_i.end \leq t_j.end - 1$. This means that t_i is already stored in the Log of p when certification happens. We know that $x \in t_j.readset$ and $(x, v') \in t_i.updates$. If we had $t_i.end > t_j.start$, then the certification procedure would compare the T_j's readset against the T_i's updates and return $failure$, thus aborting T_j. But we know that T_j is committed. Therefore, $t_i.end \leq t_j.start$.
   b) T_j is an SM transaction. For any committed updating SM transaction T (with transaction descriptor t) the following holds: $t.start + 1 = t.end$ (by Proposition 27, execution of T cannot interleave with the execution of another SM transaction or processing a transaction descriptor of a DU transaction after it is delivered). Since $t_i.end < t_j.end$ we know that $t_i.end < t_j.start + 1$. Thus $t_i.end \leq t_j.start$.

By Proposition 31 and the fact that $t_i.end \leq t_j.start$, we know that t_i is stored in Log of p (process that executes T_j) before T_j starts. It means that inside the same lock statement, LC is incremented (lines 63 and 88) and its value is assigned to $t_i.end$ (lines 64 and 89), t_i is appended to Log (lines 65 and 90) and $t_i.updates$ are applied to the local state (lines 66 and 91). Therefore, the updates of T_i are applied to the local state of p before T_j starts.

Now, unless there is some transaction T_k (with transaction descriptor t_k), such that T_k modified x, $t_k.updates$ are applied to the local state of p after $t_i.updates$ are applied but before r returns, r would have to return v'. But it is impossible, because we assumed that r returns $v \neq v'$. Therefore let us assume that there is such T_k. We have two cases to consider:

1. T_k is a committed updating DU transaction or a committed updating SM transaction executed by p. If $t_k.updates$ are indeed applied by p after $t_i.updates$ are, then $t_k.end > t_i.end$ (p increments LC each time p applies updates of some transaction, line 63 or 88). By construction of S, T_k would

have to appear in S after T_i and before r returns. However, then w would not be the most recent write operation on x prior to r in S, a contradiction.

2. T_k is an aborted SM transaction executed by p, such that T_k's execution resulted from delivery of some request r_k using TOB. Since p applies $t_k.updates$ after $t_i.updates$, $t_k.end > t_i.end$ (lines 63 and 88). By the definition of SMreduce, the updates of T_k are applied to the local state of p only if there exists a committed updating transaction T'_k (with transaction descriptor t'_k) whose execution also resulted from delivery of r_k. By Proposition 29, $t_k = t'_k$, and thus $t_k.end = t'_k.end$. Hence, $t'_k.end > t_i.end$. By construction of S, it means that T'_k appears in S after T_i and before r returns. However, then w would not be the most recent write operation on x prior to r in S, a contradiction.

Since both cases yield contradiction, the assumption that there exists such transaction T_k is false. Therefore r has to return $v = v'$ thus concluding the proof by contradiction. Therefore HTR guarantees update-real-time opacity under SMreduce. □