DEFERRED-UPDATE DATABASE REPLICATION: THEORY AND ALGORITHMS

THÈSE N^{O} 4022 (2008)

PRÉSENTÉE LE 7 MARS 2008

À LA FACULTÉ INFORMATIQUE ET COMMUNICATIONS
LABORATOIRE DE SYSTÈMES D'EXPLOITATION
PROGRAMME DOCTORAL EN INFORMATIQUE, COMMUNICATIONS ET INFORMATION

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Rodrigo MALTA SCHMIDT

M.Sc. in computer science, State University of Campinas, Campinas, Brésil
et de nationalité brésilienne

acceptée sur proposition du jury:

Prof. C. Petitpierre, président du jury
Prof. W. Zwaenepoel, Prof. F. Pedone, directeurs de thèse
Dr M. Aguilera, rapporteur
Prof. B. Garbinato, rapporteur
Prof. A. Schiper, rapporteur

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE
Suisse
2008

Abstract

This thesis is about the design of high-performance fault-tolerant computer systems. More specifically, it focuses on how to develop database systems that behave correctly and with good performance even in the event of failures. Both performance and dependability can be improved by means of the same technique, namely replication. If several database replicas are available, performance can be improved by distributing the load among them. Moreover, if one of the replicas cannot be accessed due to failures, users can still rely on the other ones. However, providing the interface of a single database system out of several replicas is not an easy task since one has to ensure they are always consistent with each other. Allowing replicas to diverge would easily break the illusion of having a single high-performance fault-tolerant database system.

Although we would like to have replicas as independent of each other as possible for performance and dependability reasons, we must keep them synchronized if we want to provide a consistent interface to users. In this work, we study how we can balance this trade-off to provide good performance and fault-tolerance without compromising consistency. Our basis is a widely used technique for database replication known as the deferred update technique. In this technique, transactions are initially executed in a single replica. Passive transactions, which do not change the state of the database, can commit locally to the replica they execute. Active transactions, which change the database state, must be synchronized with the transactions running on other replicas.

This thesis makes four major contributions. First, we introduce an abstract specification that generalizes the deferred update technique. This specification provides a strong model to prove lower bounds on replication algorithms, design new correct-by-construction protocols tailor-made for specific settings, and prove existing protocols correct more easily, in a standard way. Using this model, we show that the problem of termination of active transactions in deferred-update protocols is highly related to the problem of sequence agreement among a set of processes. In this context, we study the problem of implementing latency-optimal fault-tolerant solutions to sequence agreement and present a novel, highly-dynamic, algorithm that can quickly adapt to system changes in order to preserve its optimal latency. Our algorithm is based on a new agreement problem we introduce that seems to be more suitable to solve problems like sequence agreement than previously used abstractions.

Our last two contributions are in the context of specific deferred-update algorithms, where we present two new fault-tolerant protocols derived from our general abstraction. The first algorithm uses no extra assumptions about database replicas. Yet, it has very little overhead associated with the termination of active transactions, propagating only strictly necessary information to replicas. Our second protocol uses strong assumptions about the concurrency control mechanism used by database replicas to reduce even more the latency and the burden associated with transaction termination. These algorithms are good examples of how our general abstraction can be extended to create new protocols and prove them correct.

Keywords: distributed systems, fault tolerance, crash-recovery, deferred-update replication, serializability, sequence agreement, consensus, Paxos, collision-fast, in-memory databases, transaction termination.

Résumé

Cette thèse aborde le projet de systèmes tolérants aux pannes de hautes performances. Plus précisément, elle se concentre sur le développement de bases des données qui se comportent correctement et avec bonne performance même en cas de défaillance. Tant la performance comme la fiabilité peut être améliorées par la même technique, c'est-à-dire la réplication. Si plusieurs répliques de base de données sont disponibles, la performance peut être améliorée en répartissant la charge entre eux. De plus, si l'une des répliques n'est pas accessible en raison des défaillances, les utilisateurs peuvent toujours compter sur les autres. Toutefois, fournir l'interface d'un seul système de base de données sur plusieurs répliques n'est pas une tâche facile car il faut assurer qu'ils soient toujours consistent les uns avec les autres. La permission de divergence des répliques peut facilement briser l'illusion d'avoir un simple système tolérant aux pannes de haute performance.

Bien que nous aimerions avoir des répliques comme indépendantes les uns des autres pour des raisons de fiabilité et de performance, nous devons garder les répliques synchronisées si nous voulons fournir une interface cohérente pour les utilisateurs. Dans ce travail, nous étudions comment nous pouvons équilibrer cet échange pour fournir de bonne performance et tolérants aux pannes sans compromettre la cohésion. Notre base est une technique largement utilisée pour réplication de base de données connue sous le nom de technique de mise à jour différée (deferred update technique). Dans cette technique, les transactions sont d'abord exécutées en une seule réplique. Les transactions passives, qui ne modifient pas l'état de la base de données, peuvent être validées (committed) localement à la réplique qu'elles exécutent. Les transactions actives, qui changent l'état de la base de données, doivent être synchronisées avec les transactions en exécution sur les autres répliques.

Cette thèse comporte quatre contributions majeures. Tout d'abord, nous présentons une spécification abstraite qui généralise la technique de mise à jour différée. Cette spécification fournit un modèle fort pour prouver bornes inférieures (lower bounds) sur les algorithmes de réplication, projeter des nouveaux protocoles corrects pour construction spécifiques pour des contextes particuliers, et prouver protocoles existants corrects plus facilement, et de façon standard. Par l'utilisation de ce modèle, nous montrons que le problème de la terminaison des transactions actives en protocoles de mise à jour différées est fortement lié

au problème d'accord de séquence (sequence agreement) entre un ensemble de processus. Dans ce contexte, nous étudions le problème d'implémentation des solutions tolérantes aux pannes de latence optimal pour l'accord de séquence et présentons un algorithme original et très dynamique, qui peut rapidement s'adapter aux modifications du système afin de préserver sa latence optimale. Notre algorithme est basé sur un nouveau problème d'accord qui semble être plus approprié pour résoudre des problèmes comme l'accord de séquence que les abstractions utilisées précédemment.

Nos deux dernières contributions sont dans le contexte des algorithmes spécifiques de mise à jour différés, où nous présentons deux nouveaux protocoles tolérants aux pannes dérivés de notre abstraction général. Le premier algorithme ne utilise pas de suppositions supplémentaires concernant les répliques de base de données. Cependant, il a très peu d'overhead associé à la terminaison des transactions actives, propageant seulement des informations strictement nécessaires aux répliques. Notre deuxième protocole utilise des fortes hypothèses sur le mécanisme de contrôle de concurrence utilisé par les répliques de la base de données pour réduire encore plus la latence et la charge associée à la terminaison de transactions. Ces algorithmes sont de bons exemples de la façon dont notre abstraction générale peut être étendu pour créer de nouveaux protocoles et les prouver correctement.

Mots-clés : systèmes distribués, tolérance aux fautes, technique de mise à jour différée, consistance, accord de séquence, consensus, Paxos, collision, base des données en mémoire, terminaison de transactions.

To Bianca,
for her love and support

Acknowledgments

I am very grateful to many people who helped in one way or another for this work. First of all, I would like to thank Fernando Pedone and Willy Zwaenepoel for their confidence in me, for accepting me in their research groups and for supervising me in this work.

I wish to thank all my lab colleagues at EPFL and USI for the important scientific discussions, collaborations, and friendship: Lásaro Camargos, Emmanuel Cecchet, Olivier Crameri, Steven Dropsho, Dan Dumitriu, Sameh Elnikety, Ming-Yee Iu, Aravind Menon, Nicolas Schiper, Marija Stamenkovic, Duy VO Duc, Marcin Wieloch, and Vaide Zuikeviciute.

I am thankful to Prof. Claude Petitpierre for accepting to preside the exam, and to the members of the jury, Dr. Marcos Aguilera, Prof. Benoît Garbinato, and Prof. André Schiper for the time they spent examining this thesis and for their valuable comments.

Several other people played important roles in my doctorate. Among them, my warm thanks to Suzanne Eichenberger and Madeleine Robert for their crucial help whenever it was necessary.

Preface

This thesis concerns the Ph.D. work I did under the supervision of Prof. Fernando Pedone and Prof. Willy Zwaenepoel at the School of Computer and Communication Sciences, EPFL, from 2004 to 2007. During this period, I also worked on (1) message diffusion in unreliable environments [GPS04a, AGPSS07, GPS04b], (2) checkpointing and rollback-recovery [SGPB05, SM06], (3) recovery of in-memory database federations [SP05], and (4) partial database replication [SSP06]. More recently, I have also worked on the use of multicoordination on Paxos [CSP07].

This thesis focuses on deferred-update database replication and its relation to the problem of distributed sequence agreement. Part of the results presented in it appear in three previous papers: [SP07, SCP07, CPS06].

[SP07] R. Schmidt and F. Pedone. A Formal Analysis of the Deferred Update Technique. In *Proceedings of the 11th International Conference on Principles of Distributed Systems (OPODIS-2007)*, Guadeloupe, French West Indies, December 2007.
Brief Announcement in *Proceedings of the 21st International Symposium on Distributed Computing (DISC-2007)*, Cyprus, September 2007.

[SCP07] R. Schmidt, L. Camargos, and F. Pedone. On Collision-fast Atomic Broadcast. EPFL Technical Report, Switzerland, 2007.

[CSP07] L. Camargos, R. Schmidt, and F. Pedone. Multicoordinated Paxos. EPFL Technical Report, Switzerland, December 2007.
Brief Announcement in *Proceedings of the 26th Annual ACM Symposium on Principles of Distributed Computing (PODC-2007)*, Portland, USA, August 2007.

[AGPSS07] M. Alani, B. Garbinato, F. Pedone, R. Schmidt, and M. Stamenković. Local-Knowledge Algorithms for Structured Message Diffusion in Unreliable Environments. In *2nd Workshop on Locality Preserving Distributed Computing Methods (LOCALITY 2007)*, Portland, USA, August 2007.

[SSP06] N. Schiper, R. Schmidt, and F. Pedone. Optimistic Algorithms for Partial Database Replication. In *Proceedings of the 10th International Conference on Principles of*

Distributed Systems (OPODIS-2006), Bordeaux, France, December 2006.
Brief Announcement in *Proceedings of the 20th International Symposium on Distributed Computing (DISC-2006)*, Stockholm, Sweden, 2006.

[CPS06] L. Camargos, F. Pedone, and R. Schmidt. A Primary-Backup Protocol for In-Memory Database Replication. In *Proceedings of the 5th IEEE International Symposium on Network Computing and Applications (NCA-2006)*, Cambridge, USA, July 2006.

[SM06] R. Schmidt and P. Murray. Woodfrog: A Persistence Library for Smartfrog Components. Technical Report HPL-2006-37, HP Labs, Bristol, UK, March 2006.

[SP05] R. Schmidt and F. Pedone. Consistent Main-Memory Database Federations under Deferred Disk Writes. In *Proceedings of the 24th IEEE Symposium on Reliable Distributed Systems (SRDS-2005)*, Orlando, USA, October 2005.

[SGPB05] R. Schmidt, I. C. Garcia, F. Pedone, and L. E. Buzato. Asynchronous Garbage Collection for RDT Checkpointing Protocols. In *Proceedings of the 25th International Conference on Distributed Computing Systems (ICDCS-2005)*, Columbus, USA, June 2005.
Brief Announcement in *Proceedings of the 23rd Annual ACM Symposium on Principles of Distributed Computing (PODC-2004)*, St. John's, Canada, July 2004.

[GPS04a] B. Garbinato, F. Pedone, and R. Schmidt. An Adaptive Algorithm for Efficient Message Diffusion in Unreliable Environments. In *Proceedings of the 2004 International Conference on Dependable Systems and Networks (DSN-2004)*, Florence, Italy, July 2004.

[GPS04b] B. Garbinato, F. Pedone, and R. Schmidt. A Modular Approach to Optimizing Highly-Dynamic Distributed Systems. Fast Abstract in *Proceedings of the 2004 International Conference on Dependable Systems and Networks (DSN-2004)*, Florence, Italy, 2004.

Contents

Chapter 1

Introduction

The beginning is the most important part of the work.
Plato

The focus of this thesis is on the design of high-performance fault-tolerant database systems that behave correctly and with good performance even in the event of failures. In this context, replication can be used to provide both performance and dependability. If several database replicas are available, performance can be improved by distributing the load among them. Moreover, if one of the replicas cannot be accessed due to failures, resource users can still rely on the other ones. However, providing the interface of a single database system out of several replicas is not an easy task since one has to ensure they are always consistent with each other. Allowing replicas to diverge would easily break the illusion of having a single high-performance fault-tolerant database system.

1.1 Database Replication

The problem of database replication is not recent and has been an active area of research for the last 30 years [Gif79, Sto79, Tho79]. Yet, the trade-off between the synchronization required to provide strong consistency and the performance overhead it generates is still very strong, even with recent technologies, leaving space for new contributions in the field.

1.1.1 Distributed Locking Approach

Early methods to handle database replication used distributed locks to synchronize replicas. The idea was basically to extend distributed locking protocols used for partitioned databases to handle replication as well. In this scenario, each data replica is considered as a different data item in the system. Read operations can acquire a single read lock from any of the replicas, but write transactions must acquire write locks on all data replicas in order to execute. Gray et al. have shown that this approach performs poorly as the number of replicated database sites increase, with a rapid growth on the number distributed deadlocks in the system [GHOS96].

Later works on database replication started to consider using group communication primitives with some ordering guarantees in order to avoid the problem of distributed deadlocks [AAAS97, PGS97, HAA99, PMJPKA00, RMA+02, KA00a, WPS+00a, WPS+00b]. Submitting operations to all replicas in total order helps to avoid deadlocks but adds a new overhead to the system since ensuring total order is expensive. This favored a class of optimistic protocols that executed transactions initially at a single replica and only synchronized replicas during transaction termination. Since updates are only propagated after the transaction has executed all its operations, this approach receives the name of *deferred update technique*.

More recently, research has also been pursued on mixing ordering guarantees with distributed locking to tolerate byzantine failures at the same time that distributed deadlocks are avoided [VBLM07].

1.1.2 Deferred-Update Replication

The main idea of this technique consists in executing all operations of a transaction initially on a single database. Transactions that do not change the database state can commit locally to the replica they executed, but other transactions must be globally certified and, if committed, have their update operations (those that change the database state) submitted to all database replicas. This technique is adopted by a number of database replication protocols in different contexts (e.g., [AT02, KA00b, PMJPKA00, PF00, PGS03, SSP06]) for its good performance in practical scenarios. Since most practical workloads have a vast majority of read-only transactions, it allows for a good balance of the load amongst the replicas, which execute such transactions completely independent of each other.

Replicas still have to synchronize to commit transactions that change the database state. However, the technique usually packs all the transaction operations in a single message, reducing the overhead as compared to techniques that propagate updates operation by operation.

1.1.3 Consensus and Paxos

Most deferred update algorithms rely directly on some sort of agreement abstraction to terminate update transactions. Consensus is one of the most basic agreement problems in distributed computing. In this problem, processes must agree on a single value, out of a set of proposed ones. Despite its simplicity, it is a powerful abstraction. State-machine replication [Lam78] is probably the most important application of consensus. In this approach, a reliable service is implemented by replicating it in several failure-independent processors, where replicas consistently change their states by applying deterministic commands from an agreed sequence. A consensus instance is used to decide on each command in the sequence.

Paxos is a very efficient and fault-tolerant consensus protocol originally intended for state-machine replication [Lam98]. It allows efficient implementations and is very resilient to failures, which favors its use in practical systems. For this reason, we use Paxos as the basis of most algorithms in this work. During normal execution of Paxos, a set of proposer processes (e.g., clients) send their proposed commands to an elected leader. Upon the receipt of a command C, the leader selects the next consensus instance to which no command has been proposed and forwards C, under the selected consensus instance, to a set of acceptor processes. Acceptors "accept" C and send a notification to a set of learner processes (e.g., replicas). Learners learn the decision of a consensus instance when they receive a notification coming from a quorum of acceptors. To tolerate the failure of the leader, each instance of consensus is further subdivided into rounds, explained later. Instances and rounds are similar in definition but completely different in purpose and the reader must be careful not to confuse them.

1.2 Research Contributions

This thesis provides the following four major contributions

Abstraction of the Deferred Update Technique. We present a formal abstraction of the deferred update technique for database replication. Despite its wide use, there was no formalization of it up to now, forcing the design, analysis, and correctness proof of this type of protocols to be done by non-standard mechanisms. Our abstraction allows one to come up with general results concerning this family of protocols (lower and upper bounds), prove their correctness, and easily design novel correct-by-construction protocols. Based on this model, we show that, contrary to the assumptions made by previous works on deferred-update replication, the technique can cope with a concurrency control mechanism weaker than strict order-preserving serializability on database replicas. In fact, this motivated us to introduce the concept of active order-preserving serializability, which generalizes the be-

havior of some optimistic concurrency control algorithms and can be safely employed on general deferred-update algorithms. Using this model, we show that the problem of termination of active transactions in deferred-update protocols is highly related to the problem of sequence agreement among a set of process.

M-Consensus and Collision-fast Paxos. Collision-fast Paxos a very efficient fault-tolerant solution to the problem of sequence agreement, a sequence-based specification of the atomic broadcast problem[HT93]. Our latency-optimal algorithm, derived from the celebrated Paxos consensus protocol [Lam98], is very dynamic and can quickly reconfigure and adapt to failures, which distinguishes it from previous approaches achieving similar bounds. Our algorithm is based on a variant of the consensus problem we call M-Consensus. M-Consensus is more general than the original consensus problem, being much more suitable as a building block for efficient sequence agreement implementations. We have extensively proved safety and liveness of our solutions to both M-Consensus and sequence agreement and we believe that some of the techniques we used are general enough to be applied to other agreement problems.

Certification-based Deferred-Update Algorithm. This is a general deferred-update algorithm that requires no extra assumptions about the database engines and, yet, has very little overhead associated with the termination of active transactions, propagating only strictly necessary information to replicas. We use the knowledge obtained from working with the Paxos protocol to implement termination very efficiently, with the same latency and degree of fault tolerance as the original algorithm. It can certify and propagate active transactions to replicas within three communication steps as seen from the client. We know of no previous protocol that can ensure this latency propagating the same amount of information we do.

In-memory Primary-Backup Replication Algorithm. In this algorithm, we use strong assumptions about the concurrency control mechanism used by database replicas to reduce even more the latency and the burden associated with transaction termination, requiring only two communication steps and no extra certification procedure to commit proposed transactions. This algorithm can be nicely coupled with in-memory databases to provide very good performance in practice. Besides its practical relevance, it also shows how our deferred-update abstraction can help the design and analysis of protocols even if termination depends on stronger assumptions about the consistency guarantees of database replicas.

1.3 Methodology

Our methodology for this work is to use correctness proofs based on refinement mappings from complex algorithms to simpler, easily-proven, abstractions, and write our algorithms and abstractions in terms of formal specifications.

1.3.1 Refinement Mappings

There are basically two ways of proving correctness of algorithms: behavioral and assertional reasoning. In behavioral reasoning, proofs are based on the possible behaviors of the algorithm and the actions whose execution led to one or another state. In assertional reasoning, proofs are based on invariants kept by the actions performed by the specification. Complex distributed algorithms can be very hard to reason about in terms of behaviors. The huge number of possible execution paths and corner cases can make it very easy to forget checking one or another possibility. For this reason, due to the complexity of our algorithms, we decided to use assertional reasoning to prove their correctness.

We use Lamport's transition axiom method [Lam89] to specify and reason about problems and algorithms. In this assertional method, correctness is proved by finding a correct refinement mapping from an algorithm to an abstract specification [AL91]. These refinement mappings are proved with the help of invariants and assertional reasoning about the algorithm. The advantage of this method is that it breaks correctness proofs in multiple independent parts and allows for very precise proofs.

To increase our confidence in the results we present, we wrote our proofs very carefully. Nevertheless, some steps can become very complicated and must be divided into sub-steps to avoid mistakes. The best method we found to structure our proofs without risking making mistakes is the one explained in [Lam95]. This is a very simple method that uses itemization to break a proof into multiple steps and help their presentation.

1.3.2 Formal Specifications

Practical systems and algorithms are usually complex and difficult to analyze and prove correct. Experience has shown that the "social process" of mathematics does not operate the same way in the world of system design [MLP79]. We will hardly see engineers discussing about correctness proofs of complicated systems, specially if they did not participate in their design. Correctness of complex systems depend mainly, if not uniquely, on how careful their designers were when developing their ideas. Careless design can lead to problems much more complicated than simple programming bugs since incorrect algorithms may force a

system to be completely redesigned.

In this work, we wanted to make our algorithms as practical as possible and, in order to cope with the problem above, we used formal specifications to reason about abstractions and algorithms in this thesis. This forced us to care about many important details and helped us find many mistakes in the original design of our ideas. More specifically, we used the TLA$^+$ specification language [Lam02] for its power and simplicity with respect to the specification of distributed systems. Moreover, the existing tools for TLA$^+$ allowed us to find and correct many mistakes and the possibility of model checking our specifications increased the confidence in our results.

In the chapters that follow, we use natural language and simple pseudo-code to describe our specifications of abstractions and algorithms. However, at the end of each chapter, we also present our unambiguous TLA$^+$ specifications. We have put them there to break any ambiguity or doubt that our higher-level description might create. We tried to put as many comments as possible in our specifications to make them more easily readable in case of necessity.

TLA$^+$ specifications are not very different from the pseudo-code based on atomic actions usually used to describe distributed algorithms. The most important part of any specification is the description of the atomic actions it allows, so we can concentrate on that for this brief introduction. In TLA$^+$, each action is specified as a logical expression on old and new variable values, where operator $'$ refers to the state a variable has after the action takes place. As an example, an action that simply assigns to variable x its previous value plus 3 is represented by the logic expression $x' = x + 3$.

Some aspects about its syntax might make expressions that define actions look more complicated than they actually are. One of these aspects is the use of indentation to avoid using parentheses. In TLA$^+$, we can write expression $(((a \lor b) \land (c \lor d)) \lor e)$ as shown in Figure 1.1. In fact, this visual representation of expressions makes it much easier to understand the complicated enabling conditions found in distributed algorithms.

$$
\begin{array}{l}
\lor\ \land\ \lor\ a \\
\qquad\ \lor\ b \\
\quad\ \land\ \lor\ c \\
\qquad\ \lor\ d \\
\lor\ e
\end{array}
$$

Figure 1.1: TLA$^+$ Example

To allow the composition of specifications, each action must specify the state change of every variable in the system. Leaving a variable unspecified means that it can assume any value if the action is executed. Operator UNCHANGED is used to simplify the task

for the list of variables that are not changed by an action. Another useful operator worth mentioning here is the operator EXCEPT . The expression $[q \text{ EXCEPT } ![x] = y]$, where q is a vector, returns a vector almost equal to q except for entry x, which is set to value y. This operator is extremely helpful when dealing with vectors.

In summary, TLA$^+$ specifications are given by a disjunction of action expressions. These actions are specified as the conjunction of their pre- and post-conditions, in a very intuitive way. As an example, to define an action $ChangeState(v)$ that sets entry x of vector vec to v when variable y equals $vec[x]$, we write the following TLA$^+$ expression.

$$\begin{array}{lll} ChangeState(v) & \triangleq & \land\ vec[x] = y \quad \text{pre-condition} \\ & & \land\ vec' = [vec \text{ EXCEPT } ![x] = v] \\ & & \land\ \text{UNCHANGED } \langle y,\ x \rangle \end{array}$$

1.4 Thesis Organization

The thesis is organized as follows. Chapter 2 introduces our deferred-update abstraction. It explains the database models we assume and proves some lower and upper bounds concerning termination of active transactions, including the necessity of solving the sequence agreement problem. This chapter also gives a simple example of how our abstract algorithm can be extended into a distributed implementation. Chapter 3 presents our latency-optimal and very dynamic solution to the sequence agreement problem, based on the Paxos consensus protocol. It also introduces an abstraction different from consensus, more suitable for fast sequence agreement implementations. Chapter 4 shows how the abstraction presented in Chapter 2 can be used to design new correct-by-construction deferred-update protocols. It introduces a general algorithm with very good latency and low overhead and a more specific algorithm that depends on special consistency guarantees given by the replicas to offer even lower latency and overhead. In Chapter 5, we summarize the major results of this work and outline future research directions.

Chapter 2

Formal Analysis of the Deferred Update Technique

If people do not believe that mathematics is simple,
it is only because they do not realize how complicated life is.
John Louis von Neumann

In the deferred update technique, a number of database replicas are used to implement a single serializable database interface. This technique is adopted by a number of database replication protocols in different contexts (e.g., [PGS97, KA00b, PMJPKA00, PF00, PGS03, SSP06]) for its good performance in general scenarios. The class of deferred update protocols is very heterogeneous, including algorithms that can optimistically apply updates of uncertified transactions [PMJPKA00], certify transactions locally to the database that executed them [KA00b], execute all concurrent update transactions at the same database [PF00], reorder transactions during certification [PGS97, PGS03], and even cope with partial database replication [SSP06]. However, all of them share the same basic structure, giving them some common characteristics and constraints.

Despite its wide use, we are not aware of any work that explored more deeply the theoretical aspects of deferred-update database replication. Ours seems to be the first attempt in this direction. In this chapter, we specify a general abstract deferred update algorithm that embraces all the protocols we know of. This general specification allows us to formally characterize the deferred update technique and rigorously prove its properties. As an example, we isolated the building block responsible for committing and propagating update transactions and showed its specific relation with distributed agreement problems. Our abstraction also eases the task of proving protocols correct, since it suffices to show a correct refinement mapping. Moreover, being a generalization of the technique, the abstract algorithm can also be used to derive new correct-by-construction protocols from it.

2.1 Serializability

The consistency criterion for transactional systems in general is *Serializability*, which is defined in terms of the equivalence between the system's actual execution and a serial execution of the submitted transactions [BHG87]. Traditional definitions of equivalence between two executions of transactions referred to the internal scheduling performed by the algorithms and their ordering of conflicting operations. This approach has led to different notions of equivalence and, therefore, different subclasses of Serializability [Pap79]. In a distributed scenario, however, defining equivalence in terms of the internal execution of the scheduler is not straightforward since there is usually no central scheduler responsible for ordering transaction operations. To compare a serial centralized schedule with a general distributed one (e.g., in a replicated database), one has to create mappings between the operations performed in both schedules and extend the notion of conflicting operations to deal with sets of operations, since a single operation in the serial centralized schedule may be mapped to a set of operations executed on different sites in the distributed one [BHG87]. This approach is highly dependent on the implemented protocol and, as explained in [LMWF94], does not generalize well.

Differently, we specify a general serializable database system, which responds to requests according to some internal serial execution of the submitted transactions. A database protocol satisfies serializability if it implements the general serializable database specification, that is, if its interface changes could be generated by the general serializable database. This sort of analysis is very common in distributed systems for its compromise between abstraction and rigorousness [Lam02, Lyn96, LMWF94].

In our specification of serializability, we first define all valid state transitions for normal interactions between the clients and the database, without caring about the values returned as responses to issued operations, but rather storing them internally as part of the transaction state. The database is free to abort a transaction at any time during the execution of its operations. However, a transaction t can only be committed if its commit request was issued and there exists a sequential execution order for all committed transactions and t that corresponds to the results these transactions provided. We say the transaction is *decided* if the database has aborted or committed it. Operations issued for decided transactions get the final decision as its result.

We assume each transaction has a unique identifier and let Tid be the set of all identifiers. We call Op the set of all possible transaction operations, which execute over a database state in set $DBState$ and generate a result in set $Result$ and a new database state. We abstract the correct execution of an operation by the predicate $CorrectOp(op, res, dbst, newdbst)$, which is true iff operation op, when executed over database state $dbst$, may generate res as the operation result and $newdbst$ as the new database state. This makes our specification completely independent of the set of allowed operations, coping with predicate-based or nondeterministic ones. As a simple example, one

could define a database with two integer variables x and y with read and write operations for each variable. In this case, $DBstate$ corresponds to all possible combinations of values for x and y, Op is the combination of an identifier for x or y with a read tag or an integer (in case of a write), and $Result$ is the set of integers. $CorrectOp(op, res, dbst, newdbst)$ is satisfied iff $newdbst$ and res correspond to the results for the read or write operation op applied to $dbst$.

Two special requests, $Commit$ and $Abort$, both not present in Op, are used to terminate a transaction, that is, to force a decision to be taken. Two special responses, $Committed$ and $Aborted$, not present in $Result$, are used to tell the database user if the transaction has been committed or aborted. We also define some auxiliary sets to help us deal more easily with these special requests and responses. Specifically, we define $Decided$ to equal the set $\{Committed, Aborted\}$, $Request$ to equal $Op \cup \{Commit, Abort\}$, and $Reply$ to equal $Result \cup Decided$.

During a transaction's execution, operations are issued and responses are given until the client issues a $Commit$ or $Abort$ request or the transaction is aborted by the database for some internal reason. We represent the history of a transaction execution by a sequence of elements in $Op \times Result$, corresponding to the sequence of operations executed on the transaction's behalf and their respective results. We say that a transaction history h is *atomically correct* with respect to initial database state $initst$ and final database state $finalst$ iff it satisfies the recursive predicate defined below, where $THist$ is the set of all possible transaction histories and $Head$ and $Tail$ are the usual operators for sequences. Moreover, for notation simplicity, we identify the first and second elements of a tuple t in $Op \times Result$ by $t.op$ and $t.res$, respectively.

$$
\begin{array}{l}
CorrectAtomicHist(h \in THist, initst, finalst \in DBState) \triangleq \\
\quad \textbf{if} \quad h = \langle\rangle \quad \textbf{then} \quad initst = finalst \\
\qquad\qquad\quad \textbf{else} \quad \exists ist \in DBState : \quad CorrectOp(Head(h).op, Head(h).res, initst, ist) \wedge \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad CorrectAtomicHist(Tail(h), ist, finalst)
\end{array}
$$

Intuitively, a transaction history is atomically correct with respect to $initst$ and $finalst$ iff there are intermediate database states so that all operations in the history can be executed in their correct order and generate their correct results.

During the system's execution, many transactions are started and terminated (possibly concurrently). We represent the current history of all transactions by a data structure called *history vector* that maps each transaction to its current history. We abstract this data structure by the set $THistVector$, containing all the possible history vectors. A sequence seq of transactions and a history vector $thist$ correspond to a correct serialization with respect to initial state $initst$ and final state $finalst$ iff the recursive predicate below is satisfied, where $Seq(S)$ represents the set of all finite sequences of elements in set S.

$$\begin{array}{l} \mathit{CorrectSerialization}(\mathit{seq} \in \mathit{Seq}(\mathit{Tid}), \mathit{thist} \in \mathit{THistVector}, \mathit{initst}, \mathit{finalst} \in \mathit{DBState}) \triangleq \\ \quad \textbf{if} \quad \mathit{seq} = \langle\rangle \quad \textbf{then} \quad \mathit{initst} = \mathit{finalst} \\ \qquad\qquad\quad \textbf{else} \quad \exists \mathit{ist} \in \mathit{DBState} : \quad \mathit{CorrectAtomicHist}(\mathit{thist}(\mathit{Head}(\mathit{seq})), \mathit{initst}, \mathit{ist}) \wedge \\ \qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{CorrectSerialization}(\mathit{Tail}(\mathit{seq}), \mathit{thist}, \mathit{ist}, \mathit{finalst}) \end{array}$$

Intuitively, this predicate is satisfied iff there are intermediate database states so that all transactions in the sequence can be atomically executed in their correct order generating the correct results for their operations. We can now easily define a predicate $\mathit{IsSerializable}(S, \mathit{thist}, \mathit{initst})$ for a finite set of transaction id's S, history vector thist, and database state initst, satisfied iff there is a sequence seq containing exactly one copy of each element in S and a final database state $\mathit{finalst}$ such that $\mathit{CorrectSerialization}(\mathit{seq}, \mathit{thist}, \mathit{initst}, \mathit{finalst})$ is satisfied. Predicate $\mathit{IsSerializable}$ indicates when a set of transactions can be serialized in some order, according to their execution history, so that every operation returns its correct result when the execution is started in a given database state.

We abstract the interface of our specification by the primitives $\mathit{DBRequest}(t, \mathit{req})$, which represents the reception of a request req on behalf of transaction t, and $\mathit{DBResponse}(t, \mathit{rep})$, which represents the database response to the last request on behalf of t with reply rep. The only restriction we make with respect to the database interface is that an operation cannot be submitted on behalf of transaction t if the last operation submitted for t has not been replied yet, which releases us from the burden of using unique identifiers for operations in order to match them with their results. Notice that the system still allows a high degree of concurrency since operations from different transactions can be submitted concurrently.

Our specification is based on the following internal variables:

thist**:** A history vector, initially mapping each transaction to an empty history.

tdec**:** A mapping from each transaction to its current decision status: $\mathit{Unknown}$, $\mathit{Committed}$, or $\mathit{Aborted}$. Initially, it maps each transaction to $\mathit{Unknown}$.

q**:** A mapping from each transaction to its current request or NoReq if no request is being executed on behalf of that transaction. Initially, it maps each transaction to NoReq.

Figure 2.1 presents the pseudo-code for the atomic actions of our specification. Action $\mathit{ReceiveReq}(t, \mathit{req})$ is responsible for receiving a request on behalf of transaction t. Action $\mathit{ReplyRep}(t, \mathit{rep})$ replies to a received request. It is enabled only if the transaction has been decided and the reply is the final decision or the transaction has not been decided but the current request is an operation (neither Commit nor Abort) and the reply is in Result. This means that responses given after the transaction has been decided carry the final decision and requests to commit or abort a transaction are only replied after the transaction has

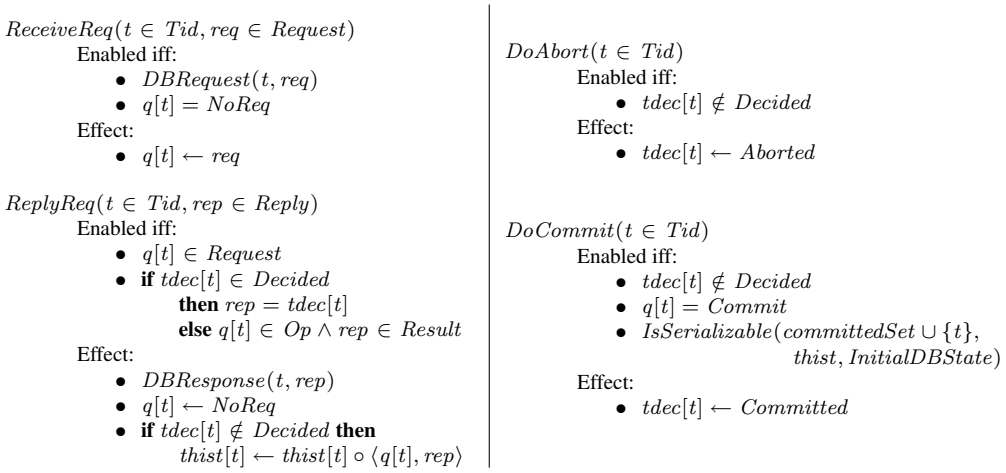

Figure 2.1: The atomic actions allowed in our specification of a serializable database.

been decided. Action $ReplyReq$ is responsible for updating the transaction history if the transaction has not been decided. It does that by appending the pair $\langle q[t], rep\rangle$ to $thist[t]$ (we use $\circ$ to represent the standard *append* operation for sequences). Action $DoAbort(t)$ simply aborts a transaction if it has not been decided yet. Action $DoCommit(t)$ commits t only if a t's commit request was issued and the set of all committed transactions (represented by $committedSet$) together with t is serializable with respect to the initial database state, denoted by the constant $InitialDBState$.

2.2 The Deferred Update Technique

2.2.1 Preliminaries

As mentioned before, deferred update algorithms initially execute transactions on a single replica. Transactions that do not change the database state (hereinafter called *passive*) may commit locally only, but *active* transactions (as opposed to passive ones) must be globally certified and, if committed, have their updates propagated to all replicas (i.e., operations that make them active). In order to correctly characterize the technique, we need to formalize the concepts of active and passive operations and transactions. An operation op is passive iff its execution never changes the database state, that is, iff the following condition is satisfied.

$$\forall st1, st2 \in DBState, rep \in Result : CorrectOp(op, rep, st1, st2) \Rightarrow st1 = st2 \quad (2.1)$$

An operation that is not passive is called active. Similarly, we define a transaction history h to be passive iff the condition below is satisfied.

$$\forall st1, st2 \in DBState : CorrectAtomicHist(h, st1, st2) \Rightarrow st1 = st2 \quad (2.2)$$

Notice that a transaction history composed of passive operations is obviously passive, but the converse is not true. A transaction that adds and subtracts 1 to a variable is passive even though its operations are active.

The deferred update technique requires some extra assumptions about the system. Operations, for example, cannot generate new database states nondeterministically for this could lead different replicas to inconsistent states. The following assumption makes sure that operations do not change the database state nondeterministically but still allows nondeterministic results to be provided to the database user.

Assumption 2.1 (State-deterministic operations) *For every operation op, and database states st and $st1$, if there is a result $res1$ such that $CorrectOp(op, res1, st, st1)$, then there is no result $res2$ and database state $st2$ such that $st1 \neq st2 \wedge CorrectOp(op, res2, st, st2)$.*

As for the database replicas, one may wrongly think that simply assuming that they are serializable is enough to ensure global serializability. However, two replicas might serialize their transactions (local and global) differently, making the distributed execution non-serializable. Consider, for example, an execution with two active transactions t_1 and t_2, operating in a database whose state is composed of two integer variables, x and y, both initially 0. Assume t_1 writes 1 to x, t_2 writes 1 to y, and all databases receive both transactions and commit them. Notice that any ordering of these two transactions is serializable. Now, assume that a single replica r_1 executes a passive transaction t_3 that reads the state $\langle x = 0, y = 1\rangle$. Since the sequence $\langle t_2, t_3, t_1\rangle$ represents a correct execution, t_3 is free to commit. Assume that a second replica r_2, which will never be notified of t_3 for it is a passive transaction, executes a passive transaction t_4 that reads the state $\langle x = 1, y = 0\rangle$. The sequence $\langle t_1, t_4, t_2\rangle$ represents a correct serial execution of the transactions submitted to replica r_2 and t_4 is also free to commit. However, no sequential execution containing all four transactions can generate their correct results, making it impossible for a serializable interface to commit all of them.

For this reason, previous works on deferred update protocols assumed the notion of *order-preserving serializability*, originally introduced by Beeri et al. in the context of nested transactions [BBG89]. In our model, order-preserving serializability ensures that the transactions' commit order represents a correct execution sequence, a condition satisfied by two-phase locking, for example. We show that this assumption can be relaxed since deferred update protocols can work with the weaker notion of *active order-preserving serializability* we introduce. Active order-preserving serializability ensures that there is an execution sequence of the committed transactions that generates their correct outputs and respects the commit order of all *active* transactions only. This notion is weaker than strict order-preserving serializability in that passive transactions do not have to provide results based on the latest committed state. Some multiversion concurrency control mechanisms [BHG87] are active order-preserving but not strict order-preserving.

$DoCommit(t \in Tid)$
Enabled iff:
- $tdec[t] \notin Decided$
- $q[t] = Commit$
- $\exists st \in DBState$:
  $CorrectSerialization(serialSeq \circ t, thist, InitialDBState, st)$

Effect:
- $tdec[t] \leftarrow Committed$
- $serialSeq \leftarrow serialSeq \circ t$

(a)

$DoCommit(t \in Tid)$
Enabled iff:
- $tdec[t] \notin Decided$
- $q[t] = Commit$
- $\exists seq \in Perm(committedSet \cup \{t\}), st \in DBState$:
  $CorrectSerialization(seq, thist, InitialDBState, st) \wedge$
  $ActiveExtension(serialSeq, t)$ is a subsequence of seq

Effect:
- $tdec[t] \leftarrow Committed$
- $serialSeq \leftarrow ActiveExtension(serialSeq, t)$

(b)

Figure 2.2: $DoCommit$ action for (a) strict and (b) active order-preserving serializability.

Specifications of order-preserving and active order-preserving serializability can be derived from our specification in Figure 2.1 by just adding a variable $serialSeq$, initially equal to the empty sequence, and changing the $DoCommit$ action. We show the required changes in Figure 2.2 below. The strict case (a) is simple and only requires that $serialSeq \circ t$ be a correct sequential execution of all committed transactions. The action automatically extends $serialSeq$ with t. The active case (b) is a little more complicated to explain and requires some extra notation. Let $Perm(S)$ be the set of all permutations of elements in finite set S (all the possible orderings of elements in S), and let $ActiveExtension(seq, t)$ be seq if $thist[t]$ is a passive history or $seq \circ t$ otherwise. The action is enabled only if there exists a sequence containing all committed transactions such that it represents a correct sequential execution and $ActiveExtension(seq, t)$ is a subsequence of it.[1] In this action, $serialSeq$ is extended with t only if t is an active transaction.

2.2.2 Abstract Algorithm

Our abstract deferred update algorithm generalizes the ideas of a handful of deferred update protocols and makes it easy to think about sufficient and necessary requirements for them

[1] sequence $subseq$ is a subsequence of seq iff it can be obtained by removing zero or more elements of seq.

to work correctly. Our specification assumes a set $Database$ of active order-preserving serializable databases, and we use the notation $DB(d)!Primitive(\_)$ to represent the execution of interface primitive $Primitive$ (either $DBRequest$ or $DBResponse$) of database d. Since transactions must initially execute on a single replica only, we let $DBof(t)$ represent the database responsible for the initial execution of transaction t. One important remark is that these internal databases receive transactions whose id set is $Tid \times \mathbb{N}$, where $\mathbb{N}$ is the set of natural numbers. This is done so because a single transaction in the system might have to be submitted multiple times to a database replica in order to ensure that it commits locally. Recall that our definition of active order-preserving serializability does not force transactions to commit. Therefore, transactions that have been committed by the algorithm and submitted to the database replicas are not guaranteed to commit unless further assumptions are made. The only way around this is to submit these transactions multiple times (with different versions) until they commit. Besides the set of databases, we assume a concurrent termination protocol, fully explained in the next section, responsible for committing active transactions and propagating their active operations to all databases.

The algorithm we present in the following orchestrates the interactions between the global database interface and the individual internal databases. It is mainly based on the following internal variables:

$thist$, q**:** Essentially the same variables as in the specification of a serializable database.

$dreq$**:** A mapping from each transaction t to the operation that is currently being submitted for execution on $DBof(t)$, or $NoReq$ if no operation is being submitted. This variable is used to implement the asynchronous communication that tells $DBof(t)$ to execute an operation of t. Initially all transactions are mapped to $NoReq$.

$dreply$**:** Similar to $dreq$, but mapping each transaction t to the last response given by $DBof(t)$.

$dcnt$**:** A mapping from each database d and transaction t to an integer representing the number of operations that executed on d for t. It counts the number of operations $DBof(t)$ has executed for t during t's initial execution and, if t is active, the number of active operations the other databases (or $DBof(t)$ if it does not manage to commit t directly after it is globally committed) have executed for t after it is globally committed. It is initially 0 for all databases and transactions.

$pdec$**:** A mapping like $tdec$ in the specification of a serializable database, used to tell whether the transaction was decided without being proposed for global termination either because it was prematurely aborted during its initial execution or because it was a passive transaction that committed on its execution database.

$vers$**:** A mapping from each database d and transaction t to an integer representing the current version of t being submitted to d. It is initially 0 for all databases and transactions.

$dcom$**:** A mapping from each database d and transaction t to a boolean telling whether t has been committed on d. It is initially false for all databases and transactions.

When a $Commit$ request is issued for a transaction whose history has been active, a decision must be taken on whether to commit or abort this transaction with respect to active transactions executed on other databases. In our specification, this is done separately by a termination protocol. The reason why we isolated this part of the specification is twofold. First, the nature of the rest of the algorithm is essentially local to the database that is executing a given transaction and it seems interesting to separate it from the part of the specification responsible for synchronizing active transactions executed on different databases. Second, the properties of the termination protocol, when isolated, can be related to properties of other agreement problems in distributed computing, which helps understand and solve it. The interface variables of the termination protocol used in our general specification are the following:

$proposed$**:** This is an input variable that keeps the set of all proposed transactions. It is initially empty.

$gdec$**:** An output variable that keeps a mapping like $pdec$ above, but managed by the termination protocol only. It tells whether a proposed transaction has already been decided or not.

$learnedSeq$**:** Another output variable mapping each database d to a sequence of globally committed active transactions. This sequence tells database d the order in which these active transactions must be committed to make the whole execution serializable. Initially, it maps each database to the empty sequence.

Our specification implements a serializable database, which can be proved by a refinement mapping from its internal variables to those of a general serializable database. Actually, the only internal variable of our specification of a serializable database not directly implemented in our abstract algorithm is $tdec$, given by joining the values of $pdec$ and $gdec$ in the following way:

$$tdec[t] \triangleq \textbf{if } t \in proposed \textbf{ then } gdec[t] \textbf{ else } pdec[t] \tag{2.3}$$

For simplicity, we use this definition of $tdec$ in some parts of our specification. Another extra definition used in our algorithm is the $ActHist(t)$ operator that returns a subsequence of $thist[t]$ containing all its active operations. The atomic actions of our abstract algorithm, without the internal actions of the individual databases and the termination protocol, are shown in Figure 2.3.

Action $ReceiveReq$ treats the receipt of a transaction request. If the transaction responsible for the operation has been decided (either for $pdec$ or $gdec$ according to the definition

$ReceiveReq(t \in Tid, req \in Request)$
Enabled iff:
- $DBRequest(t, req)$
- $q[t] = NoReq$

Effect:
- $q[t] \leftarrow req$
- **if** $tdec[t] \notin Decided$ **then**
  if $req = Commit \ \wedge \ thist[t]$ "is active"
  then $proposed \leftarrow proposed \cup \{t\}$
  else $dreq[t] \leftarrow req$

$ReplyReq(t \in Tid, rep \in Reply)$
Enabled iff:
- $q[t] \in Request$
- **if** $tdec[t] \in Decided$
  then $rep = tdec[t]$
  else $q[t] \in Op \wedge rep \in Result \wedge$
  $dcnt[DBof(t)][t] > Len(thist[t]) \wedge$
  $rep = dreply[t]$

Effect:
- $DBResponse(t, rep)$
- $q[t] \leftarrow NoReq$
- **if** $tdec[t] \notin Decided$ **then**
  - $thist[t] \leftarrow thist[t] \circ \langle q[t], rep \rangle$
  - $dreq[t] \leftarrow NoReq$

$PrematureAbort(t \in Tid)$
Enabled iff:
- $t \notin proposed$
- $pdec[t] \notin Decided$

Effect:
- $pdec[t] \leftarrow Aborted$

$PassiveCommit(t \in Tid)$
Enabled iff:
- $t \notin proposed$
- $pdec[t] \notin Decided$
- $dreply[t] = Committed$

Effect:
- $pdec[t] \leftarrow Committed$

$DBReq(d \in Database, t \in Tid, req \in Request)$
Enabled iff **any** of the conditions below hold.

Condition 1: (external operation request)
- $d = DBof(t)$
- $dreq[t] = req$
- $dcnt[d][t] = Len(thist[t])$

Condition 2: (operation after termination)
- $t \in proposed$
- $dcnt[d][t] < Len(ActHist(t))$
- $req = ActHist(t)[dcnt[d][t] + 1].op$

Condition 3: (commit after termination)
- $req = Commit$
- $\exists i \in 1..Len(learnedSeq[d]) :$
  $learnedSeq[d][i] = t \wedge$
  $\forall j < i : dcom[d][learnedSeq[d][j]]$
- **either** $d = DBof(t) \ \wedge \ vers[d][t] = 0$
  or $dcnt[d][t] = Len(ActHist(t))$

Effect:
- $DB(d)!DBRequest(\langle t, vers[d][t] \rangle, req)$

$DBRep(d \in Database, t \in Tid, rep \in Reply)$
Enabled iff:
- $DB(d)!DBResponse(\langle t, vers[d][t] \rangle, rep)$

Effect:
- **if** $d = DBof(t)$ **then** $dreply[t] \leftarrow rep$
- **if** $rep = Aborted \ \wedge \ t \in proposed$ **then**
  - $vers[d][t] \leftarrow vers[d][t] + 1$
  - $dcnt[d][t] \leftarrow 0$

  else if $rep \in Result$ **then**
  - $dcnt[d][t] \leftarrow dcnt[d][t] + 1$

  else
  - $dcom[d][t] \leftarrow rep = Committed$

Figure 2.3: The atomic actions our abstract protocol.

of $tdec$ given above), then it only changes $q[t]$. Otherwise, it either proposes t for the termination protocol or sends the request to $DBof(t)$ through variable $dreq[t]$. Our complete specification allows passive transactions to be submitted for the termination protocol too and this is why we wrote "is active" between quotation marks. We allow that because sometimes it might not be possible to identify all passive transactions. Therefore, our specification also embraces algorithms that identify only a subset of the passive transactions as passive and conservatively propose the others for global termination.

Action $ReplyReq$ replies a transaction request. It is very similar to the original $ReplyReq$ action of our serializable database specification. The small differences only make sure that the value replied for a normal operation comes from $DBof(t)$ and, in this case, $dreq[t]$ is set back to $NoReq$ to wait for the next operation. Actions $PrematureAbort$ and

$PassiveCommit$ abort or commit a transaction that has not been proposed for global termination. It can only be committed if a commit request was correctly replied by $DBof(t)$, which can only happen if t has a passive history.

Action $DBReq$ submits a request to a database. There are three conditions that enable this action. The first one represents a normal request during the transaction's initial execution or a commit request for a passive transaction. The second one represents an operation request for an active transaction that has been proposed to the termination protocol. Notice that operations of proposed transactions can be optimistically submitted to the database before they commit or appear in some $learnedSeq$. Some algorithms do that to save processing time after the transaction is committed, reducing the latency for propagating transactions to the replicas. The third condition that enables this action represents a commit request for a transaction that has been committed by the termination protocol. For that to happen, the transaction must be present in $learnedSeq[d]$ and all transactions previous to it in the sequence must have been committed on that database. Moreover, all active operations of that transaction must have been applied to the database already, which is true if the database is the one originally responsible for the transaction and it has not changed the transaction version or the operations counter $dcnt[d][t]$ equals the number of active operations in the transaction history. Recall that, by the definition of a serializable database, a request can only be submitted if there is no pending request for the same transaction. This is actually an implicit pre-condition for $DBReq$ given by the specification of a serializable database.

Action $DBRep$ treats the receipt of a response coming from a database. If the database is the one responsible for initially executing the transaction, it sets $drepy[t]$ to the value returned. If the transaction is aborted but it has been proposed for global termination, it changes the version of that transaction on that database and sets the operation counter to zero so that the transaction's operations can be resubmitted for its new version; otherwise, it just increments the operation counter and sets $dcom$ accordingly.

2.2.3 Termination Protocol

The termination protocol gives a final decision to proposed transactions and, if they are committed, forwards them to the database replicas. It "reads" from variables $proposed$ and $thist$ (it relies on the transaction history to decide on whether to commit or abort it), and changes variables $gdec$ and $learnedSeq$. As explained before, variable $gdec$ simply assigns the final decision to a transaction; $learnedSeq$, however, represents the order in which each database should submit the active transactions committed by the termination protocol. These are the three safety properties that define the termination protocol:

Nontriviality For any transaction t, t is decided ($gdec[t] \in Decided$) only if it was proposed.

Stability For any transaction t, if t is decided at any time, then its decision does not change at any later time; and, for any database d, the value of $learnedSeq[d]$ at any time is a prefix of its value at all later times.

Consistency There exists a sequence seq containing exactly one copy of every committed transaction (according to $gdec$) and a database state st such that $CorrectSerialization(seq, thist, InitialDBState, st)$ is true and, for every database d, $learnedSeq[d]$ is a prefix of seq.

The following theorem asserts that our complete abstract specification of a deferred update protocol is serializable. This result shows that every protocol that implements our specification automatically satisfies serializability. The proofs of our theorems are given in Section 2.4.

Theorem 2.1 *Our abstract deferred update algorithm implements the specification of a serializable database given in Section 2.1.*

This theorem results in an interesting corollary, stated below. It shows that indeed databases are not required to be strict order-preserving serializable, an assumption that can be relaxed to our weaker definition of active order-preserving serializability.

Corollary 2.2 *Serializability is guaranteed by our specification if databases are active order-preserving serializable instead of strict order-preserving serializable.*

The three aforementioned safety properties are not strictly necessary to ensure serializability. Nontriviality can be relaxed so that non-proposed transactions may be aborted before they are proposed and Serializability is still guaranteed. However, we see no practical use of this since our algorithm already allows a transaction to be aborted at any point of the execution before it is proposed. Committing a transaction before proposing depends on making sure that the history of the transaction will not change and, in case it is active, on whether there are alternative sequences that ensure the Consistency properties if the transaction is committed or not, a rather complicated condition to be used in practice. Stability can be relaxed by allowing changes on suffixes of $learnedSeq[d]$ that have not been submitted to the database yet. However, keeping knowledge of what part of the sequence has already been submitted to the database and possibly changing the rest of it is equivalent to implementing our abstract algorithm with $learnedSeq[d]$ being the exact sequence locally submitted to the database. As a result, we see no practical advantage in relaxing Stability.

Consistency can be relaxed in a more complicated way. In fact, the different sequences $learnedSeq[d]$ can differ, as long as the set of intermediate states they generate (states in between transactions) are a subset of the intermediate states generated by a sequence seq corresponding to a permutation of all globally committed transactions that satisfies $CorrrectSerialization(seq, thist, InitialDBState, st)$ for some state st. Ensuring this property without forcing the $learnedSeq$ sequences to prefix a common sequence is hard and may lead to situations in which committed transactions cannot be added to a sequence $learnedSeq[d]$ for they would generate states that are not present in any sequence that could satisfy our consistency criterion.

One might think, for example, that the consistency property can be relaxed to allow commuting transactions that are not related (i.e., operate on disjunct parts of the database state) in the sequences $learnedSeq[d]$. For that, however, we have to make some assumptions about the database state in order to define what we mean by disjunct parts of the database state. For simplicity, let us assume our database state is a mapping from objects in a set $Object$ to values in a set $Value$ and operations can read or write a single object value. We define the objects of a transaction history h, represented by $Obj(h)$, to be the set of objects the operations in h read or write. A consistency property based on the commutativity of transactions that have no intersecting object sets can be intuitively defined as follows:

Alternative Consistency There exists a sequence seq containing exactly one copy of every committed transaction (according to $gdec$) and a database state st such that $CorrectSerialization(seq, thist, InitialDBState, st)$ is true and, for every database d, $learnedSeq[d]$ contains exactly one copy of some committed transactions (according to $gdec$) and, for every transaction t in $learnedSeq[d]$, the following conditions are satisfied:

- Every transaction t' that precedes t in seq and shares some objects with t also precedes t in $learnedSeq[d]$, and
- Every transaction t' that precedes t in $learnedSeq[d]$ either precedes t in seq or shares no objects with t.

Although this new consistency condition seems a little complicated, it is weaker than our original property for it allows the sequences $learnedSeq[d]$ differ in their order for transactions that operate on different objects. The following theorem shows that this property is not enough to ensure Serializability in our abstract algorithm.

Theorem 2.3 *Our abstract deferred update algorithm with the Consistency property for termination changed for the Alternative Consistency property defined above does not implement the specification of a serializable database given in Section 2.1.*

This result basically means that one cannot profit much from using Generic Broadcast [PS02] algorithms to propagate committed transactions. Our properties as originally defined seem to be the weakest practical conditions for ensuring Serializability in deferred update protocols. In fact, we are not aware of any deferred update replication algorithm whose termination protocol does not satisfy the three properties above.

So far, we have not defined any liveness property for the termination protocol. Although we do not want to force protocols to commit transactions in any situation (since this might rule out some deferred update algorithms that conservatively abort transactions), we think that a termination protocol that does not update the sequences $learnedSeq[d]$ eventually, after having committed a transaction, is completely useless. Therefore, we add the following liveness property to our specification of the termination protocol:

Liveness If t is committed at a given time, then $learnedSeq[d]$ eventually contains t.

As it happens with agreement problems like Consensus, this property must be revisited in failure-prone scenarios, since it cannot be guaranteed for databases that have crashed. Independently of that, one can easily spot some similarities between the properties we have defined and those of Sequence Agreement as explained in [Lam04]. Briefly, in the sequence agreement problem, a set of processes agree on an ever-growing sequence of values, built out of proposed ones. The problem is specified in terms of proposer processes that propose values to be learned by learner processes, where $learned[l]$ represents the sequence of values learned by learner l. Sequence Agreement is defined by the following properties:

Nontriviality For any learner l, the value of $learned[l]$ is always a sequence of proposed values.

Stability For any learner l, the value of $learned[l]$ at any time is a prefix of its value at any later time.

Consistency For any learners l_1 and l_2, it is always the case that one of the sequences $learned[l_1]$ and $learned[l_2]$ is a prefix of the other.

Liveness If value V has been proposed, then eventually the sequence $learned[l]$ will contain V as an element.

This problem is a sequence-based specification of the celebrated atomic broadcast problem [HT93]. The exact relation between the termination protocol and Sequence Agreement is given by the following theorem.

Theorem 2.4 *The four properties Nontriviality, Stability, Consistency, and Liveness above satisfy the safety and liveness properties of Sequence Agreement for transactions that commit.*

One possible way of reading this theorem is that any implementation of the termination protocol is free to abort transactions, but it must implement Sequence Agreement for the transactions it commits. As a consequence, any lower bound or impossibility result for atomic broadcast and consensus applies to the termination protocol.

2.3 A Simple Implementation

It is not very difficult to extend the previous algorithm to more specific distributed implementations. In this section, we give the main idea of how it can be done with a simple example. The system is composed of clients, databases and a Sequence Agreement implementation where clients propose and databases learn, as shown in Figure 2.4 below. Clients access the replicated serializable interface through a local driver. For the execution of operations, these drivers contact the databases directly. Passive transactions can commit locally at the database they are executed but active transactions have their histories broadcast to all databases through the Sequence Agreement primitive. Databases certify these transactions based on the order they are delivered and apply their updates in case they are committed.

More specifically, when a client starts a new transaction t, its local driver connects to $DBof(t)$ (chosen randomly or given by some load balancing mechanism not depicted here). The local driver keeps variables $q[t]$, $dreq[t]$, $thist[t]$, and $pdec[t]$, for all transactions t that can have requests issued by that client (we tacitly assume that a single transaction cannot have requests issued by different clients). The driver also keeps the subset of $proposed$ containing all proposed transactions coming from that client.

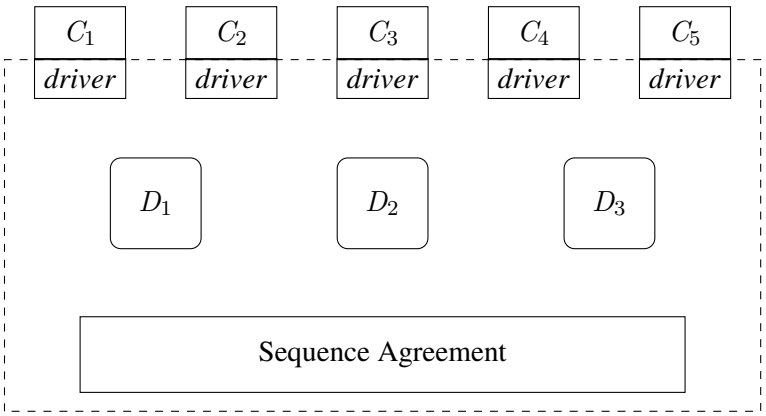

Figure 2.4: Structure of our architecture.

When a client issues an operation request (neither $Commit$ nor $Abort$), the driver executes action $ReceiveReq$ locally and forwards the requests to the database responsible for that transaction through unreliable numbered messages. The number in each message re-

flects the length of $thist[t]$ at the time the message was sent and the highest-numbered message, while undelivered, contains the current value of $dreq[t]$. Since these messages can be lost, the driver may want to resend them in case it suspects of message loss.

Database d is wrapped by a local proxy that keeps, for any transaction t, variables $dcnt[d][t]$, $vers[d][t]$, $dcom[d][t]$, $learned[d]$ (output variable of the sequence agreement implementation), and $learnedSeq[d]$ (output variable of the termination protocol). Moreover, it keeps variable $dreply[t]$ for all transactions t such that $DBof(t) = d$ and a variable $dgdec[d][t]$ that keeps a local copy of $gdec[t]$ at d, for every transaction t. When the proxy receives a numbered message from a client driver, it submits the message operation to its local database only if its transaction is not currently executing an operation and the message number corresponds to the next operation to be executed, which clearly implements action $DBReq$ enabled by Condition 1. Notice that, in this case, the proxy can safely assume that the transaction has not been proposed and will not be proposed until after the database gives a result back. When the database responds to the request, the proxy executes action $DBRep$ as specified in the abstract algorithm, knowing that $d = DBof(t)$ and t has not been proposed, which means that the first **if** condition evaluates to $true$ and the second one to $false$. In this case, the proxy sends a message containing the database response, which equals $dreply[t]$, and the new value of $dcnt[d][t]$ to the driver of the client responsible for t. When the driver receives a response message such that the value of $dcnt[d][t]$ it contains is greater than $Len(thist[t])$, it is guaranteed that neither $dcnt[d][t]$ nor $dreply[t]$ has changed since the message was sent. It can then locally execute $ReplyReq$ and give a response back to the client.

The client keeps on issuing operations and getting responses for t according to the aforementioned procedure, until it decides to terminate the transaction by an $Abort$ or $Commit$ request. If the client issues an $Abort$ request, the driver can locally execute the sequence of actions $\langle ReceiveReq, PrematureAbort, ReplyReq\rangle$, without sending any messages. However, if the client issues a $Commit$ request, different actions must be taken. If the current transaction history is passive, the driver behaves exactly as if the request was a normal operation. $DBof(t)$ will receive the message containing the $Commit$ request, apply it to the database, and return a message back with the database response. At this point, instead of executing $ReplyReq$, disabled because $q[t] \notin Op$, the driver will choose between executing $PrematureAbort$ or $PassiveCommit$ depending on the response given by the database.

If the client has issued a $Commit$ request for an active transaction t, the driver will add it to its local $proposed$ set. At this point, it will propose tuple $\langle t, thist[t]\rangle$ to the sequence agreement primitive. Databases build the sequence $learnedSeq[d]$ based on $learned[d]$ and some deterministic function $DSelect(seq)$ that returns a subsequence $subseq$ of seq that satisfies $CorrectSerialization(subseq, thist, InitialDBState, st)$ for some (unique) database state st. We explain in the following how $DSelect$ can be implemented. For now, let us concentrate on how it can be used to implement the termination protocol. In order to simplify our algorithm, we also assume that $DSelect(seq)$ is always a prefix of $DSelect(seq \circ v)$, for any sequence v. In this implementation, $learnedSeq[d]$ is given by $DSelect(learned[d])$

and $dgdec[d][t]$ is $Unknown$ if t does not appear in $learned[d]$, $Committed$ if it appears in $learnedSeq[d]$, and $Aborted$ in case t appears in $learned[d]$ but it is not selected by $DSelect$. When $dgdec[DBof(t)][t]$ changes from $Unknown$ to $Committed$ or $Aborted$, $DBof(t)$ sends a message to the driver of the client responsible for t notifying the transaction outcome. If the driver suspects $DBof(t)$ to have crashed, it can simply ask any other database for this outcome. Recall that every database d can calculate $dgdec[d][t]$ and since $DSelect$ is deterministic, all databases will have the same decision for t.

The simplest implementation of $DSelect(seq)$ is given by returning always the empty sequence. This, however, would force all active transactions to abort, never changing the database state. A more plausible implementation for $DSelect(seq)$ can be recursively defined as follows. If seq is empty, it returns the empty sequence. Otherwise, let l be the last element of seq, and $pseq$ be the prefix of seq that includes all its elements but l. We first recursively calculate $DSelect(pseq)$ and call it $pseqsel$. If $CorrectSerialization(pseqsel \circ \langle l.id \rangle, thist, InitialDBState, st)$ is satisfied, it returns $pseqsel \circ \langle l \rangle$; otherwise, it returns $pseqsel$ only.

This simplistic algorithm description avoids many important implementation details. In fact, a number optimizations are possible and some redundant variables can be eliminated from the algorithm. Our purpose, however, is just to give a simple idea of how our specification can be extended without much effort. The algorithm we presented in this section is a generalization of the Database State Machine (DBSM) algorithm given in [PGS03]. In the original paper, some extra assumptions about the databases' concurrency control mechanism are made to allow capturing and using the set of items read only instead of read operations and results. This assumption also simplifies the evaluation of predicate $CorrectSerialization$, making it more practical. Moreover, the original DBSM also requires changing the internals of the database in order to force certified active transaction to be committed in their first execution. Our general specification deals nicely with this case by allowing transactions to have multiple versions inside each database and guaranteeing that only one of them commits.

2.4 Theorem Proofs

We now prove the three theorems presented in Section 2.2.3.

2.4.1 Proof of Theorem 2.1

The first theorem is the hardest one to prove since it involves reasoning about the actions defined in our abstract algorithm. To avoid mistakes, we divided the proof into parts and proved each one in a structured way.

2.4.1.1 Main Invariants

In order to prove Theorem 2.1, we state three invariants satisfied by our abstract algorithm. The invariants are very intuitive, given the algorithm's expected behavior. However, a rigorous proof that the algorithm actually satisfies them is given in Section 2.4.1.3. Before presenting these invariants, though, we introduce some auxiliary notation. We let $CommittedAt(d)$ be the set of all transactions that have been committed at database d under any version number. That is,

$$CommittedAt(d) \triangleq \{t \in Tid : \exists v \in \mathbb{N} : DB(d)!tdec[\langle t, v\rangle] = Committed\}.$$

Moreover, we let $NVserialSeq(d)$ be the standard projection of sequence $DB(d)!serialSeq$ without the transactions' version numbers.

Our first invariant relates the databases' internal states to the global variables $thist$ and $learnedSeq$. It is mainly based on the fact that databases are active order-preserving serializable and transactions proposed to the termination protocol (which includes all active ones) have their $Commit$ requests submitted to database d according to the order specified by $learnedSeq[d]$.

Database Invariant

For every database d, there exists a sequence $seq \in Perm(CommittedAt(d))$, and a database state st such that all conditions below hold:

1. $CorrectSerialization(seq, thist, InitialDBState, st)$,
2. $NVserialSeq(d)$ is the subsequence of seq containing all its active transactions, and
3. $NVserialSeq(d)$ is the subsequence of a prefix of $learnedSeq[d]$ that contains all its active transactions.

Besides the invariant above, our proof uses the following two auxiliary invariants.

tdec **Invariant** For every transaction t, if $t \notin proposed$ and $pdec[t] = Committed$, then $thist[t]$ is passive and $t \in CommittedAt(DBof(t))$.

dreply **Invariant** For every transaction t, if $dreply[t] = Committed$ and $t \notin proposed$, then $thist[t]$ is passive and t belongs to $CommittedAt(DBof(t))$.

2.4.1.2 Theorem Proof

Theorem 2.1 *Our abstract deferred update algorithm implements the specification of a serializable database given in Section 2.1.*

PROOF: The proof is by a refinement mapping where $thist$ and q are implemented by the variables with the same name and $tdec$ is implemented according to the definition in terms of $proposed$, $pdec$, and $gdec$ given in the explanation of our abstract deferred update algorithm. Below, we show that each action executed by the abstract algorithms implements an action of our serializable database specification.

1. Action $ReceiveReq$ implements the action with the same name.
   PROOF SKETCH: Pre- and post-conditions on variables q and $thist$ are exactly the same. The action may change variable $proposed$, influencing $tdec$. However, t is proposed only if $tdec[t] \notin Decided$ and the Nontriviality property of the termination protocol ensures that $tdec$ remains the same.
2. Action $ReplyReq$ implements the action with the same name.
   PROOF SKETCH: The actions' pre- and post-conditions are obviously stricter than those of the original action.
3. Action $PrematureAbort$ implements action $DoAbort$
   PROOF SKETCH: The action changes $pdec$ iff its changes reflect the changes on $tdec$ performed by action $DoAbort$.

We now skip action $PassiveCommit$ for it deserves a slightly more complicated analysis. It is explained right after the simple actions below.

4. Actions $DBReq$, $DBRep$, and internal actions performed by any database represent stuttering steps in our specification of a serializable database
   PROOF SKETCH: Such actions do not change variables q, $thist$, $proposed$, $pdec$ and $gdec$, not influencing the mapping.
5. Changes on $learnedSeq$ performed by the termination protocol also implement stuttering steps
   PROOF SKETCH: Such changes have no influence on variables q, $thist$, $proposed$, $pdec$ and $gdec$.

The two cases below deserve a special analysis and a higher degree of rigorousness.

6. Action $PassiveCommit$ implements action $DoCommit$
   PROOF: Let $committedSet$ be the set of all committed transactions (according to the definition of $tdec$ in terms of $pdec$ and $gdec$). We must show that $IsSerializable(committedSet \cup \{t\}, thist, InitialDBState)$ is true before the execution of $PassiveCommit$. We prove that in the following proof steps.
   6.1. Choose a sequence $gseq$ that contains exactly one copy of every transaction mapped to $Committed$ in $gdec$ and satisfies the two conditions below
   1. $\exists st \in DBState : CorrectSerialization(gseq, thist, InitialDBState, st)$
   2. $\forall d \in Database : learnedSeq[d]$ is a prefix of $gseq$
   PROOF: This sequence exists for the Consistency property of the termination protocol.
   LET: $subgseq$ be the subsequence of $gseq$ containing all its active transactions.
   6.2. For every database d, $NVSerialSeq(d)$ is a prefix of $subgseq$.
   PROOF: By step 6.1 and the third item of the Database Invariant.

6.3. For any transaction t contained by $subgseq$, let $stgen(t)$ be the unique database state st such that $CorrectSerialization(preft, thist, InitialDBState, st)$ is satisfied for the prefix $preft$ of $subgseq$ limited by (and containing) t.
PROOF: Such state exists by the step $PC1$ and the definition of $subgseq$, and it is unique by Assumption 2.1 (State-deterministic Operations).

6.4. For every passive transaction t that belongs to $CommittedAt(DBof(t))$, that is, every transaction committed with some version at its delegate database, either one of the two conditions below is satisfied:

- $CorrectAtomicHist(thist[t], InitialDBState, InitialDBSTAte)$, or
- $\exists t_w \in Tid$:
  - t_w appears in $subgseq$ and
  - $CorrectAtomicHist(thist[t], stgen(t_w), stgen(t_w))$

PROOF: By the Database Invariant and the fact that t belongs to $CommittedAt(DBof(t))$, there exists a sequence seq such that:
1. seq contains t,
2. there exists a database state st such that
$$CorrectSerialization(seq, thist, InitialDBState, st),$$
3. $NVserialSeq(DBof(t))$ is the subsequence of seq containing all its active transactions,
4. $NVserialSeq(DBof(t))$ is the subsequence of a prefix of $learnedSeq[d]$ containing all its active transactions.

Let $strippedseq$ be the subsequence of seq containing all its active transactions and t, only. Since only passive transactions are taken out, by the definition of a correct serialization, $strippedseq$ also represents a correct serialization with respect to $thist$, $InitialDBState$, and st. Now take the longest prefix of $strippedseq$ that does not contain t and let us call it $strippedpref$. If $strippedpref$ is empty, then the definition of a correct serialization and the fact that t is passive imply that $thist[t]$ is atomically correct with respect to $InitialDBState$ (first condition of step 6.4). Otherwise, $strippedpref$ is a prefix of $subgseq$, and Assumption 2.1 (State-deterministic Operations) implies that $thist[t]$ is atomically correct with respect to $stgen(t_w)$, where t_w is the transaction immediately before t in $strippedseq$, which satisfies the second condition of step 6.4.

6.5. Q.E.D.
PROOF: By the Consistency and Nontriviality properties of the termination protocol, $gseq$ contains every transaction t such that t is proposed and $gdeq[t]$ equals $Committed$. We first extend $gseq$ with all other committed transactions. By our mapping of $tdec$, these are the transactions not in proposed but mapped to $Committed$ by $pdec$. However, the $tdec$ Invariant tells us that these transactions are passive and internally committed at their delegate databases. Step 6.4 tells us that they can be inserted at some position of $gseq$ and still generate a correct serialization, by the definition of $CorrectSerialization$. Last, the $dreply$ Invariant and the pre-condition of action $PassiveCommit$ also imply that t is passive and internally committed at $DBof(t)$. Therefore, by step 6.4, t can also be inserted at some posi-

tion of $gseq$ and generate a correct serialization with initial state $InitialDBState$.

7. Changes on $gdec[t]$ due to the termination protocol implement either $DoCommit(t)$ or $DoAbort(t)$
   PROOF SKETCH: Here we assume the termination protocol changes only one entry of $gdec$ at a time. An implementation that does not do that can be easily proved equivalent to this behavior by the creation of "dummy" states that change one entry at a time with the introduction of prophecy variables [AL91]. If $gdec[t]$ is changed to $Aborted$, the Nontriviality and Stability properties automatically imply the pre- and post-conditions of $DoAbort(t)$. Otherwise, we must follow basically the same steps as in step 6. The only two (small) differences are the following:
   - Step 1 should be based on the consistency property guaranteed after $gdec$ is changed, producing a sequence $gseq$ that already contains t.
   - The Q.E.D. step does not have to add t to the built sequence since it is originally in the $gseq$ sequence initially created.

2.4.1.3 Proving the Basic Invariants

In order to prove the basic invariants, we have to define a number of auxiliary invariants. We divided the auxiliary invariants into two types: transaction invariants and database-transaction invariants. The first group refers to invariants that are based on transactions only. The second group refers to invariants that relate transactions and databases.

The only extra notation we introduce in these auxiliary invariants is the definition of an operator $Substr(seq, begin, end)$ for a sequence seq and naturals $begin$ and end, used in invariant DTI5. This operator returns the substring of seq from index $begin$ until index end. If $end < begin$, it is assumed to return an empty sequence.

Transaction Invariants (TI) For every transaction t:

1. $(tdec[t] \notin Decided \wedge q[t] \in Op) \Rightarrow$
   (a) $\forall d \in Database : t \notin CommittedAt(d)$ and
   (b) $t \notin proposed$
2. $(tdec[t] \notin Decided \wedge q[t] = NoReq) \Rightarrow$
   (a) $\forall d \in Database : t \notin CommittedAt(d)$,
   (b) $t \notin proposed$,
   (c) $vers[DBof(t)][t] = 0$,

(d) $thist[t] = DB(DBof(t))!thist[\langle t, 0\rangle]$,
(e) $dcnt[DBof(t)][t] = Len(thist[t])$,
(f) $\forall d \in Database : DB(d)!q[\langle t, vers[d][t]\rangle] = NoReq$, and
(g) $dreq[t] = NoReq$

3. $dcnt[DBof(t)][t] > Len(thist[t]) \Rightarrow$
   (a) $\forall d \in Database : DB(d)!q[\langle t, vers[d][t]\rangle] = NoReq$,
   (b) $vers[DBof(t)][t] = 0$,
   (c) $thist[t] \circ \langle dreq[t], dreply[t]\rangle = DB(DBof(t))!thist[\langle t, 0\rangle]$, and
   (d) $dcnt[DBof(t)][t] = Len(thist[t]) + 1$
   (e) $t \notin proposed$
4. $t \in proposed \Rightarrow$
   (a) $thist[t] = DB(DBof(t))!thist[\langle t, 0\rangle]$,
   (b) $tdec[t] \in Decided \vee q[t] = Commit$,
   (c) $dcnt[DBof(t)][t] = Len(thist[t]) \vee vers[DBof(t)][t] > 0$, and
   (d) $dreq[t] = NoReq$
5. $dreq[t] \in Request \wedge dcnt[d][t] = Len(thist[t]) \Rightarrow$
   (a) $t \notin proposed$,
   (b) $vers[DBof(t)][t] = 0$, and
   (c) $thist[t] = DB(DBof(t))!thist[\langle t, 0\rangle]$
6. $(tdec[t] \notin Decided \wedge t \notin proposed) \Rightarrow dreq[t] = q[t]$
7. $dreq[t] = Commit \Rightarrow$
   (a) $thist[t]$ is passive and
   (b) $t \notin proposed$

Database-Transaction Invariants (DTI) For every database d and transaction t:

1. $DB(d)!q[\langle t, v\rangle] \neq NoReq \Rightarrow v = vers[d][t]$
2. $\forall v \neq vers[d][t] : DB(d)!tdec[\langle t, v\rangle] \neq Committed$
3. If $DB(d)!q[\langle t, vers[d][t]\rangle] = Commit$, then either
   (a) $thist[t] = DB(d)!thist[\langle t, vers[d][t]]$ or

(b) the projection of the operations in $DB(d)!thist[\langle t, vers[d][t]\rangle]$ equals the projection of the operations in $ActHist(t)$.

4. If $DB(d)!tdec[\langle t, vers[d][t]\rangle] = Committed$ and $thist[t]$ is active, then for all database states $st1$, $st2$, and $st3$:
$\wedge CorrectAtomicHist(DB(d)!thist[\langle t, vers[d][t]], st1, st2)$
$\wedge CorrectAtomicHist(thist[t], st1, st3)$
$\Rightarrow st2 = st3$

5. If $\neg(d = DBof(t) \wedge vers[d][t] = 0)$, then the projection of the operations in $DB(d)!thist[\langle t, vers[d][t]\rangle]$ equals the projection of the operations in $Substr(ActHist(t), 1, dcnt[d][t])$.

6. If $DB(d)!tdec[\langle t, vers[d][t]\rangle] = Committed$ and t is proposed, then t appears in $learnedSeq[d]$ and every transaction t' that precedes t in $learnedSeq[d]$ satisfies $dcom[d][t']$.

7. $dcom[d][t] \Rightarrow t \in CommittedAt(d)$

8. If $DB(d)!q[\langle t, vers[d][t]\rangle] = Commit$, then either $tdec[t] \in Decided$ or $q[t] = Commit$.

9. If $DB(d)!q[\langle t, vers[d][t]\rangle] = Commit$ and t is proposed, then t appears in $learnedSeq[d]$ and every transaction t' that precedes t in $learnedSeq[d]$ satisfies $dcom[d][t']$.

10. $DB(d)!q[\langle t, vers[d][t]\rangle] \in Request \wedge t \notin proposed \Rightarrow$

    (a) $d = DBof(t)$,
    (b) $dcnt[d][t] = Len(thist[t])$,
    (c) $DB(d)!q[\langle t, vers[d][t]\rangle] = dreq[t]$,
    (d) $vers[DBof(t)][t] = 0$, and
    (e) $thist[t] = DB(DBof(t))!thist[\langle t, 0\rangle]$

11. $DB(d)!q[\langle t, vers[d][t]\rangle] \in Op \wedge t \in proposed \Rightarrow$

    (a) $d \neq DBof(t) \vee vers[d][t] \neq 0$ and
    (b) $DB(d)!q[\langle t, vers[d][t]\rangle] = ActHist(t)[dcnt[d][t] + 1].op$

12. $\forall v > vers[d][t] : DB(d)!thist[\langle t, v\rangle] = \langle\rangle$

The intuition of the proof is quite simple. It is relatively easy to check the invariants for the initial state of the abstract algorithm. We then assume that they are true and show that they remain true after the execution of each of the algorithm's atomic actions no matter what was the state upon which the action was executed (as long as the invariants were satisfied on it). In the following we analyze action by action and sketch the proof for each of the invariants we have previously defined.

Action $ReceiveReq$

Database Invariant This action does not change any of the variables involved in the Database Invariant.

$tdec$ **Invariant** With respect to the $tdec$ Invariant, this action can only propose a transaction, which does not invalidate the invariant.

$dreply$ **Invariant** As in the previous case, this action can only propose a transaction, which does not invalidate the invariant.

TI1 This action sets $q[t]$ to a request and may add t to $proposed$. Invariant TI2(a,b) and the fact that t is added to $proposed$ only if $q[t]$ is set to $Commit$, which is not in Op, imply that TI1 is preserved.

TI2 The action sets $q[t]$ to a request (different from $NoReq$), which preserves the invariant, since it invalidates the implication condition for transaction t.

TI3 Invariant TI2(e) and the action's pre-condition invalidate the implication condition of this invariant for transaction t.

TI4 If t is proposed, then TI2(d) ensures TI4(a). TI4(b) is ensured because t is proposed only if $q[t]$ is set to $Commit$, TI4(c) is ensured by TI2(e), and TI4(d) is ensured by TI2(g).

TI5 If $dreq[t]$ is set to a value in $Request$, the invariant is ensured by TI2(b-d).

TI6 The action sets $q[t]$ to req. It does nothing else if $tdec \in Decided$, but this condition invalidates the invariant's implication condition. Otherwise, the action either proposes t, which also invalidates the invariant's implication condition, or it makes $dreq[t]$ equal to $q[t]$. In all the cases, the invariant is preserved.

TI7 Condition (a) is easily verified. Condition (b) is ensured in case the action sets $dreq[t]$ to $Commit$ by TI2(b).

DTI1-5,7 Automatically preserved.

DTI6 Transaction t is proposed only if $tdec[t] \notin Decided$ and the action's pre-condition together with invariant TI2(a) implies that t has not been committed at any database under any version, which invalidates this invariant's implication condition.

DTI8 For the sake of contradiction, assume there is a database d such that $DB(d)!q[\langle t, vers[d][t]\rangle] = Commit$, $tdec[t] \notin Decided$, and $q[t]$ is set to $Commit$ by this action. Then, invariant TI2(f) with these assumptions and the action's pre-condition imply that $DB(d)!q[\langle t, vers[d][t]\rangle] = NoReq$, a contradiction with our first assumption.

DTI9 Transaction t is proposed only if $tdec[t] \notin Decided$ and the action's pre-condition together with invariant TI2(f) implies that $DB(d)!q[\langle tvers[d][t]\rangle] = NoReq$, which invalidates this invariant's implication condition.

DTI10 For the sake of contradiction, assume there is a database d such that $DB(d)!q[\langle t, vers[d][t]\rangle] \in Request$, $t \notin proposed$, and $dreq[t]$ is set to a

value different from $DB(d)!q[\langle t, vers[d][t]\rangle]$ (we concentrate on condition (c) since the verification of the other conditions and the implication itself are simple). Then, invariant TI2(f) with these assumptions and the fact that $dreq[t]$ is only changed if $tdec[t] \notin Decided$ imply that $DB(d)!q[\langle t, vers[d][t]\rangle] = NoReq$, a contradiction with our first assumption.

DTI11 If t is proposed by this action, then $tdec[t] \notin Decided$ and invariant TI2(f) invalidates this invariant's implication condition.

DTI12 Automatically preserved.

Action *ReplyReq*

Database Invariant The action changes $thist$, which could affect the Database Invariant. However, TI1(a) implies that t has not been committed at any database, preserving the Database Invariant.

$tdec$ **Invariant** The action only changes $thist[t]$ if $tdec[t] \notin Decided$, which is not true if $t \notin proposed$ and $pdec[t] = Committed$, by the definition of $tdec$. Therefore, the invariant is preserved.

$dreply$ **Invariant** According to the action's pre-condition, $thist[t]$ is only changed if $rep \in Result$ and $rep = dreply[t]$, which implies that $dreply[t] \neq Committed$ and automatically preserves the invariant.

TI1 The action changes $q[t]$ to $NoReq$, which automatically preserves this invariant.

TI2 The action's pre-condition implies that it is executed for a transaction t such that $tdec[t] \notin Decided$ only if $q[t] \in Op$. Invariant $TI1$ implies that no database has committed t in this case and t has not been proposed. Conditions (c-f) are ensured by TI3(a-d) and condition (g) is ensured by the action definition.

TI3 By invariant TI3 and the action's definition, if $thist[t]$ changes, its length will equal $dcnt[DBof(t)][t]$, which just invalidates TI3's implication condition.

TI4 As for conditions (a), (c), and (d), the action only changes $thist[t]$ and $dreq[t]$ if $tdec[t] \notin Decided$ and $q[t] \in Op$. Invariant TI1(b) implies that $t \notin proposed$, which contradicts the invariant's implication condition. As for condition (b), assume $t \in proposed$, $tdec[t] \notin Decided$ and $q[t]$ is changed from $Commit$ to $NoReq$ by this action. Such assumptions conflict with the action definition since $q[t]$ must be in Op for it to be enabled when $tdec[t] \notin Decided$.

TI5 The action only changes $thist[t]$ if it sets $dreq[t]$ to $NoReq$, which automatically preserves the invariant.

TI6 Easily verified.

TI7 If the action changes $thist[t]$, it also sets $dreq[t]$ to $NoReq$, preserving the invariant.

DTI1-2 Automatically preserved.

DTI3 The only variable related to the invariant that is changed by the action is $thist$. However, $thist[t]$ is only changed if $tdec[t] \notin Decided$, $q[t] \in Op$, and $dcnt[DBof(t)][t] > Len(thist[t])$. Invariant TI1(b) validates the implication condition of TI3 for t and TI3(a) automatically invalidates the implication condition of DTI3, preserving the invariant.

DTI4 Again, the only variable of interest is $thist$, and it is changed only if $tdec[t] \notin Decided$ and $q[t] \in Op$. In this case, invariant TI1(a) invalidates the implication condition of DTI4, preserving the invariant.

DTI5 The action may only extend $thist[t]$, which automatically preserves this invariant.

DTI6-7 Automatically preserved.

DTI8 Assume, for the sake of contradiction, that $DB(d)!q[\langle t, vers[d][t]\rangle] = Commit$, $tdec[t] \notin Decided$ and $q[t]$ is changed from $Commit$ to $NoReq$ by this action. However, the action is only enabled when $tdec[t] \notin Decided$ if $q[t] \in Op$, which contradicts the fact that $q[t]$ equals $Commit$ before the action is executed.

DTI9 Automatically preserved.

DTI10 The action only changes $thist[t]$ and $dreq[t]$ if $tdec[t] \notin Decided$ and, in this case, the action's pre-condition implies that $dcnt[d][t] > Len(thist[t])$, which conflicts with the invariant's condition (b) and contradicts its validity before the action execution, unless the implication condition is not satisfied. Since the action does not change the variables involved in the implication condition, the invariant is preserved.

DTI11 This action only changes $thist[t]$ if $tdec[t] \notin Decided$ and invariant TI2(f) invalidates DTI11's implication condition, preserving the invariant.

DTI12 Automatically preserved.

Action $PrematureAbort$

Database Invariant Automatically preserved since this invariant does not involve $pdec$.

$tdec$ **Invariant** The invariant preserved since it involves only transactions t such that $t \notin proposed$ and $pdec[t] = Committed$. $PrematureAbort$ executes for a transaction t only if $t \notin proposed$ and $pdec[t] \notin Decided$ and it changes $pdec[t]$ to $Aborted$, not interfering with the invariant condition.

$dreply$ **Invariant** Automatically preserved.

TI1-2,6 This action can only change $tdec[t]$ from $Unknown$ to $Aborted$, which preserves these invariants since $Aborted \in Decided$.

TI3 Automatically preserved.

TI4 Conditions (a), (c) , and (d) are automatically preserved. As for condition (b), this action can only change $tdec[t]$ to a value in $Decided$ ($Aborted$), preserving the invariant as well.

TI5,7 Automatically preserved

DTI1-7,9-12 Automatically preserved.

DTI8 Easily verified.

Action $PassiveCommit$

Database Invariant Automatically preserved.

$tdec$ **Invariant** If $tdec[t]$ is changed to $Committed$, the action's pre-condition implies that $dreply[t]$ equals $Committed$ and the $dreply$ Invariant ensures that $thist[t]$ is passive and t belongs to $CommittedAt(DBof(t))$.

$dreply$ **Invariant** Automatically preserved.

TI1-2,6 This action can only change $tdec[t]$ from $Unknown$ to $Committed$, which preserves these invariants since $Committed \in Decided$.

TI3 Automatically preserved.

TI4 Condition (a), (c), and (d) are automatically preserved. As for condition (b), this action can only change $tdec[t]$ to a value in $Decided$ ($Committed$), preserving the invariant as well.

TI5,7 Automatically preserved

DTI1-7,9-12 Automatically preserved.

DTI8 Easily verified.

Action $DBReq$ **joint with** $DB(d)!ReceiveReq$

Database Invariant Automatically preserved.

$tdec$ **Invariant** Automatically preserved.

$dreply$ **Invariant** Automatically preserved.

TI1 Automatically preserved.

TI2 This action could break condition (f) of invariant TI2 for some transaction t. If the action is enabled by its first condition, invariants TI5(a) and TI6 imply that $q[t] \in Request$, contradicting TI2's implication condition. If the action is enabled by its second or third condition, then the termination properties ensure that $t \in proposed$ and invariant TI4(b) contradict TI2's implication condition.

TI3 This action sets $DB(d)!q[\langle t, vers[d][t]\rangle]$ to a value different from $NoReq$ and could break TI3(a) for t. However, condition 1 requires that $dcnt[DBof(t)][t] = Len(thist[t])$, contradicting TI3's implication condition. Conditions 2 and 3 (with the Nontriviality property of the termination protocol) imply that t has been proposed, also contradicting TI3's implication condition.

TI4-7 Automatically preserved.

DTI1 Obviously preserved.

DTI2 Automatically preserved.

DTI3 This action can only set $Db(d)!q[\langle t, vers[d][t]\rangle]$ to $Commit$ by conditions 1 or 3. In the first case, the invariant is guaranteed by invariant TI5(b-c). If condition 3 enables this action, there are two cases to consider.

$d = DBof(t) \wedge vers[d][t] = 0$**:** The Nontriviality and Consistency properties of the termination protocol imply that $t \in proposed$ and invariant TI4(a) ensures that DTI3 is preserved.

$dcnt[d][t] = Len(ActHist(t))$**:** In this case, DTI3 is ensured by DTI5.

DTI4-7 Automatically preserved.

DTI8 If the action is triggered by condition 1 and sets $DB(d)!q[\langle t, vers[d][t]\rangle]$ to $Commit$, then $dreq[t] = Commit$. Invariant TI7 implies that $t \notin proposed$ and invariant TI6 ensures DTI8. If the action is triggered by condition 2, it cannot set $DB(d)!q[\langle t, vers[d][t]\rangle]$ to $Commit$. Finally, if the action is triggered by condition 3, then the Consistency and Nontriviality properties of the termination protocol ensure that $tdec[t] = Committed$.

DTI9 If the action is triggered by condition 1, then $t \notin proposed$ by TI7(b). If the action is triggered by condition 3, this condition itself ensures DTI9.

DTI10 The only enabling condition that could interfere with this invariant for this action is condition 1. However, it can be easily verified that it ensures DTI10(c).

DTI11 The only enabling condition that could interfere with this invariant for this action is condition 2. TI4(c) ensures DTI11(a) and DTI11(b) is easily verified.

DTI12 Automatically preserved.

Action $DB(d)!DoAbort$

Database, $tdec$**, and** $dreply$ **Invariants, and TI1-2**
The action can only change $DB(d)!tdec$ by internally aborting a transaction, which does not change $CommittedAt(d)$.

TI3-7 Automatically preserved.

DTI1 Automatically preserved.

DTI2 Easily verified since it changes $DB(d)!tdec[\langle t, v\rangle]$ from $Unknown$ to $Aborted$.

DTI3 Automatically preserved.

DTI4 Easily verified since it changes $DB(d)!tdec[\langle t, v\rangle]$ from $Unknown$ to $Aborted$.

DTI5 Automatically preserved.

DTI6 Easily verified since it changes $DB(d)!tdec[\langle t, v\rangle]$ from $Unknown$ to $Aborted$.

DTI7-12 Automatically preserved.

Action $DB(d)!DoCommit$

Database Invariant There are two cases to consider.

$t \in proposed$ Take the sequence seq of the Database Invariant before the action is executed. Invariants DTI9 and DTI7 imply that all transactions previous to t in $learnedSeq[d]$ are already present in seq. Invariants DTI6 and DTI7 imply that all proposed transactions committed at d appear before t in $learnedSeq[d]$, otherwise t would have already been committed at d and the action would not be enabled. This fact and conditions 2 and 3 of the Database Invariant imply that the sequence of states generated by seq is the same as the one generated by the longest prefix not including t of the sequence defined in the Consistency property for termination. Let st be the last state generated by this sequence. The Consistency property ensures that there is a state $st2$ such that $CorrectAtomicHist(thist[t], st, st2)$ is true. As a result, we can add t to the end of seq and satisfy condition 1 of the Database Invariant after the action is executed. By DTI3, Assumption 2.1, the definition of $ActHist$, if $thist[t]$ is active, so is $DB(d)!thist[\langle t, vers[d][t]\rangle]$ and t should be added to $DB(d)!serialSeq$, satisfying condition 2 of the Database Invariant. Condition 3 is satisfied because, as we pointed out in the very beginning of this case's analysis, a proposed transaction is committed at d iff it appears before t in $learnedSeq[d]$.

$t \notin proposed$ As before, take sequence seq of the Database Invariant before the action is executed. A simple induction on the size of $DB(d)!serialSeq$ and $NVerialSeq(d)$ taking into consideration the Database Invariant as well as DTI4 shows that the sequence of different states generated by these two sequences with respect to $DB(d)!thist$ and $thist$, respectively, is exactly the the same. DTI10(c), TI7 and the definition of action $DoCommit$ imply that t is passive and it can be atomically executed after some of the states mentioned in the previous step. We can place t exactly after that state is generated in seq, satisfying condition 1 of the Database Invariant after the action is executed. Conditions 2 and 3 are automatically satisfied since t is passive.

$tdec$ **Invariant** Easily preserved, since $CommittedAt(d)$ can only be increased.

$dreply$ **Invariant** Easily preserved, since $CommittedAt(d)$ can only be increased.

TI1-2 Easily preserved, given DTI7.

TI3-7 Automatically preserved.

DTI1 Automatically preserved.

DTI2 Easily verified given DTI1.

DTI3,5 Automatically preserved.

DTI4 By DTI3.

DTI6 By DTI9.

DTI7 Obviously preserved, since $CommittedAt(d)$ can only be increased.

DTI8-12 Automatically preserved.

Action $DBRep$ **joint with** $DB(d)!ReplyReq$

Database and $tdec$ **Invariants** Automatically preserved.

$dreply$ **Invariant** $dreply[t]$ is set to $Committed$ only if $DB(d)!q[\langle t, vers[d][t]\rangle]$ equals $Commit$. Invariants DTI10(c) and $TI7$(a) imply that $thist[t]$ is passive, and invariant DTI10(a) with the definition of action $DB(d)!ReplyReq$ ensures that t will belong to $CommittedAt(DBof(t))$.

TI1 Automatically preserved.

TI2 The fact that $t \in proposed$ contradicts TI2(b) and imply that the implication condition of TI2 is false. Therefore, we have to consider only the case in which $t \notin proposed$. In this case, DTI10(c) implies that $dreq[t]$ is different from $NoReq$ and TI6 implies that so is $q[t]$, a contradiction with the implication condition of TI2.

TI3 Easily verified by DTI10 and the action definition.

TI4 DTI11(a) implies that TI4(a) is preserved. TI4(b) is automatically preserved, and TI4(c) is easily verified by TI4(c) itself and the action definition. TI4(d) is also automatically preserved.

TI5 If $t \in proposed$, TI4(d) implies that $dreq[t] = NoReq$, contradicting the implication condition and preserving the invariant. If $t \notin proposed$, DTI10(b) and the action definition imply that $dcnt[d][t]$ is set to $Len(thist[t]) + 1$, also contradicting the implication condition and preserving the invariant.

TI6-7 Automatically preserved.

DTI1 The action sets $DB(d)!q[\langle t, vers[d][t]\rangle]$ to $NoReq$, preserving the invariant.

DTI2 $vers[d][t]$ is increased only if $DB(d)!tdec[\langle t, v\rangle]$ equals $Aborted$.

DTI3 Easily verified by DTI1 and the action definition since $DB(d)!q[\langle t, vers[d][t]\rangle]$ is set to $NoReq$.

DTI4 If $vers[d][t]$ is changed, DTI2 preserves DTI4. Otherwise, if $DB(d)!tdec[\langle t, vers[d][t]\rangle] = Committed$, $DB(d)!thist$ is not changed and the invariant is preserved.

DTI5 If $vers[d][t]$ is changed, then it is increased and $dcnt[d][t]$ is set to 0. In this case, invariant DTI12 preserves DTI5. If $vers[d][t]$ is not changed, there are to cases to analyze. If $t \notin proposed$ the invariant's implication condition is invalidated by DTI10(a,d); If $t \in proposed$, then DTI11(b) and the action definition preserve DTI5.

DTI6 Easily verified by DTI2 in case $vers[d][t]$ changes.

DTI7 Easily verified by the action definition.

DTI8-9 Easily verified in case $vers[d][t]$ changes by DTI1.

DTI10-11 The action sets $DB(d)!q[\langle t, vers[d][t]\rangle]$ to $NoReq$, invalidating these invariants' implication condition.

DTI12 By DTI12 and the fact that $vers[d][t]$ can only be increased.

Termination action changing $gdec[t]$ from $Unknown$ to a value in $Decided$

Database, $tdec$, and $dreply$ Invariants Automatically preserved.

TI1-2 Easily verified since this action can only change $tdec[t]$ to a value in $Decided$, invalidating these invariants' implication condition.

TI3 Automatically preserved.

TI4 Conditions (a) and (c-d) are automatically preserved. Condition (b) is easily verified since this action changes $tdec[t]$ to a value in $Decided$.

TI5,7 Automatically preserved.

TI6 Easily verified since this action can only change $tdec[t]$ to a value in $Decided$, invalidating the invariant's implication condition.

DTI1-7,9-12 Automatically preserved.

DTI8 Easily verified since this action can only change $tdec[t]$ to a value in $Decided$, invalidating the invariant's implication condition.

Termination action changing $learnedSeq[d]$ — Recall that this action can only extend $learnedSeq[d]$ by the Stability property of the termination protocol.

Database Invariant Easily verified since $learnedSeq[d]$ is only extended.

$tdec$ **and** $dreply$ **Invariants** Automatically preserved.

TI1-7 Automatically preserved.

DTI1-5,7-8,10-12 Automatically preserved.

DTI6,9 Easily verified since $learnedSeq[d]$ is only extended.

2.4.2 Proof of Theorem 2.3

Theorem 2.3 *Our abstract deferred update algorithm with the Consistency property for termination changed for the Alternative Consistency property defined above does not implement the specification of a serializable database given in Section 2.1.*

PROOF SKETCH: To understand why, consider the case with two active transactions t_1 and t_2 that write distinct database objects, x and y, respectively, and do not read anything. Transaction t_1 can execute on database d_1 and transaction t_2 can execute on database d_2. Both transactions are free to commit and can be proposed to the termination protocol. Executing either t_1 before t_2 or t_2 before t_1 leads to the same final state and they both can be committed in $gdec$. Assume, then, that database d_1 follows the ordering $\langle t_1, t_2 \rangle$ and executes and commits t_1 first. Database d_2 follows the ordering $\langle t_2, t_1 \rangle$, executing and committing t_2 first. At this point, if a passive transaction reads the whole state of database d_1, it will

see the execution of t_1 but not the execution of t_2, which implies that t_1 must be serialized before t_2. If a passive transaction reads d_2, it will imply that t_2 must be serialized before t_1. Since passive transactions are free to execute completely at the databases responsible for them, all these transactions are free to commit locally and this scenario would break the global serializability.

2.4.3 Proof of Theorem 2.4

Theorem 2.4 *The four properties Nontriviality, Stability, Consistency, and Liveness of our Termination Protocol specification satisfy the Nontriviality, Stability, Consistency, and Liveness properties of Sequence Agreement for transactions that commit where values are transactions and* $learnedSeq$ *implements* $learned$*.*

PROOF SKETCH: For any execution of the Termination Protocol, consider only the set of proposed transactions that eventually commit ($gdec[t]$ is set to $Committed$) as the set of proposed transactions in an execution of Sequence Agreement. We show that all properties of Sequence Agreement are guaranteed in the following:

Nontriviality Guaranteed by Consistency and Nontriviality of Termination.

Stability Trivially guaranteed by Stability of Termination.

Consistency By the Consistency property of Termination, all $learnedSeq$ sequences are prefixes of a common sequence seq of committed (proposed, for Sequence Agreement) transactions, which guarantees that, for every two of them, one is a prefix of the other.

Liveness By the Liveness property of Termination.

2.5 TLA$^+$ Specifications

In this section, we present our specifications more formally using the unambiguous TLA$^+$ specification language [Lam02]. The existing tools for TLA$^+$ allowed us to find and correct many design mistakes and the possibility of model checking our specifications increased the confidence in our results.

2.5.1 Module $DatabaseConstants$

This module contains general database definitions.

MODULE $DatabaseConstants$

EXTENDS $Sequences$, $FiniteSets$, $Naturals$

The specification is based on the following constants:
- Tid: Set of transaction ids, where each id identifies a single transaction.
- Op: Set of possible transaction operations different from $Commit$ or $Abort$.
- $Commit$, $Abort$: Special operations for committing/aborting a transaction.
- $Result$: Set of operation results.
- $Committed$, $Aborted$: Special results returned when a transaction is committed/aborted.
- $CorrectOp(op, res, dbstate, newdbstate)$: Predicate that tells if operation op, when executed upon database state $dbstate$, may give res as a result and generate new database state $newdbstate$.
- $DBState$: Set of database states.
- $InitialDBState$: Initial database state.
- $FSeq$: A substitute for Seq – just a trap for the model checker.
- $Universe$: A set to bound unbounded CHOOSE statements – another trap for the model checker.

CONSTANTS Tid, Op, $Commit$, $Abort$, $Result$, $Committed$, $Aborted$,
$CorrectOp(\_, \_, \_, \_)$, $DBState$, $InitialDBState$, $FSeq(\_)$, $Universe$

We define $Unknown$ as a transaction status in which the transaction has been neither committed nor aborted.

$Decided \triangleq \{Committed, Aborted\}$
$Unknown \triangleq$ CHOOSE $v \in Universe : v \notin Decided$

Request is the set of all possible requests and $NoReq$ is defined to be something that is not a (valid) request.

$Request \triangleq Op \cup \{Commit, Abort\}$
$NoReq \triangleq$ CHOOSE $noreq \in Universe : noreq \notin Request$

Reply is the set of all possible replies for a request.

$Reply \triangleq Result \cup Decided$

Assumptions

The values used as transaction decisions (Committed and $Aborted$) must be different from operation results because we assume the decision is given as the response for operations issued after the transaction has been committed or aborted, so that the client is told that the operation was not performed because the transaction has been decided. If $Committed$ or $Aborted$ corresponds to a correct operation result, the client cannot tell if the operation executed or the transaction terminated.

ASSUME $Committed \notin Result$
ASSUME $Aborted \notin Result \cup \{Committed\}$

We must also assume that $Commit$ and $Abort$ requests are different and not present in Op.

ASSUME $Commit \notin Op$
ASSUME $Abort \notin Op \cup \{Commit\}$

$InitialDBState$ must belong to $DBState$

ASSUME $InitialDBState \in DBState$

$CorrectOp$ must be a correct predicate on op, res, $dbstate$, and $newdbstate$

ASSUME $\forall\, op \in Op, res \in Result,$
$dbstate \in DBState, newdbstate \in DBState :$

$CorrectOp(op,\ res,\ dbstate,\ newdbstate) \in \text{BOOLEAN}$

Auxiliar Expressions

OpRec represents a tuple in $Op \times Result$ as a record with two fields: op and res.

$$OpRec \triangleq [op : Op,\ res : Result]$$

THist is the set of all possible transaction histories.

$$THist \triangleq FSeq(OpRec)$$

THistVector is the set of all possible history vectors.

$$THistVector \triangleq [Tid \to THist]$$

CorrectAtomicHist verifies if the operations in transaction history h, when sequentially applied to the database state $initst$, can provide the same results they provided in h and generate the final database state *finalst*. It is defined as a recursive function that tests operation by operation, in order, with a simple tail recursion. *CorrectAtomicHist* is defined so that even nondeterministic operations are allowed. A single operation can provide nondeterministic results or change the database nondeterministically.

$$\begin{array}{l} CorrectAtomicHist[h \in THist,\ initst \in DBState,\ finalst \in DBState] \triangleq \\ \quad \text{IF } h = \langle\rangle \\ \qquad \text{THEN } initst = finalst \\ \qquad \text{ELSE } \exists\, ist \in DBState : \wedge\ CorrectOp(Head(h).op,\ Head(h).res,\ initst,\ ist) \\ \qquad\qquad\qquad\qquad\qquad \wedge\ CorrectAtomicHist[Tail(h),\ ist,\ finalst] \end{array}$$

$Perm(S)$ represents all sequences containing exactly one copy of each element in set S. It represents all the possible orderings of elements in S. The name $Perm$ comes from permutations although a permutation is a function from S to S, and not a sequence derived from S. For want of a better name, we kept $Perm$.

$$\begin{array}{l} Perm(S) \triangleq \text{LET } N \triangleq Cardinality(S) \\ \qquad\qquad \text{IN} \quad \{s \in [1\,..\,N \to S] : \{s[i] : i \in 1\,..\,N\} = S\} \end{array}$$

CorrectSerialization verifies if sequence seq of transaction ids represents a correct serial execution of its transactions with respect to their histories in history vector $thist$, initial database state $initst$, and final database state *finalst*. It is defined as as a recursive function, like *CorrrectAtomicHist*, that verifies transaction by transaction with a simple tail recursion.

$$\begin{array}{l} CorrectSerialization[seq \in FSeq(Tid),\ thist \in THistVector,\ initst \in DBState, \\ \qquad\qquad finalst \in DBState] \triangleq \\ \quad \text{IF } seq = \langle\rangle \\ \qquad \text{THEN } initst = finalst \\ \qquad \text{ELSE } \exists\, ist \in DBState : \wedge\ CorrectAtomicHist[thist[Head(seq)],\ initst,\ ist] \\ \qquad\qquad\qquad\qquad\qquad \wedge\ CorrectSerialization[Tail(seq),\ thist,\ ist,\ finalst] \end{array}$$

$IsSerializable(S,\ thist,\ initst)$ verifies if set S can have a sequence containing each of its elements exactly once such that its execution is serializable with respect to history vector $thist$ and initial database state $initst$.

$$\begin{array}{l} IsSerializable(S,\ thist,\ initst) \triangleq \\ \quad \exists\, seq \in Perm(S),\ db \in DBState : CorrectSerialization[seq,\ thist,\ initst,\ db] \end{array}$$

$PassiveOp(op)$ is satisfied iff operation op is passive.

$PassiveOp(op) \triangleq$
  $\forall\, st1,\, st2 \in DBState,\, res \in Result :$
    $CorrectOp(op,\, res,\, st1,\, st2) \Rightarrow st1 = st2$

$PassiveHist(h)$ is satisfied iff history h is passive.

$PassiveHist(h) \triangleq$
  $\forall\, st1,\, st2 \in DBState :$
    $CorrectAtomicHist[h,\, st1,\, st2] \Rightarrow st1 = st2$

2.5.2 Module *SerializableDB*

This module presents a TLA$^+$ version of our serializable database specification. It extends module *DBInterface* that defines interface operators *DBRequest* and *DBResponse* in terms of an interface variable *DBinter*. Our specifications are practically oblivious to how these operators are defined as long as their definitions are disjoint. For the sake of simplicity, we do not present our specification of module *DBInterface*.

MODULE *SerializableDB*

EXTENDS *DatabaseConstants*, *DBInterface*

VARIABLES *thist*, *tdec*, *q* and *DBinter* from *DBInterface*

Types and Auxiliar Expressions

Type Invariants guaranteed by the specification

$thistType \triangleq THistVector$
$tdecType \triangleq [Tid \to Decided \cup \{Unknown\}]$
$qType \triangleq [Tid \to Request \cup \{NoReq\}]$

$TypeInv \triangleq \land thist \in thistType$
$\land tdec \in tdecType$
$\land q \in qType$

Set of all committed transactions

$committedSet \triangleq \{t \in Tid : tdec[t] = Committed\}$

Actions

$ReceiveReq(t,\, req)$ deals with the receipt of a transaction request. It stores the received request in $q[t]$ for it to be processed later.

$ReceiveReq(t,\, req) \triangleq \land DBRequest(t,\, req)$
$\land q[t] = NoReq$
$\land q' = [q \text{ EXCEPT } ![t] = req]$

$\land$ UNCHANGED $\langle thist,\ tdec \rangle$

ReplyReq$(t,\ rep)$ deals with the response of an executing request. It checks whether transaction t has already been decided; if so, the response to t's executing request is its final decision. If t has not been decided yet, then the action is enabled only if op is in Op and rep is in $Result$. In such a case, the operation and its result are enqueued in t's history.

$$
\begin{array}{ll}
ReplyReq(t,\ rep) \triangleq & \land\ q[t] \in Request \\
& \land\ DBResponse(t,\ rep) \\
& \land\ q' = [q \text{ EXCEPT } ![t] = NoReq] \\
& \land \text{ IF } tdec[t] \in Decided \\
& \quad \text{THEN} \quad \land\ rep = tdec[t] \\
& \qquad\qquad \land \text{ UNCHANGED } \langle thist,\ tdec \rangle \\
& \quad \text{ELSE} \quad \land\ q[t] \in Op \\
& \qquad\qquad \land\ rep \in Result \\
& \qquad\qquad \land\ thist' = [thist \text{ EXCEPT } ![t] = \\
& \qquad\qquad\qquad\qquad Append(@,\ [op \mapsto q[t], \\
& \qquad\qquad\qquad\qquad\qquad res \mapsto rep])] \\
& \qquad\qquad \land \text{ UNCHANGED } \langle tdec \rangle
\end{array}
$$

Action *DoAbort*(t) aborts t by setting $tdec[t]$ to $Aborted$. This can be done at any time as long as t has not been decided yet.

$$
\begin{array}{ll}
DoAbort(t) \triangleq & \land\ tdec[t] \notin Decided \\
& \land\ tdec' = [tdec \text{ EXCEPT } ![t] = Aborted] \\
& \land \text{ UNCHANGED } \langle thist,\ q,\ DBinter \rangle
\end{array}
$$

Action *DoCommit*(t) commits t by setting $tdec[t]$ to $Committed$, which is done only if t has not been decided and t's commit request has been issued.

$$
\begin{array}{ll}
DoCommit(t) \triangleq & \land\ tdec[t] \notin Decided \\
& \land\ q[t] = Commit \\
& \land\ tdec' = [tdec \text{ EXCEPT } ![t] = Committed] \\
& \land\ IsSerializable(committedSet',\ thist,\ InitialDBState) \\
& \land \text{ UNCHANGED } \langle thist,\ q,\ DBinter \rangle
\end{array}
$$

Specification

Initialization.

$$
\begin{array}{ll}
Init \triangleq & \land\ InitInterface \\
& \land\ thist = [i \in Tid \mapsto \langle\rangle] \\
& \land\ tdec\ = [i \in Tid \mapsto Unknown] \\
& \land\ q = [t \in Tid \mapsto NoReq]
\end{array}
$$

Next defines the possible "next" steps in a correct execution.

$$
\begin{array}{ll}
Next \triangleq & \exists\, t \in Tid : \\
& \quad \lor\ \lor \exists\, req \in Request : ReceiveReq(t,\ req) \\
& \qquad \lor \exists\, rep \in Reply\ \ : ReplyReq(t,\ rep)
\end{array}
$$

$\lor\ DoCommit(t)$
$\lor\ DoAbort(t)$

Final specification.

$Spec \triangleq Init \land \Box[Next]_{\langle thist,\ tdec,\ q,\ DBinter\rangle}$

Type Invariant Theorem

THEOREM $Spec \Rightarrow \Box\, TypeInv$

2.5.3 Module $OPSerializableDB$

This module presents our specification of an order-preserving serializable database.

MODULE $OPSerializableDB$

EXTENDS $SerializableDB$ Simply extends a serializable database

serialSeq keeps the commit order

VARIABLES $serialSeq$

$serialSeqType \triangleq \{s \in FSeq(Tid) : \forall\, i,\ j \in \text{DOMAIN } s : i \neq j \Rightarrow s[i] \neq s[j]\}$

Actions

The only action we change is $DoCommit$, which is replaced by the action defined below

$OPDoCommit(t) \triangleq$
$\quad \land\ tdec[t] = Unknown$
$\quad \land\ q[t] = Commit$
$\quad \land\ tdec' = [tdec \text{ EXCEPT } ![t] = Committed]$
$\quad \land\ serialSeq' = Append(serialSeq,\ t)$
$\quad \land \exists\, st \in DBState : CorrectSerialization[serialSeq',\ thist,$
$\qquad\qquad\qquad\qquad\qquad\qquad InitialDBState,\ st]$
$\quad \land$ UNCHANGED $\langle thist,\ q,\ DBinter\rangle$

Specification

Initialization.

$OPInit \triangleq \land\ Init$
$\qquad\qquad \land\ serialSeq = \langle\rangle$

Next defines the possible "next" steps in a correct execution.

$OPNext \triangleq \exists\, t \in Tid :$

$$
\begin{array}{l}
\lor\ \land\ \lor \exists\, req \in Request : ReceiveReq(t,\, req) \\
\qquad\ \lor \exists\, rep \in Reply\ \ \ : ReplyReq(t,\, rep) \\
\qquad\ \lor DoAbort(t) \\
\quad \land \text{UNCHANGED } serialSeq \\
\lor OPDoCommit(t)
\end{array}
$$

Final specification.

$$OPSpec \triangleq OPInit \land \Box[OPNext]_{\langle thist,\, tdec,\, q,\, DBinter,\, serialSeq\rangle}$$

Type Invariant Theorem

$$\text{THEOREM } OPSpec \Rightarrow \Box(TypeInv \land\ serialSeq \in serialSeqType)$$

2.5.4 Module $AOPSerializableDB$

This module presents our specification of an active order-preserving serializable database.

MODULE $AOPSerializableDB$

EXTENDS $SerializableDB$

serialSeq keeps the commit order

VARIABLES $serialSeq$

$$serialSeqType \triangleq \{s \in FSeq(Tid) : \forall\, i,\, j \in \text{DOMAIN } s : i \neq j \Rightarrow s[i] \neq s[j]\}$$

Actions

Function *IsSubSeq* below verifies if *smallseq* is a subsequence of *bigseq*.

$$
\begin{array}{l}
IsSubSeq[smallseq \in FSeq(Tid),\, bigseq \in FSeq(Tid)] \triangleq \\
\quad (smallseq \neq \langle\rangle) \Rightarrow \\
\qquad \exists\, i \in 1\,..\,Len(bigseq) : \\
\qquad\quad \land\ bigseq[i] = Head(smallseq) \\
\qquad\quad \land \forall\, j \in 1\,..\,Len(bigseq) : \\
\qquad\qquad bigseq[j] = Head(smallseq) \Rightarrow j \geq i \\
\qquad\quad \land\ IsSubSeq[Tail(smallseq),\, SubSeq(bigseq,\, i+1,\, Len(bigseq))]
\end{array}
$$

The only action we change is $DoCommit$, which is replaced by the action defined below

$$
\begin{array}{ll}
AOPDoCommit(t) \triangleq & \land\ tdec[t] = Unknown \\
& \land\ q[t] = Commit \\
& \land\ tdec' = [tdec \text{ EXCEPT } ![t] = Committed] \\
& \land \text{ IF } PassiveHist(thist[t]) \\
& \qquad \text{THEN UNCHANGED } serialSeq
\end{array}
$$

$$
\begin{array}{l}
\quad \text{ELSE } serialSeq' = Append(serialSeq,\ t) \\
\land\ \exists\, seq \in Perm(committedSet'),\ st \in DBState : \\
\quad \land\ CorrectSerialization[seq,\ thist,\ InitialDBState,\ st] \\
\quad \land\ IsSubSeq[serialSeq',\ seq] \\
\land\ \text{UNCHANGED } \langle thist,\ q,\ DBinter \rangle
\end{array}
$$

Specification

Initialization.

$$
\begin{array}{l}
AOPInit \triangleq\ \land\ Init \\
\qquad\qquad\ \land\ serialSeq = \langle\rangle
\end{array}
$$

Next defines the possible "next" steps in a correct execution.

$$
\begin{array}{l}
AOPNext \triangleq\ \exists\, t \in Tid : \\
\qquad \lor\ \land\ \lor \exists\, req \in Request : ReceiveReq(t,\ req) \\
\qquad\qquad\ \lor \exists\, rep \in Reply\ \ \ : ReplyReq(t,\ rep) \\
\qquad\qquad\ \lor\ DoAbort(t) \\
\qquad\quad\ \land\ \text{UNCHANGED } serialSeq \\
\qquad \lor\ AOPDoCommit(t)
\end{array}
$$

Final specification.

$$AOPSpec \triangleq\ AOPInit \land \Box[AOPNext]_{\langle thist,\ tdec,\ q,\ DBinter,\ serialSeq \rangle}$$

Type Invariant Theorem

$$\text{THEOREM } AOPSpec \Rightarrow \Box(TypeInv \land\ \ serialSeq \in serialSeqType)$$

2.5.5 Module *GeneralDeferredUpdate*

This is the TLA$^+$ specification of our abstract deferred-update algorithm.

MODULE *GeneralDeferredUpdate*

EXTENDS *DatabaseConstants*, *DBInterface*

CONSTANTS *Database*, *DBof*(_), *StripPassive*(_)

VARIABLES *thist*, *q*, *dreq*, *pdec*, — Client variables
dreply, *dcnt*, *vers*, *dcom*, — Database variables
ldinter, *dthist*, *dtdec*, *dq*, *dserialSeq*, — Internal database variables
proposed, *learnedSeq*, *gdec* — Termination variables

Extra Assumptions

ASSUME $\forall\, t \in Tid : DBof(t) \in Database$

ASSUME $\forall\, hist \in THist,\ st1,\ st2 \in DBState :$
$\quad \wedge\ StripPassive(hist) \in THist$
$\quad \wedge\ CorrectAtomicHist[hist,\ st1,\ st2] \Rightarrow$
$\qquad CorrectAtomicHist[StripPassive(hist),\ st1,\ st2]$

We assume state-deterministic operations

ASSUME $\forall\, op \in Op,\ res1,\ res2 \in Result,\ st,\ st1,\ st2 \in DBState :$
$\quad \wedge\ CorrectOp(op,\ res1,\ st,\ st1)$
$\quad \wedge\ CorrectOp(op,\ res2,\ st,\ st2)$
$\quad \Rightarrow st1 = st2$

Auxiliar Expressions and Replicas' Instantiation

Definition of *tdec* based on *pdec* e *gdec*

$tdec \triangleq [t \in Tid \mapsto$ IF $t \in proposed$
$\qquad$ THEN $gdec[t]$
$\qquad$ ELSE $pdec[t]]$

Each database accepts transactions with ids in the form $\langle tid,\ version\rangle$ where tid is an element of Tid and version is a Natural. This allows "a single" transaction to be submitted to a database multiple times.

$LocalTid \triangleq Tid \times Nat$

The definition below instantiates each local database used by the general algorithm.

$DBS(d) \triangleq$ INSTANCE $AOPSerializableDB$ WITH $Tid \leftarrow LocalTid,$
$DBinter \leftarrow ldinter[d],$
$thist \leftarrow dthist[d],$
$tdec \leftarrow dtdec[d],$
$q \leftarrow dq[d],$
$serialSeq \leftarrow dserialSeq[d]$

NoRep is defined to be some value that is not a valid reply.

$NoRep \triangleq$ CHOOSE $v : v \notin Reply$

The definition below creates an instance of the termination protocol specification.

$GT \triangleq$ INSTANCE $GeneralTermination$

Auxiliary definitions to help dealing with the declared variables.

$cvars \triangleq \langle thist,\ q,\ dreq,\ pdec\rangle$
$ldvars \triangleq \langle ldinter,\ dthist,\ dtdec,\ dq,\ dserialSeq\rangle$
$gdvars \triangleq \langle dreply,\ dcnt,\ vers,\ dcom\rangle$
$dvars \triangleq \langle gdvars,\ ldvars\rangle$
$tvars \triangleq \langle proposed,\ learnedSeq,\ gdec\rangle$

$ActHist(t)$ returns the current history of transaction t with some of its passive operations taken out of the sequence (according to operator $StripPassive$).

$$ActHist(t) \triangleq StripPassive(thist[t])$$

$DBvars(d)$ returns the internal variables of database d. It is used by the auxiliar action defined below.

$$DBvars(d) \triangleq \langle ldinter[d],\ dthist[d],\ dtdec[d],\ dq[d],\ dserialSeq[d]\rangle$$

$OtherDBsStutter(d)$ is an auxiliar action that forces all databases but d to execute a stuttering step, that is, a step in which their internal variables do not change values. For simplicity, our specification does not allow interleaving of database actions. In fact, as we explain in the following, it does not allow interleaving at all.

$$\begin{array}{l} OtherDBsStutter(d) \triangleq \\ \quad \text{LET } dbfn \triangleq [nd \in (Database \setminus \{d\}) \mapsto DBvars(nd)] \\ \quad \text{IN } \ dbfn' = dbfn \end{array}$$

Actions

These are the atomic actions of the general deferred update technique, not including the internal database actions and the internal actions of the termination protocol. In order to model check this specification, we had to make it noninterleaving, that is, we had to specify it in terms of actions that cannot occur concurrently (even considering that they are executed by different specification components). This prevented us from using the $DBRequest$ and $DBResponse$ primitives to interact with the internal databases. Instead, we used the $ReceiveReq$ and $ReplyReq$ actions directly to submit an operation and get a response from a database.

The $ReceiveReq$ action.

$$\begin{array}{l} ReceiveReq(t,\ req) \triangleq \\ \quad \land\ DBRequest(t,\ req) \\ \quad \land\ q[t] \notin Request \\ \quad \land\ q' = [q \text{ EXCEPT } ![t] = req] \\ \quad \land \text{ IF } tdec[t] \notin Decided \\ \qquad \text{THEN } \lor\ \land\ req = Commit \\ \qquad\qquad\qquad \land\ GT!Propose(t) \\ \qquad\qquad\qquad \land \text{ UNCHANGED } \langle thist,\ dreq,\ pdec,\ dvars\rangle \\ \qquad\qquad \lor\ \land\ req = Commit \Rightarrow PassiveHist(thist[t]) \\ \qquad\qquad\qquad \land\ dreq' = [dreq \text{ EXCEPT } ![t] = req] \\ \qquad\qquad\qquad \land \text{ UNCHANGED } \langle thist,\ pdec,\ dvars,\ tvars\rangle \\ \qquad \text{ELSE } \text{UNCHANGED } \langle thist,\ dreq,\ pdec,\ dvars,\ tvars\rangle \end{array}$$

The $ReplyReq$ action.

$$\begin{array}{l} ReplyReq(t,\ rep) \triangleq \\ \quad \land\ q[t] \in Request \\ \quad \land\ DBResponse(t,\ rep) \\ \quad \land\ q' = [q \text{ EXCEPT } ![t] = NoReq] \\ \quad \land \text{ IF } tdec[t] \in Decided \\ \qquad \text{THEN } \land\ rep = tdec[t] \\ \qquad\qquad\quad \land \text{ UNCHANGED } \langle thist,\ dreq,\ pdec,\ dvars,\ tvars\rangle \\ \qquad \text{ELSE } \ \land\ q[t] \in Op \end{array}$$

$$
\begin{array}{l}
\land\ rep \in Result \\
\land\ dcnt[DBof(t)][t] > Len(thist[t]) \\
\land\ rep = dreply[t] \\
\land\ thist' = [thist \text{ EXCEPT } ![t] = Append(@, [op \mapsto q[t], \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad res \mapsto rep])] \\
\land\ dreq' = [dreq \text{ EXCEPT } ![t] = NoReq] \\
\land\ \text{UNCHANGED } \langle pdec,\ dvars,\ tvars \rangle
\end{array}
$$

The *PrematureAbort* action.

$$
\begin{array}{l}
PrematureAbort(t) \triangleq \land\ t \notin proposed \\
\qquad\qquad\qquad\qquad\ \land\ pdec[t] \notin Decided \\
\qquad\qquad\qquad\qquad\ \land\ pdec' = [pdec \text{ EXCEPT } ![t] = Aborted] \\
\qquad\qquad\qquad\qquad\ \land\ \text{UNCHANGED } \langle thist,\ q,\ dreq,\ dvars,\ tvars,\ DBinter \rangle
\end{array}
$$

The *PassiveCommit* action.

$$
\begin{array}{l}
PassiveCommit(t) \triangleq \land\ t \notin proposed \\
\qquad\qquad\qquad\qquad\ \land\ pdec[t] \notin Decided \\
\qquad\qquad\qquad\qquad\ \land\ dreply[t] = Committed \\
\qquad\qquad\qquad\qquad\ \land\ pdec' = [pdec \text{ EXCEPT } ![t] = Committed] \\
\qquad\qquad\qquad\qquad\ \land\ \text{UNCHANGED } \langle thist,\ q,\ dreq,\ dvars,\ tvars,\ DBinter \rangle
\end{array}
$$

The *DBReq* action with its three enabling conditions.

$$
\begin{array}{l}
DBReq(d,\ t,\ req) \triangleq \\
\quad \land \quad \lor \land\ d = DBof(t) \qquad \text{Condition 1} \\
\qquad\quad\ \ \land\ dreq[t] = req \\
\qquad\quad\ \ \land\ dcnt[d][t] = Len(thist[t]) \\
\qquad\quad \lor \land\ t \in proposed \qquad \text{Condition 2} \\
\qquad\quad\ \ \land\ dcnt[d][t] < Len(ActHist(t)) \\
\qquad\quad\ \ \land\ req = ActHist(t)[dcnt[d][t] + 1].op \\
\qquad\quad \lor \land\ req = Commit \qquad \text{Condition 3} \\
\qquad\quad\ \ \land\ \exists\, i \in 1\,..\,Len(learnedSeq[d]) : \\
\qquad\qquad\qquad \land\ learnedSeq[d][i] = t \\
\qquad\qquad\qquad \land\ \forall\, j \in 1\,..\,i-1 : dcom[d][learnedSeq[d][j]] \\
\qquad\quad\ \ \land\ \lor\ d = DBof(t) \land vers[d][t] = 0 \\
\qquad\qquad\quad\ \lor\ dcnt[d][t] = Len(ActHist(t)) \\
\quad \land \quad DBS(d)!ReceiveReq(\langle t,\ vers[d][t] \rangle,\ req) \\
\quad \land \quad OtherDBsStutter(d) \\
\quad \land \quad \text{UNCHANGED } \langle cvars,\ gdvars,\ tvars,\ DBinter \rangle
\end{array}
$$

The *DBRep* action.

$$
\begin{array}{l}
DBRep(d,\ t,\ rep) \triangleq \\
\quad \land \quad DBS(d)!ReplyReq(\langle t,\ vers[d][t] \rangle,\ rep) \\
\quad \land \quad OtherDBsStutter(d) \\
\quad \land \quad \text{IF } d = DBof(t) \text{ THEN } dreply' = [dreply \text{ EXCEPT } ![t] = rep]
\end{array}
$$

```
                                ELSE UNCHANGED dreply
 ∧    IF rep = Aborted ∧ t ∈ proposed
        THEN ∧ vers' = [vers EXCEPT ![d][t] = @ + 1]
             ∧ dcnt' = [dcnt EXCEPT ![d][t] = 0]
             ∧ UNCHANGED dcom
        ELSE ∧ dcnt' = [dcnt EXCEPT ![d][t] = @ + 1]
             ∧ dcom' = [dcom EXCEPT ![d][t] = (rep = Committed)]
             ∧ UNCHANGED vers
 ∧    UNCHANGED ⟨cvars, tvars, DBinter⟩
```

Specification

Initialization.

$$
\begin{aligned}
Init \triangleq\ & \land InitInterface \\
& \land q = [t \in Tid \mapsto NoReq] \\
& \land dreq = [t \in Tid \mapsto NoReq] \\
& \land dreply = [t \in Tid \mapsto NoRep] \\
& \land pdec = [t \in Tid \mapsto Unknown] \\
& \land vers = [d \in Database \mapsto [t \in Tid \mapsto 0]] \\
& \land dcom = [d \in Database \mapsto [t \in Tid \mapsto \text{FALSE}]] \\
& \land dcnt = [d \in Database \mapsto [t \in Tid \mapsto 0]] \\
& \land \forall\, d \in Database : DBS(d)!AOPInit \\
& \land GT!Init \quad \text{includes } thist
\end{aligned}
$$

The next-state action in terms of noninterleaving actions.

$$
\begin{aligned}
Next \triangleq\ & \lor \exists\, t \in Tid : \lor \exists\, req \in Request : ReceiveReq(t, req) \\
& \qquad\qquad\ \lor \exists\, rep \in Reply\ \ : ReplyReq(t, rep) \\
& \qquad\qquad\ \lor PrematureAbort(t) \\
& \qquad\qquad\ \lor PassiveCommit(t) \\
& \lor \exists\, d \in Database : \lor \exists\, t \in Tid : \lor \exists\, req \in Request : DBReq(d, t, req) \\
& \qquad\qquad\qquad\qquad\qquad\qquad\ \lor \exists\, rep \in Reply : \ \ DBRep(d, t, rep) \\
& \qquad\qquad\qquad\qquad\ \lor \land \text{UNCHANGED } ldinter[d] \land DBS(d)!AOPNext \\
& \qquad\qquad\qquad\qquad\quad \land OtherDBsStutter(d) \\
& \qquad\qquad\qquad\qquad\quad \land \text{UNCHANGED } \langle cvars, gdvars, tvars, DBinter\rangle \\
& \lor \land \text{UNCHANGED } \langle cvars, dvars, DBinter\rangle \\
& \quad \land GT!TNext
\end{aligned}
$$

The final safety specification

$$Safety \triangleq Init \land \Box[Next]_{\langle cvars,\, dvars,\, tvars,\, DBinter\rangle}$$

The final specification, including the liveness condition of the termination protocol.

$$Spec \triangleq Safety \land GT!Liveness$$

2.5.6 Module *GeneralTermination*

This module presents our specification of the Termination Protocol.

MODULE *GeneralTermination*

EXTENDS *DatabaseConstants*

CONSTANTS *Database*

VARIABLES *proposed*, *learnedSeq*, *gdec*, *thist*

Auxiliar Expressions

$vars \triangleq \langle proposed,\ learnedSeq,\ gdec \rangle$

$committedSet \triangleq \{t \in Tid : gdec[t] = Committed\}$

IsPrefix(*smallseq*, *bigseq*) verifies if *smallseq* is a prefix of *bigseq*.

$IsPrefix(smallseq,\ bigseq) \triangleq \exists\, n \in 0\,..\,Len(bigseq) : smallseq = SubSeq(bigseq,\ 1,\ n)$

The consistency property in TLA+ The other properties are automatically guaranteed by the atomic actions below.

$$
\begin{array}{l}
Consistency \triangleq \\
\quad \exists\, seq \in Perm(committedSet),\ st \in DBState : \\
\qquad \land\ CorrectSerialization[seq,\ thist,\ InitialDBState,\ st] \\
\qquad \land\ \forall\, d \in Database : IsPrefix(learnedSeq[d],\ seq)
\end{array}
$$

Actions

Propose(t) proposes a transaction t for termination.

$$
\begin{array}{l}
Propose(t) \triangleq \\
\quad \land\ t \notin proposed \\
\quad \land\ proposed' = proposed \cup \{t\} \\
\quad \land\ \text{UNCHANGED}\ \langle learnedSeq,\ gdec \rangle
\end{array}
$$

Decide(t) makes a final decision (Committed or *Aborted*) about proposed transaction t.

$$
\begin{array}{l}
Decide(t) \triangleq \\
\quad \land\ t \in proposed \\
\quad \land\ gdec[t] = Unknown \\
\quad \land\ \exists\, v \in Decided : gdec' = [gdec\ \text{EXCEPT}\ ![t] = v] \\
\quad \land\ \text{UNCHANGED}\ \langle proposed,\ learnedSeq \rangle \\
\quad \land\ Consistency'
\end{array}
$$

Learn(d, seq) extends *learnedSeq*[d], but only if the new value ensures consistency.

$$
\begin{array}{l}
Learn(d,\ seq) \triangleq \\
\quad \land\ \ IsPrefix(learnedSeq[d],\ seq) \land Len(seq) > Len(learnedSeq[d])
\end{array}
$$

$\land \quad learnedSeq' = [learnedSeq \text{ EXCEPT } ![d] = seq]$
$\land \quad \text{UNCHANGED } \langle proposed,\ gdec \rangle$
$\land \quad Consistency'$

Specification

The following two definitions have to do with our weak liveness requirement for termination.

$$LivenessDatabase(t,\ d) \triangleq$$
$$\quad gdec[t] = Aborted \ \Rightarrow$$
$$\qquad \Diamond(t \in \{learnedSeq[d][i] : i \in \text{DOMAIN } learnedSeq[d]\})$$

$$Liveness \triangleq$$
$$\quad \Box(\forall\, t \in Tid,\ d \in Database : LivenessDatabase(t,\ d))$$

The following action simply helps model checking. It changes the transactions' history vector.

$$ChangeHist(t) \triangleq$$
$$\quad \land\ t \notin proposed$$
$$\quad \land \exists\, o \in Op,\ r \in Result :$$
$$\qquad thist' = [thist \text{ EXCEPT } ![t] = Append(@,\ [op \mapsto o,$$
$$\qquad\qquad\qquad\qquad\qquad\qquad res \mapsto r])]$$
$$\quad \land \text{UNCHANGED } vars$$

TNext allows any action but $Propose(v)$. It is used in the specification of our general deferred update protocol.

$$TNext \triangleq$$
$$\quad \lor \exists\, t \in Tid : Decide(t)$$
$$\quad \lor \exists\, d \in Database,\ seq \in FSeq(Tid) : Learn(d,\ seq)$$

Next allows all the actions and is used by to model check termination in an isolated way.

$$Next \triangleq$$
$$\quad \lor\ TNext \land \text{UNCHANGED } thist$$
$$\quad \lor \exists\, t \in Tid : \lor Propose(t) \land \text{UNCHANGED } thist$$
$$\qquad\qquad\qquad\quad \lor ChangeHist(t)$$

Initialization.

$$Init \triangleq$$
$$\quad \land\ proposed \quad = \{\}$$
$$\quad \land\ learnedSeq = [d \in Database \mapsto \langle\rangle]$$
$$\quad \land\ gdec \qquad = [t \in Tid \mapsto Unknown]$$
$$\quad \land\ thist \qquad = [t \in Tid \mapsto \langle\rangle]$$

Final specification.

$$Spec \triangleq Init \land \Box[Next]_{\langle vars,\ thist \rangle} \land Liveness$$

2.6 Related Work and Final Remarks

Database replication techniques can be classified in eager or lazy, according to the way updates are propagated [GHOS96]. In eager replication, updates are applied to database replicas as part of the original transaction (i.e., if the transaction commits, it is guaranteed that correct replicas will eventually apply them locally). In lazy replication, a transaction first commits at one replica, and then its updates are propagated to the others. Clearly, lazy replication may not preserve a serializable interface, even in the absence of failures.

Eager replication admits two variations [BHG87]. Immediate-update replication propagates updates to replicas during the transaction execution, whereas deferred-update replication propagates updates to replicas after the transaction has had its operations executed and a commit request has been issued. Deferred-update algorithms reduce the amount of synchronization between replicas and ensure a serializable interface to the database users, which makes them of practical interest [WPS+00a, WPS+00b]. Nevertheless, we are not aware of any work that explored more deeply the theoretical aspects and limitations of deferred-update database replication.

Consistency criteria for databases systems have been defined in many different ways. It is known that definitions based on anomalies allowed and disallowed by the system lead to some inconsistencies [BBG+95]. A more correct way to specify a consistent database is done in terms of the equivalence between the system's actual execution and a serial execution of the submitted transactions [BHG87]. Traditional definitions of equivalence between two executions of transactions referred to the internal scheduling performed by the algorithms and their ordering of conflicting operations. This approach has led to different notions of equivalence and, therefore, different subclasses of Serializability [Pap79].

Schedule-comparison specifications are very good for the specific algorithms people had in mind when designing them [BHG87]. However, they are hard to adapt to more complicated algorithms [LMWF94]. A more general approach to specifying database consistency criteria consists of specifying a state machine that corresponds to the behavior an idealized correct database system should have [LMWF94]. Proving algorithms correct using this approach boils down to finding a refinement mapping between the algorithm and the correct specification [AL91]. A special issue of *Distributed Computing* devoted to the formal specification of concurrent systems had different specifications of a Serializable database, based on different specification techniques [Bro92, KS92, LS92]. Our model-based specification of Serializability resembles the specifications in [LMWF94] and [LS92].

Our methodology for proving our abstraction correct resembles the work done in [LLOR99] to prove that a lazy-caching protocol satisfies sequential consistency. Nevertheless, their work referred to proving a specific protocol correct, which leads to no general results concerning a class of algorithms. In our case, since we are dealing with an abstraction, results generalize to all possible algorithmic implementations.

In this chapter, we have formalized the deferred update technique for database replication and stated some intrinsic characteristics and limitations of it. Previous works have only considered new algorithms, with independent specifications, analysis, and correctness proofs. To the best of our knowledge, our work is first effort to formally characterize this family of algorithms and establish its requirements. Our general abstraction can be used to derive other general limitation results as well as to create new algorithms and prove existing ones correct. Some algorithms can be easily proved correct by a refinement mapping to ours. Others may require an additional effort due to the extra assumptions they make, but the task seems still easier than with previous formalisms. Chapter 4 shows how our abstraction can be used to obtain interesting protocols and correctness proofs.

Chapter 3

Collision-fast Sequence Agreement and Paxos

A distributed system is one in which the failure of a computer you didn't even know existed can render your computer unusable.
Leslie Lamport

As shown in the previous chapter, the Sequence Agreement problem is highly related to the termination of transactions in deferred update database replication protocols since any practical termination protocol must implement sequence agreement for the transactions it commits. Besides that, as presented in Section 2.3, sequence agreement itself can be used to implement the termination protocol in a very elegant way. Therefore, understanding the limitations of this problem and being able to efficiently solve it is paramount to designing and implementing deferred update replication.

In this chapter we explore this direction and present a very efficient fault-tolerant solution to sequence agreement. Our latency-optimal algorithm, derived from the Paxos consensus protocol [Lam98], is very dynamic and can quickly reconfigure and adapt to failures, which distinguishes it from previous approaches achieving similar bounds. Our algorithm is based on a variant of the consensus problem we call M-Consensus. M-Consensus is more general than the original consensus problem, being much more suitable as a building block for efficient sequence agreement implementations. Last, but not least, we have extensively proved safety and liveness of our solutions to both M-Consensus and sequence agreement and we believe that some of the techniques we used are general enough to be applied to other agreement problems.

3.1 Sequence Agreement and Consensus

As mentioned in Chapter 2, sequence agreement is a sequence-based specification of the celebrated atomic broadcast problem [HT93], often solved using the consensus problem as a building block. In fact, the two problems are equivalent with respect to solvability but consensus has a simpler definition since learners must eventually learn only a single value out of the set of proposed ones. The equivalence between consensus and sequence agreement, though, brings out some interesting results. First, it extends to sequence agreement the famous FLP impossibility result stating that consensus is not deterministically solvable in asynchronous systems subject to failures [FLP85]. Moreover, since the reduction from consensus to sequence agreement is direct (learners learn only the first element of the agreed sequence), any lower bounds for consensus also apply to sequence agreement.

Generally speaking, one can solve sequence agreement by means of a totally ordered succession of consensus instances. A proposer that wants to propose a value proposes it in the first instance for which it has not proposed or learned anything yet. Consensus ensures that the decision reliably reaches all nonfaulty learners, and the sequence order is given by the ordering of the instances, that is, the i^{th} instance's decision gives the i^{th} element in the learned sequence. Proposers must be also consensus learners so that they can check if their proposal in some instance was decided or not and repropose it in a different instance in case it was not the consensus decision. Clearly, the performance of any implementation of this general approach is highly dependent on the consensus protocol it hinges upon.

This solution to sequence agreement has a problem, though: Because the decision of each instance is bounded to a single proposal, values proposed but not decided in a given instance must be reproposed on subsequent instances until they get decided, increasing their learning delay. Notice that even implementations in which proposals are composed of sets of messages (e.g., [CT96]) may suffer from this problem since there is no guarantee that all processes propose always the same sets in all instances.

Some consensus algorithms for the asynchronous model rely on a leader to coordinate the agreement procedure and this can be used to bypass the problem above. In such algorithms, proposals are sent to the leader, which selects one as the possible decision and continues with the algorithm execution. In a sequence agreement implementation, all the consensus instances could share the same leader, as done in Paxos [Lam98]. Instead of selecting an instance of consensus by the time a value is proposed, proposals could be just forwarded to the leader. The leader selects the first instance it has not used and continues with the algorithm as if the received proposal related to that instance. This gives to the sequence agreement implementation the same latency as the consensus protocol—three message steps in general, or two for values proposed by the leader.

There are consensus protocols that can achieve the latency of two message steps for multiple proposers by employing stricter conditions for a proposal to be decided (e.g., Fast

Paxos [Lam06a]). In such algorithms, there is no leader involved in the general case for getting a proposal decided but quorums are necessarily bigger. Moreover, the absence of a leader to circumvent the FLP impossibility result creates a problem called *collision*, which happens when two concurrent proposals are issued but none gets decided after two message steps [Lam06b]. To resolve a collision, extra message steps are required. It is possible to ensure a latency of two message steps in normal runs and avoid collisions. An asynchronous consensus algorithm that achieves this is called collision-fast. In [Lam06b], Lamport states the two conditions in which collision-fast asynchronous consensus algorithms are possible. The first case restricts fault tolerance to a single failure and is solved by a simple variant of Paxos, which allows the optimization we mentioned in the previous paragraph for sequence agreement. As for the second condition, which does not require strong failure assumptions, its algorithm applied to solving sequence agreement cannot solve the problem of different proposals for the same instance of consensus resulting in a single decision. Thus, non-decided proposals must be resubmitted in different instances, delaying their learning.

Indeed, when more than one process can fail, it seems impossible to use the standard reduction from sequence agreement to consensus and obtain a collision-fast sequence agreement protocol, that is, a sequence agreement implementation in which values proposed are learned within two message steps in normal runs. Differently, we reduce sequence agreement to a variation of consensus we call M-Consensus. In M-Consensus, processes decide not on a single proposed value, but on a bounded composition of them. This way, if concurrent proposals happen, *all* of them may take part in the final decision. To implement sequence agreement, we use a succession of M-Consensus instances as done before with standard consensus. Wise collision-fast implementations of M-Consensus, however, can produce a collision-fast sequence agreement. Collision-fast Paxos, our solution to M-Consensus, extends the original Paxos algorithm to allow multiple proposers, and not only the leader, to have their proposals decided in two message steps. As we show in the paper, our protocol can be used to implement a collision-fast sequence agreement that tolerates as many failures as the original Paxos.

3.2 Model and Definitions

3.2.1 Model

Instead of using processes, we state our definitions in terms of agents that perform actions in the system; processes can aggregate the roles of several agents. We assume an asynchronous crash-recovery model in which agents communicate by exchanging messages, with no bounds on the time it takes for messages to be transmitted or actions to be executed. Messages can be lost or duplicated but not corrupted; agents can fail by stopping only and never perform incorrect actions. Agents are assumed to have some sort of local stable storage to keep their state in between failures so that finite periods of absence are not

distinguishable from excessive slowness. Although we assume agents may recover, they are not obliged to do so once they have failed. An agent is considered to be nonfaulty iff it never stops executing enabled actions.

3.2.2 Sequence Agreement

As briefly explained in the previous chapter, given two sets of agents, namely *proposers* and *learners*, the sequence agreement problem consists of ensuring that learners learn increasing prefixes of a single sequence built out of the proposed value. In order to deal more easily with sequences throughout the chapter, we introduce some useful notation. We represent a sequence s as the tuple of its elements $\langle v_1, v_2, \ldots, v_n \rangle$, where n is the length of s and v_i equals $s[i]$, the sequence's i^{th} element. We say that sequence s is a prefix of sequence t, noted as $s \sqsubseteq t$, iff the length of s is less than or equal to the length of t and, for all i from 1 to the length of s, $s[i] = t[i]$; s and t are equal iff $s \sqsubseteq t$ and $t \sqsubseteq s$. The empty sequence $\langle\rangle$ has length zero and is a prefix of any other sequence. Sequence agreement's safety properties can then be stated as follows, where $learned[l]$ refers to the prefix currently learned by learner l, initially $\langle\rangle$.

Nontriviality For any learner l, $learned[l]$ contains only proposed values.

Stability For any learner l, if $learned[l] = s$ at some time, then $s \sqsubseteq learned[l]$ at all later times.

Consistency For any pair of learners l_1 and l_2, either $learned[l_1] \sqsubseteq learned[l_2]$ or $learned[l_2] \sqsubseteq learned[l_1]$.

The liveness requirement stated in the previous chapter, forcing every proposed value to be eventually learned by all learners is too strong for real systems. In an actual system, client applications propose commands and learn the result of their execution, tasks possibly associated with proposers and learners in our model. Since we cannot require clients not to fail, we define liveness in terms of another set of agents: the *acceptors*. Let a *quorum* be any finite set of acceptors large enough to ensure liveness. The liveness property of sequence agreement is defined as follows.

Liveness For any proposer p and learner l, if p, l and a quorum of acceptors are nonfaulty and p proposes value v, then eventually $learned[l]$ contains v.

3.2.3 Algorithms

In this section, we formally define sequence agreement algorithms and what it means for them to be collision-fast. The formal definitions we give are mostly borrowed from [Lam06b]; as in that work, we start by describing events.

An *event* is an action performed at some agent either spontaneously or triggered by the reception of a message. Each event e performed by agent e_{agent} sends exactly one message e_{msg}, receivable by any agent, including itself. We assume that events are totally ordered at the agents performing them, that is, we assume that each event e performed by agent e_{agent} is uniquely identified by the positive integer e_{num}, indicating that e was the $e_{num}{}^{\text{th}}$ event performed by e_{agent}. For an event e triggered by the reception of a message, we let e_{rcvd} equal the triple $\langle m, a, i\rangle$, where m is the received message, a is the agent that sent it, and i the index e_{num} of m's sending event e.

A *scenario* is the set of events performed in some single (partial) execution of an algorithm. For every event in a scenario, all other events that could have causally influenced it must also be in the scenario. To formally define a scenario, we let $\preceq_S$ be, for any set S of events, the transitive closure of the relation $\rightarrow$ on S such that $e \rightarrow f$ iff either (i) $e_{agent} = f_{agent}$ and $e_{num} \leq f_{num}$ or (ii) f is a message-receiving event and $f_{rcvd} = \langle e_{msg}, e_{agent}, e_{num}\rangle$.

Definition 3.1 (Scenario [Lam06b]) *A* scenario S *is a set of events such that:*

- *for any agent a, the set of events in S performed by a consists of k_a events numbered from 1 through k_a, for some natural number k_a;*
- *for every message-receiving event $e \in S$, there exists $d \in S$, $d \neq e$, such that $e_{rcvd} = \langle d_{msg}, d_{agent}, d_{num}\rangle$; and*
- *$\preceq_S$ is a partial order on S.*

A scenario obtained by removing the last events of a scenario S, according to the precedence relation $\preceq_S$, is called a *prefix* of S.[1]

Definition 3.2 (Prefix [Lam06b]) *A subset S of a scenario T is a* prefix *of T, written $S \sqsubseteq T$, iff for any events d in T and e in S, if $d \preceq_T e$ then d is in S.*

[1] For simplicity, we use the same nomenclature and notation to define the prefix relation between sequences, scenarios, and, as shown later, v-mappings. These sets are different and used in different contexts, which makes us believe this cannot be a source of confusion.

An algorithm can be seen as the set of non-empty scenarios it allows. However, we are only interested in algorithms that are compliant with our model. We define an *asynchronous algorithm* as follows, where $Agents(S)$ is the set of agents that performed events in S.

Definition 3.3 (Asynchronous Algorithm [Lam06b]) *An* asynchronous algorithm Alg *is a set of scenarios such that:*

- *every prefix of a scenario in Alg is in Alg; and*
- *if T and U are scenarios of Alg and S is a prefix of both T and U such that $Agents(T \setminus S)$ and $Agents(U \setminus S)$ are disjoint sets, then $T \cup U$ is a scenario of Alg.*

We define a *source* of a scenario S as an event $e \in S$ that is minimal in the ordering $\preceq_S$, and we let the depth of an event be the number of message steps that precede the event.

Definition 3.4 (Event Depth [Lam06b]) *The* depth *of an event e in a scenario S equals 0 if e is a source of S, otherwise it equals the maximum of*

(i) the depths of all events d with $d_{agent} = e_{agent}$ and $d_{num} < e_{num}$, and

(ii) if e is an event that receives a message sent by event b, then 1 plus the depth of b.

We now must define what a collision-fast sequence agreement protocol is. For simplicity, the definition we present considers only scenarios in which values are proposed in the source events. As a result, an algorithm might be collision-fast according to this simplified definition even if it does not ensure the same delivery latency for non-source proposals. Nonetheless, we believe that algorithms that satify our definition can be usually adapted to ensure the same delivery latency for all values proposed in normal runs, as this is the case for our solution.

A *normal scenario* is one in which the execution starts by one or more proposers proposing values, messages are not lost or duplicated, timeouts do not occur, messages are received in FIFO order, and no event receives a message with depth lower than its own minus one, that is, message reception is not delayed for two message steps or more.

Definition 3.5 (Normal Scenario [Lam06b]) *A scenario S is* normal *iff:*

- *the only sources of S are propose events;*

- *the message sent by any single event is not received twice by the same agent;*
- *every non-source event is a message receiving event;*
- *if* $d1$ *and* $d2$ *are events in* S *with* $d1_{agent} = d2_{agent}$ *and* $d1 \preceq_S d2$*, and* $e2$ *is an event in* S *that receives the message sent by* $d2$*, then there exists an event* $e1$ *in* S *with* $e1_{agent} = e2_{agent}$ *and* $e1 \preceq_S e2$ *such that* $e1$ *receives the message sent by* $d1$*; and,*
- *if* d *and* e *are events in* S *and* e *receives the messages sent by* d*, then* e_{depth} *equals 1 plus* d_{depth} *in* S*.*

Our definition of collision-fast sequence agreement states that the values initially proposed are delivered in two message steps. In order to measure that, we use the definition below.

Definition 3.6 (Complete to Depth [Lam06b]) *An agent* a *is* complete to depth δ *in a scenario* S *iff either* $\delta = 0$ *or every agent in* $Agents(S)$ *is complete to depth* $\delta - 1$ *and* a *receives every message sent by an event in* S *with depth less than* δ*.*

We consider a sequence agreement algorithm to be collision-fast iff there is a set M of agents and a set P of at least two proposers such that all values initially proposed by any subset O of the proposers in P are delivered by a learner l when l is complete to depth 2 in a normal scenario in which no agent in $M \cup O \cup \{l\}$ crashes. Our formal definition below is derived from the definition of Collision-fast Accepting in [Lam06b].

Definition 3.7 (Collision-fast Algorithm) *An asynchronous sequence agreement algorithm* Alg *is* collision-fast *iff there is a set* M *of agents and a set* P *of proposers with at least two proposers such that, for every nonempty subset* $\{p_1, \ldots, p_k\}$ *of* P *with* p_i *all distinct:*

- *for any proposable values* $v_1, \ldots, v_k$ *there is a scenario* $\{e_1, \ldots, e_k\}$ *in* Alg *such that each* e_i *is a source event in which* p_i *proposes* v_i*; and,*
- *for every learner* l *and every normal scenario* S *of* Alg *with* $Agents(S) = \{l, p_1, \ldots, p_k\} \cup M$ *that contains* $\{e_1, \ldots, e_k\}$ *as a prefix, if* l *is complete to depth 2 in* S*, then* $learned[l]$ *contains* $v_1, \ldots, v_k$*.*

3.3 M-Consensus

As mentioned before, instead of solving sequence agreement based on Consensus, we do it based on a different problem we call M-Consensus. The problem with standard Consensus

is that it does not allow multiple proposals to take part of a single decision. Differently, in the M-Consensus problem, where M stands for mapping, agents must agree on an increasing mapping from proposers to either proposed values or to the special value Nil. Before formalizing the problem, though, we define the value mapping data structure, v-mapping for short, it depends upon.

3.3.1 Value Mapping Sets

In order to introduce v-mappings, we must define some function notation. As usual, we let $f(d)$ be the result of function f for its domain element d. We represent the set of all functions with domain D and range R by $[D \rightarrow R]$, and the domain of a function f by $Dom(f)$. Moreover, we assume the existence of a special function $\perp$ such that $Dom(\perp) = \{\}$.

A value mapping set is a data structure defined in terms of sets $Domain$ and $Value$. Each pair $\langle Domain, Value \rangle$ corresponds to a different set $ValMap$ of value mappings, defined as all functions from subsets of $Domain$ to $Value \cup \{Nil\}$, where Nil is a special value not present in $Value$. More formally, $ValMap = \bigcup\{[D \rightarrow R] : D \subseteq Domain \wedge R = Value \cup \{Nil\}\}$. A v-mapping is therefore a function that maps some elements of $Domain$ to either a value in $Value$ or Nil. Notice that, since $\{\} \subseteq Domain$ for any set $Domain$, $\perp$ is present in every v-mapping set. To ease the presentation, hereinafter we consistently use uppercase letters for values in $Value \cup \{Nil\}$ and lowercase letters for v-mappings in $ValMap$.

We call a pair $\langle d, V \rangle$, where $d \in Domain$ and $V \in Value \cup \{Nil\}$, a ***single mapping***, or s-mapping for short, and define the append operation $v \bullet \langle d, V \rangle$, where v is a v-mapping and $\langle d, v \rangle$ is an s-mapping, to equal v-mapping f such that:

- $Dom(f) = Dom(v) \cup \{d\}$ and
- $\forall q \in Dom(f)$: IF $q \in Dom(v)$ THEN $f(q) = v(q)$ ELSE $f(q) = V$.

Informally, $v \bullet \langle d, V \rangle$ extends v with the s-mapping $\langle d, V \rangle$ iff d is not in the domain of v. The append operator defines a partial order relation on a v-mapping set. We say that v-mapping v is a *prefix* of v-mapping w, and w is an extension of v ($v \sqsubseteq w$), iff w can be generated from v by a series of append operations. The precedence between v and w can be easily checked since $v \sqsubseteq w$ iff $Dom(v) \subseteq Dom(w)$ and $\forall d \in Dom(v) : v(d) = w(d)$. We define $v \sqsubset w$ to be true iff $v \sqsubseteq w$ and $v \neq w$.

Given a set $T \subseteq ValMap$, we say that v-mapping v is a lower bound of T iff $v \sqsubseteq w$ for all w in T. A greatest lower bound (glb) of T is a lower bound v of T such that

$w \sqsubseteq v$ for every lower bound w of T. Similarly, we say that v is an upper bound of T iff $w \sqsubseteq v$ for all w in T. A least upper bound (lub) of T is an upper bound v of T such that $v \sqsubseteq w$ for every upper bound w of T. There is always a unique glb for a set T of v-mappings. The existence of a lub, however, depends on whether the set T is compatible, but if it exists, then it is unique. Two v-mappings v and w are defined to be *compatible* iff there exists a v-mapping u such that $v \sqsubseteq u$ and $w \sqsubseteq u$. A set S of v-mappings is compatible iff its elements are pairwise compatible. Compatibility can be easily checked since two v-mappings are compatible iff the elements in the intersection of their domains are mapped to the same values. Given a set T, we represent its glb by $\sqcap T$, and its lub by $\sqcup T$. Moreover, for simplicity, we use $v \sqcap w$ and $v \sqcup w$ to represent $\sqcap\{v, w\}$ and $\sqcup\{v, w\}$, respectively.

We say that a value mapping is *complete* iff its domain equals $Domain$. It is easy to see that a complete v-mapping does not have any strict extension, since no append operation applied to it can result in a different v-mapping. An interesting complete v-mapping is the one that maps every element in $Domain$ to Nil. This v-mapping is independent of the set $Value$ and, for this reason, we call it the *trivial* v-mapping. A v-mapping is *nontrivial* iff it is different from the trivial one.

3.3.2 Problem Definition

As we have done for sequence agreement, we define the M-Consensus problem in terms of the sets of proposer, acceptor, and learner agents, and a set of proposable values. The problem considers the v-mapping set with $Domain$ equal to the set of proposers and $Value$ equal to the set of proposable values. Proposers propose values and learners learn v-mappings that can differ but must always be compatible, can only be extended, and must eventually equal the same complete nontrivial v-mapping. We say that a v-mapping is *proposed* iff all elements of its domain are mapped either to Nil or to a proposed value and we let $learned[l]$ represent the v-mapping currently learned by learner l, initially $\perp$. Based on that, the properties of M-Consensus are defined as follows:

Nontriviality For any learner l, $learned[l]$ is always a nontrivial proposed v-mapping.

Stability For any learner l, if $learned[l] = v$ at some time, then $v \sqsubseteq learned[l]$ at all later times.

Consistency The set of learned v-mappings is always compatible and has a nontrivial lub.

Liveness For any proposer p and learner l, if p, l and a quorum of acceptors are nonfaulty and p proposes a value, then eventually $learned[l]$ is complete.

Learners initially know $\perp$, which is a valid prefix for any v-mapping. As proposers make proposals, learners can extend their learned v-mappings as long as they are always

proposed and nontrivial. Note that a v-mapping that maps all its domain to Nil but does not cover all elements in $Domain$ is nontrivial and can be learned by a learner, which is not a problem since the remaining elements of $Domain$ can still be mapped to some value. Consistency ensures that all currently learned values can be extended to a common v-mapping that satisfies the Nontriviality property. The existence of a nontrivial lub is also implied by Liveness and Nontriviality, but its presence in the Consistency property makes the problem specification machine-closed [AL91], isolating safety from liveness properties. The Liveness property states that all correct learners will eventually learn a complete v-mapping, which implies that, like consensus and differently from sequence agreement, an instance of M-Consensus eventually terminates.

With respect to solvability, M-Consensus is equivalent to consensus. It is easy to see that an algorithm that solves consensus can solve M-Consensus by just having learners learn a mapping in which a specific proposer is mapped to the decided value and all the others are mapped to Nil. An algorithm that solves M-Consensus also trivially solves consensus by just totally ordering the set of proposers and picking up the value mapped to the first proposer not mapped to Nil. Actually, this equivalence lends to M-Consensus all known lower bounds and impossibility results for consensus. The advantage of M-Consensus, though, has to do with the implementation of sequence agreement since it allows two concurrent proposals to appear in the problem solution, mapped to different proposers. This avoids the proposal collision problem present in consensus-based sequence agreement and explained in Section 3.1.

3.4 Collision-fast Paxos

This section describes Collision-fast Paxos, our solution to M-Consensus. Collision-fast Paxos builds on top of the original Paxos consensus algorithm [Lam98]. It extends the original protocol to allow multiple processes, instead of a single coordinator, to propose values that will take part in the final decision. We later employ Collision-fast Paxos to solve sequence agreement.

3.4.1 Basic Algorithm

We first describe our basic algorithm, which satisfies safety but does not guarantee liveness, a topic addressed in the next section. The algorithm is structured in rounds and the only assumption we make about them is that they are totally ordered by a relation $\leq$. For simplicity, it can be assumed that rounds correspond to the natural numbers unless we explicitly state it differently (Section 3.4.3). As in the original Paxos protocol, every round has a single coordinator assigned to it. Coordinators represent a different sort of agent besides

proposers, acceptors, and learners.

We also assign to each round r a subset of the proposers we call the collision-fast proposers of r. The collision-fast proposers of a round are the only proposers allowed to have their proposals learned in two communication steps at that round. As we explain later, making all proposers collision-fast for all rounds would restrict the algorithm's resilience.

At some round r, a collision-fast proposer p *fast-proposes* an s-mapping $\langle p,\ V\rangle$ at most once. It does that when it has a value to be proposed or when it notices that another collision-fast proposer of round r has fast proposed a non-Nil value—a situation in which p fast-proposes $\langle p, Nil\rangle$. If the fast proposal contains a mapping with a proposed value, it is sent to the acceptors and other collision-fast proposers; otherwise it is sent directly to the learners. An acceptor may *accept* multiple v-mappings as long as the newly accepted v-mapping extends the previous one. The v-mappings accepted by the acceptors are generated from the non-Nil s-mappings fast-proposed and, therefore, always map at least one proposer to a non-Nil value.

We say that a v-mapping v is *chosen* at round r iff there exists a (possibly empty) subset P of the collision-fast proposers of r such that the two conditions below hold:

- every proposer $p \in P$ has fast-proposed s-mapping $\langle p, Nil\rangle$ and
- there exists a quorum Q of acceptors such that every acceptor $a \in Q$ has accepted a v-mapping w such that v is a prefix of w extended with $\langle p, Nil\rangle$ for every proposer $p \in P$.

More intuitively, one can think that if a collision-fast proposer p has fast-proposed $\langle p, Nil\rangle$ at round r, then every acceptor that has accepted or later accepts some v-mapping at r will "automatically", though not explicitly, extend it with $\langle p, Nil\rangle$. Thinking this way, a v-mapping is chosen at round r if it is a prefix of every v-mapping accepted by some quorum acceptor Q at r.

Chosen v-mappings are guaranteed to be compatible and a learner can extend $learned[l]$ by setting it to the lub between $learned[l]$ and any chosen v-mapping. If at least one collision-fast proposer fast-proposes a value, no process crashes, and messages are correctly delivered, it is easy to see that learners learn a complete nontrivial v-mapping within two message steps. However, new rounds might have to be started due to failures. To ensure consistency in this case v-mappings chosen in some round must be made compatible with v-mappings chosen in other rounds.

The algorithm keeps the invariant that if a v-mapping is or might be chosen at some round r then any v-mapping accepted at a higher-numbered round extends the possibly chosen one. This is guaranteed by the actions taken to start a new round. A new round's

coordinator queries a quorum of acceptors to discover if some v-mapping has been or might be chosen at a lower-numbered round. If this is the case, the coordinator extends such v-mapping with Nil mappings to make it complete and sends it to the acceptors for it to be accepted and chosen directly. If no v-mapping has been or might be chosen at a lower-numbered round, the collision-fast proposers of the current round are notified that they can fast-propose for that round (collision-fast proposers wait for this confirmation before fast-proposing at some round).

For the coordinator to be able to identify if some value has been or might be chosen at a lower-numbered round by just querying a quorum of acceptors, we need the following assumption about quorums:

Assumption 3.1 (Quorum Requirement) *If Q and R are quorums, then $Q \cap R \neq \emptyset$.*

In fact, any general algorithm for asynchronous consensus (and, therefore, M-Consensus) must satisfy a similar requirement, as shown by the Accepting Lemma in [Lam06b]. A simple way to ensure this is defining quorums as any majority of the acceptors.

To make our algorithm description precise, we must explain the variables required by each agent. A proposer p has the following variables:

$prnd[p]$: The current round of p. Initially 0.

$pval[p]$: The value p has fast-proposed at round $prnd[p]$ or special value $none$ if p has not fast-proposed anything at round $prnd[p]$. Initially $none$.

The variables of a coordinator c are:

$crnd[c]$: The current round of c. Initially 0.

$cval[c]$: The initial v-mapping for round $crnd[c]$, if c has already queried a quorum of acceptors for $crnd[c]$ or special value $none$ otherwise. Initially $\perp$ for the coordinator of round 0 and $none$ for all the others.

An acceptor a keeps three variables:

$rnd[a]$: The current round of a. Initially 0.

$vrnd[a]$: The round at which a has accepted its latest value. Initially 0.

$vval[a]$: The v-mapping a has accepted at $vrnd[a]$ if it has accepted something at $vrnd[a]$, or special value $none$ otherwise. Initially $none$.

Each learner l keeps only the v-mapping it has learned so far.

$learned[l]$: The v-mapping currently learned by l. Initially $\bot$.

In the following, we present the basic atomic actions that compose the algorithm.

$Propose(p, V)$ Executed by proposer p when it wants to propose value V. p sends message $\langle$"`propose`", $V\rangle$ to some collision-fast proposer for round $prnd[p]$. It is just a local message (from one agent to another inside the same process) if p is a collision-fast proposer of $prnd[p]$.

$Phase1a(c, r)$ Executed by coordinator c to start round r. It is enabled iff:

- c is the coordinator of round r and
- $crnd[c] < r$.

It sets $crnd[c]$ to r, $cval[c]$ to $none$, and sends message $\langle$"`1a`", $r\rangle$ to the acceptors.

$Phase1b(a, r)$ Executed by acceptor a, for round r. It is enabled iff:

- a has received a $\langle$"`1a`", $r\rangle$ message and
- $rnd[a] < r$

It sets $rnd[a]$ to r and sends message $\langle$"`1b`", r, a, $vrnd[a]$, $vval[a]\rangle$ to the coordinator of round r. Setting $rnd[a]$ to r makes sure that no mapping will be further accepted by a at a round lower than r and the "`1b`" message tells the coordinator of r that the last value accepted by a for a round lower than r was $vval[a]$ at round $vrnd[a]$.

$Phase2Start(c, r)$ Executed by coordinator c of round r. This action picks up an initial v-mapping for round r based on the "`1b`" messages the coordinator c received for round r from a quorum of acceptors. It is enabled iff:

- $r = crnd[c]$,
- $cval[c] = none$, and
- c has received a "`1b`" message for round r from every acceptor in a quorum Q.

Let k be the highest $vrnd$ field received in the "`1b`" messages mentioned above and let S be the set of all v-mappings (different from $none$) received in the "`1b`" messages with field $vrnd$ equal to k. If S is empty, then no v-mapping has been or might be chosen at a lower-numbered round and c can pick up v-mapping $\bot$ to start round r. In this case, it sets $cval[c]$ to $\bot$ and sends message $\langle$"`2S`", $r, \bot\rangle$ to all proposers, allowing them to fast-propose when they are ready. Acceptors need not be notified in this case.

If S is not empty, then it might be the case that some v-mapping has been or might be chosen at a round lower than or equal to k. As mentioned before, the algorithm guarantees that if a v-mapping was or might be chosen at some round lower than k, then it is a prefix of all values accepted in k, including those in S. Moreover, if any v-mapping has been or might be chosen at round k, then, by the quorum assumption, it must have been accepted by some acceptor in Q and, thus, is present in S. As we explain in action $Phase2b(a, r)$, v-mappings accepted by acceptors for the same round are always compatible and this obviously guarantees the compatibility of set S. Therefore, $\sqcup S$ extends both the v-mappings possibly chosen at rounds lower than k and the v-mappings possibly chosen at k. Because acceptors only accept v-mappings that map at least one proposer to a non-Nil value, $\sqcup S$ also satisfies this property and extending it with s-mappings $\langle p, Nil\rangle$ for every proposer p does not generate the trivial mapping. Let v be $\sqcup S$ extended with $\langle p, Nil\rangle$ for every proposer p; coordinator c sets $cval[c]$ to v and sends message $\langle$"2S", r, $v\rangle$ to all acceptors and proposers.

$Phase2Prepare(p, r)$ Executed by proposer p, for round r. It is enabled iff:

- $prnd[p] < r$ and
- p has received a message $\langle$"2S", r, $v\rangle$.

First, it sets $prnd[p]$ to r. If $v = \bot$, it sets $pval[p]$ to $none$; otherwise, it sets $pval[p]$ to $v(p)$. Recall, from action $Phase2Start(c, r)$ above, that a "2S" message for any round contains either $\bot$ or a complete v-mapping.

$Phase2a(p, r, V)$ Executed by proposer p, where r is the current round of p and V is either a proposed value or Nil. It is enabled iff:

- $prnd[p] = r$,
- p is a collision-fast proposer of r,
- $pval[p] = none$, and
- either p has received message $\langle$"propose", $V\rangle$ or V equals Nil and p has received message $\langle$"2a", r, $\langle q, W\rangle\rangle$, where $\langle q, W\rangle$ is an s-mapping from any proposer q to a non-Nil value W.

It sets $pval[p]$ to V and sends message $\langle$"2a", r, $\langle p, V\rangle\rangle$ either to the acceptors and other collision-fast proposers of r, if V does not equal Nil, or directly to the learners otherwise. In this action, proposer p fast-proposes, giving its opinion about the value it should be mapped to. It is triggered by the receipt of a "propose" message with a proposed value (a local 0-latency message if p sent it to itself) or by the receipt of a "2a" message from another collision-fast proposer, which forces p to set its opinion to Nil.

$Phase2b(a, r)$ Executed by acceptor a, for round r and v-mapping v. It is enabled iff:

- $rnd[a] \leq r$ and

- Either one of the two following conditions is satisfied:
  a) a has received message $\langle$"2S", $r, v\rangle$, where $v \neq \bot$, and $vrnd[a] < r$ or $vval[a] = none$
  b) a has received message $\langle$"2a", $r, \langle p, V\rangle\rangle$, where $V \neq Nil$.

It sets $rnd[a]$ and $vrnd[a]$ to r and changes $vval[a]$ depending on whether condition (a) or (b) above is satisfied. If condition (a) is true, it sets $vval[a]$ to v. If condition (b) is true and $vrnd[a] < r$ or $vval[a] = none$, then it sets $vval[a]$ to $\bot \bullet \langle p, V\rangle$ extended with $\langle q, Nil\rangle$ for every proposer q that is not collision-fast for r; otherwise, it sets $vval[a]$ to its previous value extended with $\langle p, V\rangle$, that is, $vval[a] \bullet \langle p, V\rangle$. It then sends message $\langle$"2b", $r, a, vval[a]\rangle$ to all learners, with the updated value of $vval[a]$.

Condition (a) implies that the coordinator of round r has picked up v-mapping $v \neq \bot$ for round r based on the votes of a quorum of acceptors for lower-numbered rounds. As explained in action $Phase2Start(c, r)$, this v-mapping v is complete and different from the trivial mapping.

Condition (b) implies that the coordinator of round r has picked up $\bot$ for the initial v-mapping of r and collision-fast proposers were allowed to fast-propose. In this case, the first mapping acceptor a accepts for round r maps the proposer p that sent the "2a" message to the (non-Nil) value it sent and maps every proposer that is not collision-fast for r to Nil, since they are not allowed to fast-propose. When a receives the "2a" messages from other collision-fast proposers of round r with non-Nil values, a just appends the received s-mapping to the previously accepted v-mapping.

It is not possible that, for some round r, an acceptor executes this action due to condition (a) and another acceptor executes it due to condition (b). If acceptors execute this action for a round r satisfying condition (a), they must accept the same complete v-mapping v. If acceptors execute this action for r satisfying condition (b), they must accept v-mappings that map a proposer p either to Nil, if p is not collision-fast for r, or to the value p sent in its "2a" message, if p is collision-fast for r. Since no proposer can send different "2a" messages for the same round, all v-mappings accepted by condition (b) must be compatible. This argument shows that it is safe to calculate the lub of any set of v-mappings accepted for the same round, as done in action $Phase2Start(c, r)$.

$Learn(l)$ Executed by learner l. It is enabled iff l has received "2b" messages for some round r from a quorum Q and message $\langle$"2a", $r, \langle p, Nil\rangle\rangle$ from every proposer p in a (possibly empty) subset P of the collision fast proposers of round r. It calculates the lub of the chosen v-mappings based on the received information in order to update the currently learned v-mapping of l. Let $Q2bVals$ be the set of all v-mappings received in the "2b" messages for round i from acceptors in Q, and let $newv$ be $\sqcap Q2bVals$ extended with $\langle p, Nil\rangle$ for every proposer p in P. The action sets $learned[l]$ to $learned[l] \sqcup newv$.

3.4.2 Ensuring Liveness

The previous actions ensure safety, but if messages are lost, coordinators or collision-fast proposers crash, or coordinators keep on starting new rounds, then they will not ensure progress. We now extend the algorithm for that. Some of the assumptions we make are very basic. It is clear that no algorithm can ensure progress if messages can be indiscriminately lost and non-crashed agents indefinitely refuse to take actions that are enabled. Therefore, we assume that if agents a and b do not crash and a keeps resending message m to b, then b eventually receives m. Moreover, we assume weak fairness on the actions an agent may take, that is, no action remains enabled forever without being executed. We tacitly assume that an action is enabled only if its agent is not crashed.

The FLP result and the equivalence between consensus and M-Consensus with respect to solvability imply that these assumptions are not enough to ensure liveness for M-Consensus. As in the original Paxos protocol, we circumvent FLP by eventually electing a distinguished coordinator—the leader—responsible for starting new rounds. For it to work, we require also that every coordinator be reponsible for infinitely many higher-numbered rounds, which is easily ensured by having round numbers defined as tuples $\langle n, c\rangle$ where n is a natural number and c is its coordinator identifier.

When the leader starts a round and picks up $\perp$ as its initial value, the round will only succeed in getting a complete v-mapping chosen and learned if all its collision-fast proposers remain up. This is inherent to collision-fast consensus algorithms like ours as implies the Collision-fast Learning Theorem of [Lam06b] and, in fact, it is the main reason why we designed Collision-fast Paxos so that the set of collision-fast proposers depends on the round; had we done it differently, the failure of any collision-fast proposer would not allow our algorithm to become collision-fast again. As a result, the leader must be able to somehow identify when a collision-fast proposer of the current round has crashed in order to start a new one. We assume that a coordinator c that believes itself to be the leader keeps a set $activep[c]$ with all the proposers it believes to be currently up. We assume this set can take any valid value but, in order to ensure liveness, it must eventually satisfy some conditions we show later in this section.

For progress, we need to make a number of small changes to the algorithm we presented in Section 3.4.1:

- We add "c believes itself to be the leader" as a pre-condition to actions $Phase1a(c, r)$ and $Phase2Start(c, r)$.
- If an acceptor a receives a "`1a`", "`2S`", or "`2a`" message for round r such that $r < rnd[a]$ and the coordinators of r and $rnd[a]$ differ, then a sends a special message to the coordinator of r to inform that round $rnd[a]$ was initiated.
- The same sort of special message is sent if a proposer p receives a "`2S`" message for

round r such that $r < prnd[p]$ and the coordinators of r and $prnd[p]$ differ.

- Besides the first modification, coordinator c executes action $Phase1a(c, r)$ only if either it receives a special message informing of round j ($r > j > crnd[c]$) was initiated, or the set of collision fast-proposers of $crnd[c]$ is not a subset of $activep[c]$ but the set of collision-fast proposers of r is.
- Each proposer p that has sent a "`propose`" message keeps resending it to one of the collision-fast proposers of $prnd[p]$.
- Each coordinator that believes itself to be the leader keeps resending, to all its original receivers, the last "`1a`" or "`2S`" message it sent.
- Each proposer p that has executed action $Phase2a(p, r, V)$, for round $r = prnd[p]$ and any V, keeps resending the last "`2a`" message it sent.
- Each acceptor keeps resending the last "`1b`" or "`2b`" message it sent.

These changes do not affect safety because they incur new actions that do not change the algorithm's variables and make some actions' pre-conditions more restrictive only.

Except for the conditions related to new variable $activep$, the liveness assumption of Collision-fast Paxos is the same as the one of the original protocol (c.f. Section 2.3 of [Lam06a]). We define $LA(p, l, c, Q)$ for any proposer p, learner l, coordinator c, and quorum Q of acceptors, to be the conjunction of the following conditions:

- $\{p, l, c\} \cup Q$ are not crashed.
- p has proposed a value.
- c is the only coordinator that believes itself to be the leader.
- All proposers in $activep[c]$ are not crashed.
- For every round $r > crnd[c]$, c is the coordinator of a round $s > r$ whose collision-fast proposers are all in $activep[c]$.
- $activep[c]$ is a subset of all its future values.

If $LA(p, l, c, Q)$ holds for some proposer p, coordinator c, and quorum Q, from some point in time on, then eventually l learns a complete v-mapping. If every coordinator is itself the only collision-fast proposer for infinitely higher-numbered rounds that it coordinates, then Collision-fast Paxos could ensure liveness in the same situations where Paxos would. In fact, a round in which the only collision-fast proposer is the round coordinator itself implements a standard Paxos round.

3.4.3 Runtime Reconfiguration

In any real world scenario, distributed systems are subject to crashes, slow components, and workload variations. To handle these situations Collision-Fast Paxos explores the fact that M-Consensus poses no limitations on the number of proposers, and assumes an infinite set of proposers and coordinators. At the bootstrap, just a relatively small set of proposers and coordinators is made active. If some agent becomes unresponsive or if any other change in the environment requires it, agents can be activated or permanently deactivated.

As we mentioned before, the set of collision-fast proposers is defined per round, so that failed proposers can be excluded from the set to allow collision-fast termination even after failures. For that, we extend round numbers with the round's set of collision-fast proposers, defining round numbers as tuples of the form $\langle n, c, cf \rangle$, where n is a natural number, c is the round's coordinator, and cf is the sorted list of the round's collision-fast proposers. It is clear from this definition that a lexicographical comparison induces a total order on the round numbers. To ensure the uniqueness of the special round $Zero$, it is defined *a priori* as $\langle 0, c, cf \rangle$, for some coordinator c and list cf. This scheme grants to each coordinator an infinite number of rounds for every possible set of collision-fast proposers.

Another consequence of being able to replace crashed proposers and coordinators by brand new ones is that these agents do not have to write on disk. Since the number of proposers, coordinators, and learners, is not limited by any assumption in the algorithm, these agents do not have to keep their state on stable storage. If one of them crashes and later recovers, it can simply assume a completely new identity (based on some incarnation number) before joining the system [Agu04]. Acceptors still have to write their state on stable storage, but optimizations similar to those presented in [CSP06].

3.5 Solving Sequence Agreement

Solving sequence agreement with M-Consensus is simple; achieving a collision-fast solution, though, depends on the M-Consensus algorithm in use. We first present the general approach and then extend it to use Collision-fast Paxos.

3.5.1 General Approach

To implement sequence agreement, we use infinitely many M-Consensus instances, each one uniquely identified by a natural number. To differentiate messages and variables of different instances we superscript them with the instance's identification (e.g., $learned^i$, "$\texttt{1b}^j$"). Sequence agreement proposers act both as proposers and learners in each of the

M-Consensus instances.

To propose a value v for sequence agreement, a proposer p proposes v in the smallest instance of M-Consensus i in which it has neither proposed nor learned anything yet. Being also a learner, p eventually learns the decision of i, and checks if there exists some proposer q such that $learned^i[p](q) = v$. If there is, then p knows that v was successfully proposed and will eventually be learned by all nonfaulty learners; otherwise, p re-proposes v in the next free M-Consensus instance. This procedure can be executed in parallel for many values.

Assuming there is a known total order of proposers, learner l in the sequence agreement problem builds its sequence $learned[l]$ by appending mapped results different from Nil, following a $(instance, proposer)$ order.

3.5.2 Collision-fast Paxos Approach

Using Collision-fast Paxos, all M-Consensus instances can share the same coordinator. This also allows us to keep all instances synchronized with respect to their current round in all agents. As a result, variables $rnd[a]$, $prnd[p]$, and $crnd[c]$ can be shared amongst all instances. The other variables are not shared but could be allocated for an instance only when their value changes from the initial one.

When a coordinator executes action $Phase1a(c, r)$, it does that for all instances and sends a single "1a" message. An acceptors a that executes action $Phase1b(a, r)$, also does that for all instances and aggregates all "$1b^i$" messages it should send in a single one. Only a finite number of instances will have $vval^i[a] \neq none$, which allows the compression of this message to a finite size. After collecting these composite "1b" messages from a quorum of acceptors, a coordinator c executes $Phase2Start(c, r)$ for all instances and, similarly, generates a composite "2S" message containing the "$2S^i$" message of every instance i. A proposer p that receives such composite "2S" message, simply executes $Phase2Prepare(p, r)$ for all instances.

The actions above are executed only when the leader changes. During normal execution, things are simpler. A collision-fast proposer that wants to propose value v, fast-proposes v in the first instance i for which $pval^i[p] = none$ by executing action $Phase2a^i(p, r, M)$. If everything goes fine, the message will be eventually learned; if failures or suspicions prevent the normal case, eventually $pval^i[p]$ will change from v to Nil due to a "2S" message and p will notice that it will have to repropose v in another instance. When a proposer that is not collision-fast for its current round wants to propose a value, it simply forwards it to one of the collision-fast proposers. Notice that, since Collision-fast Paxos ensures that a collision-fast proposer eventually knows if its fast proposal was learned or not, this implementation does not require that proposers be learners too. Acceptors and Learners execute

actions $Phase2b(a, r)$ and $Learn(l)$ independently for each instance. As for progress, this sequence agreement implementation has the same liveness condition as Collision-fast Paxos.

As discussed in Section 3.4.1, in the normal case, if a collision-fast proposer p fast-proposes a message, then a v-mapping containing it is learned in two message steps. If there are no concurrent (non-Nil) fast-proposals for the same instance, this v-mapping will be complete. Otherwise, a learner complete to depth 2 plus the depth of p's fast proposal will learn a complete v-mapping containing all fast proposals, since all are learned in two steps. Because a value to be proposed by a collision-fast proposer never waits to be fast-proposed in some instance and a collision-fast proposer leaves no gaps between instances, this sequence agreement algorithm is collision-fast. In fact, according to our definition it is collision-fast for P equal to the collision-fast proposers of round 0 and M equal to $Q \cup P$ where Q is a quorum.

3.6 Correctness of Collision-fast Paxos

This section presents the proof that Collision-fast Paxos satisfies the safety properties of M-Consensus.

3.6.1 Preliminaries

We start by defining a special data structure we call a *ballot array*. Our definition is highly-inspired by the data structure with the same name presented in [Lam04]. A ballot array represents the voting history of a set of acceptors, that is, the history of v-mappings accepted by acceptors on different rounds. For every acceptor a, it keeps the current round of a, $\widehat{bA}_a$, and, for every acceptor a and round r, the vote a has cast at r, $bA_a[r]$. If an acceptor has not cast a vote at round r, then $bA_a[r]$ equals special value $none$. To ease the design of our algorithms, we force acceptors to vote only for v-mappings that have at least one element of their domain mapped to a non-Nil value. A v-mapping that satisfies this constraint is called a *valued* v-mapping. The complete definition of a ballot array is given below.

Definition 3.8 (Ballot Array) *A* ballot array bA *is a mapping that assigns to each acceptor* a *a round* $\widehat{bA}_a$ *and to each acceptor* a *and round* r *a value* $bA_a[r]$ *that is a v-mapping or equals* $none$*, such that for every acceptor* a*:*

- *The set of rounds* m *with* $bA_a[m] \neq none$ *is finite,*
- $bA_a[r] = none$ *for all rounds* $r > \widehat{bA}_a$*, and*

- $bA_a[r]$ *is either* $none$ *or a valued v-mapping for all rounds* r.

As mentioned in Section 3.4.1, a v-mapping is learned depending not only on the votes cast by acceptors but also on the Nil values proposed by proposers. Because of that, we define another data structure we call a *proposal array*. A proposal array represents a history of proposals made by proposers at different rounds. It keeps, for every proposer p and round r, the (possibly Nil) value p has proposed at round r, or special value $none$ if p has not proposed at round r.

Definition 3.9 (Proposal Array) *A* proposal array pA *is a mapping that assigns to each proposer* p *and round* r *a value* $pA_p[r]$ *that is either a proposable value, special value* Nil*, or special value* $none$*.*

In fact, proposal arrays are important only for their Nil proposals because these proposals are used to define a $chosen$ v-mapping, that is, a v-mapping that can be safely learned by a learner without jeopardizing consistency. Before we give a formal definition for a chosen v-mapping, though, we have to introduce the operator $NilExtension(v, P)$, which we refer to as the Nil-extension of v for P where v is a v-mapping and P is a set of proposers. This operator returns $none$ if v equals $none$; otherwise, it is defined as v-mapping w satisfying the three conditions below:

1. $Dom(w) = Dom(v) \cup P$
2. $\forall p \in Dom(w) \cap Dom(v) : w(p) = v(p)$
3. $\forall p \in Dom(w) \setminus Dom(v) : w(p) = Nil$

Intuitively, $NilExtension(v, P)$ extends v by mapping each proposer in $P \setminus Dom(v)$ to Nil. A different but equivalent definition of this operator appears in TLA$^+$ module $PaxosConstants$ in Section 3.8.

We say that a v-mapping v is *chosen at* some round r in pair $\langle bA, pA \rangle$, where bA is a ballot array and pA is a proposal array, iff there is a quorum Q of acceptors and a (possibly empty) set P of collision-fast proposers for r that have proposed Nil at round r such that, for every acceptor a in Q, v is a prefix of the $NilExtension$ of the v-mapping a has accepted at round r for P. For completeness, we define that $none$ is not a prefix or an extension of any v-mapping.

Definition 3.10 (Chosen at) *A v-mapping* v *is* chosen at *round* r in $\langle bA, pA \rangle$*, where* bA *is a ballot array and* pA *is a proposal array, iff there exists a set* P *of collision-fast proposers for* r *and a quorum* Q *such that:*

- $\forall p \in P : pA_p[r] = Nil$
- $\forall q \in Q : v \sqsubseteq NilExtension(bA_a[r], P)$

A v-mapping v is chosen in $\langle bA, pA\rangle$ *iff it is chosen at some round r in $\langle bA, pA\rangle$.*

We say that a v-mapping v is *choosable at* some round r if it is possible to extend the voting history represented by bA and the proposal array pA so that v satisfies the condition above to be considered $chosen$ at r in $\langle bA, pA\rangle$.

Definition 3.11 (Choosable at) *A v-mapping v is* choosable at *round r* in *pair $\langle bA, pA\rangle$, where bA is a ballot array and pA is a proposal array, if, and only if, considering P to be the set of proposers p such that $pA_p[r]$ is either Nil or $none$, there exists a quorum Q such that $v \sqsubseteq NilExtension(bA_a[r], P)$ for every acceptor a in Q with $\widehat{bA}_a > r$.*

We say that a v-mapping is safe at some round in a pair $\langle bA, pA\rangle$, where bA is a ballot array and pA is a proposal array, if it extends all v-mappings that are choosable at lower-numbered rounds in $\langle bA, pA\rangle$. We also say that a a pair $\langle bA, pA\rangle$ is safe if all v-mappings that acceptors have voted for in bA are *safe* at the rounds they were accepted in $\langle bA, pA\rangle$.

Definition 3.12 (Safe at) *A v-mapping v is* safe at *round r* in *$\langle bA, pA\rangle$, where bA is a ballot array and pA is a proposal array, iff $w \sqsubseteq v$ for every round $k < r$ and every v-mapping w that is choosable at k. A pair $\langle bA, pA\rangle$ is* safe *iff for every acceptor a and balnum k, if $bA_a[k] \neq none$ then it is safe at k in $\langle bA, pA\rangle$.*

The proposition below states that if a pair of ballot and proposal arrays is safe, then all its chosen v-mappings are compatible.

Proposition 3.1 *Let bA be a ballot array and pA be a proposal array, if $\langle bA, pA\rangle$ is safe, then the set of values that are chosen in $\langle bA, pA\rangle$ is compatible.*

PROOF: By the definition of Consistency, it suffices to
ASSUME: 1. $\langle bA, pA\rangle$ is safe
2. v-mapping v is chosen at round r in $\langle bA, pA\rangle$
3. v-mapping w is chosen at round $s \geq r$ in $\langle bA, pA\rangle$
PROVE: v and w are compatible.
1. Choose a quorum Q_v and set P_v of collision-fast proposers for r such that
   - $\forall p \in P_v : pA_p[r] = Nil$

- $\forall q \in Q_v : v \sqsubseteq \mathit{NilExtension}(bA_a[r], P_v)$

PROOF: This follows from proof assumption 2 and the definition of *chosen at*.

2. Choose a quorum Q_w and set P_w of collision-fast proposers for s such that
   - $\forall p \in P_w : pA_p[r] = \mathit{Nil}$
   - $\forall q \in Q_w : v \sqsubseteq \mathit{NilExtension}(bA_a[r], P_w)$

   PROOF: This follows from proof assumption 3 and the definition of *chosen at*.
3. CASE: $r = s$
   - 3.1. Choose an acceptor a in $Q_v \cap Q_w$
     PROOF: a exists by the Quorum Requirement (Assumption 3.1).
   - 3.2. $v \sqsubseteq \mathit{NilExtension}(bA_a[r], P_v \cup P_w)$
     - 3.2.1. $v \sqsubseteq \mathit{NilExtension}(bA_a[r], P_v)$
       PROOF: By steps 1 and 3.1.
     - 3.2.2. $\mathit{NilExtension}(bA_a[r], P_v) \sqsubseteq \mathit{NilExtension}(bA_a[r], P_v \cup P_w)$
       PROOF: By the definition of Nil-extension.
     - 3.2.3. Q.E.D.
   - 3.3. $w \sqsubseteq \mathit{NilExtension}(bA_a[r], P_v \cup P_w)$
     - 3.3.1. $w \sqsubseteq \mathit{NilExtension}(bA_a[r], P_w)$
       PROOF: By steps 2 and 3.1.
     - 3.3.2. $\mathit{NilExtension}(bA_a[r], P_w) \sqsubseteq \mathit{NilExtension}(bA_a[r], P_v \cup P_w)$
       PROOF: By the definition of Nil-extension.
     - 3.3.3. Q.E.D.
   - 3.4. Q.E.D.
     PROOF: By steps 3.2 and 3.3 and the definition of compatible.
4. CASE: $r < s$
   - 4.1. v is choosable at r in $\langle bA, pA \rangle$
     PROOF: By the definition of choosable, any v-mapping chosen at some round is also choosable at it.
   - 4.2. Choose any acceptor a in Q_w
     PROOF: The Quorum Requirement implies that quorums cannot be empty.
   - 4.3. $v \sqsubseteq bA_a[s]$
     PROOF: By the fact that $\langle bA, pA \rangle$ is safe (proof assumption 1).
   - 4.4. $w \sqsubseteq \mathit{NilExtension}(bA_a[s], P_w)$
     PROOF: By Step 2.
   - 4.5. Q.E.D.
     PROOF: Step 4.3 and the definition of Nil-extension imply that $v \sqsubseteq \mathit{NilExtension}(bA_a[s], P_w)$. Step 4.4, and the definition of compatible complete the proof.
5. Q.E.D.
   PROOF: All cases were considered since $r \leq s$ according to proof assumption 3.

We define a pair $\langle bA, pA \rangle$ to be *conservative* iff all v-mappings accepted by any acceptors a and b at the same round are compatible and if the v-mapping accepted by b maps

some proposer not mapped by the v-mapping accepted by a, then this proposer is mapped to the value it has proposed for that round.

Definition 3.13 (Conservative) *A pair $\langle bA, pA\rangle$ is* conservative *iff for every round r and all acceptors a and b, if $bA_a[r]$ and $bA_b[r]$ are different from $none$, then the two conditions below hold:*

- *$bA_a[r]$ and $bA_b[r]$ are compatible and*
- *$\forall p \in Dom(b) \setminus Dom(a) : bA_b[r][p] = pA_p[r]$.*

Below we present the definition of operator $ProvedSafe(Q, r, bA)$, which returns a v-mapping that is proved to be safe at round r in $\langle bA, pA\rangle$ for any proposal array pA based only on the votes of acceptors in Q, given that $\langle bA, pA\rangle$ is safe and conservative, and, for every acceptor a in Q, $\widehat{bA}_a \geq r$. In the definition below, we let $Proposer$ be the set of all proposers and $RNum$ be the set of round numbers.

Definition 3.14 (ProvedSafe) *For any round r, quorum Q, and ballot array bA, let:*

- $KS \triangleq \{i \in RNum \mid (i < r) \wedge (\exists a \in Q : bA_a[i] \neq none])\}$
- $k \triangleq Max(KS)$
- $AS \triangleq \{a \in Q : bA_a[k] \neq none\}$
- $G \triangleq \{bA_a[k] : a \in S\}$

If $KS = \{\}$, then $ProvedSafe(Q, r, bA)$ is defined to equal $\bot$; otherwise, $ProvedSafe(Q, r, bA)$ is defined to equal $NilExtension(\sqcup G, Proposer)$, where $Proposer$ is the set of all proposers.

The proposition below states that the value returned by $ProvedSafe(Q, r, bA)$ is indeed safe at r in $\langle bA, pA\rangle$ if $\langle bA, pA\rangle$ is safe and conservative and, for every acceptor a in Q, $\widehat{bA}_a \geq r$.

Proposition 3.2 *For any round r, quorum Q, ballot array bA, and proposal array pA, if*

- *$\langle bA, pA\rangle$ is safe,*

- $\langle bA, pA \rangle$ *is conservative, and*
- $\widehat{bA}_a \geq r$ *for all* $a \in Q$,

then $ProvedSafe(Q, r, bA)$ *is safe at* r *in* $\langle bA, pA \rangle$.

ASSUME: There exist round r, quorum Q, ballot array bA, and proposal array pA such that:
1. $\langle bA, pA \rangle$ is safe
2. $\langle bA, pA \rangle$ is conservative
3. $\forall a \in Q : \widehat{bA}_a \geq r$

PROVE: $ProvedSafe(Q, r, bA)$ is a v-mapping safe at r in $\langle bA, pA \rangle$

LET: KS be the set KS in the definition of $ProvedSafe$ for Q,r, and bA.

1. CASE: KS is empty
   - 1.1. No v-mapping v is choosable at a round $s < r$ in $\langle bA, pA \rangle$
     PROOF: By the definition of *choosable at*, it suffices to
     LET: P be the set $\{p \in Proposer : pA_p[s] = Nil\}$
     ASSUME: There exist v-mapping v, round $s < r$, and quorum Q_v such that $v \sqsubseteq NilExtension(bA_a[s], P)$, for every acceptor a in Q_v with $\widehat{bA}_a > s$
     PROVE: FALSE
     - 1.1.1. Choose any acceptor $a \in Q_v \cap Q$
       PROOF: Such acceptor exists because of the Quorum Requirement (Assumption 3.1).
     - 1.1.2. $\widehat{bA}_a > s$
       PROOF: Since a belongs to Q, proof assumption 3 states that $\widehat{bA}_a \geq r$, and the assumption of step 1.1 states that $r > s$. As a result, $\widehat{bA}_a > s$.
     - 1.1.3. $v \not\sqsubseteq NilExtension(bA_a[s], P)$
       PROOF: By the definition of KS, if KS is empty, then $bA_a[s]$ must equal $none$, otherwise KS would have s as an element. By definition, any Nil-extension of $none$ equals $none$ and no v-mapping is a prefix of $none$.
     - 1.1.4. Q.E.D.
       PROOF: Steps 1.1.2 and 1.1.3 and the fact that a belongs to Q_v given by step 1.1.1 contradict the assumption of step 1.1.
   - 1.2. Q.E.D.
     PROOF: By step 1.1, any v-mapping is safe at r in bA. Therefore, $\perp$, which is the value returned by $ProvedSafe(Q, r, bA)$, is safe too.
2. CASE: KS is not empty
   - 2.1. Choose round k and sets AS and G so that they satisfy the definitions of k, AS and G in the definition of $ProvedSafe$ for Q,r, and bA.
     PROOF: k exists since KS is not empty. AS and G exist because k exists.
   - 2.2. AS and G are not empty.
     PROOF: Given that k belongs to KS, there is at least one acceptor a in Q such that $bA_a[k] \neq none$.
   - 2.3. G is compatible.

PROOF: All elements of G are v-mappings accepted by acceptors in Q at round k. These v-mappings are guaranteed to be compatible because bA is assumed to be conservative (proof assumption 2).

2.4. $NilExtension(\sqcup G, Proposer)$ is safe at r in bA

PROOF: By the definition of *safe at*, it suffices to

ASSUME: There exist v-mapping w and round $s < r$ such that w is choosable at s in bA

PROVE: $w \sqsubseteq NilExtension(\sqcup G, Proposer)$

2.4.1. CASE: $s < k$

2.4.1.1. Choose $a \in AS$

PROOF: a exists by step 2.2, which states that AS is not empty.

2.4.1.2. $w \sqsubseteq bA_a[k]$

PROOF: By the definition of AS, $bA_a[k] \neq none$. Since bA is safe (proof assumption 1), any v-mapping choosable at a round lower-numbered than k, including w given that step 2.4.1 considers only the case where $s < k$, must be a prefix of $bA_a[k]$.

2.4.1.3. $bA_a[k] \sqsubseteq \sqcup G$

PROOF: By the definition of G and least upper bound, and the fact that $a \in AS$ (step 2.4.1.1).

2.4.1.4. Q.E.D.

PROOF: Steps 2.4.1.2 and 2.4.1.3, and the fact that $\sqsubseteq$ is a partial order relation over v-mappings imply that $w \sqsubseteq \sqcup G$. Moreover, by the definition of Nil-extension, $\sqcup G \sqsubseteq NilExtension(\sqcup G, Proposer)$. Since $\sqsubseteq$ is a partial order relation over v-mappings, $w \sqsubseteq NilExtension(\sqcup G, Proposer)$.

2.4.2. CASE: $s \geq k$

LET: P be the set $\{p \in Proposer : pA_p[s] = Nil\}$

2.4.2.1. Choose Q_w such that $w \sqsubseteq NilExtension(bA_a[s], P)$ for every acceptor a in Q_w with $\widehat{bA}_a > s$

PROOF: Q_w exists by the definition of choosable and the assumption of step 2.4.

2.4.2.2. Choose $a \in Q \cap Q_w$

PROOF: a exists by the Quorum Requirement.

2.4.2.3. $bA_a[s] \neq none$

PROOF: Proof assumption 3 states that $\widehat{bA}_a \geq r$ and the assumption of step 2.4 states that $r > s$; therefore $\widehat{bA}_a > s$. Steps 2.4.2.1 and 2.4.2.2 imply that $w \sqsubseteq NilExtension(bA_a[s], P)$, which is not possible if $bA_a[s] = none$.

2.4.2.4. $s = k$

PROOF: If $s > k$, and given that $s < r$ by the assumption of step 2.4, then step 2.4.2.3 above contradicts the definition of k since a belongs to Q, $s > k$ and $bA_a[s] \neq none$.

2.4.2.5. $w \sqsubseteq NilExtension(bA_a[k], P)$

PROOF: Steps 2.4.2.1, 2.4.2.2, and 2.4.2.4.

LET: $P^- \triangleq P \setminus Dom(bA_a[k])$

2.4.2.6. No v-mapping in G maps a proposer in P^- to a value different from Nil

PROOF: Assume, for the sake of contradiction, that there is a v-mapping in G that maps an element p of P^- to something different from Nil. Since $\langle bA, pA\rangle$ is conservative and $p \notin Dom(bA_a[k])$, $pA_p[k]$ must equal the mapped value, which contradicts the definition of P.

2.4.2.7. $NilExtension(bA_a[k], P) \sqsubseteq NilExtension(\sqcup G, Proposer)$

PROOF: Steps 2.4.2.3 and 2.4.2.4 imply that $a \in AS$ and, therefore, $bA_a[k] \in G$. The definition of a Nil-extension and step 2.4.2.6 complete the proof.

2.4.2.8. Q.E.D.

PROOF: By steps 2.4.2.5 and 2.4.2.7, and the transitivity of $\sqsubseteq$.

2.4.3. Q.E.D.

2.5. Q.E.D.

PROOF: Directly, since $NilExtension(\sqcup G, Proposer)$ is the value returned by $ProvedSafe(Q, r, bA)$ in case KS is not empty.

3. Q.E.D.

PROOF: Since KS is defined to be a set, all cases are being covered.

3.6.2 Abstract Collision-fast Paxos

We structure our proof of correctness as a series of refinement mappings [AL91]. In this section, we present an abstract algorithm that can be more easily proved correct. We then refine this algorithm in the following sections until we obtain the algorithm presented in Section 3.4.1. We are initially concerned with Safety properties only. The liveness of Collision-fast Paxos is proved in Section 3.6.5.

Our initial abstraction is based upon the following variables:

learned An array of v-mappings, where $learned[l]$ is the v-mapping currently learned by learner l. Initially, $learned[l] = \bot$ for all learners l.

proposed The set of proposed values. It initially equals the empty set.

bA A ballot array. It represents the current state of the voting. Initially, $\widehat{bA}_a = 0$ and $bA_a[r] = none$ for every acceptor a and round r.

pA A proposal array. It represents the proposal history. Initially, $pA_p[r] = none$ for every proposer p and round r.

$minTried$ An array of v-mappings, where $minTried[r]$ is either a v-mapping or equal to $none$, for every round r. Initially, $minTried[0] = \perp$ and $maxTried[r] = none$ for all $r > 0$.

The Abstract Collision-fast Paxos algorithm satisfies the following invariants, which, as we prove next, imply the properties Nontriviality and Consistency of M-Consensus.

$minTried$ **Invariant** For every round r, if $minTried[r] \neq none$, then

1. $minTried[r]$ is proposed.
2. $minTried[r]$ is safe at r in $\langle bA, pA \rangle$.
3. If $minTried[r] \neq \perp$, then $minTried[r]$ is valued and complete.

bA **Invariant** For all acceptors a and rounds r, if $bA_a[r] \neq none$, then

1. $minTried[r] \sqsubseteq bA_a[r]$.
2. $bA_a[r]$ is a valued and proposed v-mapping.
3. If $minTried[r] = \perp$ then $\forall p \in Dom(bA_a[r]) : bA_a[r](p) = pA_p[r]$; otherwise, $bA_a[r] = minTried[r]$.

pA **Invariant** For all proposers p and rounds r, if $pA_p[r] \neq none$, then $pA_p[r]$ is either Nil or a proposed value.

$learned$ **Invariant** For every learner l:

1. $learned[l]$ is a nontrivial proposed v-mapping.
2. $learned[l]$ is the lub of a finite set of v-mappings chosen in $\langle bA, pA \rangle$.

Proposition 3.3 *The* learned *invariant implies the Nontriviality property of M-Consensus.*

PROOF: By part 1 of the $learned$ invariant.

Proposition 3.4 *Invariants* minTried*,* bA*, and* learned *imply the Consistency property of M-Consensus.*

PROOF: By the definition of Consistency, it suffices to assume that invariants bA and $learned$ are true, and prove that, for every pair of learners l_1 and l_2, $learned[l_1]$ and $learned[l_2]$ are compatible. The proof is divided into four steps, presented below:
1. $\langle bA, pA \rangle$ is safe.

PROOF: This follows from part 1 of the bA invariant, part 2 of the *minTried* invariant, the fact that the extension of a safe v-mapping is also safe by definition, and the definition of a safe ballot array (Definition 3.12).

LET: $S = \{v : v \text{ is chosen in } \langle bA, pA\rangle\}$

2. S is compatible.
   PROOF: By step 1 and Proposition 3.1.
3. For every learner l, $learned[l] \sqsubseteq \sqcup S$.
   PROOF: This is true by part 2 of the $learned$ invariant and the definition of least upper bound, which implies that if set S is compatible, then the lub of S is equal to or extends the lub of any subset of S.
4. Q.E.D.
   PROOF: By step 3 and the definition of compatible c-structs.

Abstract Collision-fast Paxos has six atomic actions, described below. A complete specification of the algorithm in TLA$^+$ is given in Section 3.8.

$Propose(V)$, for any value V. It is enabled iff $V \notin proposed$ and sets $proposed$ to $proposed \cup \{V\}$.

$JoinRound(a, r)$, for any acceptor a and round r. It is enabled iff $\widehat{bA}_a < r$ and sets $\widehat{bA}_a$ to r.

$StartRound(r, Q)$, for any round r and quorum Q of acceptors. It is enabled iff

- $minTried[r] = none$ and
- $\forall a \in Q : r \leq \widehat{bA}_a$.

It sets $minTried[r]$ to $ProvedSafe(Q, r, bA)$.

$Suggest(p, r, V)$, for proposer p, round r, and (possibly Nil) value V, where p is a collision-fast proposer of r. It is enabled iff

- $pA_p[r] = none$ and
- either (i)$minTried[r] \notin \{\bot, none\}$ and $V = minTried[r](p)$, (ii) V is a proposed value, or (iii) $V = Nil$ and there is a collision-fast proposer q of r such that $pA_q[r] \notin \{Nil, none\}$.

It sets $pA_p[r]$ to V.

$ClassicVote(a, r, v)$, for acceptor a, round r, and v-mapping v. Let P^- be the subset of collision-fast proposers of r such that $p \in P \iff pA_p[r] = none$, and let $MaxT$ equal the v-mapping that maps each proposer q in $Proposer \setminus P^-$ to $pA_q[r]$ if q is collision-fast for r or to Nil otherwise. This action is enabled iff

- $\widehat{bA}_a \leq r$,

- v is a valued v-mapping,
- $minTried[r] \neq none$,
- $v \sqsubseteq MaxT$, if $minTried[r] = \bot$, or $v = minTried[r]$, othewise, and
- either $bA_a[r] = none$ or $bA_a[r] \sqsubset v$.

It sets $\widehat{bA}_a$ to r and $bA_a[r]$ to v.

$AbstractLearn(l, v)$, for any learner l and v-mapping v. It is elabled iff v is chosen in $\langle bA, pA \rangle$ and sets $learned[l]$ to $learned[l] \sqcup v$.

The following proposition proves that the algorithm also satisfies the Stability property of M-Consensus.

Proposition 3.5 *Abstract Collision-fast Paxos satisfies the Stability property of M-Consensus.*

PROOF: For any learner l, the only action that changes the value of $learned[l]$ is $AbstractLearn(l, v)$. Since, by the definition of lub, this action can only extend the value of $learned[l]$, Stability is ensured.

It is easy to verify that the algorithm's actions keep the type invariant of the variables it uses. The most complicated case concerns the ballot array bA, updated by actions $JoinRound(a, r)$ and $ClassicVote(a, r, v)$. However, action $JoinRound(a, r)$ only increases the value of $\widehat{bA}_a$ and action $ClassicVote(a, r, v)$ sets $bA_a[r]$ to v, where r always equals $\widehat{bA}_a$ after the action is executed. These changes to bA keep it a ballot array according to the definition.

It remains to prove that the abstract algorithm satisfies the invariants $minTried$, bA, pA, and $learned$. For the sake of simplicity, however, we use some extra notation in the proof. When analyzing the execution of an action, we use ordinary expressions such as exp to represent the value of that expression before the action is executed, and we let exp' be the value of that expression after the action execution.

Proposition 3.6 *Abstract Collision-fast Paxos satisfies the invariants* minTried*,* bA*,* pA*, and* learned*.*

PROOF: The invariants are trivially satisfied in the initial state. Therefore, it suffices to assume that the invariants are true and prove that, for every action α, they remain true if α is executed. We do that in the following, analyzing case by case.

1. CASE: Action $Propose(V)$ is executed, where V is a non-Nil value.

PROOF SKETCH: Action $Propose(V)$ only changes variable $proposed$, which is the set of proposed values, and does that by adding a new element to it. Invariant conditions that do not refer to this set are obviously preserved. The others are kept true since the set $proposed$ only increases and v-mappings composed of proposed values remain composed of proposed values.

2. CASE: Action $JoinRound(a, r)$ is executed, where a is an acceptor and r is a round number.
PROOF SKETCH: Action $JoinRound(a, r)$ only changes $\widehat{bA}_a$, setting it to r, which is bigger than $\widehat{bA}_a$. Invariant conditions that do not refer to $\widehat{bA}_a$ are obviously preserved. It remains to check that safe or chosen v-mappings in $\langle bA, pA\rangle$ are kept safe or chosen in $\langle bA', pA'\rangle$. The definition of *chosen* does not involve $\widehat{bA}_e$ for any acceptor e. The definition of *safe* is based upon the definition of *choosable at*, which does refer to $\widehat{bA}_e$, but implies that a v-mapping w that is choosable at round k in $\langle bA', pA'\rangle$ is also choosable at k in $\langle bA, pA\rangle$. By the definition of *safe*, this implies that a value x that is safe at round s in $\langle bA, pA\rangle$ is also safe at s in $\langle bA', pA'\rangle$.

3. CASE: Action $StartRound(r, Q)$ is executed, where r is a round and Q is a quorum of acceptors.
PROOF SKETCH: Action $StartRound(r, Q)$ changes $minTried[r]$ from $none$ to $ProvedSafe(Q, r, bA)$. The action does not change the other variables.The bA invariant implies that the pair $\langle bA, pA\rangle$ is conservative, which ensures that $ProvedSafe(Q, r, bA)$ is safe at r in $\langle bA, pA\rangle$. Moreover, the bA invariant states that all accepted v-mappings are proposed and valued, which guarantees that $ProvedSafe(Q, r, bA)$ is either $\perp$ or a proposed, valued, and complete v-mapping by the definition of $ProvedSafe(Q, r, bA)$. All these things imply that the action preserves the $minTried$ invariant. It preserves the bA invariant because it does not change bA and $bA_e[r]$ is ensured to equal $none$, for any acceptor e, by the bA invariant itself. The other invariants are obviously preserved because the variables they refer to do not change.

4. CASE: Action $Suggest(p, r, V)$ is executed, where p is a collision-fast proposer for round r and V is either a proposed value or Nil.
PROOF SKETCH: This action only changes $pA_p[r]$ from $none$ to V and clearly keeps all invariants.

5. CASE: Action $ClassicVote(a, r, v)$ is executed, where a is an acceptor, r is a round number, and v is a v-mapping.
PROOF SKETCH: The definition of *choosable at* implies that if a value is choosable at round s in $\langle bA', pA'\rangle$, then it is chossable at s in $\langle bA, pA\rangle$. This fact, by the definition of *safe at*, implies that a value that is safe at round s in $\langle bA, pA\rangle$ is necessarily safe at s in $\langle bA', pA'\rangle$. This, together with the fact that no variable but bA is updated by this action, implies that the action preserves invariants $minTried$, pA, and $learned$. As for the bA invariant, there are two cases to consider. If $minTried[r] \neq \perp$, then this action sets

$bA_a[r]$ to $minTried[r]$, which is ensured to be valued, safe, and complete. In this case, the bA invariant is clearly preserved. Now, let us assume $minTried[r] = \bot$ and check if the action preserves the bA invariant with respect to $bA_a[r]$, which is the only entry of bA changed in this action. Condition 1 of the bA invariant is trivially true because $minTried[r] = \bot$. Condition 2 is true because $MaxT$ is proposed by the pA invariant and v is ensured to be valued by the action's pre-condition. Condition 3 is ensured by the definition of $MaxT$.

6. CASE: Action $AbstractLearn(l, v)$ is executed, where l is a learner and v is a v-mapping. PROOF SKETCH: Action $AbstractLearn(l, v)$ only changes variable $learned$, which is the array of learned v-mappings, and does that by extending one entry to the lub of it with a chosen v-mapping. Invariants $maxTried$ and bA are obviously preserved. The first part of the $learned$ invariant is preserved because of the pA invariant and the fact that v-mappings accepted by acceptors are both proposed and valued. The second part is obviously preserved.

3.6.3 Distributed Abstract Collision-fast Paxos

As an intermediate step in our proof, we introduce a distributed version of the abstract algorithm in the previous section. This algorithm is based on the same variables as the previous algorithm plus a variable $msgs$, used to simulate a message passing system by holding the messages sent between agents. Variable initialization is done as before for the common variables, and $msgs$ is set to $\{\langle \text{“}\texttt{2S}\text{”}, 0, \bot\rangle\}$ initially, which implies that a $2S$ message for round 0 is implicitly sent when the algorithm starts. Message duplication is implemented by never taking messages out of set $msgs$, which would permanently enable actions that depend on an existing message. Since we are proving only safety, we do not have to implement the loss of messages because a message loss would only imply that some actions would not be executed.

The distributed abstract algorithm is described in terms of the following actions. Its formal specification in TLA$^+$ is given in the appendix section 3.8.

Propose(V)**,** for any value V. It is enabled iff $V \notin proposed$. It sets $proposed$ to $proposed \cup \{V\}$ and adds message $\langle \text{“}\texttt{propose}\text{”}, V\rangle$ to $msgs$.

Phase$1a(c, r)$, executed by coordinator c, for round r. The action is enabled iff $minTried[r] = none$. It sends the message $\langle \text{“}\texttt{1a}\text{”}, r\rangle$ to acceptors (adds it to $msgs$).

Phase$1b(a, r)$, executed by acceptor a, for round r. The action is enabled iff

- $\widehat{bA}_a < r$ and
- $\langle \text{“}\texttt{1a}\text{”}, r\rangle \in msgs$

It sets $\widehat{bA}_a$ to r and adds message $\langle$"1b", r, a, $bA_a\rangle$ to $msgs$.

$Phase2Start(r)$, for round r. The action is enabled iff:

- $minTried[r] = none$ and
- There exists a quorum Q such that for all $a \in Q$, there is a message $\langle$"1b", r, a, $\rho\rangle$ in $msgs$, for some ρ.

Let $v = ProvedSafe(Q, r, \beta)$, where β is any ballot array such that, for every acceptor a in Q, $\widehat{\beta}_a = r$ and there exists message $\langle$"1b", r, a, $\rho\rangle$ in $msgs$ with $\rho = \beta_a$. This action sets $minTried[r]$ to v and adds message $\langle$"2S", r, $v\rangle$ to $msgs$.

$Phase2Prepare(p, r)$, executed by proposer p, for round r. It is enabled iff:

- $pA_p[r] = none$ and
- There exists message $\langle$"2S", r, $v\rangle$ in $msgs$ with $v \neq \bot$

It sets $pA_p[r]$ to $v(p)$.

$Phase2a(p, r, V)$, executed by proposer p, for round r and (possibly Nil) value V. The action is enabled iff:

- p is a collision-fast proposer of r,
- $pA_p[r] = none$,
- $\langle$"2S", r, $\bot\rangle \in msgs$, and
- either $\langle$"propose", $V\rangle \in msgs$ or $V = Nil$ and there exists a message $\langle$"2a", r, $\langle q, U\rangle\rangle$ in $msgs$ with $U \neq Nil$.

This action sets $pA_p[r]$ to V and adds message $\langle$"2a", r, $\langle p, V\rangle\rangle$ to $msgs$.

$Phase2b(a, r, v)$, executed by acceptor a, for round r and v-mapping v. It is enabled iff $\widehat{bA}_a \leq r$ and either one of the following conditions hold:

a) $bA_a[r] = none$ and message $\langle$"2S", r, $v\rangle$ exists in $msgs$, where $v \neq Nil$, or

b) message $\langle$"2a", r, $\langle p, V\rangle\rangle$ exists in $msgs$, where $V \neq Nil$, and either one of the two following conditions hold:

   b1) $bA_a[r] = none$ and $v = NilExtension(\bot \bullet \langle p, V\rangle, P)$, where P is the set of all proposers that are not collision-fast for round r, or

   b2) $bA_a[r] \neq none$ and $v = bA_a[r] \bullet \langle p, V\rangle$.

The action sets $\widehat{bA}_a$ to r and $bA_a[r]$ to v, and adds message $\langle$"2b", r, a, $v\rangle$ to $msgs$.

$Learn(l, v)$, executed by learner l, for v-mapping v. It is enabled iff there exist round r, quorum Q, and set P of collision-fast proposers for r such that the two conditions below hold:

- $\forall p \in P : \langle$"2a", r, $\langle p, Nil\rangle\rangle \in msgs$ and

- $\forall a \in Q : \langle \text{“2b”}, r, a, u \rangle \in msgs$, where $v \sqsubseteq u$.

It sets $learned[l]$ to $learned[l] \sqcup v$.

The distributed abstract algorithm implements the the non-distributed version in the sense that all behaviors of the former are also behaviors of the latter.

Proposition 3.7 *Distributed Abstract Collision-fast Paxos implements the Abstract Collision-fast Paxos specification.*

PROOF SKETCH: The initial state of both algorithms with respect to their shared variables is exactly the same. As a result, to prove this proposition we must only show that every action in the distributed algorithm implements an action of the non-distributed algorithm with respect to the variable states before and after the action is taken [AL91]. In the following we analyze each action of the distributed version.

$Propose(V)$ This action clearly implements the action with the same name in the non-distributed algorithm. The only difference has to do with variable $msgs$ which is not present in the non-distributed version.

$Phase1a(c, r)$ This action changes only variable $msgs$, and implements a no-action (stuttering) step in the non-distributed algorithm, since it keeps the rest of the state the same as before.

$Phase1b(a, r)$ This action clearly implements action $JoinRound$ of the non-distributed algorithm. It is more restrictive, though, since it requires a “1b” message for r to be present in $msgs$.

$Phase2Start(r)$ This action implements action $StartRound$ of the non-distributed algorithm. Let Q be the quorum of action $Phase2Start$; the reception of the “1b” messages for round r coming from acceptors a in Q implies that every acceptor $a \in Q$ has set $\widehat{bA}_a$ to r. Since $\widehat{bA}_a$ is never decreased, we can conclude that $\widehat{bA}_a \geq r$ for every acceptor a in Q, as required by action $StartRound$. By the definition of $ProvedSafe$ and the fact that the vectors sent in the “1b” messages are consistent with the current state of bA, one can easily verify that $ProvedSafe(Q, r, \beta)$ returns exactly the same value as $ProvedSafe(Q, r, bA)$.

$Phase2Prepare(p, r)$ This action implements $Suggest(p, r, V)$ when $minTried[r]$ is different from $\bot$ (identified by the “2S” message) and $V = minTried[r](p)$.

$Phase2a(p, r, V)$ This action implements $Suggest(p, r, V)$ when $minTried[r]$ equals $\bot$ (identified by the received “2S” message). Notice that V is either a proposed value (received in a “propose” message) or it equals Nil but other collision-fast proposer q for round r has set $pA_q[r]$ to a non-Nil value, a situation identified by a “2a” message.

$Phase2b(a, r, v)$ This action implements $ClassicVote(a, r, v)$. There are three cases to consider:

- $bA_a[r] = none$ and message $\langle$"2S", $r, v\rangle$ ($v \neq Nil$) exists in $msgs$, where $v \neq Nil$.

  In this case, $minTried[r] = v$ and the implementation of $ClassicVote(a, r, v)$ is easily verified.

- message $\langle$"2a", $r, \langle p, V\rangle\rangle$ ($V \neq Nil$) exists in $msgs$, $bA_a[r] = none$, and $v = NilExtension(\bot \bullet \langle p, V\rangle, P)$, where P is the set of all proposers that are not collision-fast for round r.

  In this case, since a "2a" message was sent and it is only sent by a collision-fast proposer of r when $minTried[r] = \bot$, we can infer that p is a collision-fast proposer of r and $minTried[r] = \bot$. By the definition of $MaxT$ in $ClassicVote$, it is easy to see that v satisfies the pre-condition of this action. The rest of the action implementation is easily checked.

- message $\langle$"2a", $r, \langle p, V\rangle\rangle$ ($V \neq Nil$) exists in $msgs$, $bA_a[r] \neq none$, and $v = bA_a[r] \bullet \langle p, V\rangle$.

  Once again, the existence of a "2a" message for round r implies that p is collision-fast for r and $minTried[r] = \bot$. Notice that no value $pA_p[r]$ can change after it is set to something different from $none$ and the same happens with $minTried[r]$. Since entry $bA_a[r]$ is only changed by a $Phase2b(a, r, v)$ action, by the definition of $MaxT$ it is easy to see that $bA_a[r] \sqsubseteq MaxT$. Given that $v = bA_a[r] \bullet \langle p, V\rangle$, it also follows from the definition of $MaxT$ that $v \sqsubseteq MaxT$.

$Learn(l, v)$ This action implements action $AbstractLearn(l, v)$ by the definition of a chosen v-mapping and the fact that "2a" and "2b" messages reflect stable changes, in the sense that no further changes can happen, to entries in pA and bA respectively.

3.6.4 Collision-fast Paxos

To prove correctness of algorithm presented in Section 3.4.1, we first add the following history variables to the algorithm presented in the previous section.

$prnd$: An array of round numbers, where $prnd[p]$ represents the current round of proposer p. Initially 0.

$pval$: An array of v-mappings, where $pval[p]$ represents the v-mapping fast-proposed by proposer p on round $prnd[p]$ or $none$, if p has not fast-proposed in that round. Initially $none$.

$crnd$: An array of round numbers, where $crnd[c]$ represents the current round of coordinator c. Initially 0.

$cval$: An array of v-mappings, where $cval[c]$ represents the latest v-mapping sent by coordinator c in a phase "2S" message for round $crnd[c]$. Initially $\perp$ for the coordinator of round 0 and $none$ for all the others.

rnd : An array of round numbers, where $rnd[a]$ is the current round of acceptor a, that is, the highest-numbered round a has heard of. Initially 0.

$vrnd$: An array of round numbers, where $vrnd[a]$ is the round at which acceptor a has accepted the latest v-mapping. Initially 0.

$vval$: An array of v-mappings, where $vval[a]$ is the v-mapping acceptor a has accepted at $vrnd[a]$ or $none$. Initially $none$.

$msgs2$: Counterparts of the messages sent by the original protocol, but built with the values of history variables. Initially $\{\langle$"2S", $0, \perp\rangle\}$.

$Propose(V)$, for any value V. It is enabled iff $V \notin proposed$. It sets $proposed$ to $proposed \cup \{V\}$ and adds message $\langle$"propose", $V\rangle$ to $msgs$ and $msgs2$.

$Phase1a(c, r)$, executed by coordinator c, for round r. The action is enabled iff $minTried[r] = none$. It sets $crnd[c]$ to r and $cval[c]$ to $none$, and adds a message $\langle$"1a", $c, m\rangle$ to $msgs$ and $msgs2$.

$Phase1b(a, r)$, executed by acceptor a, for round r. The action is enabled iff

- $\widehat{bA}_a < r$ and
- $\langle$"1a", $r\rangle \in msgs$

It sets $\widehat{bA}_a$ to r and $rnd[a]$ to r and adds the message $\langle$"1b", r, a, $bA_a\rangle$ to $msgs$ and "1b"r,a,vrnd[a],vval[a] to $msgs2$.

$Phase2Start(r)$, executed by the coordinator c of round r, for round r. The action is enabled iff:

- $minTried[r] = none$ and
- There exists a quorum Q such that for all $a \in Q$, there is a message $\langle$"1b", r, a, $\rho\rangle$ in $msgs$, for some ρ.

Let $v = ProvedSafe(Q, r, \beta)$, where β is any ballot array such that, for every acceptor a in Q, $\widehat{\beta}_a = r$ and there exists message $\langle$"1b", $r, a, \rho\rangle$ in $msgs$ with $\rho = \beta_a$. This action sets $minTried[r]$ and $cval[c]$ to v, $crnd[c]$ to r, and adds message $\langle$"2S", $r, v\rangle$ to $msgs$ and $msgs2$.

$Phase2Prepare(p, r)$, executed by proposer p, for round r. It is enabled iff:

- $pA_p[r] = none$ and
- There exists message $\langle$"2S", $r, v\rangle$ in $msgs$

If $v \neq \bot$, it sets $pA_p[r]$ and $pval[p]$ to $v(p)$, and $prnd[p]$ to r. Otherwise, if v equals $\bot$, $pval[p]$ is set to $none$ and $prnd[p]$ to r.

$Phase2a(p, r, V)$, executed by proposer p, for round r and (possibly Nil) value V. The action is enabled iff:

- p is a collision-fast proposer of r,
- $pA_p[r] = none$,
- $\langle$"2S", $r, \bot\rangle \in msgs$, and
- either $\langle$"propose", $V\rangle \in msgs$ or $V = Nil$ and there exists a message $\langle$"2a", $r, \langle q, U\rangle\rangle$ in $msgs$ with $U \neq Nil$.

This action sets $pA_p[r]$ and $pval[p]$ to V and adds message $\langle$"2a", $r, \langle p, V\rangle\rangle$ to $msgs$ and $msgs2$.

$Phase2b(a, r, v)$, executed by acceptor a, for round r and v-mapping v. It is enabled iff $\widehat{bA}_a \leq r$ and either one of the following conditions hold:

a) $bA_a[r] = none$ and message $\langle$"2S", $r, v\rangle$ exists in $msgs$, where $v \neq Nil$, or

b) message $\langle$"2a", $r, \langle p, V\rangle\rangle$ exists in $msgs$, where $V \neq Nil$, and either one of the two following conditions hold:

b1) $bA_a[r] = none$ and $v = NilExtension(\bot \bullet \langle p, V\rangle, P)$, where P is the set of all proposers that are not collision-fast for round r, or

b2) $bA_a[r] \neq none$ and $v = bA_a[r] \bullet \langle p, V\rangle$.

The action sets $\widehat{bA}_a$, $rnd[a]$, and $vrnd[a]$ to r, $bA_a[r]$ and $vval[a]$ to v, and adds message $\langle$"2b", $r, a, v\rangle$ to $msgs$ and $msgs2$.

$Learn(l, v)$, executed by learner l, for v-mapping v. It is enabled iff there exist round r, quorum Q, and set P of collision-fast proposers for r such that the two conditions below hold:

- $\forall p \in P : \langle$"2a", $r, \langle p, Nil\rangle\rangle \in msgs$ and
- $\forall a \in Q : \langle$"2b", $r, a, u\rangle \in msgs$, where $v \sqsubseteq u$.

It sets $learned[l]$ to $learned[l] \sqcup v$.

Variables $prnd$, $pval$, $crnd$, $cval$, rnd, $vrnd$, $vval$, and $msgs2$ appear in no pre-condition and, therefore, are clearly history variables satisfying conditions H1-5 of [AL91]. This implies that the resulting algorithm is equivalent to (i.e., accepts the same behaviors as) the previous one without such variables. The following invariants can be easily proved for this new algorithm:

InvDA1: $crnd[c] = k \Rightarrow \forall j > k : c$ is coordinator of $j : minTried[j] = none$

InvDA2: $minTried[crnd[c]] = cval[c]$

InvDA3: $rnd[a] = \widehat{bA}_a$

InvDA4: $vrnd[a] = k \iff \begin{array}{ll} \wedge & bA_a[k] \neq none \\ \wedge & \forall j > k : bA_a[j] = none \end{array}$

InvDA5: $vval[a] = bA_a[vrnd[a]]$

InvDA6: $prnd[p] = k \Rightarrow \forall j > k : pA_p[j] = none$

InvDA6.5: $pval[p] = pA_p[prnd[p]]$

InvDA7: $\langle$"1a", $m\rangle \in msgs \iff \langle$"1a", $m\rangle \in msgs2$

InvDA8: $\langle$"1b", $m, \rho\rangle \in msgs \iff \langle$"1b", $m, vval, vrnd\rangle \in msgs2$, where $vrnd$ is the highest balnum k such that $\rho[k] \neq none$ and $vval$ equals $\rho[vrnd]$.

InvDA9: $\langle$"2S", $m, v\rangle \in msgs \iff \langle$"2S", $m, v\rangle \in msgs2$

InvDA10: $\langle$"2a", $m, v\rangle \in msgs \iff \langle$"2a", $m, v\rangle \in msgs2$

InvDA11: $\langle$"2b", $m, v\rangle \in msgs \iff \langle$"2b", $m, v\rangle \in msgs2$

We can use these invariants to rewrite the pre-conditions of the previous algorithm's actions in the following way:

$Propose(V)$, for any value V. It is enabled iff $V \notin proposed$. It sets $proposed$ to $proposed \cup \{V\}$ and adds message $\langle$"propose", $V\rangle$ to $msgs$ and $msgs2$.

The action remains the same.

$Phase1a(c, r)$, executed by coordinator c, for round r. The action is enabled iff

- c is the coordinator of round r and
- $crnd[c] \leq r$.

It sets $crnd[c]$ to r and $cval[c]$ to $none$, and adds a message $\langle$"1a", $c, m\rangle$ to $msgs$ and $msgs2$.

By invariant InvDA1.

$Phase1b(a, r)$, executed by acceptor a, for round r. The action is enabled iff

- $rnd[a] < r$ and
- $\langle$"1a", $r\rangle \in msgs2$

It sets $\widehat{bA}_a$ and $rnd[a]$ to r, and adds the message $\langle$"1b", r, a, $bA_a\rangle$ to $msgs$ and "1b"r,a,vrnd[a],vval[a] to $msgs2$.

By invariants InvDA3 and InvDA7.

$Phase2Start(r)$, executed by the coordinator c of round r, for round r. The action is enabled iff:

- $crnd[c] = r$
- $cval[c] = none$ and
- There exists a quorum Q such that for all $a \in Q$, there is a message $\langle$"1b", r, a, $vval$, $vrnd\rangle$ in $msgs2$.

Let $v = ProvedSafe(Q, r, \beta)$, where β is any ballot array such that, for every acceptor a in Q, $\widehat{\beta}_a = r$ and there exists message $\langle$"1b", r, a, $vrnd$, $vval\rangle$ in $msgs2$ with $\beta_a vrnd = vval$ and $\beta_a or = none$ for any round $or \neq vrnd$. This action sets $minTried[r]$ and $cval[c]$ to v, $crnd[c]$ to r, and adds message $\langle$"2S", r, $v\rangle$ to $msgs$ and $msgs2$.

By invariants InvDA2 and InvDA8. $ProvedSafe(Q, r, \beta)$ still gives the expected result because it only uses the latest values accepted by each acceptor, the only value in β.

$Phase2Prepare(p, r)$, executed by proposer p, for round r. It is enabled iff:

- $prnd[p] \leq r$ and
- There exists message $\langle$"2S", r, $v\rangle$ in $msgs2$

If $v \neq \bot$, it sets $pA_p[r]$ and $pval[p]$ to $v(p)$, and $prnd[p]$ to r. Otherwise, if v equals $\bot$, $pval[p]$ is set to $none$ and $prnd[p]$ to r.

By invariants InvDA6, Inv6.5, and InvDA9. If $v \neq \bot$, then this action implements its previous version. Otherwise, it implements a stuttering step of its previous specification.

$Phase2a(p, r, V)$, executed by proposer p, for round r and (possibly Nil) value V. The action is enabled iff:

- $prnd[p] = r$
- p is a collision-fast proposer of r,
- $pval[r] = none$,
- $\langle$"2S", r, $\bot\rangle \in msgs2$, and
- either $\langle$"propose", $V\rangle \in msgs2$ or $V = Nil$ and there exists a message $\langle$"2a", r, $\langle q, U\rangle\rangle$ in $msgs2$ with $U \neq Nil$.

This action sets $pA_p[r]$ and $pval[p]$ to V and adds message $\langle$"2a", r, $\langle p, V\rangle\rangle$ to $msgs$ and $msgs2$.

By invariants InvDA6 and InvDA6.5. Since it is more restrictive than its previous version (requires the prior action to execute), and the other pre-conditions are equivalent, the action implements its previous version.

$Phase2b(a, r, v)$, executed by acceptor a, for round r and v-mapping v. It is enabled iff $rnda \leq r$ and either one of the following conditions hold:

a) $vrnd[a] < r \vee vval[a] = none$ and message $\langle$"2S", $r, v\rangle$ exists in $msgs2$, where $v \neq Nil$, or

b) message $\langle$"2a", $r, \langle p, V\rangle\rangle$ exists in $msgs2$, where $V \neq Nil$, and either one of the two following conditions hold:

b1) $vrnd[a] < r \vee vval[a] = none$ and $v = NilExtension(\bot \bullet \langle p, V\rangle, P)$, where P is the set of all proposers that are not collision-fast for round r, or

b2) $vrnd[a] = r \wedge vval[a] \neq none$ and $v = vval[a] \bullet \langle p, V\rangle$.

The action sets $\widehat{bA}_a$, $rnd[a]$, and $vrnd[a]$ to r, $bA_a[r]$ and $vval[a]$ to v, and adds message $\langle$"2b", $r, a, v\rangle$ to $msgs$ and $msgs2$.

By invariants InvDA4, InvDA5, InvDA9, InvDA10, and because $rnd[a] \geq vrnd[a]$ (By InvDA3, InvDA4 and the definition of ballot array).

$Learn(l, v)$, executed by learner l, for v-mapping v. It is enabled iff there exist round r, quorum Q, and set P of collision-fast proposers for r such that the two conditions below hold:

- $\forall p \in P : \langle$"2a", $r, \langle p, Nil\rangle\rangle \in msgs2$ and
- $\forall a \in Q : \langle$"2b", $r, a, u\rangle \in msgs2$, where $v \sqsubseteq u$.

It sets $learned[l]$ to $learned[l] \sqcup v$.

By invariant InvDA11.

The resulting algorithm now has variables bA, pA, $minTried$ and $msgs$ as history variables, since they do not appear on any action's pre-condition and are only updated. This algorithm is, therefore, equivalent to one that does not contain such variables, which we present below.

$Propose(V)$, for any value V. It is enabled iff $V \notin proposed$. It sets $proposed$ to $proposed \cup \{V\}$ and adds message $\langle$"propose", $V\rangle$ to $msgs2$.

$Phase1a(c, r)$, executed by coordinator c, for round r. The action is enabled iff

- c is the coordinator of round r and
- $crnd[c] \leq r$.

It sets $crnd[c]$ to r and $cval[c]$ to $none$, and adds a message $\langle$"1a", $c, m\rangle$ to $msgs2$.

$Phase1b(a, r)$, executed by acceptor a, for round r. The action is enabled iff

- $rnd[a] < r$ and

- $\langle$"1a", $r\rangle \in msgs2$

It sets $rnd[a]$ to r and adds the message "1b"r,a,vrnd[a],vval[a] to $msgs2$.

$Phase2Start(r)$, executed by the coordinator c of round r, for round r. The action is enabled iff:

- $crnd[c] = r$
- $cval[c] = none$ and
- There exists a quorum Q such that for all $a \in Q$, there is a message $\langle$"1b", r, a, $vval$, $vrnd\rangle$ in $msgs2$.

Let $v = ProvedSafe(Q, r, \beta)$, where β is any ballot array such that, for every acceptor a in Q, $\widehat{\beta}_a = r$ and there exists message $\langle$"1b", r, a, $vrnd$, $vval\rangle$ in $msgs2$ with $\beta_a vrnd = vval$ and $\beta_a or = none$ for any round $or \neq vrnd$. This action sets $cval[c]$ to v, $crnd[c]$ to r, and adds message $\langle$"2S", r, $v\rangle$ to $msgs2$.

$Phase2Prepare(p, r)$, executed by proposer p, for round r. It is enabled iff:

- $prnd[p] \leq r$ and
- There exists message $\langle$"2S", r, $v\rangle$ in $msgs2$

If $v \neq \bot$, it sets $pval[p]$ to $v(p)$, and $prnd[p]$ to r. Otherwise, if v equals $\bot$, $pval[p]$ is set to $none$ and $prnd[p]$ to r.

$Phase2a(p, r, V)$, executed by proposer p, for round r and (possibly Nil) value V. The action is enabled iff:

- $prnd[p] = r$
- p is a collision-fast proposer of r,
- $pval[r] = none$,
- $\langle$"2S", $r, \bot\rangle \in msgs2$, and
- either $\langle$"propose", $V\rangle \in msgs2$ or $V = Nil$ and there exists a message $\langle$"2a", $r, \langle q, U\rangle\rangle$ in $msgs2$ with $U \neq Nil$.

This action sets $pval[p]$ to V and adds message $\langle$"2a", $r, \langle p, V\rangle\rangle$ to $msgs2$.

$Phase2b(a, r, v)$, executed by acceptor a, for round r and v-mapping v. It is enabled iff $rnda \leq r$ and either one of the following conditions hold:

a) $vrnd[a] < r \vee vval[a] = none$ and message $\langle$"2S", r, $v\rangle$ exists in $msgs2$, where $v \neq Nil$, or

b) message $\langle$"2a", $r, \langle p, V\rangle\rangle$ exists in $msgs2$, where $V \neq Nil$, and either one of the two following conditions hold:

   b1) $vrnd[a] < r \vee vval[a] = none$ and $v = NilExtension(\bot \bullet \langle p, V\rangle, P)$, where P is the set of all proposers that are not collision-fast for round r, or

b2) $vrnd[a] = r \wedge vval[a] \neq none$ and $v = vval[a] \bullet \langle p, V \rangle$.

The action sets $rnd[a]$ and $vrnd[a]$ to r, $vval[a]$ to v, and adds message $\langle$"2b", r, a, $v\rangle$ to $msgs2$.

$Learn(l, v)$, executed by learner l, for v-mapping v. It is enabled iff there exist round r, quorum Q, and set P of collision-fast proposers for r such that the two conditions below hold:

- $\forall p \in P : \langle$"2a", $r, \langle p, Nil\rangle\rangle \in msgs2$ and
- $\forall a \in Q : \langle$"2b", $r, a, u\rangle \in msgs2$, where $v \sqsubseteq u$.

It sets $learned[l]$ to $learned[l] \sqcup v$.

The algorithm presented in Section 3.4.1 is a stricter implementation of the algorithm above, which can be easily verified by simply comparing their actions. This concludes the proof that Collision-fast Paxos satisfies the safety requirements of M-Consensus.

3.6.5 The Liveness of Collision-fast Paxos

We now prove that the extended Collision-fast Paxos algorithm presented in Section 3.4.2 satisfies the Liveness property of M-Consensus, given that its liveness condition is eventually satisfied.

Proposition 3.8 *If there exist proposer p, learner l, coordinator c, and quorum Q, such that $LA(p, l, c, Q)$ holds from some time t_0 on, then eventually $learned[l]$ is complete.*

PROOF: The proof is divided into following steps:

1. No coordinator other than c executes any action after t_0
   PROOF SKETCH: The extended algorithm states that coordinators only execute actions if they believe to be the leader and the definition of $LA(p, l, c, Q)$ states that only c believes to be the leader after t_0.

2. There is a time $t_1 \geq t_0$ after which $crnd[c]$ does not change
   PROOF SKETCH: $crnd[c]$ can only be changed by action $Phase1a$. In the extended algorithm, though, this can only happen if c receives a special message informing about a higher-numbered round already started or if not all collision-fast proposers for $crnd[c]$ are in $activep[c]$. As for the first condition, step 1 implies there is only a finite number of (possibly higher-numbered) rounds started before t_0. As for the second one, the definition of LA states that $activep[c]$ contains only nonfaulty processes after t_0 and no element is taken out of the set. In the extended algorithm, if c starts a new round r after t_0, it is guaranteed that its collision-fast proposers are in $activep[c]$, which makes sure

the second condition will not trigger the execution of action $Phase1a$ more than once after t_0.

3. There is a time $t_2 \geq t_1$ after which action $Phase2Start(c, crnd[c])$ will have been executed
   PROOF SKETCH: Let us assume, for the sake of contradiction, that c never executes action $Phase2Start(c, crnd[c])$. By step 2 and the extended algorithm's specification, coordinator c keeps re-sending the "1a" message for round $crnd[c]$ to all acceptors. Acceptors in Q do not crash after t_0 by the definition of LA and must receive such messages. If they all execute action $Phase1b$ for round $crnd[c]$, then they will keep re-sending their $1b$ messages and c will eventually execute $Phase2Start$, contradicting our assumption. Therefore, there must be an acceptor $a \in Q$ such that $rnd[a] > crnd[c]$, which prevents the execution of action $Phase1b(a, crnd[c])$. However, in the extended algorithm a would send an infinite number of special messages to c, indicating that a round higher than $crnd[c]$ has been started and this would eventually lead c to execute action $Phase1a$ for a higher-numbered round, contradicting step 2.

4. From t_2 on, $cval[c]$ does not change
   PROOF SKETCH: By steps 2 and 3 and the algorithm's specification.

5. Eventually l learns a complete v-mapping
   By step 4, there are two cases to consider after t_2:
   5.1. CASE: $cval[c] = \bot$

      5.1.1. From t_2 on, $prnd[q] \leq crnd[c]$ for any proposer q such that q does not crash after t_2
      PROOF SKETCH: Assume $prnd[q] > crnd[c]$ after t_2, for some proposer q that does not crash after t_2. In the extended algorithm, by steps 2 and 3, coordinator c keeps sending the "2S" message for round $crnd[c]$ to the set of proposers. As a result, q will keep replying to c with special messages indicating that a round higher-numbered than $crnd[c]$ has been started and this will force c to start an even higher-numbered round. This contradicts step 2.

      5.1.2. There is a time $t_3 \geq t_2$ after which all proposers q that do not crash after t_3 will have executed action $Phase2Prepare(q, crnd[c])$ and set $prnd[q]$ to $crnd[c]$ and $pval[q]$ to $none$
      PROOF SKETCH: By steps 2 and 3, coordinator c keeps sending the "2S" message for round $crnd[c]$ and, by step 5.1.1 and the definition of action $Phase2Prepare$, every nonfaulty proposer q must eventually execute action $Phase2Prepare(q, crnd[c])$ based on this "2S" message. By the action's definition, $prnd[q]$ is set to $crnd[c]$ and, since the message carries v-mapping $\bot$ (as the value for $cval[c]$), $pval[q]$ is set to $none$.

      5.1.3. There is a time t_4 after which some collision-fast proposer q for round $crnd[c]$ will have executed action $Phase2a(q, crnd[c], V)$, where V is a

proposed value
PROOF SKETCH: By steps 5.1.1 and 5.1.2 and the definition of LA, each collision-fast proposer of $crnd[c]$ is eventually prepared to execute action $Phase2a$ for round $crnd[c]$ triggered by a "propose" or "2a" message. Since a "2a" message is only sent by action $Phase2a$, some "propose" message must trigger the first execution of action $Phase2a$ for round $crnd[c]$. The existence of such a "propose" message is guaranteed by steps 5.1.1 and 5.1.2 since they ensure that proposer p (from the definition of LA) will eventually keep sending its "propose" message to some collision-fast proposer of round $crnd[c]$. Because the first $Phase2a$ action executed for round $crnd[c]$ is necessarily triggered by a "propose" message, its parameter V is a proposed value by the action's definition.

5.1.4. From t_4 on, $rnd[a] \leq crnd[c]$ for any acceptor a such that a does not crash after t_2
PROOF SKETCH: Assume $rnd[a] > crnd[c]$ after t_4, for some acceptor a that does not crash after t_4. By steps 5.1.1 and 5.1.3 and the definition of LA, at least one nonfaulty collision-fast proposer of $crnd[c]$ will eventually keep sending a "2a" message for round $crnd[c]$ to a. As a result, a will keep sending notification messages to c indicating that a round higher-numbered than $crnd[c]$ has been started. This would force c to start a new higher-numbered round, contradicting step 2.

5.1.5. There is a time $t_5 \geq t_4$ after which all collision-fast proposers of round $crnd[c]$ will have executed action $Phase2a$ for round $crnd[c]$
PROOF SKETCH: Assume there is a collision-fast proposer q for round $crnd[c]$ such that q never executes action $Phase2a$ for round $crnd[c]$. By steps 5.1.1 and 5.1.2 and the definition of LA, q is eventually prepared to execute action $Phase2a$ for round $crnd[c]$ triggered by a "propose" or "2a" message. By steps 5.1.1 and 5.1.3, at least one nonfaulty collision-fast proposer of $crnd[c]$ will eventually keep sending a "2a" message for $crnd[c]$ to q, which must eventually trigger its execution of action $Phase2a$ for round $crnd[c]$. This contradicts our initial assumption that q does not execute action $Phase2a$ for $crnd[c]$.

5.1.6. There is a time $t_6 \geq t_5$ after which, if action $Phase2a(q, crnd[c], V)$ has been executed for any proposer q and non-Nil value V, then all acceptors in Q will have executed action $Phase2b$ for round $crnd[c]$ triggered by the "2a" message sent by q
PROOF SKETCH: By step 5.1.1, if action $Phase2a(q, crnd[c], V)$ is executed, q will keep sending "2a" messages to the acceptors. By step 5.1.4, acceptors will be able to execute action $Phase2a$ for any "2a" message for round $crnd[c]$ with non-Nil values. Since LA ensures that all collision-fast proposers of $crnd[c]$ are nonfaulty, the "2a" message from q will be eventually received by the acceptors in Q (also nonfaulty by LA) and trigger the

execution of a $Phase2a$ action for round $crnd[c]$.

5.1.7. Q.E.D.
PROOF SKETCH: After t_6, by steps 5.1.6 and 5.1.4, acceptors in Q keep sending "2b" messages to the learners with the same v-mapping v. By the definition of action $Phase2b$, v maps each proposer that is not collision-fast for $crnd[c]$ to Nil. Moreover, according to step 5.1.6 and the definition of action $Phase2b$, v maps each proposer q that is collision-fast for r and has executed action $Phase2a(q, crnd[c], V)$, where $V \neq Nil$, to V. All other proposers are not mapped by v. According to steps 5.1.1 and 5.1.5, proposers that have executed action $Phase2a$ for round $crnd[c]$ and value Nil keep sending their "2a" messages to the learners. The "2b" messages from the acceptors and the "2a" messages from the collision-fast proposers allow l to eventually learn a complete v-mapping.

5.2. CASE: $cval[c] \neq \bot$

5.2.1. $cval[c]$ is complete
By the definition of action $Phase2Start$ and the fact that $cval[c] \neq \bot$.

5.2.2. From t_2 on, $rnd[a] \leq crnd[c]$ for any acceptor a such that a does not crash after t_2
PROOF SKETCH: Assume $rnd[a] > crnd[c]$ after t_2, for some acceptor a that does not crash after t_2. By steps 2 and 3, coordinator c keeps sending the "2S" message for round $crnd[c]$ to the set of acceptors. In the extended algorithm, though, a will keep replying to c with special messages indicating that a round higher-numbered than $crnd[c]$ has been started and this would force c to start an even higher-numbered round, which contradicts step 2.

5.2.3. Eventually all acceptors a in Q execute action $Phase2b(a, r)$ and set $vrnd[a]$ to $crnd[c]$ and $vval[a]$ to $cval[c]$
PROOF SKETCH: By steps 2 and 3, coordinator c keeps sending the "2S" message for round $crnd[c]$ and, by step 5.2.2, all acceptors in Q must eventually execute action $Phase2b(a, crnd[c])$ based on this "2S" message.

5.2.4. Q.E.D.
PROOF SKETCH: By steps 5.2.2 and 5.2.3, all acceptors in Q will eventually keep sending "2b" messages for round $crnd[c]$ with a complete v-mapping, that is, $cval[c]$. Learner l will eventually receive such messages and learn this complete v-mapping.

6. Q.E.D.

3.7 Correctness of the Sequence Agreement Algorithm

In this section we prove that our sequence agreement protocol indeed satisfies the safety and liveness properties stated in Section 3.2. We start by presenting the complete specification of the protocol and then proceed with the proofs.

3.7.1 Complete Algorithm Specification

In Section 3.5.2, we have presented our collision-fast sequence agreement algorithm. This protocol uses infinitely many Collision-fast Paxos instances (Section 3.4.2), each of them identified by a natural number i and referred as $CFP(i)$ in the protocol. Actions and variables specific of an instance i are identified by the prefix $CFP(i)!$ (instead of the superscript of Section 3.5.2). The protocol forces the Collision-fast Paxos instances to execute the same rounds at the same time. As a consequence, some of their variables, namely, $proposed$, rnd, $prnd$, $crnd$, and $activep$, are always equal. Intead of keeping multiple copies of these variables, we let all instances share the same copy (and drop the prefix $CFP(i)!$ to simplify the notation). The protocol introduces no other variable.

All actions of the algorithm execute the homonymous action either in one of the Collision-fast Paxos instances, or in all of them at once in a composed manner. All composed actions pre-conditions are defined only over shared variables and, therefore, either all actions in the composition are enabled or all are disabled. The only exception is action $Phase1a$.

Action $CFP(i)!Phase1a$, for some Collision-fast Paxos instance i, has one pre-condition over $CFP(i)!msgs$. We define $NewPhase1a$ as a replacement to $CFP(i)!Phase1a$ that changes the said pre-condition to be satisfied if true for $CFP(j)!msgs$, for any instance j. $NewPhase1a$ and the other actions of the algorithm are defined bellow.

$Propose(V)$ Executed to propose a message V. It is the composition of action $CFP(i)!Propose(V)$ for all Collision-fast Paxos instances i. Logically, each instance sends the message $\langle$"`propose`", $V\rangle$. Since they are all the same, they can be replaced by a single message valid for all Collision-fast Paxos instances.

$NewPhase1a(i, c, r)$ Executed by coordinator c to start round r in instances i. The action executes iff:

- c believes itself to be the leader,
- c is the coordinator of round r,
- $crnd[c] \prec r$,

- either c received some special message informing of a round j ($r > j > crnd[c]$) was initiated in any M-Consensus instance, *or* the set of collision-fast proposers of round $crnd[c]$ is not a subset of $activep[c]$.

The action sets $crnd[c]$ to r and $CFP(i)!cval[i]$ to $none$, and sends a message $\langle$"1a", $r\rangle$ in this instance.

$Phase1a(c, r)$ Executed by coordinator c to start round r. It is the composition action $NewPhase1a(i, c, r)$ for all Collision-fast Paxos instances i, where action $NewPhase1a(i, c, r)$ is defined previously. Logically, each instance sends its own "1a" message but, since they are all equal, a single message is sent instead.

$Phase1b(a, r)$ Executed by acceptor a on round r when it receives the message $\langle$"1a", $r\rangle$. It is the composition of action $CFP(i)!Phase1b(a, r)$ for all Collision-fast Paxos instances i. Each instance sends its own "1b" message but they are all bundled together in a single composite message.

$Phase2Start(c, r)$ Executed by the coordinator c of round r when it receives a composite "1b" message. It is the composition of action $CFP(i)!Phase2Start(c, r)$ for all Collision-fast Paxos instances i. c sends a "2S" message in each instance, but these messages are bundled together in a single physical message. Since only a finite number of instances send "2S" messages different from $\langle$"2S", $r, \perp\rangle$, the composite message has a finite size.

$Phase2Prepare(p, r)$ Executed by proposer p when it receives a composite "2S" message for round r. It is the composition of the actions $CFP(i)!Phase2Prepare(p, r)$ of all Collision-fast Paxos instances i.

$Phase2a(p, r, V)$ Executed by proposer p to fast-propose message V on round r. It is executed iff p has not fast-proposed V on any Collision-fast Paxos instance before. The action proposes V on the smaller Collision-fast Paxos instance i it is allowed to propose (those instances j such that $CFP(j)!pval[p] = none$); it does so by executing action $CFP(i)!Phase2a(p, r, V)$.

$Phase2b(a, r)$ Executed by acceptor a to accept some v-mapping on some instance i. It does so by executing action $CFP(i)!Phase2b(a, r)$.

$Learn(l, v)$ Executed by learner l to learn the v-mapping v in some Collision-fast Paxos instance i. The action executes $CFP(i)!Learn(l, v)$.

The sequence learned by a learner l in the sequence agreement problem is a function of the v-mappings learned by l in all M-Consensus instances. To define this function properly, we assume a total order $<_P$, on the set of proposer agents, and a function $R_P(e, S)$, that gives the rank of an element e of a set S with respect to the other elements in S, according to the order $<_P$ (e.g., $R_P(3, \{3, 1, 5\}) = 2$). The recursive definition of $learned[l]$ is as follows.

$$
\begin{array}{l}
learned[l] \triangleq \\
\quad \text{LET } defSet(m,s) \triangleq \\
\qquad \{p \in Dom(m) : \\
\qquad\qquad \wedge\, m[p] \neq Nil \\
\qquad\qquad \wedge\, \neg\exists i \in 1..Len(s) : s[i] = m[p] \\
\qquad\qquad \wedge\, \forall q \in Dom(m) : q <_P p \Rightarrow m[q] \neq m[p] \\
\qquad\qquad \wedge\, \forall q \in Proposer : q <_P p \Rightarrow q \in Dom(m)\} \\
\\
\qquad defSeq(m,s) \triangleq \langle e_1, e_2, \ldots, e_n\rangle : \\
\qquad\quad \wedge\, n = |defSet(m,s)| \\
\qquad\quad \wedge\, \forall p \in defSeq(m,s) : R_P(p, defSet(m,s)) = i \Leftrightarrow e_i = m[p] \\
\\
\qquad deliver(l,i,s) \triangleq \\
\qquad\quad \text{IF } Domain = Dom(CF(i)!learned[l]) \\
\qquad\quad \text{THEN } deliver(l, i+1, s \circ defSeq(CF(i)!learned[l], s)) \\
\qquad\quad \text{ELSE } s \circ defSeq(CF(i)!learned[l], s) \\
\\
\quad \text{IN } deliver(l, 0, \langle\rangle)
\end{array}
$$

Informally, for each learner l, this function builds sequence $learned[l]]$ by iterating over the M-Consensus instances, from 0 to the first one in which l has not learned a complete mapping yet. In each instance, the iteration proceeds over proposers, ascendingly in the total order $<_P$, adding the mapped value of each proposer to the sequence (if it has not been added before). For the instance with an incomplete v-mapping, the iteration proceeds only until a non-mapped proposer is found.

3.7.2 Safety

In the previous section we described all the actions of our collision-fast sequence agreement algorithm. Except for one action, all the others simply execute the actions of its Collision-fast Paxos instances. The exception is action $Phase1a$, that executes $NewPhase1a$ instead of the $Phase1a$ of Collision-fast Paxos, defined in Section 3.4. However, comparing the two definitions, it is easy to see that the first is in fact a more restricted version of the latter, and therefore implements it. The proposition below formally states this property.

Proposition 3.9 *For any Collision-fast Paxos instance i, coordinator c, and round r, $NewPhase1a(i,c,r)$ implements $CFP(i)!Phase1a(c,r)$.*

ASSUME: There exist a natural number i, a coordinator c and a round number r such that $NewPhase1a(i,c,r)$ is enabled.

PROVE: $CFP(i)!Phase1a(c, r)$ is enabled.

PROOF: Since all pre-conditions of $CFP(i)!Phase1a(c, r)$ are also present in $NewPhase1a(i, c, r)$ and they are all satisfied by the assumption, the action $CFP(i)!Phase1a(c, r)$ must also be enabled.

Proposition 3.9 implies that all Collision-fast Paxos instances satisfy their safety properties. We use it to prove that our sequence agreement implementation also satisfies its safety properties, according to our specification of sequence agreementgiven in Section 3.2.

Proposition 3.10 *Our sequence agreementimplementation satisfies the safety specification of the sequence agreementproblem given in Section 3.2.*

1. At the initial state, $learned[l] = \langle\rangle$, for any learner l.
   PROOF SKETCH: By the definition of $learned[l]$, since initialization guarantees that $CFP(i)!learned[l] = \bot$, for any learner l and M-Consensus instance i.
2. For any learner l, $learned[l]$ contains only proposed values.
   PROOF SKETCH: Our definition of $learned[l]$ states that it contains only values decided in a mapping for some M-Consensus instance. From the Nontriviality property of M-Consensus, any value $V \neq Nil$ in the decided mapping of an instance must have been proposed in that instance. From our protocol's specification, a value is only proposed to an M-consensus instance if it has been proposed to sequence agreement.
3. For any learner l, if $delivered[l] = s$ at some time, then $s \sqsubseteq delivered[l]$ at all later times.
   PROOF SKETCH: For some learner l, let k be the smallest instance for which l learned an incomplete v-mapping at some point in time t. For all instances $i < k$, l has learned a complete v-mapping for i (i.e., $CFP(i)!learned[l]$ is complete). Because of the Stability property of M-Consensus, for any instances $i < k$ at any instant $t' > t$, $CFP(i)!learned[l]$ will equal its value at instant t. As for instance k, $CFP(k)!learned[l]$ at time t will be a prefix of its value at time t'.
   By its definition, $learned[l]$ is built by an ascending iteration over the instances i, $0 \leq i \leq k$. For each instance i, the procedure iterates over the proposers p that have been mapped to some value in $CFP(i)!learned[l]$, in the ascending order defined by $<_P$, and appends $V = CFP(i)!learned[l][p]$ to the sequence being created if V is not in the sequence yet. On instance k, the iteration proceeds until the bigger proposer q for which all the smaller proposers have been mapped.
   Given the determinism in the iteration and the observations in the first paragraph, the construction of $learned[l]$ at any instant $t' > t$ will proceed over instances 0 to $k - 1$ and then in instance k exactly as at the instant t, building the same sequence, and then possibly extend it with other values mapped in k and bigger instances. Hence, $learned[l]$ at instant t is a prefix of $learned[l]$ at any instant $t' > t$.
4. For any pair of learners l_1 and l_2, either $learned[l_1] \sqsubseteq learned[l_2]$ or $learned[l_2] \sqsubseteq learned[l_1]$.

PROOF SKETCH: Let k, be the smallest M-Consensus instance for which either l_1 or l_2 has not learned a complete v-mapping. By the definition of $learned[l]$ and the Consistency property of M-Consensus, for all instances $i \leq k$, $CFP(k)!learned[l_1] = CFP(k)!learned[l_2]$. As a result, both sequences share the same prefix $pref$ generated based on such instances. As for instance k, the Consistency property of M-Consensus also guarantees that no two values are mapped to the same proposer for different learners. The learner with the shortest sequence $learned[l]$ will have $pref$ extended with the values mapped to some proposers in k. The same values must be mapped to these proposers in instance k for the other learner, which ensures that one sequece is a prefix of the other by the definition of $learned[l]$.

5. Q.E.D.

3.7.3 Liveness

In this section we prove that values proposed using our sequence agreement protocol are eventually delivered if the liveness condition of Collision-fast Paxos is eventually satisfied.

Proposition 3.11 *If there exist proposer p, learner l, coordinator c, and quorum Q, such that $LA(p, l, c, Q)$ holds from some time t_0 on, then eventually $learned[l]$ contains the values proposed by p.*

PROOF SKETCH: The proof is divided into the following steps:

1. No coordinator other than c executes any action after t_0.
   By the definition of $LA(p, l, c, Q)$, after t_0, no coordinator except for c will ever believe itself to be to leader again. Since this is a pre-condition for all coordinator actions, none will be executed after t_0.
2. There is a time $t_1 > t_0$ after which $crnd[c]$ does not change.
   The value of $crnd[c]$ is only changed by executing action $NewPhase1a$ for c. By its defintion, this action can be executed only when c receives a special message informing about a round numbered higher than $crnd[c]$ already started, or if the set of collision-fast proposers for round $crnd[c]$ is not a subset of $activep[c]$.
   Step 1 implies there is just a finite number of (possibly higher-numbered) rounds started before t_0 and that could satisfy the first condition. As for the second condition, the definition of LA states that $activep[c]$ contains only correct processes after t_0 and no element is taken out of the set. Therefore, if c starts a new round r after t_0, it is guaranteed that its collision-fast proposers are in $activep[c]$, which makes sure the second condition will not trigger the execution of action $NewPhase1a$ and, consequently, $Phase1a$ for a higher-numbered round.
3. Q.E.D.
   PROOF: By steps 1 and 2, by the time t_1 c will have started a round r bigger than any other started and will not start any bigger one. Moreover, all collision-fast proposers

of r are correct and accessible. For each Collision-fast Paxos instance, this situation is equivalent to the one in which its $Phase1a$ has been successfully executed for r and $LA(p, l, c, Q)$ holds. As we have shown in Section 3.6.5, Collision-fast Paxos satisfy the liveness property of M-Consensus under these conditions.
All Collision-fast Paxos instances in the sequence agreement protocol satisfy such property. Therefore, if p proposes value V, it will eventually be seen by some collision-fast proposer q (not necessarily different from p) of round r. q will eventually fast-propose V in the first instance for which it has not fast-proposed yet. The "2a" message it generates will eventually trigger the termination of the instance and have l learn V.

3.8 TLA$^+$ Specifications

3.8.1 Module *SAgreement*

This module presents a model-based specification of the safety properties of the sequence agreement problem.

MODULE *SAgreement*

EXTENDS *Sequences*, *Naturals*

CONSTANTS *Proposer*, *Learner*, *Value*

VARIABLES *proposed*, *learned*

TypeInv asserts a type invariant; the assertion that *TypeInv* is always true is a property of (implied by) the specification

$$\begin{aligned} TypeInv \triangleq\ & \land\ proposed \subseteq Value \\ & \land\ learned\ \in [Learner\ \to Seq(Value)] \end{aligned}$$

Init is the initial predicate.

$$\begin{aligned} Init \triangleq\ & \land\ proposed = \{\} \\ & \land\ learned = [l \in Learner \mapsto \langle\rangle] \end{aligned}$$

$$\begin{aligned} & s1 \sqsubseteq s2 \triangleq \\ & \quad \land\ Len(s1) \leq Len(s2) \\ & \quad \land\ \forall\, i \in 1\,..\,Len(s1) : s1[i] = s2[i] \end{aligned}$$

We now define the two actions of proposing a value and learning a sequence. The Learn action sets $learned[l]$ to a sequence extending its present value.

$$\begin{aligned} Propose \triangleq\ & \land\ \exists\, m \in Value : \\ & \qquad proposed' = proposed \cup \{m\} \\ & \land\ \text{UNCHANGED}\ \langle learned \rangle \end{aligned}$$

$$Learn(l) \triangleq\ \land\ \exists\, v \in proposed :$$

$$
\begin{array}{l}
\land \neg\exists\, i \in 1 \,..\, Len(learned[l]) : \\
\quad learned[l][i] = v \\
\land \forall\, l2 \in Learner : \\
\quad \lor\ learned[l] \circ \langle v \rangle \sqsubseteq learned[l2] \\
\quad \lor\ learned[l2] \sqsubseteq learned[l] \\
\land\ learned' = [learned \text{ EXCEPT } ![l] = @ \circ \langle v \rangle] \\
\land \text{ UNCHANGED } \langle proposed \rangle
\end{array}
$$

Next is the complete next-state action; *Spec* is the complete specification.

$$Next \triangleq Propose \lor \exists\, l \in Learner : Learn(l)$$

$$Spec \triangleq Init \land \Box[Next]_{\langle proposed,\, learned \rangle}$$

We now define the three safety properties as temporal formulas and assert that they and the type-correctness invariant are properties of the specification.

$$
\begin{array}{ll}
Nontriviality & \triangleq \forall\, l \in Learner : \\
 & \quad \Box(learned[l] \in Seq(proposed)) \\
 & \\
Stability & \triangleq \forall\, l \in Learner : \\
 & \quad \Box(\exists\, s \in Seq(Value) : learned[l] = s \\
 & \qquad \Rightarrow \Box(s \sqsubseteq learned[l])) \\
 & \\
Consistency & \triangleq \forall\, l1,\, l2 \in Learner : \\
 & \quad \Box(\ \lor\ learned[l1] \sqsubseteq learned[l2] \\
 & \qquad \lor\ learned[l2] \sqsubseteq learned[l1])
\end{array}
$$

THEOREM $Spec \Rightarrow (\Box\, TypeInv) \land Nontriviality \land Stability \land Consistency$

3.8.2 Module *VMapping*

Module *VMapping* specifies constants and operators for dealing with v-mappings.

MODULE *VMapping*

This module defines constants and operators for dealing with value mappings.

LOCAL INSTANCE *FiniteSets* The *FiniteSets* module defines the operation *Len*

We declare the sets *Domain* and *Value* as parameters.

CONSTANTS $Domain,\ Value,\ Nil,\ none$

Nil is a no-value used to map an element in *Domain* to nothing.

$Nil \triangleq$ CHOOSE $n : n \notin Value$

ASSUME $Nil \notin Value$

ValMap defines the set of all valid value mappings. It is composed of any function that maps a subset of the domain to values or *Nil*.

$$ValMap \triangleq \text{UNION } \{[PS \to Value \cup \{Nil\}] : PS \in \text{SUBSET } Domain\}$$

none is defined to be something that is not a *ValMap*

$$none \triangleq \text{CHOOSE } n : n \notin ValMap$$
$$\text{ASSUME } none \notin ValMap$$
$$\text{ASSUME } none \neq Nil$$
$$\text{ASSUME } none \notin Value$$

We define *Bottom* to be the "empty" *ValMap*, that is, a *ValMap* function whose domain is the empty set. In TLA, a function with empty domain is defined to be equal to the empty sequence, which allows the simplification below. We use *Bottom* instead of $\perp$.

$$Bottom \triangleq \langle\rangle$$

For simplicity, $Dom(f)$ is defined to be the domain of function f.

$$Dom(f) \triangleq \text{DOMAIN } f$$

A *SingleMap* maps a single domain element to a *Value* or *Nil*.

$$SingleMap \triangleq [p : Domain,\ v : Value \cup \{Nil\}]$$

$SM(p,\ v)$ defines a *SingleMap* from domain element p to value v.

$$SM(p,\ v) \triangleq [p \mapsto p,\ v \mapsto v]$$

The basic operation over a *ValMap* is $vm \bullet sm$. It agregates a *SingleMap* to a *ValMap*. It is well-defined only if vm is a *ValueMap* and sm is a *SingleMap*.

$$vm \bullet sm \triangleq [p \in Dom(vm) \cup \{sm.p\} \mapsto \text{IF } p \in Dom(vm) \text{ THEN } vm[p] \text{ ELSE } sm.v]$$

ValMap is a c-struct and admits all existing operators for c-structs. We define them in the following specifically for the *ValMap* type, which allows many simplifications and optimizations.

A *ValMap* v is a prefix of a *ValMap* w ($v \sqsubseteq w$) if it can be extended to w by a sequence of $\bullet$ applications with single mappings. This can be verified in a simplified way by checking if the domain of v is a subset of the domain of w and for every element in the domain of v, its mapped value in v is equal to its mapped value in w. If $v \sqsubseteq w$, we say that v is a prefix of w or that w extends v. We extend the definition of $v \sqsubseteq w$ so that it is true if both v and w equals none.

$$\begin{array}{lll} v \sqsubseteq w \triangleq & \lor & \land\ v \neq none \\ & & \land\ w \neq none \\ & & \land\ Dom(v) \subseteq Dom(w) \\ & & \land\ \forall\, e \in Dom(v) : v[e] = w[e] \\ & \lor & \land\ v = none \\ & & \land\ w = none \end{array}$$

A *ValMap* v is a strict prefix of a *Valmap* w ($v \sqsubset w$) if it is a prefix of w and it is different from w.

$$v \sqsubset w \triangleq (v \sqsubseteq w) \land (v \neq w)$$

$GLB(T)$ is the greatest lower bound of a set of value mappings. It is a *ValMap* u that is a prefix to all *ValMaps* in T but is not a prefix of any other *ValMap* that is also a prefix of all *ValMaps* in T. It is more simply defined as a function that maps each element that belongs to the domain intersection of all mappings and whose mapped value in all mappings is the same to its mapped value in all value mappings.

$$\begin{array}{l} GLB(T) \triangleq \text{LET } witness \triangleq \text{CHOOSE } f \in T : \text{TRUE} \\ \qquad\qquad CInter \triangleq \{p \in Dom(witness) : \\ \qquad\qquad\qquad\qquad \forall f \in T : \land p \in Dom(f) \\ \qquad\qquad\qquad\qquad\qquad\qquad \land f[p] = witness[p]\} \\ \qquad\qquad \text{IN } [p \in CInter \mapsto witness[p]] \end{array}$$

$v \sqcap w$ is defined to be the greatest lower bound for the set $\{v, w\}$

$$v \sqcap w \triangleq GLB(\{v, w\})$$

A *ValMap* v is defined to be compatible with a *ValMap* w if they are both prefixes of a *ValMap* u. It is equivalent to verifying if their common domain elements are mapped to the same values.

$$AreCompatible(v, w) \triangleq \forall e \in Dom(v) \cap Dom(w) : v[e] = w[e]$$

A set of *ValMaps* is compatible if its elements are pairwise compatible.

$$IsCompatible(S) \triangleq \forall v, w \in S : AreCompatible(v, w)$$

$LUB(T)$ is the least upper bound of a set of value mappings. It is a *ValMap* u that extends all *ValMaps* in T but does not extend any other *ValMap* that also extends all *ValMaps* in T. It is more simply defined as a function that maps each element that belongs to the domain of any of the mappings to its mapped value on any of the mappings whose domain it belongs to. It is only well-defined if T is a set of compatible value mappings.

$$\begin{array}{l} LUB(T) \triangleq [p \in \text{UNION } \{Dom(f) : f \in T\} \mapsto \\ \qquad\qquad (\text{CHOOSE } f \in T : p \in Dom(f))[p]] \end{array}$$

$v \sqcup w$ is defined to be the least upper bound for the set $\{v, w\}$

$$v \sqcup w \triangleq LUB(\{v, w\})$$

3.8.3 Module *MConsensus*

This module presents a model-based specification of the safety properties of the M-Consensus problem.

MODULE *MConsensus*

CONSTANTS *Proposer*, *Learner*, *Value*, *Nil*, *none*

INSTANCE *VMapping* WITH *Domain* $\leftarrow$ *Proposer*

VARIABLES *proposed*, *learned*

TypeInv asserts a type invariant; the assertion that *TypeInv* is always true is a property of (implied by) the specification

$$\begin{array}{l} TypeInv \triangleq \land proposed \subseteq Value \\ \qquad\qquad\ \land learned \in [Learner \rightarrow ValMap] \end{array}$$

Init is the initial predicate.

$$Init \triangleq \land\ proposed = \{\}$$
$$\land\ learned = [l \in Learner \mapsto Bottom]$$

$$IsProposed(m) \triangleq \forall\, p \in Dom(m) : m[p] \in (proposed \cup \{Nil\})$$

$$IsTrivial(m) \triangleq m = [p \in Proposer \mapsto Nil]$$

We now define the two actions of proposing a value and learning a mapping. The Learn action sets $learned[l]$ to a mapping extending its present value.

$$Propose \triangleq \land \exists\, v \in Value :$$
$$proposed' = proposed \cup \{v\}$$
$$\land \text{ UNCHANGED } \langle learned \rangle$$

$$Learn(l) \triangleq \land \exists\, v \in ValMap :$$
$$\land\ IsProposed(v)$$
$$\land \forall\, l2 \in Learner : AreCompatible(v,\ learned[l2])$$
$$\land \neg IsTrivial(LUB(\{learned[r] : r \in Learner\}) \sqcup v)$$
$$\land\ learned' = [learned \text{ EXCEPT } ![l] = @ \sqcup v]$$
$$\land \text{ UNCHANGED } \langle proposed \rangle$$

Next is the complete next-state action; *Spec* is the complete specification.

$$Next \triangleq Propose \lor \exists\, l \in Learner : Learn(l)$$

$$Spec \triangleq Init \land \Box[Next]_{\langle proposed,\, learned \rangle}$$

We now define the three safety properties as temporal formulas and assert that they and the type-correctness invariant are properties of the specification.

$$Nontriviality \triangleq \forall\, l \in Learner :$$
$$\Box(IsProposed(learned[l]) \land \neg IsTrivial(learned[l]))$$

$$Stability \triangleq \forall\, l \in Learner :$$
$$\Box(\exists\, v \in ValMap : learned[l] = v \Rightarrow \Box(v \sqsubseteq learned[l]))$$

$$Consistency \triangleq \Box(\land IsCompatible(\{learned[l] : l \in Learner\})$$
$$\land \neg IsTrivial(LUB(\{learned[r] : r \in Learner\})))$$

THEOREM $Spec \Rightarrow (\Box\, TypeInv) \land Nontriviality \land Stability \land Consistency$

3.8.4 Module *PaxosConstants*

Module *PaxosConstants* defines constants and operators used by all our Paxos abstractions.

MODULE $PaxosConstants$

This module defines the parameters and data structures for our algorithms.

EXTENDS $FiniteSets$

$RNum$ is the set of round numbers and $\preceq$ defines an ordering relation amongst the set of rounds. The module also has as a parameter an initial round number called $Zero$.

CONSTANTS $RNum$, $\_ \preceq \_$, $Zero$

We assume $\preceq$ is a total ordering of the set $RNum$ of round numbers.

ASSUME LET $PO \triangleq$ INSTANCE $OrderRelations$ WITH $S \leftarrow RNum$, $\sqsubseteq \leftarrow \preceq$
IN $PO!IsTotalOrder$

We define $i \prec j$ to be true iff $i \preceq j$ for two different rounds i and j.

$i \prec j \triangleq (i \preceq j) \land (i \neq j)$

If B is a set of round numbers that contains a maximum element, then $Max(B)$ is defined to equal that maximum. Otherwise, its value is unspecified.

$Max(B) \triangleq$ CHOOSE $i \in B : \forall\, j \in B : j \preceq i$

Are parameters of this module:
- A set $Proposer$ of proposer agents,
- A set $Learner$ of learner agents,
- A set $Acceptor$ of acceptor agents,
- An operator $Quorum$ that returns the acceptor quorums of a round,
- An operator $CfProposer$ that returns the collision-fast proposers of a round,
- And a set $Value$ of proposable Values.

CONSTANTS $Proposer$, $Learner$, $Acceptor$, $Quorum(\_)$, $CfProposer(\_)$, $Value$

$Nil \triangleq$ CHOOSE $n : n \notin Value$
$none \triangleq$ CHOOSE $n : n \notin Value$

The problem of $MConsensus$ is defined in terms of a value mapping set whose $Domain$ is the set of proposers

INSTANCE $VMapping$ WITH $Domain \leftarrow Proposer$

We assume that quorums are finite subsets of the acceptors and every pair of quorums has a non-empty intersection.

$QuorumAssumption \triangleq$
$\forall\, i \in RNum :$
$\land\, Quorum(i) \subseteq$ SUBSET $Acceptor$
$\land \forall\, Q \in Quorum(i) : IsFiniteSet(Q)$
$\land \forall\, j \in RNum :$
$\forall\, Q \in Quorum(i),\, R \in Quorum(j) : Q \cap R \neq \{\}$

ASSUME $QuorumAssumption$

Over the set of collision-fast proposers for a round i, we only assume they are a finite subset of the proposers.

$CfProposerAssumption \triangleq$
$\forall\, i \in RNum :$

$\land\ CfProposer(i) \in \text{SUBSET } Proposer$
$\land\ IsFiniteSet(CfProposer(i))$

ASSUME $CfProposerAssumption$

We say that a value mapping is valued iff at least one element of its domain is mapped to a value ($\neq Nil$). Our algorithm makes sure that acceptors will only accept valued mappings, so that we can guarantee $Nontriviality$.

$IsValued(m) \triangleq \exists\, p \in Dom(m) : m[p] \neq Nil$

We define the Nil-extension of a valued mapping v for a set P of proposers to be the $ValMap$ resulting from adding to v the single mapping $p \to Nil$, for every proposer p in $P \setminus Dom(v)$.

$NilExtension(v,\ P) \triangleq$
  IF $v = none$ THEN $none$
    ELSE $[p \in Dom(v) \cup P \mapsto$ IF $p \in Dom(v)$ THEN $v[p]$
                                  ELSE $Nil]$

We define $BallotArray$ to be the set of all ballot arrays. We represent a ballot array as a record, where we write $\beta_a[m]$ as $\beta.vote[a][m]$ and $\hat{\beta}_a$ as $\beta.rnd[a]$.

$BallotArray \triangleq$
 $\{beta \in [vote : [Acceptor \to [RNum \to ValMap \cup \{none\}]],$
          $rnd\ : [Acceptor \to RNum]] :$
       $\forall\, a \in Acceptor :$
          $\land\ IsFiniteSet(\{m \in RNum : beta.vote[a][m] \neq none\})$
          $\land\ \forall\, m \in RNum :$
               $\land\ (beta.rnd[a] \prec m) \Rightarrow (beta.vote[a][m] = none)$
               $\land\ (beta.vote[a][m] \neq none) \Rightarrow IsValued(beta.vote[a][m])\}$

We define $CfPropArray$ to be the set of all proposal arrays.

$PropArray \triangleq [Proposer \to [RNum \to Value \cup \{Nil\} \cup \{none\}]]$

We now formalize the definitions of *chosen at*, *safe at*, etc. We translate the *English* terms into obvious operator names. For example, $IsChosenAt(v,\ m,\ \beta, \gamma\ )$ is defined to be true iff v is chosen at m in $\langle\ \beta, \gamma\ \rangle$, assuming that v is a $ValMap$, m is a round number, β is a ballot array, and γ is a proposal array. (We don't care what $IsChosenAt(v,\ m,\ \beta, \gamma\ )$ means for other values of v, m, β, and γ.) We also assert the two propositions as theorems.

$IsChosenAt(v,\ m,\ beta,\ gamma) \triangleq$
  LET $NilP \triangleq \{p \in CfProposer(m) : gamma[p][m] = Nil\}$
  IN $\exists\, Q \in Quorum(m) :$
       $\forall\, a \in Q : (v \sqsubseteq NilExtension(beta.vote[a][m],\ NilP))$

$IsChosenIn(v,\ beta,\ gamma) \triangleq \exists\, m \in RNum : IsChosenAt(v,\ m,\ beta,\ gamma)$

$IsChoosableAt(v,\ m,\ beta,\ gamma) \triangleq$
  $\exists\, Q \in Quorum(m) :$
   LET $P \triangleq \{p \in Proposer : gamma[p][m] \in \{Nil,\ none\}\}$
   IN $\forall\, a \in Q :$
        $(m \prec beta.rnd[a]) \Rightarrow$

$$(v \sqsubseteq \mathit{NilExtension}(\mathit{beta}.\mathit{vote}[a][m],\ \mathit{Proposer}))$$

$\mathit{IsSafeAt}(v,\ m,\ \mathit{beta},\ \mathit{gamma}) \triangleq$
$\forall\, k \in \mathit{RNum} :$
$(k \prec m) \Rightarrow \forall\, w \in \mathit{ValMap} :$
$\mathit{IsChoosableAt}(w,\ k,\ \mathit{beta},\ \mathit{gamma}) \Rightarrow (w \sqsubseteq v)$

$\mathit{IsSafe}(\mathit{beta},\ \mathit{gamma}) \triangleq$
$\forall\, a \in \mathit{Acceptor},\ k \in \mathit{RNum} :$
$(\mathit{beta}.\mathit{vote}[a][k] \neq \mathit{none}) \Rightarrow \mathit{IsSafeAt}(\mathit{beta}.\mathit{vote}[a][k],\ k,\ \mathit{beta},\ \mathit{gamma})$

$\mathit{Proposition}1 \triangleq$
$\forall\, \mathit{beta} \in \mathit{BallotArray},\ \mathit{gamma} \in \mathit{PropArray} :$
$\mathit{IsSafe}(\mathit{beta},\ \mathit{gamma}) \Rightarrow$
$\mathit{IsCompatible}(\{v \in \mathit{ValMap} : \mathit{IsChosenIn}(v,\ \mathit{beta},\ \mathit{gamma})\})$

THEOREM $\mathit{Proposition}1$

$\mathit{IsConservative}(\mathit{beta},\ \mathit{gamma}) \triangleq$
$\forall\, m \in \mathit{RNum},\ a,\ b \in \mathit{Acceptor} :$
$\land\ \mathit{beta}.\mathit{vote}[a][m] \neq \mathit{none}$
$\land\ \mathit{beta}.\mathit{vote}[b][m] \neq \mathit{none}$
$\Rightarrow\ \land\ \mathit{AreCompatible}(\mathit{beta}.\mathit{vote}[a][m],\ \mathit{beta}.\mathit{vote}[b][m])$
$\land\ \forall\, p \in \mathit{Dom}(\mathit{beta}.\mathit{vote}[b][m]) \setminus \mathit{Dom}(\mathit{beta}.\mathit{vote}[a][m]) :$
$\mathit{beta}.\mathit{vote}[b][m][p] = \mathit{gamma}[p][m]$

$\mathit{ProvedSafe}(Q,\ m,\ \mathit{beta}) \triangleq$
IF $\forall\, a \in Q,\ i \in \mathit{RNum} : (i \prec m) \Rightarrow (\mathit{beta}.\mathit{vote}[a][i] = \mathit{none})$
THEN Bottom
ELSE LET $k \triangleq \mathit{Max}(\{i \in \mathit{RNum} :$
$(i \prec m) \land (\exists\, a \in Q : \mathit{beta}.\mathit{vote}[a][i] \neq \mathit{none})\})$
$\mathit{AS} \triangleq \{a \in Q : \mathit{beta}.\mathit{vote}[a][k] \neq \mathit{none}\}$
$G \triangleq \{\mathit{beta}.\mathit{vote}[a][k] : a \in \mathit{AS}\}$
IN $\mathit{NilExtension}(\mathit{LUB}(G),\ \mathit{Proposer})$

$\mathit{Proposition}2 \triangleq$
$\forall\, m \in \mathit{RNum},\ \mathit{beta} \in \mathit{BallotArray},\ \mathit{gamma} \in \mathit{PropArray} :$
$\forall\, Q \in \mathit{Quorum}(m) :$
$\land\ \mathit{IsSafe}(\mathit{beta},\ \mathit{gamma})$
$\land\ \mathit{IsConservative}(\mathit{beta},\ \mathit{gamma})$
$\land\ \forall\, a \in Q : m \preceq \mathit{beta}.\mathit{rnd}[a]$
$\Rightarrow\ \mathit{IsSafeAt}(\mathit{ProvedSafe}(Q,\ m,\ \mathit{beta}),\ m,\ \mathit{beta},\ \mathit{gamma})$

THEOREM $\mathit{Proposition}2$

3.8.5 Module $AbstractCFPaxos$

Module $AbstractCFPaxos$ provides the formal specification of the Abstract Collision-fast Paxos algorithm.

MODULE $AbstractCFPaxos$

Abstract algorithm

EXTENDS $PaxosConstants$

The algorithm's variables:
- proposed: set of proposed values
- learned: array that maps each learner to its currently learned $ValMap$
- bA: a ballot array that keeps current round and history of votes for each acceptor.
- pA: a proposal array that keeps the history of proposals for each proposer.
- $minTried$: a vector with the safe initial value to be accepted at each round.

VARIABLES $proposed$, $learned$, bA, pA, $minTried$

The type invariant asserts that the specification preserves the types of the variables according to the definition below.

$$\begin{aligned} TypeInv \triangleq\ & \land\ proposed \subseteq Value \\ & \land\ learned \in [Learner \to ValMap] \\ & \land\ bA \in BallotArray \\ & \land\ pA \in PropArray \\ & \land\ minTried \in [RNum \to ValMap \cup \{none\}] \end{aligned}$$

Initial state of the specification

$$\begin{aligned} Init \triangleq\ & \land\ proposed = \{\} \\ & \land\ learned = [l \in Learner \mapsto Bottom] \\ & \land\ bA = [vote \mapsto [a \in Acceptor \mapsto [m \in RNum \mapsto none]], \\ & \qquad\ rnd\ \mapsto [a \in Acceptor \mapsto Zero]] \\ & \land\ pA = [p\ \ \in Proposer \mapsto [i \in RNum \mapsto none]] \\ & \land\ minTried = [i \in RNum \mapsto \text{IF } i = Zero \text{ THEN } Bottom \\ & \qquad\qquad\qquad\qquad\qquad\qquad \text{ELSE } none\ ] \end{aligned}$$

$Propose(V)$ adds value V to the set *proposed* if it is not there yet.

$$\begin{aligned} & Propose(V) \triangleq \\ & \ \land\ V \notin proposed \\ & \ \land\ proposed' = proposed \cup \{V\} \\ & \ \land\ \text{UNCHANGED } \langle learned,\ bA,\ pA,\ minTried\rangle \end{aligned}$$

$JoinRound(a,\ m)$ changes the current round of acceptor a to m.

$$\begin{aligned} & JoinRound(a,\ m) \triangleq \\ & \ \land\ bA.rnd[a] \prec m \\ & \ \land\ bA' = [bA \text{ EXCEPT } !.rnd[a] = m] \\ & \ \land\ \text{UNCHANGED } \langle proposed,\ learned,\ pA,\ minTried\rangle \end{aligned}$$

StartRound$(m,\ Q)$ sets $minTried[m]$ to a value safe at m in bA, according to the definition of *ProvedSafe*$(Q,\ m,\ bA)$.

$$
\begin{array}{l}
StartRound(m,\ Q) \triangleq \\
\quad \land\ minTried[m] = none \\
\quad \land\ \forall\, a \in Q : m \preceq bA.rnd[a] \\
\quad \land\ minTried' = [minTried \text{ EXCEPT } ![m] = ProvedSafe(Q,\ m,\ bA)] \\
\quad \land \text{ UNCHANGED } \langle proposed,\ learned,\ bA,\ pA \rangle
\end{array}
$$

Suggest$(p,\ m,\ V)$ changes $pA[p][m]$ from none to a value or *Nil* (this last, only if other proposer has suggested a value or $minTried[m][p]$ equals *Nil*).

$$
\begin{array}{l}
Suggest(p,\ m,\ V) \triangleq \\
\quad \land\ \lor\ minTried[m] \notin \{Bottom,\ none\} \land V = minTried[m][p] \\
\qquad \lor\ V \in proposed \\
\qquad \lor\ \land\ V = Nil \\
\qquad\quad \land \exists\, pv \in CfProposer(m) : pA[pv][m] \in Value \\
\quad \land\ pA[p][m] = none \\
\quad \land\ pA' = [pA \text{ EXCEPT } ![p][m] = V] \\
\quad \land \text{ UNCHANGED } \langle proposed,\ learned,\ bA,\ minTried \rangle
\end{array}
$$

ClassicVote$(a,\ m,\ v)$ extends the vote of acceptor a for round m, changing it for value v if the conditions below are satisfied.

$$
\begin{array}{l}
ClassicVote(a,\ m,\ v) \triangleq \\
\quad \land\ bA.rnd[a] \preceq m \\
\quad \land\ IsValued(v) \\
\quad \land\ minTried[m] \neq none \\
\quad \land\ minTried[m] \sqsubseteq v \\
\quad \land \text{ LET } sp(p) \triangleq SM(p,\ pA[p][m]) \\
\qquad\qquad\ pS \triangleq \{p \in CfProposer(m) : pA[p][m] \neq none\} \\
\qquad\qquad\ maxTried \triangleq LUB(\{minTried[m] \bullet sp(p) : p \in pS\}) \\
\qquad \text{IN } \ v \sqsubseteq NilExtension(maxTried,\ Proposer \setminus CfProposer(m)) \\
\quad \land\ \lor\ bA.vote[a][m] = none \\
\qquad \lor\ bA.vote[a][m] \sqsubset v \\
\quad \land\ bA' = [bA \text{ EXCEPT } !.rnd[a] = m,\ !.vote[a][m] = v] \\
\quad \land \text{ UNCHANGED } \langle proposed,\ learned,\ pA,\ minTried \rangle
\end{array}
$$

AbstractLearn$(l,\ v)$ extends $learned[l]$ with v iff v is chosen in $\langle bA,\ pA \rangle$.

$$
\begin{array}{l}
AbstractLearn(l,\ v) \triangleq \\
\quad \land\ IsChosenIn(v,\ bA,\ pA) \\
\quad \land\ learned' = [learned \text{ EXCEPT } ![l] = learned[l] \sqcup v] \\
\quad \land \text{ UNCHANGED } \langle proposed,\ bA,\ pA,\ minTried \rangle
\end{array}
$$

Next defines the next-state relation and *Spec* is the complete specification.

$$
\begin{array}{l}
Next \triangleq \\
\quad \lor\ \ \exists\, V \in Value : Propose(V) \\
\quad \lor\ \ \exists\, a \in Acceptor,\ m \in RNum : JoinRound(a,\ m)
\end{array}
$$

$\lor \exists\, m \in RNum :$
$\quad \lor \exists\, Q \in Quorum(m) : StartRound(m,\, Q)$
$\quad \lor \exists\, p \in CfProposer(m),\; V \in Value \cup \{Nil\} : Suggest(p,\, m,\, V)$
$\lor \exists\, a \in Acceptor,\, m \in RNum,\, v \in ValMap : ClassicVote(a,\, m,\, v)$
$\lor \exists\, l \in Learner,\, v \in ValMap : AbstractLearn(l,\, v)$

$Spec \triangleq Init \land \Box[Next]_{\langle proposed,\, learned,\, bA,\, pA,\, minTried\rangle}$

The theorems below asserts that the spec ensures the type invariant and implements the general specification of $MConsensus$.

THEOREM $Spec \Rightarrow \Box\, TypeInv$

$MC \triangleq$ INSTANCE $MConsensus$
THEOREM $Spec \Rightarrow MC!Spec$

3.8.6 Module $DistAbsCFPaxos$

This module specifies our distributed abstract algorithm, which we later extend to our final specification.

MODULE $DistAbsCFPaxos$

EXTENDS $PaxosConstants$

The algorithm's variables are the same as in the abstract algorithm plus $msgs$: set of system messages. Since we are specifying only safety, message loss is simply implemented by not executing actions that depend on the message, without having to take it explicitly out of the set $msgs$. Moreover, duplicate messages are implemented by keeping messages in $msgs$, since they could possibly trigger the same action multiple times.

VARIABLES $proposed,\; learned,\; bA,\; pA,\; minTried,\; msgs$

Msg is the set containing all possible messages by the algorithm. For clarity, we use records instead of sequences to represent messages.

$Msg \triangleq [type : \{\text{“propose”}\},\, val : Value] \cup$
$\quad [type : \{\text{“1a”}\},\, rnd : RNum] \cup$
$\quad [type : \{\text{“1b”}\},\, rnd : RNum,\, acc : Acceptor,$
$\qquad\qquad vote : [RNum \to ValMap \cup \{none\}]] \cup$
$\quad [type : \{\text{“2S”}\},\, rnd : RNum,\, val : ValMap] \cup$
$\quad [type : \{\text{“2b”}\},\, rnd : RNum,\, acc : Acceptor,\, val : ValMap] \cup$
$\quad [type : \{\text{“2a”}\},\, rnd : RNum,\, val : SingleMap]$

Type Invariant

$TypeInv \triangleq \land\, proposed \subseteq Value$
$\qquad \land\, learned \in [Learner \to ValMap]$
$\qquad \land\, bA \in BallotArray$
$\qquad \land\, pA \in PropArray$
$\qquad \land\, minTried \in [RNum \to ValMap \cup \{none\}]$

$\land\ msgs \subseteq Msg$

Initial state

$$\begin{aligned} Init \triangleq\ & \land\ proposed = \{\} \\ & \land\ learned = [l \in Learner \mapsto Bottom] \\ & \land\ bA = [vote \mapsto [a \in Acceptor \mapsto [m \in RNum \mapsto none]], \\ & \qquad\ rnd \mapsto [a \in Acceptor \mapsto Zero]] \\ & \land\ pA = [p \in Proposer \mapsto [i \in RNum \mapsto none]] \\ & \land\ minTried = [i \in RNum \mapsto \text{IF } i = Zero \text{ THEN } Bottom \\ & \qquad\qquad\qquad\qquad\qquad\qquad \text{ELSE } none] \\ & \land\ msgs = \{[type \mapsto \text{“2S”},\ rnd \mapsto Zero,\ val \mapsto Bottom]\} \end{aligned}$$

Actions

Action $Send(msg)$ implements the sending of message msg.

$$Send(msg) \triangleq msgs' = msgs \cup \{msg\}$$

$Propose(V)$ executes a value proposal. In the specification we make no distinction between a proposal made by a collision-fast proposer and one made by an ordinary external proposer. The difference lies on whether the “propose” message will be local to a processor or not.

$$\begin{aligned} & Propose(V) \triangleq \\ & \land\ V \notin proposed \\ & \land\ proposed' = proposed \cup \{V\} \\ & \land\ Send([type \mapsto \text{“propose”},\ val \mapsto V]) \\ & \land\ \text{UNCHANGED } \langle learned,\ bA,\ pA,\ minTried\rangle \end{aligned}$$

Action $Phase1a(m)$ triggers the start of a new round m. It sends a phase “1a” message for round m.

$$\begin{aligned} & Phase1a(m) \triangleq \\ & \land\ minTried[m] = none \\ & \land\ Send([type \mapsto \text{“1a”},\ rnd \mapsto m]) \\ & \land\ \text{UNCHANGED } \langle proposed,\ learned,\ bA,\ pA,\ minTried\rangle \end{aligned}$$

Action $Phase1b(a,\ m)$ is executed by acceptor a in response to a phase “1a” message. The action is executed only once per round. It changes the current round of acceptor a to m and sends a phase “1b” message containing the current voting situation of a.

$$\begin{aligned} & Phase1b(a,\ m) \triangleq \\ & \land\ [type \mapsto \text{“1a”},\ rnd \mapsto m] \in msgs \\ & \land\ bA.rnd[a] \prec m \\ & \land\ bA' = [bA \text{ EXCEPT } !.rnd[a] = m] \\ & \land\ Send([type \mapsto \text{“1b”},\ rnd \mapsto m,\ acc \mapsto a,\ vote \mapsto bA.vote[a]]) \\ & \land\ \text{UNCHANGED } \langle proposed,\ learned,\ pA,\ minTried\rangle \end{aligned}$$

Action $Phase2Start(m)$ “starts” round m. It is enabled iff the round has not been started and a quorum of acceptors has sent phase “1b” messages for round m. It uses these “1b” messages to pick up a safe $ValMap$ for round m, using $ProvedSafe(Q,\ m,\ beta)$ as defined in module $PaxosConstants$. $minTried[m]$ is set to this safe value and a phase “2S” message is sent to inform it.

```
Phase2Start(m) ≜
 ∧ minTried[m] = none
 ∧ ∃ Q ∈ Quorum(m) :
    ∧ ∀ a ∈ Q : ∃ msg ∈ msgs : ∧ msg.type = "1b"
                               ∧ msg.rnd = m
                               ∧ msg.acc = a
    ∧ LET 1bMsg ≜ [a ∈ Q ↦ CHOOSE msg ∈ msgs :
                                ∧ msg.type = "1b"
                                ∧ msg.rnd = m
                                ∧ msg.acc = a]
          beta  ≜ [vote ↦ [a ∈ Q ↦ 1bMsg[a].vote],
                   rnd ↦ [a ∈ Q ↦ m]]
          v     ≜ ProvedSafe(Q, m, beta)
      IN  ∧ minTried' = [minTried EXCEPT ![m] = v]
          ∧ Send([type ↦ "2S", rnd ↦ m, val ↦ v])
 ∧ UNCHANGED ⟨proposed, learned, bA, pA⟩
```

Action $Phase2Prepare(p, m)$ is executed by proposer p, for round m. It is enabled iff $pA[p][m]$ is different from none and p has received a "2S" message containing a v-mapping different from $Bottom$. The action sets $pA[p][m]$ to $v[p]$.

```
Phase2Prepare(p, m) ≜
 ∧ pA[p][m] = none
 ∧ ∃ v ∈ ValMap :
    ∧ [type ↦ "2S", rnd ↦ m, val ↦ v] ∈ msgs
    ∧ v ≠ Bottom
    ∧ pA' = [pA EXCEPT ![p][m] = v[p]]
 ∧ UNCHANGED ⟨proposed, learned, bA, minTried, msgs⟩
```

Action $Phase2a(p, m, V)$ is executed by proposer p, for round m and value V. It is enabled iff p is a collision-fast proposer for m, it has received a phase "2S" message containing $Bottom$ and either a "propose" or a "2a" message for m, and $pA[p][m]$ equals none. The action sets $pA[p][m]$ to V if p received a ⟨"propose", V⟩ message or to Nil otherwise (p received a phase "2a" message from another proposer). It also sends a phase "2a" message for round m with a single mapping from p to V.

```
Phase2a(p, m, V) ≜
 ∧ p ∈ CfProposer(m)
 ∧ pA[p][m] = none
 ∧ [type ↦ "2S", rnd ↦ m, val ↦ Bottom] ∈ msgs
 ∧ ∨ [type ↦ "propose", val ↦ V] ∈ msgs
   ∨ ∧ V = Nil
     ∧ ∃ q ∈ CfProposer(m), U ∈ Value :
         [type ↦ "2a", rnd ↦ m, val ↦ SM(q, U)] ∈ msgs
 ∧ pA' = [pA EXCEPT ![p][m] = V]
 ∧ Send([type ↦ "2a", rnd ↦ m, val ↦ SM(p, V)])
 ∧ UNCHANGED ⟨proposed, learned, bA, minTried⟩
```

Action $Phase2b(a,\ m,\ v)$ is executed by acceptor a, for round m and $ValMap$ v. It is enabled only if m is higher than or equal to the current round of a, either v is valued and came on a phase "2S" message or v is built out of a phase "2a" message whose value is a mapping from a proposer to a (non-Nil) value. Moreover, the current vote of a for m must be either none or a prefix of v. The action sets the current round of a to m and a's vote at m to v.

$$
\begin{array}{l}
Phase2b(a,\ m,\ v) \triangleq \\
\quad \land\ bA.rnd[a] \preceq m \\
\quad \land\ \lor\ \land\ [type \mapsto \text{“2S”},\ rnd \mapsto m,\ val \mapsto v] \in msgs \\
\qquad\quad \land\ IsValued(v) \\
\qquad\quad \land\ bA.vote[a][m] = none \\
\qquad \lor\ \exists\, s \in SingleMap : \\
\qquad\quad \land\ [type \mapsto \text{“2a”},\ rnd \mapsto m,\ val \mapsto s] \in msgs \\
\qquad\quad \land\ s.v \neq Nil \\
\qquad\quad \land\ \lor\ \land\ bA.vote[a][m] = none \\
\qquad\qquad\quad \land\ v = NilExtension(Bottom \bullet s,\ Proposer \setminus CfProposer(m)) \\
\qquad\qquad \lor\ \land\ bA.vote[a][m] \neq none \\
\qquad\qquad\quad \land\ v = bA.vote[a][m] \bullet s \\
\quad \land\ bA' = [bA \text{ EXCEPT } !.rnd[a] = m,\ !.vote[a][m] = v] \\
\quad \land\ Send([type \mapsto \text{“2b”},\ rnd \mapsto m,\ acc \mapsto a,\ val \mapsto v]) \\
\quad \land\ \text{UNCHANGED } \langle proposed,\ learned,\ pA,\ minTried \rangle
\end{array}
$$

Action $Learn$ is executed by learner l, for a $ValMap$ v. Let P be the set of proposers from which l has received phase "2a" messages with single mappings from them to Nil. The action is enabled if there is a quorum Q of acceptors from which l has received phase "2b" messages such that v is a prefix of the of the values in each of these messages Nil-extended for P. The action sets $learned[l]$ to the lub between its previous value and v.

$$
\begin{array}{l}
Learn(l,\ v) \triangleq \\
\quad \land\ \exists\, m \in RNum : \\
\qquad \exists\, Q \in Quorum(m),\ P \in \text{SUBSET } CfProposer(m) : \\
\qquad\quad \land\ \forall\, p \in P : [type \mapsto \text{“2a”},\ rnd \mapsto m,\ val \mapsto SM(p,\ Nil)] \in msgs \\
\qquad\quad \land\ \forall\, a \in Q : \exists\, msg \in msgs : \land\ msg.type\ = \text{“2b”} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \land\ msg.rnd = m \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \land\ msg.acc\ = a \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \land\ v \sqsubseteq NilExtension(msg.val,\ P) \\
\quad \land\ learned' = [learned \text{ EXCEPT } ![l] = @ \sqcup v] \\
\quad \land\ \text{UNCHANGED } \langle proposed,\ bA,\ pA,\ minTried,\ msgs \rangle
\end{array}
$$

Next defines the next-state relation and *Spec* is the complete specification.

$$
\begin{array}{rl}
Next \triangleq & \lor\, \exists\, V \in Value : Propose(V) \\
& \lor\, \exists\, m \in RNum : Phase1a(m) \\
& \lor\, \exists\, a \in Acceptor,\ m \in RNum : Phase1b(a,\ m) \\
& \lor\, \exists\, m \in RNum : Phase2Start(m) \\
& \lor\, \exists\, p \in Proposer,\ m \in RNum,\ V \in Value \cup \{Nil\} : \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad Phase2a(p,\ m,\ V) \\
& \lor\, \exists\, a \in Acceptor,\ m \in RNum,\ v \in ValMap : Phase2b(a,\ m,\ v) \\
& \lor\, \exists\, l \in Learner,\ v \in ValMap : Learn(l,\ v)
\end{array}
$$

$Spec \triangleq Init \wedge \Box[Next]_{\langle proposed, learned, bA, pA, minTried, msgs\rangle}$

The theorems below asserts that the spec ensures the type invariant and implements the general specification of $MConsensus$.

THEOREM $Spec \Rightarrow \Box TypeInv$

$MC \triangleq$ INSTANCE $MConsensus$

THEOREM $Spec \Rightarrow MC!Spec$

3.8.7 Module $DistCFPaxosLiv$

This module completely specifies our Collision-fast Paxos protocol with its liveness requirementes.

MODULE $DistCFPaxosLiv$

EXTENDS $FiniteSets$ Standard module with basic operations for finite sets.

$RNum$ is the set of round numbers and $\preceq$ defines an ordering relation amongst the set of rounds. The module also has as a parameter an initial round number called $Zero$.

CONSTANTS $RNum$, $\_ \preceq \_$, $Zero$

We assume $\preceq$ is a total ordering of the set $RNum$ of round numbers. Module $OrderRelations$ can be found in our complete technical report.

ASSUME LET $PO \triangleq$ INSTANCE $OrderRelations$ WITH $S \leftarrow RNum$, $\sqsubseteq \leftarrow \preceq$

IN $PO!IsTotalOrder$

We define $i \prec j$ to be true iff $i \preceq j$ for two different rounds i and j.

$i \prec j \triangleq (i \preceq j) \wedge (i \neq j)$

If B is a set of round numbers that contains a maximum element, then $Max(B)$ is defined to equal that maximum. Otherwise, its value is unspecified.

$Max(B) \triangleq$ CHOOSE $i \in B : \forall j \in B : j \preceq i$

Are parameters of this module:
- A set $Proposer$ of proposer agents,
- A set $Learner$ of learner agents,
- A set $Coord$ of coordinator agents,
- An operator $CoordOf$ that returns the coordinator of a round,
- A set $Acceptor$ of acceptor agents,
- An operator $Quorum$ that returns the acceptor quorums of a round,
- An operator $CfProposer$ that returns the collision-fast proposers of a round,
- A set $Value$ of proposable Values,
- A special value Nil not in $Value$,
- And a special value none not in $Value$.

CONSTANTS $Proposer$, $Learner$, $Coord$, $CoordOf(\_)$, $Acceptor$,

$Quorum(\_),\ CfProposer(\_),\ Value,\ Nil,\ none$

ASSUME $Nil \notin Value$
ASSUME $none \notin Value$

The problem of $MConsensus$ is defined in terms of a value mapping set whose $Domain$ is the set of proposers

INSTANCE $VMapping$ WITH $Domain \leftarrow Proposer$

We make the assumption that, for every round r, r has a single coordinator responsible for it and every coordinator is is responsible for a round higher-numbered than r.

$$CoordAssumption \triangleq \forall\, r \in RNum : \land\, CoordOf(r) \in Coord \land \forall\, c \in Coord : \exists\, r2 \in RNum : \land\, r \prec r2 \land\, c = CoordOf(r2)$$

ASSUME $CoordAssumption$

We assume that quorums are finite subsets of the acceptors and every pair of quorums has a non-empty intersection.

$$QuorumAssumption \triangleq \forall\, i \in RNum : \land\, Quorum(i) \subseteq \text{SUBSET } Acceptor \land \forall\, Q \in Quorum(i) : IsFiniteSet(Q) \land \forall\, j \in RNum : \forall\, Q \in Quorum(i),\ R \in Quorum(j) : Q \cap R \neq \{\}$$

ASSUME $QuorumAssumption$

Over the set of collision-fast proposers for a round i, we only assume they are a finite subset of the proposers.

$$CfProposerAssumption \triangleq \forall\, i \in RNum : \land\, CfProposer(i) \in \text{SUBSET } Proposer \land\, IsFiniteSet(CfProposer(i))$$

ASSUME $CfProposerAssumption$

We say that a value mapping is valued iff at least one element of its domain is mapped to a value ($\neq Nil$). Our algorithm makes sure that acceptors will only accept valued mappings, so that we can guarantee $Nontriviality$.

$$IsValued(m) \triangleq \exists\, p \in Dom(m) : m[p] \neq Nil$$

We define the Nil-extension of a valued mapping v for a set P of proposers to be the $ValMap$ resulting from adding to v the single mapping $p \rightarrow Nil$, for every proposer p in $P \setminus Dom(v)$.

$$NilExtension(v,\ P) \triangleq \text{IF } v = none \text{ THEN } none \text{ ELSE } [p \in Dom(v) \cup P \mapsto \text{IF } p \in Dom(v) \text{ THEN } v[p] \text{ ELSE } Nil]$$

The algorithm's variables are the following:

- proposed: set of proposed values
- learned: array mapping each learner to its currently learned $ValMap$
- rnd: array mapping each acceptor to its current round.
- $vrnd$: array mapping each acceptor to the last round at which it accepted something.
- $vval$: array mapping each acceptor a to the $ValMap$ accepted in $vrnd[a]$.
- $prnd$: array mapping each proposer to its current round.
- $pval$: array mapping each proposer p to the value it fast-proposed at round $prnd[p]$.
- $crnd$: array mapping each coordinator to its current round.
- $cval$: array mapping each coordinator to the initial $ValMap$ it has picked for round $crnd[p]$.
- $msgs$: set of messages that implements the message passing subsystem
- $noncrashed$: set of non-crashed agents in the system.
- $amLeader$: array mapping each coordinator to TRUE or FALSE depending on whether it believes to be the leader or not
- $activep$: array mapping each coordinator to the set of proposers it currently believes to be active.

VARIABLES $proposed,\ learned,\ rnd,\ vrnd,\ vval,\ prnd,\ pval,\ crnd,\ cval,$
$msgs,\ noncrashed,\ amLeader,\ activep$

$$
\begin{aligned}
aVars &\triangleq \langle rnd,\ vrnd,\ vval\rangle\\
pVars &\triangleq \langle prnd,\ pval\rangle\\
cVars &\triangleq \langle crnd,\ cval\rangle\\
oVars &\triangleq \langle proposed,\ learned,\ noncrashed,\ amLeader,\ activep\rangle
\end{aligned}
$$

Msg is the set containing all possible messages sent by the algorithm. For clarity, we use records instead of sequences to represent messages.

$$
\begin{aligned}
Msg \triangleq\ & [type : \{\text{“propose”}\},\ val : Value] \cup\\
& [type : \{\text{“1a”}\},\ rnd : RNum] \cup\\
& [type : \{\text{“1b”}\},\ rnd : RNum,\ acc : Acceptor,\\
& \qquad vrnd : RNum,\ vval : ValMap \cup \{none\}] \cup\\
& [type : \{\text{“2S”}\},\ rnd : RNum,\ val : ValMap] \ \cup\\
& [type : \{\text{“2b”}\},\ rnd : RNum,\ acc : Acceptor,\ val : ValMap] \cup\\
& [type : \{\text{“2a”}\},\ rnd : RNum,\ val : SingleMap]
\end{aligned}
$$

Type Invariant

$$
\begin{aligned}
&TypeInv \triangleq\\
&\quad \land\ proposed \subseteq Value\\
&\quad \land\ learned\ \in [Learner \to ValMap]\\
&\quad \land\ rnd\ \in [Acceptor \to RNum]\\
&\quad \land\ vrnd \in [Acceptor \to RNum]\\
&\quad \land\ vval\ \in [Acceptor \to ValMap \cup \{none\}]\\
&\quad \land\ prnd \in [Proposer \to RNum]\\
&\quad \land\ pval\ \in [Proposer \to Value \cup \{Nil\} \cup \{none\}]\\
&\quad \land\ crnd \in [Coord \to RNum]\\
&\quad \land\ cval\ \in [Coord \to ValMap \cup \{none\}]\\
&\quad \land\ msgs \subseteq Msg\\
&\quad \land\ noncrashed \subseteq Acceptor \cup Coord \cup Proposer \cup Learner\\
&\quad \land\ amLeader\ \in [Coord \to \text{BOOLEAN}\ ]\\
&\quad \land\ activep \in [Coord \to \text{SUBSET}\ Proposer]
\end{aligned}
$$

Initial state

$$
\begin{aligned}
Init \triangleq\ & \wedge proposed = \{\} \\
& \wedge learned = [l \in Learner \mapsto Bottom] \\
& \wedge rnd = [a \in Acceptor \mapsto Zero] \\
& \wedge vrnd = [a \in Acceptor \mapsto Zero] \\
& \wedge vval = [a \in Acceptor \mapsto none] \\
& \wedge prnd = [p \in Proposer \mapsto Zero] \\
& \wedge pval = [p \in Proposer \mapsto none] \\
& \wedge crnd = [c \in Coord \mapsto Zero] \\
& \wedge cval = [c \in Coord \mapsto \text{IF } c = CoordOf(Zero) \\
& \qquad\qquad \text{THEN } Bottom \\
& \qquad\qquad \text{ELSE } none] \\
& \wedge msgs = \{\} \\
& \wedge noncrashed = Acceptor \cup Coord \cup Proposer \cup Learner \\
& \wedge amLeader = [c \in Coord \mapsto \text{IF } c = CoordOf(Zero) \\
& \qquad\qquad \text{THEN TRUE} \\
& \qquad\qquad \text{ELSE FALSE}] \\
& \wedge activep = [c \in Coord \mapsto Proposer]
\end{aligned}
$$

Agent Actions

Action $Send(msg)$ implements the sending of message msg.

$$Send(msg) \triangleq msgs' = msgs \cup \{msg\}$$

$Propose(V)$ executes a value proposal. In the specification we make no distinction between a proposal made by a collision-fast proposer and one made by an ordinary external proposer. The difference lies on whether the "propose" message will be local to a processor or not.

$$
\begin{aligned}
& Propose(V) \triangleq \\
& \quad \wedge V \notin proposed \\
& \quad \wedge proposed' = proposed \cup \{V\} \\
& \quad \wedge Send([type \mapsto \text{"propose"}, val \mapsto V]) \\
& \quad \wedge \text{UNCHANGED } \langle aVars, pVars, cVars, learned, noncrashed, amLeader, activep \rangle
\end{aligned}
$$

Action $Phase1a(c, r)$ is executed by the coordinator c of round r as specified in the paper. To ensure Liveness, c can only execute this action if it believes to be the leader and either c has received a message related to a round between $crnd[c]$ and r, or it suspects one of the current collision-fast proposers to have failed.

$$
\begin{aligned}
& Phase1a(c, r) \triangleq \\
& \quad \wedge amLeader[c] \\
& \quad \wedge c = CoordOf(r) \\
& \quad \wedge crnd[c] \prec r \\
& \quad \wedge \vee \exists msg \in msgs \setminus [type : \{\text{"propose"}\}, val : Value] : \\
& \qquad\quad \wedge crnd[c] \prec msg.rnd \\
& \qquad\quad \wedge msg.rnd \prec r \\
& \qquad \vee \wedge \neg(CfProposer(crnd[c]) \subseteq activep[c])
\end{aligned}
$$

```
        ∧ CfProposer(r) ⊆ activep[c]
   ∧ crnd' = [crnd EXCEPT ![c] = r]
   ∧ cval' = [cval EXCEPT ![c] = none]
   ∧ Send([type ↦ "1a", rnd ↦ r])
   ∧ UNCHANGED ⟨aVars, pVars, oVars⟩
```

Action $Phase1b(a, r)$ is executed by acceptor a, for round r.

```
Phase1b(a, r) ≜
   ∧ [type ↦ "1a", rnd ↦ r] ∈ msgs
   ∧ rnd[a] ≺ r
   ∧ rnd' = [rnd EXCEPT ![a] = r]
   ∧ Send([type ↦ "1b", rnd ↦ r, acc ↦ a,
                        vrnd ↦ vrnd[a], vval ↦ vval[a]])
   ∧ UNCHANGED ⟨vrnd, vval, pVars, cVars, oVars⟩
```

$DistProvedSafe(Q, r, 1bMsg)$ returns a safe initial v-mapping for round r based on the "1b" messages for r sent by acceptors in quorum Q. It returns $Bottom$ if no v-mapping has been or might be chosen in a lower-numbered round or a complete v-mapping that extends any v-mapping possibly chosen in a lower-numbered round.

```
DistProvedSafe(Q, r, 1bMsg) ≜
  IF ∀ a ∈ Q : 1bMsg[a].vval = none
     THEN Bottom
     ELSE LET k  ≜ Max({1bMsg[a].vrnd : a ∈ Q})
              AS ≜ {a ∈ Q : ∧ 1bMsg[a].vrnd = k
                            ∧ 1bMsg[a].vval ≠ none}
              S  ≜ {1bMsg[a].vval : a ∈ AS}
          IN  NilExtension(LUB(S), Proposer)
```

The action $Phase2Start(c, r)$ follows the description given in the algorithm description. However, it is only executed if c believes to be the current leader.

```
Phase2Start(c, r) ≜
   ∧ crnd[c] = r
   ∧ cval[c] = none
   ∧ amLeader[c]
   ∧ ∃ Q ∈ Quorum(r) :
       ∧ ∀ a ∈ Q : ∃ msg ∈ msgs : ∧ msg.type = "1b"
                                  ∧ msg.rnd  = r
                                  ∧ msg.acc  = a
       ∧ LET 1bMsg ≜ [a ∈ Q ↦ CHOOSE msg ∈ msgs :
                                  ∧ msg.type = "1b"
                                  ∧ msg.rnd  = r
                                  ∧ msg.acc  = a]
             v     ≜ DistProvedSafe(Q, r, 1bMsg)
         IN  ∧ cval' = [cval EXCEPT ![c] = v]
             ∧ Send([type ↦ "2S", rnd ↦ r, val ↦ v])
```

$\land$ UNCHANGED $\langle aVars,\ pVars,\ crnd,\ oVars\rangle$

$Phase2Prepare(p,\ r)$ simply follows the algorithm description.

$$
\begin{array}{l}
Phase2Prepare(p,\ r)\ \triangleq \\
\quad \land\ prnd[p] \prec r \\
\quad \land\ \exists\, v \in ValMap : \\
\qquad \land\ [type \mapsto \text{“2S”},\ rnd \mapsto r,\ val \mapsto v] \in msgs \\
\qquad \land\ \lor\ \land\ v = Bottom \\
\qquad\qquad\ \land\ pval' = [pval \text{ EXCEPT } ![p] = none] \\
\qquad\quad \lor\ \land\ v \neq Bottom \\
\qquad\qquad\ \land\ pval' = [pval \text{ EXCEPT } ![p] = v[p]] \\
\quad \land\ prnd' = [prnd \text{ EXCEPT } ![p] = r] \\
\quad \land \text{ UNCHANGED } \langle aVars,\ cVars,\ oVars\rangle
\end{array}
$$

Action $Phase2a(p,\ r,\ V)$ also just follows its previous description.

$$
\begin{array}{l}
Phase2a(p,\ r,\ V)\ \triangleq \\
\quad \land\ prnd[p] = r \\
\quad \land\ p \in CfProposer(r) \\
\quad \land\ pval[p] = none \\
\quad \land\ \lor\ [type \mapsto \text{“propose”},\ val \mapsto V] \in msgs \\
\qquad \lor\ \land\ V = Nil \\
\qquad\quad \land\ \exists\, q \in CfProposer(r),\ U \in Value : \\
\qquad\qquad [type \mapsto \text{“2a”},\ rnd \mapsto r,\ val \mapsto SM(q,\ U)] \in msgs \\
\quad \land\ pval' = [pval \text{ EXCEPT } ![p] = V] \\
\quad \land\ Send([type \mapsto \text{“2a”},\ rnd \mapsto r,\ val \mapsto SM(p,\ V)]) \\
\quad \land \text{ UNCHANGED } \langle prnd,\ aVars,\ cVars,\ oVars\rangle
\end{array}
$$

The same for action $Phase2b(a,\ r)$.

$$
\begin{array}{l}
Phase2b(a,\ r)\ \triangleq \\
\quad \land\ rnd[a] \preceq r \\
\quad \land\ \exists\, v \in ValMap : \\
\qquad \land\ \lor\ \land\ [type \mapsto \text{“2S”},\ rnd \mapsto r,\ val \mapsto v] \in msgs \\
\qquad\qquad\ \land\ IsValued(v) \\
\qquad\qquad\ \land\ vrnd[a] \prec r \lor vval[a] = none \\
\qquad\quad \lor\ \exists\, s \in SingleMap : \\
\qquad\qquad\ \land\ [type \mapsto \text{“2a”},\ rnd \mapsto r,\ val \mapsto s] \in msgs \\
\qquad\qquad\ \land\ s.v \neq Nil \\
\qquad\qquad\ \land\ \lor\ \land\ vrnd[a] \prec r \lor vval[a] = none \\
\qquad\qquad\qquad\ \land\ v = NilExtension(Bottom \bullet s,\ Proposer \setminus CfProposer(r)) \\
\qquad\qquad\quad \lor\ \land\ vrnd[a] = r \land vval[a] \neq none \\
\qquad\qquad\qquad\ \land\ v = vval[a] \bullet s \\
\qquad \land\ vval' = [vval \text{ EXCEPT } ![a] = v] \\
\qquad \land\ rnd' = [rnd \text{ EXCEPT } ![a] = r] \\
\qquad \land\ vrnd' = [vrnd \text{ EXCEPT } ![a] = r]
\end{array}
$$

$\land\ Send([type \mapsto \text{“2b”},\ rnd \mapsto r,\ acc \mapsto a,\ val \mapsto v])$
$\land$ UNCHANGED $\langle pVars,\ cVars,\ oVars\rangle$

The $Learn$ action is defined differently from the explanation in the algorithm description. Here, for simplicity, we let the value being merged with $learned[l]$ be an action parameter.

$$
\begin{array}{l}
Learn(l,\ v)\ \triangleq \\
\quad \land\ \exists\, r \in RNum : \\
\qquad \exists\, Q \in Quorum(r),\ P \in \text{SUBSET } CfProposer(r) : \\
\qquad\quad \land\ \forall\, p \in P : [type \mapsto \text{“2a”},\ rnd \mapsto r,\ val \mapsto SM(p,\ Nil)] \in msgs \\
\qquad\quad \land\ \forall\, a \in Q : \exists\, msg \in msgs : \land\ msg.type = \text{“2b”} \\
\qquad\qquad\qquad\qquad\qquad\qquad\ \land\ msg.rnd\ = r \\
\qquad\qquad\qquad\qquad\qquad\qquad\ \land\ msg.acc\ = a \\
\qquad\qquad\qquad\qquad\qquad\qquad\ \land\ v \sqsubseteq NilExtension(msg.val,\ P) \\
\quad \land\ learned' = [learned \text{ EXCEPT } ![l] = @ \sqcup v] \\
\quad \land \text{ UNCHANGED } \langle aVars,\ pVars,\ cVars,\ proposed,\ msgs, \\
\qquad\qquad\qquad noncrashed,\ amLeader,\ activep\rangle
\end{array}
$$

Message Loss/Retransmission Actions

The following operator returns the last message sent by coordinator c

$$
\begin{array}{l}
CoordLastMsg(c)\ \triangleq \\
\quad \text{IF } cval[c] = none \\
\qquad \text{THEN } [type \mapsto \text{“1a”},\ rnd \mapsto crnd[c]] \\
\qquad \text{ELSE } [type \mapsto \text{“2S”},\ rnd \mapsto crnd[c],\ val \mapsto cval[c]]
\end{array}
$$

The following operator returns the last message sent by proposer p. It is sound only if $pval[p] \neq none$, $crnd[CoordOf(prnd[p])] = prnd[p]$, and $cval[CoordOf(prnd[p])] = Bottom$. This condition is true only if p has fast-proposed a value at round $prnd[p]$.

$$
\begin{array}{l}
ProposerLastMsg(p)\ \triangleq \\
\quad [type \mapsto \text{“2a”},\ rnd \mapsto prnd[p],\ val \mapsto SM(p,\ pval[p])]
\end{array}
$$

The following operator returns the last message sent by acceptor a.

$$
\begin{array}{l}
AcceptorLastMsg(a)\ \triangleq \\
\quad \text{IF } vrnd[a] = rnd[a] \\
\qquad \text{THEN } [type\ \mapsto \text{“2b”},\ rnd \mapsto rnd[a],\ acc \mapsto a,\ val \mapsto vval[a]] \\
\qquad \text{ELSE } [type\ \mapsto \text{“1b”},\ rnd \mapsto rnd[a],\ acc \mapsto a, \\
\qquad\qquad\quad vrnd \mapsto vrnd[a],\ vval \mapsto vval[a]]
\end{array}
$$

$LoseMsg(m)$ implements the loss of message m. Any message may be lost except for: - a “propose” message: Indeed the algorithm is resilient to their loss, but implementing
retransmission of a “propose” message would make our specification needlessly more complicated.
- the last message sent by a good coordinator that believes to be the leader
- the fast-proposal sent by a good proposer for its current round.
- the last message sent by a good acceptor.
Therefore, the algorithm only requires that such messages be always available for the agents they were sent to.

$$LoseMsg(m)\ \triangleq$$

$\land \neg \lor m.type \in \{$"propose"$\}$
  $\lor \land m.type \in \{$"1a", "2S"$\}$
    $\land m = CoordLastMsg(CoordOf(m.rnd))$
    $\land CoordOf(m.rnd) \in noncrashed$
    $\land amLeader[CoordOf(m.rnd)]$
  $\lor \land m.type \in \{$"2a"$\}$
    $\land pval[m.val.p] \neq none$
    $\land crnd[CoordOf(m.rnd)] = m.rnd$
    $\land cval[CoordOf(m.rnd)] = Bottom$
    $\land m = ProposerLastMsg(m.val.p)$
    $\land m.val.p \in noncrashed$
  $\lor \land m.type \in \{$"1b", "2b"$\}$
    $\land m = AcceptorLastMsg(m.acc)$
    $\land m.acc \in noncrashed$
$\land msgs' = msgs \setminus \{m\}$
$\land$ UNCHANGED $\langle aVars, cVars, pVars, oVars\rangle$

Even though the $LoseMsg(m)$ action above does not apply for some messages, retransmission is necessary because an agent can fail, have its message lost, and then recover. In this case it is necessary to re-send the lost message. Below you will find the actions responsible for retransmission.

$CoordRetransmit(c) \triangleq$
$\land amLeader[c]$
$\land Send(CoordLastMsg(c))$
$\land$ UNCHANGED $\langle aVars, cVars, pVars, oVars\rangle$

$AcceptorRetransmit(a) \triangleq$
$\land Send(AcceptorLastMsg(a))$
$\land$ UNCHANGED $\langle aVars, cVars, pVars, oVars\rangle$

$ProposerRetransmit(p) \triangleq$
$\land pval[p] \neq none$
$\land crnd[CoordOf(prnd[p])] = prnd[p]$
$\land cval[CoordOf(prnd[p])] = Bottom$
$\land Send(ProposerLastMsg(p))$
$\land$ UNCHANGED $\langle aVars, cVars, pVars, oVars\rangle$

Other Actions

Action $LeaderSelection$ allows an arbitrary change to the values of $amLeader[c]$, for all coordinators c. Since this action may be performed at any time, the specifiction makes no assumption about the outcome of leader selection. (However, progress is guaranteed only under an assumption about the values of $amLeader[c]$.)

$LeaderSelection \triangleq$
$\land amLeader' \in [Coord \to$ BOOLEAN $]$
$\land$ UNCHANGED $\langle aVars, cVars, pVars, proposed, learned,$
$noncrashed, activep, msgs\rangle$

Action *SuspectOrTrust* arbitrarily changes the values of $activep[c]$, for all coordinators c.

$$
\begin{array}{l}
SuspectOrTrust \triangleq \\
\quad \land\ activep' \in [Coord \to \text{SUBSET } Proposer] \\
\quad \land\ \text{UNCHANGED } \langle aVars,\ cVars,\ pVars,\ proposed,\ learned, \\
\qquad\qquad\qquad noncrashed,\ amLeader,\ msgs\rangle
\end{array}
$$

Action *FailOrRecover* also allows an arbitrary change to the value of $noncrashed$.

$$
\begin{array}{l}
FailOrRecover \triangleq \\
\quad \land\ noncrashed' \in \text{SUBSET } (Acceptor \cup Coord \cup Proposer \cup Learner) \\
\quad \land\ \text{UNCHANGED } \langle aVars,\ cVars,\ pVars,\ proposed,\ learned, \\
\qquad\qquad\qquad amLeader,\ activep,\ msgs\rangle
\end{array}
$$

Final Specification

CoordNext(c) specifies the execution of some action by coordinator c.

$$
\begin{array}{l}
CoordNext(c) \triangleq \\
\quad \lor \exists\, r \in RNum : \lor\ Phase1a(c,\ r) \\
\qquad\qquad\qquad\qquad \lor\ Phase2Start(c,\ r) \\
\quad \lor\ CoordRetransmit(c)
\end{array}
$$

ProposerNext(p) specifies the execution of some action by proposer p.

$$
\begin{array}{l}
ProposerNext(p) \triangleq \\
\quad \lor \exists\, r \in RNum,\ V \in Value \cup \{Nil\} : Phase2a(p,\ r,\ V) \\
\quad \lor\ ProposerRetransmit(p)
\end{array}
$$

AcceptorNext(a) specifies the execution of some action by acceptor a.

$$
\begin{array}{l}
AcceptorNext(a) \triangleq \\
\quad \lor \exists\, r \in RNum : \lor\ Phase1b(a,\ r) \\
\qquad\qquad\qquad\qquad \lor\ Phase2b(a,\ r) \\
\quad \lor\ AcceptorRetransmit(a)
\end{array}
$$

LearnerNext(a) specifies the execution of some action by learner l.

$$
\begin{array}{l}
LearnerNext(l) \triangleq \\
\quad \exists\, v \in ValMap : Learn(l,\ v)
\end{array}
$$

Next defines the next-state action of the specification.

$$
\begin{array}{lll}
Next \triangleq & \lor \exists\, V & \in Value : Propose(V) \\
& \lor \exists\, c & \in Coord \quad \cap noncrashed : CoordNext(c) \\
& \lor \exists\, p & \in Proposer \cap noncrashed : ProposerNext(p) \\
& \lor \exists\, a & \in Acceptor \cap noncrashed : AcceptorNext(a) \\
& \lor \exists\, l & \in Learner \cap noncrashed : LearnerNext(l) \\
& \lor \exists\, m & \in msgs : LoseMsg(m) \\
& \lor LeaderSelection \lor SuspectOrTrust \lor FailOrRecover &
\end{array}
$$

$vars \triangleq \langle aVars,\ pVars,\ cVars,\ oVars,\ msgs \rangle$

$Fairness \triangleq$
$\land \forall\, c \in Coord :$
$\quad \land \mathbf{WF}_{vars}(c \in noncrashed \land CoordNext(c))$
$\quad \land \mathbf{WF}_{vars}(c \in noncrashed \land (\exists\, r \in RNum : Phase1a(c,\ r)))$
$\land \forall\, p \in Proposer : \mathbf{WF}_{vars}(p \in noncrashed \land ProposerNext(p))$
$\land \forall\, a \in Acceptor : \mathbf{WF}_{vars}(a \in noncrashed \land AcceptorNext(a))$
$\land \forall\, l \in Learner : \mathbf{WF}_{vars}(l \in noncrashed \land LearnerNext(l))$

$Spec \triangleq Init \land \Box[Next]_{vars} \land Fairness$

The theorems below asserts that the spec ensures the type invariant and implements the safety part of the *MConsensus* specification.

THEOREM $Spec \Rightarrow \Box TypeInv$

$MC \triangleq$ INSTANCE $MConsensus$
THEOREM $Spec \Rightarrow MC!Spec$

$LA(l,\ c,\ Q)$ defines the liveness assumption required by the algorithm. Since our specification does not allow a "propose" message to be lost, LA is slightly simpler than what we described in the paper.

$LA(l,\ c,\ Q) \triangleq$
$\quad \land \{c,\ l\} \cup Q \subseteq noncrashed$
$\quad \land proposed \neq \{\}$
$\quad \land \forall\, c2 \in Coord : amLeader[c2] \equiv (c = c2)$
$\quad \land activep[c] \subseteq noncrashed$
$\quad \land \forall\, r \in RNum :$
$\qquad \exists\, r2 \in RNum : \land r \prec r2$
$\qquad\qquad\qquad\qquad \land c = CoordOf(r2)$
$\qquad\qquad\qquad\qquad \land CfProposer(r2) \subseteq activep[l]$
$\quad \land activep[c] \subseteq activep[c]'$

The theorem below asserts that the algorithm's specification satisfies Liveness if the liveness assumption eventually holds forever.

THEOREM $\forall\, l \in Learner :$
$\qquad \land Spec$
$\qquad \land \exists\, Q \in$ SUBSET $Acceptor :$
$\qquad\quad \land \forall\, r \in RNum : Q \in Quorum(r)$
$\qquad\quad \land \exists\, c \in Coord : \Diamond\Box[LA(l,\ c,\ Q)]_{vars}$
$\qquad \Rightarrow \Diamond(\text{DOMAIN } learned[l] = Proposer)$

3.8.8 Module *CFPaxosSAgreement*

Module *CFPaxosSAgreement* specifies our complete collision-fast sequence agreement algorithm based on Collision-fast Paxos.

MODULE *CFPaxosSAgreement*

This module presents the specification of an efficient collision-fast sequence agreement algorithm based on the Collision-fast *Paxos* algorithm for M-Consensus.

EXTENDS *Naturals*, *Sequences*, *FiniteSets* Definitions concerning the natural numbers and sequences.

RNum is the set of round numbers and $\preceq$ defines an ordering relation amongst the set of rounds. The module also has as a parameter an initial round number called *Zero*.

CONSTANTS $RNum$, $\_ \preceq \_$, $Zero$

The definition of $i \prec j$

$$i \prec j \triangleq (i \preceq j) \land (i \neq j)$$

The specification has the same parameters as the Collision-Fast *Paxos* algorithm.

CONSTANTS $Proposer$, $Learner$, $Coord$, $CoordOf(\_)$, $Acceptor$, $Quorum(\_)$, $CfProposer(\_)$, $Value$, Nil, $none$

The Sequence Agreement algorithm is based on an infinite number of Collision-fast *Paxos* (*CFPaxos*) instances. However, some variables such as proposed, *rnd*, *prnd*, *crnd*, *noncrashed*, *amLeader*, and *activep* are shared by all instances. The other variables are implemented by arrays indexed by the instance. We represent this arrays by the original variable name with the prefix x as it can be seen below.

VARIABLES $proposed$, $xlearned$, rnd, $xvrnd$, $xvval$, $prnd$, $xpval$, $crnd$, $xcval$, $xmsgs$, $noncrashed$, $amLeader$, $activep$

vars is a sequence containing all the specification variables.

$$vars \triangleq \langle proposed, xlearned, rnd, xvrnd, xvval, prnd, xpval, crnd, xcval, xmsgs, noncrashed, amLeader, activep \rangle$$

$CFP(i)$, where i is a Natural number, is the Collision-fast *Paxos* instance number i.

$$\begin{aligned} CFP(i) \triangleq \text{INSTANCE } DistCFPaxosLiv \text{ WITH } & learned \leftarrow xlearned[i], \\ & vrnd \leftarrow xvrnd[i], \\ & vval \leftarrow xvval[i], \\ & pval \leftarrow xpval[i], \\ & msgs \leftarrow xmsgs[i], \\ & cval \leftarrow xcval[i] \end{aligned}$$

Initial state of the specification

$$\begin{aligned} Init \triangleq & \land proposed = \{\} \\ & \land xlearned = [i \in Nat \mapsto [l \in Learner \mapsto CFP(i)!Bottom]] \\ & \land rnd = [a \in Acceptor \mapsto Zero] \\ & \land xvrnd = [i \in Nat \mapsto [a \in Acceptor \mapsto Zero]] \\ & \land xvval = [i \in Nat \mapsto [a \in Acceptor \mapsto none]] \end{aligned}$$

$$
\begin{array}{l}
\land\, prnd \;\; = [p \in Proposer \mapsto Zero] \\
\land\, xpval \;\; = [i \in Nat \mapsto [p \in Proposer \mapsto none]] \\
\land\, crnd \;\; = [c \in Coord \;\; \mapsto Zero] \\
\land\, xcval \;\; = [i \in Nat \mapsto [c \in Coord \;\; \mapsto \text{IF } c = CoordOf(Zero) \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \text{THEN } CFP(i)!Bottom \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \text{ELSE } none]] \\
\land\, xmsgs = [i \in Nat \mapsto \{\}] \\
\land\, noncrashed = Acceptor \cup Coord \cup Proposer \cup Learner \\
\land\, amLeader \;\; = [c \in Coord \mapsto \text{IF } c = CoordOf(Zero) \\
\qquad\qquad\qquad\qquad\qquad\quad \text{THEN TRUE} \\
\qquad\qquad\qquad\qquad\qquad\quad \text{ELSE FALSE}] \\
\land\, activep \;\;\;\; = [c \in Coord \mapsto Proposer]
\end{array}
$$

Some actions of the algorithm have to do only with one instance of $CFPaxos$. For such cases it is interesting to have an operator that keeps non-shared variables of other instances unchanged.

$$
\begin{array}{l}
InstanceUnchanged(i) \triangleq \\
\quad \text{UNCHANGED } \langle xlearned[i],\, xvrnd[i],\, xvval[i], \\
\qquad\qquad\qquad xpval[i],\, xcval[i],\, xmsgs[i] \rangle
\end{array}
$$

Action $Phase1a(c,\, r)$ for instance i must be slightly changed so that a new round might be started if there is an interfering message for a different round number in ANY of the running instances.

$$
\begin{array}{l}
NewPhase1a(i,\, c,\, r) \triangleq \\
\quad \land\, amLeader[c] \\
\quad \land\, c = CoordOf(r) \\
\quad \land\, crnd[c] \prec r \\
\quad \land\, \lor \exists\, j \;\; \in Nat : \\
\qquad\quad \exists\, msg \in xmsgs[j] : \\
\qquad\qquad \land\, msg.type \neq \text{“propose”} \\
\qquad\qquad \land\, crnd[c] \;\; \prec msg.rnd \\
\qquad\qquad \land\, msg.rnd \prec r \\
\qquad \lor \land \neg(CfProposer(crnd[c]) \subseteq activep[c]) \\
\qquad\quad \land\, CfProposer(r) \subseteq activep[c] \\
\quad \land\, crnd' = [crnd \text{ EXCEPT } ![c] = r] \\
\quad \land\, xcval'[i] = [xcval[i] \text{ EXCEPT } ![c] = none] \\
\quad \land\, CFP(i)!Send([type \mapsto \text{“1a”},\, rnd \mapsto r]) \\
\quad \land \text{ UNCHANGED } \langle CFP(i)!aVars,\, CFP(i)!pVars,\, CFP(i)!oVars \rangle
\end{array}
$$

Agent Actions

We now present the combinated actions each agent performs.

A *Propose* action sends a “propose” message that is valid for all the $CFPaxos$ instances. This is logically implemented by executing a propose action for each instance. Notice, however, that the multiple logical “propose” messages are implemented by a single one in practice.

$Propose(V) \triangleq$
$\forall\, i \in Nat : CFP(i)!Propose(V)$

Action $Phase1a$ executes $NewPhase1a$ specified above for all $CFPaxos$ instances. It is easy to see that, since all pre-conditions of the action are based on variables shared among all instances, the action is enable for instance i iff it is enabled for instance j. As a result of this action, a "1a" message is sent for each instance. In practice, a single "1a" message is sent and it is interpreted as valid for all instances.

$Phase1a(c,\, r) \triangleq$
$\forall\, i \in Nat : NewPhase1a(i,\, c,\, r)$

Action $Phase1b(a,\, r)$ is executed when acceptor a receives the "1a" message for a higher-numbered round than its current one. Since there is a logical message for each instance, the $Phase1b$ action of every instance is equally enabled and are executed. However, different instances might generate different "1b" messages. These different logical messages are sent on the same physical one. The size of this message can be limited because only a finite number of (initial) instances will result on "1b" messages different from $\langle$"$1b$",\, r,\, a,\, $Zero$,\, $none\rangle$.

$Phase1b(a,\, r) \triangleq$
$\forall\, i \in Nat : CFP(i)!Phase1b(a,\, r)$

Action $Phase2Start(c,\, r)$ is executed by the coordinator of r when it receives the composite "$1b$" message of the previous action. The coordinator calculates the different "$2S$" messages for every instance. These "$2S$" messages are sent together in the same physical one as we have done in the previous action. Recall that only a finite number of instances will result on "$2S$" messages different from $\langle$"$2S$",\, r,\, $Bottom\rangle$, which can be used to limit the size of the composite "$2S$" message sent.

$Phase2Start(c,\, r) \triangleq$
$\forall\, i \in Nat : CFP(i)!Phase2Start(c,\, r)$

Action $Phase2Prepare(p,\, r)$ is executed by proposer p when it receives the composite "$2S$" message from the action above. The action executes $Phase2Prepare$ for every $CFPaxos$ instance.

$Phase2Prepare(p,\, r) \triangleq$
$\forall\, i \in Nat : CFP(i)!Phase2Prepare(p,\, r)$

Action $Phase2a(p,\, r,\, V)$ is executed by proposer p when it fast-proposes value V. It executes $Phase2a(p,\, r,\, V)$ for some $CFPaxos$ instance i, as long as p has not fast-proposed the same value for a different instance. All the other instances are left unchanged.

$Phase2a(p,\, r,\, V) \triangleq$
$\exists\, i \in Nat :$
$\quad \wedge\, V \notin \{xpval[j][p] : j \in Nat\}$
$\quad \wedge\, CFP(i)!Phase2a(p,\, r,\, V)$
$\quad \wedge \forall\, j \in Nat : j < i \Rightarrow xpval[j][p] \neq none$
$\quad \wedge \forall\, j \in Nat \setminus \{i\} : InstanceUnchanged(j)$

Action $Phase2b(a,\, r)$ executes $Phase2b(a,\, r)$ for some $CFPaxos$ instance i.

$Phase2b(a,\, r) \triangleq$
$\exists\, i \in Nat :$
$\quad \wedge\, CFP(i)!Phase2b(a,\, r)$
$\quad \wedge \forall\, j \in Nat \setminus \{i\} : InstanceUnchanged(j)$

Action $Learn(l,\, v)$ executes $Learn(l,\, v)$ for some $CFPaxos$ instance i.

$Learn(l,\, v) \triangleq$

$\exists\, i \in Nat :$
$\quad \land\ CFP(i)!Learn(l,\ v)$
$\quad \land\ \forall\, j \in Nat \setminus \{i\} : InstanceUnchanged(j)$

Message Loss/Retransmission Actions

$LoseMsg(m)$ takes into consideration the fact that messages "$1a$", "$2S$", and "$1b$" are composite, with a logical message for every instance but with all grouped in the same physical message. As a result, all grouped messages must be lost together. The other sorts of message are not composite and can be lost in a single instance only.

$LoseMsg(m) \triangleq$
$\quad \lor \land m.type \in \{$"1a", "2S"$\}$
$\qquad \land \forall\, i \in Nat :$
$\qquad\quad \land \exists\, m2 \in CFP(i)!Msg :$
$\qquad\qquad \land\ m2.type = m.type$
$\qquad\qquad \land\ m2.rnd\ = m.rnd$
$\qquad\qquad \land\ CFP(i)!LoseMsg(m2)$

$\quad \lor \land m.type =$ "1b"
$\qquad \land \forall\, i \in Nat :$
$\qquad\quad \land \exists\, m2 \in CFP(i)!Msg :$
$\qquad\qquad \land\ m2.type = m.type$
$\qquad\qquad \land\ m2.rnd\ = m.rnd$
$\qquad\qquad \land\ m2.acc\ = m.acc$
$\qquad\qquad \land\ CFP(i)!LoseMsg(m2)$

$\quad \lor \land m.type \in \{$"2a", "2b"$\}$
$\qquad \land \exists\, i \in Nat :$
$\qquad\quad \land\ CFP(i)!LoseMsg(m)$
$\qquad\quad \land \forall\, j \in Nat \setminus \{i\} : InstanceUnchanged(j)$

By the way coordinator actions are grouped for all the instances, its last message is allways of the same type and for the same round. Therefore, retransmission boils down to just retransmitting the last message in every instance.

$CoordRetransmit(c) \triangleq$
$\quad \forall\, i \in Nat : CFP(i)!CoordRetransmit(c)$

If the last logical message of an acceptor for every instance is of type "$1b$", it means that its last action has been a $Phase1b(a,\ r)$ for some round r. In this case, it is not clear whether the coordinator of r has received the composite "$1b$" message from a or not, so a resends it. If it is not the case that the last logical message of a for every instance is of type "$1b$", then a has accepted some value for its current round and this means that the coordinator does not need its composite "$1b$" message. So, a only re-sends the "$2b$" messages for the instances at which it has accepted some value and leave the other instances unchanged.

$AcceptorRetransmit(a) \triangleq$
$\quad \lor \forall\, i \in Nat :$
$\qquad \land\ CFP(i)!AcceptorLastMsg(a).type =$ "1b"

$\land\ CFP(i)!AcceptorRetransmit(a)$
$\lor\ \land \exists\, i \in Nat : CFP(i)!AcceptorLastMsg(a).type = \text{“2b”}$
$\land \forall\, i \in Nat :$
IF $CFP(i)!AcceptorLastMsg(a).type = \text{“2b”}$
THEN $CFP(i)!AcceptorRetransmit(a)$
ELSE $InstanceUnchanged(i)$

A proposer retransmission is only a single retransmission for some *CFPaxos* instance i.

$ProposerRetransmit(p) \triangleq$
$\exists\, i \in Nat :$
$\land\ CFP(i)!ProposerRetransmit(p)$
$\land \forall\, j \in Nat \setminus \{i\} : InstanceUnchanged(i)$

Other Actions

LeaderSelection, *SuspectOrTrust*, and *FailOrRecover* just execute the actions with the same name on some instance i and leave the other instances unchanged. These actions actually influence all the instances because they deal with shared variables.

$LeaderSelection \triangleq$
$\exists\, i \in Nat :$
$\land\ CFP(i)!LeaderSelection$
$\land \forall\, j \in Nat \setminus \{i\} : InstanceUnchanged(i)$

$SuspectOrTrust \triangleq$
$\exists\, i \in Nat :$
$\land\ CFP(i)!SuspectOrTrust$
$\land \forall\, j \in Nat \setminus \{i\} : InstanceUnchanged(i)$

$FailOrRecover \triangleq$
$\exists\, i \in Nat :$
$\land\ CFP(i)!FailOrRecover$
$\land \forall\, j \in Nat \setminus \{i\} : InstanceUnchanged(i)$

Final Specification

$CoordNext(c)$ specifies the execution of some action by coordinator c.

$CoordNext(c) \triangleq$
$\lor \exists\, r \in RNum :\ \lor Phase1a(c,\, r)$
$\lor Phase2Start(c,\, r)$
$\lor\ CoordRetransmit(c)$

$ProposerNext(p)$ specifies the execution of some action by proposer p.

$ProposerNext(p) \triangleq$
$\lor \exists\, r \in RNum,\ V \in Value \cup \{Nil\} : Phase2a(p,\, r,\, V)$

$\lor\ ProposerRetransmit(p)$

$AcceptorNext(a)$ specifies the execution of some action by acceptor a.

$$
\begin{array}{l}
AcceptorNext(a) \triangleq \\
\quad \lor \exists\, r \in RNum : \lor\ Phase1b(a,\ r) \\
\qquad\qquad\qquad\quad \lor\ Phase2b(a,\ r) \\
\quad \lor\ AcceptorRetransmit(a)
\end{array}
$$

$LearnerNext(a)$ specifies the execution of some action by learner l.

$$
\begin{array}{l}
LearnerNext(l) \triangleq \\
\quad \exists\, v \in CFP(0)!ValMap : Learn(l,\ v)
\end{array}
$$

Next defines the next-state action of the specification.

$$
\begin{array}{lll}
Next \triangleq & \lor \exists\, V \in Value : Propose(V) & \\
& \lor \exists\, c \in Coord \cap noncrashed & : CoordNext(c) \\
& \lor \exists\, p \in Proposer \cap noncrashed & : ProposerNext(p) \\
& \lor \exists\, a \in Acceptor \cap noncrashed & : AcceptorNext(a) \\
& \lor \exists\, l \in Learner \cap noncrashed & : LearnerNext(l) \\
& \lor \exists\, m \in \text{UNION}\ \{xmsgs[i] : i \in Nat\} & : LoseMsg(m) \\
& \lor LeaderSelection \lor SuspectOrTrust \lor FailOrRecover &
\end{array}
$$

The fairness condition of the specification. We need weak fairness on the agent actions for every instance, since this is part of the liveness requirement for a single instance of $CFPaxos$.

$$
\begin{array}{l}
Fairness \triangleq \\
\quad \land \forall\, c \in Coord : \\
\qquad \land\ \mathbf{WF}_{vars}(c \in noncrashed \land CoordNext(c)) \\
\qquad \land\ \mathbf{WF}_{vars}(c \in noncrashed \land (\exists\, r \in RNum : Phase1a(c,\ r))) \\
\quad \land \forall\, p \in Proposer,\ i \in Nat : \\
\qquad \mathbf{WF}_{vars}(p \in noncrashed \land CFP(i)!ProposerNext(p)) \\
\quad \land \forall\, a \in Acceptor,\ i \in Nat : \\
\qquad \mathbf{WF}_{vars}(a \in noncrashed \land CFP(i)!AcceptorNext(a)) \\
\quad \land \forall\, l \in Learner,\ i \in Nat : \\
\qquad \mathbf{WF}_{vars}(l \in noncrashed \land CFP(i)!LearnerNext(l))
\end{array}
$$

The final specification

$$Spec \triangleq Init \land \Box[Next]_{vars} \land Fairness$$

Below we define the interface mapping from the algorithm above and the specification of Sequence Agreement.

We assume the set $Proposer$ can be totally ordered by a mapping $POrd$ that matches each proposer to a natural number and no two proposers to the same one.

$$
\begin{array}{l}
\text{ASSUME}\ \exists\, POrd \in [Proposer \to Nat] : \\
\qquad\qquad \forall\, p,\ q \in Proposer : p \neq q \Rightarrow POrd[p] \neq POrd[q]
\end{array}
$$

$$POrd \triangleq \text{CHOOSE}\ POrd \in [Proposer \to Nat] :$$

$$\forall\, p,\, q \in Proposer : p \neq q \Rightarrow POrd[p] \neq POrd[q]$$

The array learned that maps each learner to its learned sequence is defined below, in terms of the instances that have already been terminated and the last partially terminated sequence.

$$\begin{array}{l}
learned \triangleq \\
\quad \text{LET } defined[m \in CFP(0)!ValMap,\, s \in Seq(Value)] \triangleq \\
\qquad \text{LET } defSet \triangleq \{p \in \text{DOMAIN } m : \\
\qquad\qquad \wedge\, m[p] \neq Nil \\
\qquad\qquad \wedge\, \neg\exists\, i \in \text{DOMAIN } s : s[i] = m[p] \\
\qquad\qquad \wedge\, \forall\, q \in \text{DOMAIN } m : POrd[q] < POrd[p] \Rightarrow m[q] \neq m[p] \\
\qquad\qquad \wedge\, \forall\, q \in Proposer \quad : POrd[q] < POrd[p] \Rightarrow q \in \text{DOMAIN } m\} \\
\qquad \text{IN} \quad \text{CHOOSE } f \in [1\,..\,Cardinality(defSet) \to defSet] : \\
\qquad\qquad \forall\, i,\, j \in \text{DOMAIN } f : i \leq j \equiv POrd[f[i]] \leq POrd[f[j]] \\
\quad\quad deliver[l \in Learner,\, i \in Nat,\, s \in Seq(Value)] \triangleq \\
\qquad \text{IF } Proposer = \text{DOMAIN } xlearned[i][l] \\
\qquad \text{THEN } deliver[l,\, i+1,\, s \circ defined[xlearned[i][l],\, s]] \\
\qquad \text{ELSE } s \circ defined[xlearned[i][l],\, s] \\
\quad \text{IN} \quad [l \in Learner \mapsto deliver[l,\, 0,\, \langle\rangle]]
\end{array}$$

The following theorem asserts that the algorithm's specification implements sequence agreement.

$$SA \triangleq \text{INSTANCE } SAgreement$$
$$\text{THEOREM } Spec \Rightarrow SA!Spec$$

$LA(V,\, l,\, c,\, Q)$ defines the liveness assumption required by the algorithm. It is the same as Collision-fast $Paxos$.

$$\begin{array}{l}
LA(V,\, l,\, c,\, Q) \triangleq \\
\quad \wedge\, \{c,\, l\} \cup Q \subseteq noncrashed \\
\quad \wedge\, V \in proposed \\
\quad \wedge\, \forall\, c2 \in Coord : amLeader[c2] \equiv (c = c2) \\
\quad \wedge\, activep[c] \subseteq noncrashed \\
\quad \wedge\, \forall\, r \in RNum : \\
\qquad \exists\, r2 \in RNum : \wedge\, r \prec r2 \\
\qquad\qquad\qquad\qquad \wedge\, c = CoordOf(r2) \\
\qquad\qquad\qquad\qquad \wedge\, CfProposer(r2) \subseteq activep[l] \\
\quad \wedge\, activep[c] \subseteq activep[c]'
\end{array}$$

The theorem below asserts that the algorithm's specification satisfies Liveness if the liveness assumption eventually holds forever.

$$\begin{array}{l}
\text{THEOREM } \forall\, l \in Learner,\, V \in Value : \\
\qquad \wedge\, Spec \\
\qquad \wedge\, \exists\, Q \in \text{SUBSET } Acceptor :
\end{array}$$

$$\begin{array}{l} \quad \wedge \forall\, r \in RNum : Q \in Quorum(r) \\ \quad \wedge \exists\, c \in Coord\ : \Diamond\Box[LA(V,\ l,\ c,\ Q)]_{vars} \\ \Rightarrow \Diamond(\exists\, j \in 1\,..\,Len(learned[l]) : learned[l][j] = V) \end{array}$$

3.9 Related Work and Final Remarks

As explained before, sequence agreement is a sequence-based specification of the famous atomic broadcast problem of group communication [HT93]. The literature about atomic broadcast is extensive [DSU04], and we concentrate on the works that relate to ours the most. There are various fault-tolerant atomic broadcast algorithms that can deliver messages (extend the learned sequence in the sequence agreement problem) in two communication steps in optimistic runs [VR02, PS02, PS03, PSUC02]. However, these protocols cannot guarantee this latency even in the total absence of failures. They are based on optimistic assumptions on the way messages are ordered by the communication network or the absence of network contention. Even if the system is behaving synchronously, if their assumptions do not hold, delivery in two communication steps will not be achieved due to internal collisions.

The same problem happens with some fault-tolerant consensus protocols such as Fast Paxos [Lam06a]. Concurrent proposals might collide and prevent the learning of any of them within two communication steps. There exist collision-fast consensus protocols [Lam06b, CBS06]; however their direct application to solving sequence agreement does not lead to a collision-fast implementation since, by the definition of consensus, if two proposals are made to the same instance, only one is given as the output and the other has to be proposed again in a different instance (increasing the learning latency).

Instead of using consensus as the basis to build our protocol, we use a different abstraction we call M-Consensus. M-Consensus resembles the interactive consistency problem defined for byzantine settings [PSL80]. There are some small differences between the two problems besides those referring to the models they were defined. However, the main reason why we give our problem a different name is that it can actually be defined based on any mapping and not only a mapping from proposers to proposals. We assume such a mapping in this chapter because this is mapping Collision-fast Paxos uses. However, the problem is more general than that.

One could think that solutions to interactive consistency to solve sequence agreement could give the same performance as collision-fast paxos. Interactive consistency is not defined or used in the context of sequence agreement or atomic broadcast, though. Its solutions usually assume all proposers propose in parallel and consider the mapping as independent consensus instances [GL06]. This type of solution does not lead to collision-

fast implementations because they do not ensure termination in two-communication steps if a single proposer proposes. The other proposers will only hear about the first proposals after one communication step and it will take at least two extra communication steps for their empty proposals to be learned. Collision-fast Paxos provides better performance because the mapping is considered as a single structure and it is architected wisely to allow proposers to contact learners directly in some special cases.

It is possible to find many variants of Paxos in the literature. Disk Paxos [GL03] is a variant of the original algorithm in which processes have access to network disks with some limited processing power. Cheap Paxos [ML04] is also a simple variant of the original protocol where cheap machines to help system configuration after failures and, with that, provide better fault tolerance. Paxos Commit [GL06] uses the original protocol with some simple optimization to solve inractive consistency in the context of transaction commit. All these algorithms differ from ours in that they all assume a unique proposer can be collision-fast for a round. Fast Paxos [Lam06a] allows rounds in which multiple proposers can propose directly to acceptors, but this may lead to collisions and delay the learning of proposed values. Collisions are also a problem with Generalized Paxos [Lam04] and Multicoordinated Paxos [CSP06].

After designing Collision-fast Paxos, we came accross two protocols that can tolerate more than a single failure and be collision-fast: [Zie06] and [KD96]. In fact, the two protocols are very similar since they both seem to extend the timestamp-based algorithm presented in [Lam78]. In contrast to the approach in [Zie06], ours considers a weaker model, where processes can crash and recover, and messages can be lost or duplicated. More importantly, our algorithm allows reconfiguration in case collision-fast proposers fail so that execution can become collision-fast again for a different set of proposers. In [Zie06], all processes are considered collision-fast proposers and a single failure makes the algorithm slow down to a non-collision-fast execution mode forever.

The approach in [KD96] allows reconfiguration, but depends on a majority of the processes being up and synchronized to allow collision-fast learning. In fact, due to the dynamic environment assumed by the algorithm, if the system is partitioned, not even a consistent delivery order is guaranteed. This is a major difference to our algorithm since we do not know how to adapt their work to make it comparable to ours. Their algorithm is based on different building boxes with their own guarantees. This makes their algorithm more expensive in terms of disk writes as compared to ours, which requires only acceptors to write on disk. Reconfiguration and recovery in [KD96] is also more expensive than with Collision-fast Paxos.

In summary, in this chapter we have discussed the design and implementation of a very efficient and dynamic collision-fast sequence agreement protocol. Since the traditional approach to implement sequence agreement based on standard consensus cannot result on a resilient collision-fast implementation, we have proposed a new agreement problem called M-Consensus that allows multiple proposals to take part in the final decision. Our solution

to M-Consensus is an extension of the Paxos protocol in which a number of proposers can have their proposals as part of the final decision in two message steps. Using Collision-fast Paxos to implement a collision-fast sequence agreement algorithm is simple and provides a very efficient fault-tolerant protocol.

Chapter 4

Optimized Algorithms

We think in generalities, but we live in detail.
Alfred North Whitehead

In Chapter 2, we presented a simple algorithm that can be directly derived from our deferred-update abstraction. This simple and general algorithm has some problems, though. First of all, complete information about what was read and written by active transactions must be forwarded to all databases through a sequence agreement protocol. As we have seen in the previous chapter, fault-tolerant sequence agreement algorithms are expensive and one should avoid sending too much information through them if good performance is a system requirement. Moreover, the algorithm requires all databases to perform the same certification test for all proposed transactions, even though its outcome is deterministic and, therefore, guaranteed to have the same outcome. Comparing histories for certification requires considerable processing time and having this task replicated in all database replicas in the system hurts performance and does not add anything in terms of fault tolerance. Combining our deferred update abstraction with termination protocols tailor-made for efficient replication can provide much more efficient algorithms.

In this chapter, we present two such novel algorithms derived from our general abstraction. In our first algorithm, we use the knowledge obtained from working with the Paxos protocol to implement termination very efficiently. We managed to reduce the burden of transaction certification to a single process and propagate only transaction updates (active operations) to acceptors and replicas, all that providing the same latency and degree of fault tolerance as the original Paxos protocol, being able to certify and propagate active transactions to replicas within three communication steps as seen from the client. We know of no previous protocol with these characteristics. Our second algorithm assumes that stronger properties are ensured by database replicas to achieve even better termination performance, requiring only two communication steps and no certification procedure to commit proposed

transactions. It shows that our deferred-update abstraction can help the design and analysis of protocols even if termination depends on stronger assumptions about the consistency guarantees of database replicas.

4.1 Certification-based Algorithm

In this section we present our first optimized algorithm for database replication. It assumes a model similar to the one we presented in Figure 2.4 of Section 2.3. Clients access the interface of the replicated database through a local driver responsible for interacting with databases and some extra agents we introduce to implement an efficient and fault-tolerant termination protocol.

4.1.1 Model and Definitions

Our algorithm is based on four different types of agents: clients, databases, acceptors and leaders. Clients implement the local drivers of the algorithm and databases implement the proxies around the set of active order-preserving serializable replicas. We introduce acceptors to provide fault tolerance to our termination protocol and leaders to certify transactions and propagate updates efficiently. As we did in Chapter 3, we assume an asynchronous crash-recovery model in which agents communicate by exchanging messages, with no bounds on the time it takes for messages to be transmitted or actions to be executed. Messages can be lost or duplicated but not corrupted; agents can fail by stopping only and never perform incorrect actions. Agents are assumed to have some sort of local stable storage to keep their state in between failures so that finite periods of absence are not distinguishable from excessive slowness. Although we assume agents may recover, they are not obliged to do so once they have failed. An agent is considered to be nonfaulty iff it never stops executing enabled actions.

Borrowing from our abstract deferred-update protocol, we must assume state-deterministic operations. Moreover, each transaction t has a single database $DBof(t)$ responsible for the initial execution of its operations. We also assume transaction t has a single client $ClientOf(t)$ that can submit operations on its behalf. In simpler words, this means that a transaction cannot have its operations submitted to the database by more than one client.

The algorithm is organized in a totally ordered set of rounds. For simplicity, it can be assumed that rounds correspond to the natural numbers. We assume each round has a single leader assigned to it, responsible for certifying and propagating update transactions at that round. As in the previous chapter, acceptors are necessary to ensure liveness and we let a *quorum* be any finite set of acceptors large enough to ensure liveness. To ensure consistency,

we need the following assumption about quorums:

Assumption 4.1 (Quorum Requirement) *If Q and R are quorums, then $Q \cap R \neq \emptyset$.*

4.1.2 General Idea and Data Structures

The initial execution of transaction operations is done in the same way we specified for the algorithm in Section 2.3. The main difference between that algorithm and this one lies on the implementation of the termination protocol. During normal execution, a single leader performs leader actions. In this scenario, when a client wants to propose an active transaction for termination, it sends a "`propose`" message to the current leader with complete information about the transaction history. The leader keeps on its internal state the sequence of active transactions that have been certified so far and, based on that, it can calculate the final database state after all previously certified transactions are executed. To certify a proposed transaction it simply verifies if the execution of the operations in its history produces the same results. If it is the case, the transaction is added to the sequence of certified transactions. Otherwise, it is added to a set of aborted transactions.

Whenever the leader changes its current state, after certifying a transaction, it sends a message to the acceptors containing the current round number along with the sequence of positively certified transactions and the set of aborted ones. Notice that the sequence of positively certified transactions may contain only the active history for each transaction in it, since these transactions have already passed a certification test. Different from the algorithm on Section 2.3, no database needs to receive information about passive operations performed by transactions originally executed on different databases. Having said that, when an acceptor receives a new state change notification from the leader, if it has not heard of a higher-numbered round, it extends its currently accepted values (sequence and set) for that round with the values it received in the new message from the leader.

A pair $\langle committedSeq, abortedset \rangle$ is said to be chosen at round r if it has been accepted by a quorum Q of acceptors, that is, if $committedSeq$ is a common prefix and $abortedset$ is a subset of the values currently accepted at round r for all acceptors in Q. Since acceptors can only extend the values they accept at some round, this property is stable and can never become false once it is true. The termination protocol guarantees that no incompatible pairs are chosen at different rounds and this makes chosen values safe to be learned by databases and clients.

When an acceptor accepts new values, it propagates them to databases and clients along with the round number at which the new values were accepted. A database that receives acceptance notification messages from a quorum of acceptors for the same round number calculates the maximum common prefix of the sequence of committed transactions in all the

messages. It then compares if it extends its currently stored sequence and, if that is the case, updates it. This sequence implements variable $learnedSeq$ of our abstraction with the plus that it also contains the active history (the sequence of active operations) of each transaction in it.

A client c that receives acceptance notification messages from a quorum of acceptors for the same round number calculates the maximum subset of certified and aborted transactions in all the messages. If these sets contain transactions previously proposed by c, the client can safely deliver their outcomes (whether $Committed$ or $Aborted$) to the application.

This algorithm runs nicely in this way if the leader never crashes, certifying and propagating proposed transactions to databases and clients in three communication steps only. The failure suspicion of the current leader by some other agent triggers the election of a new leader in the system. This new leader starts a new, higher-numbered, round by sending a notification to the acceptors. Acceptors that receive this notification for a round higher than any other round they have heard of simply change their current round number and send a notification to the new leader with the latest value they have accepted.

The algorithm keeps the invariant that a value accepted by an acceptor at round r extends or equals any value chosen at a lower-numbered round. Therefore, when the new leader receives the notification from a quorum of acceptors for the newly started round, it only looks at the values that were accepted at the highest-numbered round. From the set of values that satisfy this constraint, the leader picks up the pair $\langle committedSeq, abortedSet \rangle$ that extends all the other pairs. This pair becomes the new internal state of the leader and is used to certify uncertified proposed transactions. This procedure guarantees that the value picked by the leader extends all values possibly chosen in lower-numbered rounds and preserves the invariant of the algorithm.

The specific implementation details of the complete protocol are given in the next section. For now, let us concentrate on the data structures kept by each one of the agents in the system. Many variables are straightforward, if not direct, mappings of the variables of our general deferred-update abstraction. A client c keeps the following variables internally:

$cthist[c]$: A history vector mapping each transaction under c's responsibility to its current history, initially empty.

$cq[c]$: A mapping from each transaction under c to its current request or $NoReq$ if no request is being executed on behalf of that transaction. Initially, it maps each transaction to $NoReq$.

$cdreq[c]$: A mapping from each transaction t under c to the operation that is currently being submitted for execution on $DBof(t)$, or $NoReq$ if no operation is being submitted. Initially all transactions are mapped to $NoReq$.

$cpdec[c]$: A mapping used to tell whether a transaction t under c was decided without being proposed for global termination either because it was prematurely aborted during its initial execution or because it was a passive transaction that committed on its execution database.

$cproposed[c]$: Set of transaction ids proposed by c.

$cgdec[c]$: A mapping from each transaction t under c to the its termination decision.

The variables of a database d are:

$ddreply[d]$: Similar to $cdreq[c]$ above, but mapping each transaction t such that $DBof(t) = d$ to the last response given by d to one of its operations.

$dcnt[d]$: A mapping from each transaction t to an integer representing the number of operations that executed on d for t. It counts the number of operations $DBof(t)$ has executed for t during t's initial execution and, if t is active, the number of active operations the other databases (or $DBof(t)$ if it does not manage to commit t directly after it is globally committed) have executed for t after it is globally committed. It is initially 0 for all transactions.

$vers[d]$: A mapping from each transaction t to an integer representing the current version of t being submitted to d. It is initially 0 for all transactions.

$dcom[d]$: A mapping from each transaction t to a boolean telling whether t has been committed on d. It is initially false for transactions.

$dlearnedSeq[d]$: A sequence of records of type $[trans : Tid, acthist : Seq(Op)]$[1], where $acthist$ stores the sequence of operations of the active history of transaction $trans$. The projection of sequence $dlearnedSeq[d]$ considering only field $trans$ for each record gives the mapping to variable $learnedSeq[d]$ in our general deferred-update algorithm.

A leader l keeps two variables:

$lrnd[l]$: l's current round number, initially 0.

$lval[l]$: A record of type $[seq : dlSeqType, aborted :\subseteq Tid]$, where $dlSeqType$ defines the type of variable $dlearnedSeq[d]$ above. In other words, $lval[l]$ keeps a sequence seq of certified transactions, where each element in the sequence contains a transaction id and its sequence of active operations, as well as a set of aborted transactions. For simplicity of notation in the acceptor variables, let us define $ValType$ as the type of

[1]The operator $Seq(S)$, introduced in Chapter 2 represents the set of all finite sequences of elements in set S

variable $lval[l]$. As for its initialization, all leaders except the leader of round 0 have it set to a special value $none$. The leader of round 0 has its $lval[l]$ pair set to an empty sequence and an empty set.

Last, the variables kept by acceptor a are the following:

$arnd[a]$: The current round of a. Initially 0.

$ahist[a]$: The acceptance history of a, that is, a mapping from each round number to a value of type $ValType$ defined above. It is initially $none$ for all rounds except round 0. The pair $ahist[a][0]$ is set to an empty sequence and an empty set.

4.1.3 Atomic Actions

We now present more precisely the atomic actions that define the algorithm. The actions whose names also appear in our general deferred-update abstraction represent implementations of them. The extra actions define our termination protocol.

$ReceiveReq(c, t, req)$ Executed by client c, for transaction t and request req. This action deals with the receipt of a request for transaction t by client c. It follows the basic structure of the action with the same name in our deferred-update abstraction. If the transaction has not been previously decided (either locally or by the termination protocol) the request is evaluated and, depending on that, the action proposes t for the termination protocol or forwards the request to the database responsible for it.
The action is enabled iff:

- $c = ClientOf(t)$,
- $DBRequest(t, req)$ was triggered by the application layer, and
- $cq[c][t] = NoReq$

It sets $cq[c][t]$ to req and performs the following conditional action:

if $t \notin cproposed[c] \wedge cpdec[c][t] \notin Decided$ **then**
  if $req = Commit$ and t may be active **then**
    - $cproposed[c] \leftarrow cproposed[c] \cup t$
    - Send $\langle$"`propose`", t, $cthist[c][t]\rangle$ to leader
  else
    - $cdreq[c][t] \leftarrow req$
    - Send $\langle$"`dreq`", t, req, $Len(cthist[c][t])\rangle$ to $DBof(t)$

$ReplyReq(c, t, rep)$ Executed by client c, for transaction t and reply rep. This action gives a response to the application as a result of the last operation submitted on behalf of transaction t. It assumes an operator $ctdec(c, t)$ that returns the c's local view of abstract variable $tdec$, defined as follows.

$$ctdec(c, t) \triangleq \textbf{if } t \notin cproposed[c] \textbf{ then } cpdec[c][t] \textbf{ else } cgdec[c][t]$$

If c knows, based on $ctdec(c, t)$, that t has been decided, it returns the decision to the application. Otherwise, it replies the result given by a message of type $drep$ coming from $DBof(t)$ with the database result to the last operation submitted.
The action is enabled iff:

- $c = ClientOf(t)$,
- $cq[c][t] \in Request$, and
- **if** $ctdec(c, t) \in Decided$ **then**
  - $rep = ctdec(c, t)$

  else
  - $cq[c][t] \in Op$,
  - $rep \in Result$, and
  - c received message $\langle$"`drep`", t, rep, $cnt\rangle$ with $cnt > Len(cthist[c][t])$

It executes the following operations:

- $DBResponse(t, rep)$,
- $cq[c][t] \leftarrow NoReq$, and
- **if** $ctdec(c, t) \notin Decided$ **then**
  - $cthist[c][t] \leftarrow cthist[c][t] \circ \langle cq[c][t], rep\rangle$
  - $cdreq[c][t] \leftarrow NoReq$

$PrematureAbort(c, t)$ Executed by client c, for transaction t. This simple action allows c to abort local transaction t.
It is enabled iff:

- $c = ClientOf(t)$,
- $t \notin cproposed[c]$, and
- $cpdec[c][t] \notin Decided$

It simply sets $cpdec[c][t]$ to $Aborted$.

$PassiveCommit(c, t)$ Executed by client c, for transaction t. It allows c to commit passive transaction t given that $DBof(t)$ managed to locally commit it.
It is enabled iff:

- $c = ClientOf(t)$,

- $t \notin cproposed[c]$, and
- c received message $\langle$"drep", t, $Committed\rangle$ from $DBof(t)$

It simply sets $cpdec[c][t]$ to $Committed$.

$DBReq(d, t, req)$ Executed by database d, for transaction t and request req. This action submits a request req to the local active order-preserving serializable database controlled by proxy d. As we did in Chapter 2, we use the notation $DB(d)!Action$ to represent interface actions on the database replica controlled by d. As in our abstract deferred-update algorithm, three conditions, shown below, can enable this action. Notice however, that it is also subject to the pre-condition of interface action $DB(d)!DBRequest(\langle t, vers[d][t]\rangle, req)$, since its execution must be performed as an effect.

Condition 1 (external operation request)

- $d = DBof(t)$,
- $vers[d][t] = 0$, and
- d received message $\langle$"dreq", $t, req, len\rangle$ with $len = dcnt[d][t]$

Condition 2 (operation after termination)

- $\exists i \in 1..Len(dlearnedSeq[d])$:
  - $dlearnedSeq[d][i].trans = t$,
  - $dcnt[d][t] < Len(dlearnedSeq[d][i].acthist)$, and
  - $req = dlearnedSeq[d][i].acthist[dcnt[d][t] + 1]$

Condition 3 (commit after termination)

- $req = Commit$
- $\exists i \in 1..Len(dlearnedSeq[d])$:
  - $dlearnedSeq[d][i].trans = t$
  - $\forall j < i$: $dcom[d][dlearnedSeq[d][j].trans]$
- **either** $d = DBof(t) \wedge vers[d][t] = 0$
  or $dcnt[d][t] = Len(dlearnedSeq[d][i].acthist)$

It executes action $DB(d)!DBRequest(\langle t, vers[d][t]\rangle, req)$ to submit operation req to its local active order-preserving serializable database.

$DBRep(d, t, rep)$ Executed by database d, for transaction t and database reply rep. It executes as a response to action $DB(d)!ReplyReq(\langle t, vers[d][t]\rangle, rep)$, triggered by the database replica controlled by d. It follows basically the same structure of action $DBRep$ in our abstract algorithm. More specifically, it performs the following operations:

- **if** $d = DBof(t)$ **then**
  - $ddreply[d][t] \leftarrow rep$
  - **if** t does not appear in $dlearnedSeq[d]$ **then**

if $rep \in Decided$ **then** send message $\langle$"drep", t, $rep\rangle$ to $ClientOf(t)$
else send message $\langle$"drep", t, rep, $dcnt[d][t] + 1\rangle$ to $ClientOf(t)$

- **if** $rep = Aborted \ \wedge \ t$ appears in $dlearnedSeq[d]$ **then**
  - $vers[d][t] \leftarrow vers[d][t] + 1$
  - $dcnt[d][t] \leftarrow 0$

  else if $rep \in Result$ **then**
  - $dcnt[d][t] \leftarrow dcnt[d][t] + 1$

  else
  - $dcom[d][t] \leftarrow rep = Committed$

$Phase1a(l, r)$ Executed by leader l, for round r. In this action, l starts a new round r to overcome the possible failure of a previous leader.
It is enabled iff:

- l is the leader of round r and
- $lrnd[l] < r$

These pre-conditions ensures safety but can prevent progress without further constraints. We discuss liveness requirements on the following section. The action performs the following operations:

- $lrnd[l] \leftarrow r$
- $lval[l] \leftarrow none$
- Send message $\langle$"1a", $r\rangle$ to acceptors

$Phase1b(a, r)$ Executed by acceptor a to start on round r. It is enabled iff:

- a receives a message $\langle$"1a", $r\rangle$ and
- $arnd[a] < r$

The action sets $arnd[a]$ to r and sends message $\langle$"1b", r, a, $vrnd$, $vval\rangle$ back to the leader of round r, where $vrnd$ is the highest-numbered round at which a accepted something and $vval$ is the record containing the sequence of certified transactions and set of aborted transactions that a accepted at $vrnd$.

$Phase2Start(l, r)$ Executed by leader l, for round r. In this action, l installs the new round and makes it operational.
It is enabled iff:

- l is the leader of round r,
- $lrnd[l] = r$,
- $lval[l] = none$, and
- l has received "1b" messages for round r from a quorum Q of acceptors.

From the "1b" messages received, l selects those with the highest value for $vrnd$. From these selected messages, it picks up the pair $vval$ with the longest sequence and the biggest set. It sets $lval[l]$ to $vval$ and sends message $\langle$"2a", $r, vval\rangle$ to all the acceptors.

*Phase*2$a(l, r)$ Executed by leader l, for round r. After l has installed round r (i.e., has set $lval[l]$ to a value different from $none$), it can start certifying proposed transactions.
It is enabled iff:

- l is the leader of round r,
- $lrnd[l] = r$,
- $lval[l] \neq none$, and
- l has received a "propose" message for a transaction t not present in $lval[l].committedSeq$ or $lval[l].abortedSet$.

Let $FSTATE$ be the database state generated by executing the active histories of every transaction in $lval[l].committedSeq$ in order. Recall that elements of $lval[l].committedSeq$ contain both the transaction id and the active operations (updates) of its execution history. Therefore, it is possible for l to generate $FSTATE$ based only on its local information. The certification of a proposed transaction t is done by verifying if its history is compliant with an execution starting on state $FSTATE$. If it is, l extends $lval[l].committedSeq$ with a record containing t's id and its sequence of active operations. If t does not pass the certification test, l extends $lval[l].abortedSet$ with t's id. Notice that after certification passive operations and operation results referring to t's history present in the "propose" message are simply thrown away by l. Then, l sends a message $\langle$"2a", $r, val\rangle$, where val equals the new value of $lval[l]$, to all acceptors.

*Phase*2$b(a, r)$ Executed by acceptor a, for round r. In this action, a accepts a value forwarded by the leader of round r.
It is enabled iff:

- $arnd[a] \leq r$,
- a has received a message $\langle$"2a", $r, val\rangle$ and either $ahist[a][r] = none$ or val extends $ahist[a][r]$, in the sense that $ahist[a][r].committedSeq$ is a prefix of $val.committedSeq$ and $ahist[a][r].abortedSet$ is a subset of $val.abortedSet$.

The action sets $arnd[a]$ to r and $ahist[a][r]$ to the pair val received in the "2a" message. It also sends a message $\langle$"2b", $r, a, val\rangle$ to all databases and clients.

ClientLearn(c) Executed by client c. The action is enabled if c has received "2b" messages for the same round from a quorum of acceptors. c calculates the longest common prefix of all *committedSeq* sequences received and the longest common subset of all *abortedSet* sets received. If a transaction t under c's responsibility appears in any of these values and $cgdec[c][t] = Unknown$, c sets $cgdec[c][t]$ to $Committed$ if

t appears in the common sequence of committed transactions or $Aborted$ if it appears in the common set of aborted ones.

$DatabaseLearn(d)$ Executed by database d. The action is enabled if d has received "2b" messages for the same round from a quorum of acceptors. d calculates the longest common prefix of all $committedSeq$ sequences received only. If the calculated value extends $dlearnedSeq[d]$, $dlearnedSeq[d]$ is updated with the new value.

4.1.4 Correctness and Optimizations

The correctness of the algorithm above is given by a refinement mapping from its variables to the variables of our abstract deferred-update algorithm. In fact, variables $dcnt$, $vers$, and $dcom$ are the same. All the other variables but $gdec$ are implemented by the simple mappings below:

$$
\begin{aligned}
thist[t] &\triangleq cthist[ClientOf(t)][t] \\
q[t] &\triangleq cq[ClientOf(t)[t] \\
dreq[t] &\triangleq cdreq[ClientOf(t)][t] \\
pdec[t] &\triangleq cpdec[ClientOf(t)][t] \\
dreply[t] &\triangleq ddreply[DBof(t)][t] \\
proposed &\triangleq \bigcup_{c \in Client} cproposed[c] \\
learnedSeq[d] &\triangleq \text{projection of sequence } dlearnedSeq[d] \text{ over field } trans.
\end{aligned}
$$

As for $gdec$ it depends on two extra definitions. We say that transaction t is *globally committed* iff it appears in a common prefix of field seq for values accepted by a quorum of acceptors at the same round, that is, iff

$\exists Q \in Quorum, r \in \mathbb{N}, seq \in Seq(ValType)$:

- seq contains t, and
- $\forall a \in Q$:

- $ahist[a][r] \neq none$ and
- seq is a prefix of $ahist[a][r].committedSeq$

Similarly, we can define a transaction t as *globally aborted* iff it appears in the aborted set accepted by a quorum of acceptors, that is, iff

$\exists Q \in Quorum, r \in \mathbb{N}$

- $\forall a \in Q$:
  - $ahist[a][r] \neq none$ and
  - t appears in $ahist[a][r].abortedSet$

With these two definitions, we can easily create a mapping for termination variable $gdec$ as follows:

$$gdec[t] \triangleq \begin{cases} Committed & \text{, if } t \text{ is globally committed} \\ Aborted & \text{, if } t \text{ is globally aborted} \\ Unknown & \text{, otherwise} \end{cases}$$

The refinement mapping above is easily proved by analyzing the actions of the algorithm and matching them with the corresponding actions they implement in our deferred-update abstraction. Few, very intuitive, extra invariants are necessary as we show in the following.

Action $ReceiveReq$**:** This action directly implements its abstraction. The substitution of $tdec[t] \notin Decide$ for $t \notin cproposed[c] \wedge cpdec[c][t] \notin Decided$ is guaranteed to be correct by Invariant TI4(b) of Section 2.4.1.3.

Action $ReplyReq$**:** The use of expression $ctdec(c, t)$ instead of $tdec[t]$ is correct because $cgdec[c][t]$ is only changed from $Unknown$ to the value of $gdec[t]$ according to our mapping above by action $ClientLearn$. Therefore, if $ctdec(c, t) \in Decided$ then $tdec[t] \in Decided$. If $ctdec(c, t) \notin Decided$, invariant TI3(e) and the action's pre-condition ensure correctness.

Action $PrematureAbort$**:** Direct implementation.

Action $PassiveCommit$**:** A "`drep`" message carrying the decision $Committed$ implies that $dreply[t]$ had value $Committed$ at some point. It is easy to show that $dreply[t]$ never changes to a different value once it is set to $Commited$.

Action $DBReq$**:** Enabling condition 1 is the only one not directly implementing the same condition in our abstraction. However, it is easy to prove the invariant that a "`dreq`" message satisfying these conditions when $vers[d][t] = 0$ carries the correct value of $dreq[t]$ and $Len(thist[t])$.

Action $DBRep$**:** In this algorithm, it is also easy to prove the invariant that, if $DB(d)!q[\langle t, vers[d][t]\rangle] \in Request$, then $t \in proposed \iff t$ appears in $dlearnedSeq[d]$. This invariant makes it clear that the algorithm's action implement its abstraction correctly.

One might think that the most difficult part of the proof has to do with the actions implementing the termination protocol. However, it is easy to see the resemblance between these actions and those of Collision-fast Paxos presented in Chapter 3. In fact, our termination protocol is a variant of the original Paxos protocol [Lam98], based on its generalized extension presented in [Lam04]. We just could not use Paxos as a black box because our leader is responsible for certification and does not forward to the acceptors the information that does not have to be propagated to replicas. Nevertheless, Nontriviality, Stability, Consistency, and Liveness follow directly from the properties ensured by Paxos and the facts that the sequence of transactions in any pair $\langle committedSeq, abortedSet\rangle$ propagated from the leader to the acceptors is guaranteed to be serializable by the certification test and no transaction will ever belong to both $committedSeq$ and $abortedSet$.

The Liveness of termination is ensured by Paxos. For Paxos to ensure progress, we must make the same assumptions and modifications to the algorithm stated in Section 3.4.2 with some required small adaptations. Progress of Paxos also depends on an unreliable leader election algorithm that eventually elects a single nonfaulty leader. A leader l will only execute leader actions if it believes to be the output of the leader election algorithm. A nonfaulty quorum of acceptors is also required for liveness. A more complete explanation and proof of Liveness concerning Paxos is given in [Lam06a].

As for the optimizations, there are many possibilities. As a simple example, variable $cthist[c][t]$ does not have to be kept after $ctdec(c, t) \in Decided$. The same is valid for the active history and id of transactions stored by databases for proposed transactions that have already been locally committed. In fact, many other variables do not have to be stored forever, and can be released as soon as they will not be required by the algorithm. Variable $ahist$ is defined the way it is in order to ease our mapping of variable $gdec$ as well as the understanding of our termination protocol. In fact, acceptors can only keep the last value they have accepted and the round at which it happened, since these are the only values sent in "`1b`" and "`2b`" messages [Lam98, Lam04].

The sending of increasing sequences and sets in "`2a`" and "`2b`" messages during normal execution can easily degrade performance of direct implementations. This can be easily optimized by sending just complementary information with respect to the information that was sent before. For example, when the leader certifies a transaction t and adds it to sequence $lval[l].committedSeq$, it could send just the index of t in the new sequence together with the extra information about t (its id and active history). The same optimization can be done if we represent $lval[l].abortedSet$ also as a sequence (i.e., use a sequence to implement a set). An acceptor only accepts information sent like that if it correctly extends the value

the acceptor has previously accepted, with no gaps. Messages received out of order can be stored in a local buffer and ordered before being accepted. The same thing can be done for "`2b`" messages, making both "`2a`" and "`2b`" messages sent during normal execution of practical size, and without increasing the message complexity of the protocol. Message losses can be recovered by using special messages to request missing information.

As for "`1b`" messages and the initial "`2a`" message sent by the leader of a new round, it is possible to optimize them in similar ways, but this requires stronger assumptions and algorithm changes that may depend on the application scenario. If the number of the databases and clients in the system is finite (it is not possible to continuously add new ones to the system), processes can store and send sequences in messages without prefixes that are known to have been learned by all databases and clients. It is also possible to use checkpoints to avoid dealing with long sequences of committed transactions. The set of aborted transactions can be compressed if clients responsible for transactions in it have already received the information or have surely crashed.

Assuming that databases have a consistency guarantee stronger than active order-preserving serializability can also lead to optimizations. For example, if it is possible to infer the consistent state from which a transaction reads (achievable with two-phase-locking, some multiversion concurrency control mechanisms, or direct access to the internals of the database engine) one can considerably compress the history information clients must keep. In this case, it suffices to know which prefix of $dlearnedSeq$ the state read by the transaction refers to and the data items read by the transaction (no results must be kept). Then, the certification test of a transaction t boils down to checking if any active transaction after the prefix seen by t in the currently certified sequence changed any data item read by t during its execution. To compress the stored information even further, clients can keep a reference to a superset of the data items read by a transaction, keeping, for example, the reference to a table the transaction read from instead of multiple references to individual table rows. Assuming a transaction reads more than it actually reads can just force it to be aborted in situations where it could be committed, but never the opposite. In an extreme case, it is possible to assume the transaction has read the whole database state, reducing to a constant size the amount of information related to read operations kept by clients and propagated to the leader.

Notice that only the set of acceptors is assumed to be finite to ensure consistency of termination. Having an infinite number of possible leaders, clients, and databases, allows us to improve performance by avoiding writing their state on stable storage. A leader that crashes and recovers can just assume a new identity with an empty initial state before rejoining the system [Agu04]. This means that no disk writes on the databases are necessary either. However, databases are atomic and stable in their nature and most off-the-shelf databases systems will not have the option to disable their disk writes. The algorithm in the next section assumes a special class of databases that usually have this feature, thus providing very good performance.

In fact, the next algorithm is an extension of the current one based on the fact that the certification test can be done by using a database to serialize the active transactions submitted for termination. Assuming a stronger scheduler inside this database can lead to a very efficient termination protocol.

4.2 In-memory Primary-Backup Replication

This section presents a primary-backup protocol to manage replicated in-memory database systems (IMDBs). The protocol we present extends the one originally presented in [CPS06] and, likewise, exploits features inherent to in-memory databases such as simpler concurrency control mechanisms and the possibility of deferring disk writes. The algorithm is fault tolerant in that primary crashes can be suspected and, in that case, a new primary is elected without jeopardizing consistency. False failure suspicions are tolerated and never lead to incorrect behavior. The protocol uses a variant of Paxos to solve the sequence agreement problem inside termination. Under normal circumstances (i.e., no failures or false suspicions), transactions are committed after only two communication steps, as seen by the application, with no risk of collisions.

4.2.1 Motivation

Demand for high performance combined with plummeting hardware prices have led to the widespread emergence of large computing clusters [Pfi98, SS99]. Often built out of commodity components, these systems are composed of servers interconnected through very fast network switches and equipped with powerful processors and large memories. Applications in these environments often rely on shared storage systems, accessible to the application processes running on remote servers. A typical example is a multi-tier web application in which some of the tiers run within the cluster [BN97]. Storage systems are used to keep shared information, managed concurrently by different application processes, as well as to provide fault tolerance by allowing processes to save their state and later retrieve it for recovery and migration. Whatever the use, high-availability, good performance, and strong consistency are key requirements of a storage service. In such environments, in-memory databases (IMDBs) [GMS92] have been successfully used to increase the performance of transactions [BK02]. The algorithm we present in the following section introduces high-availability in a replicated IMDB setting with very small overhead.

IMDBs provide high transaction throughput and low response time by avoiding disk I/O during the execution of transaction operations. The key characteristic of an IMDB is that the database resides in the server's main memory—virtual memory can also be used, but maximum performance is achieved when data fits the server's physical memory. Since

transactions do not have to wait for data to be fetched from disk, concurrency becomes less important for performance and IMDBs rely on simple concurrency control mechanisms [GMS92]. Some of them even consider very aggressive approaches such as *multiple-readers single-writer* [BK02] in which read-only transactions can execute concurrently, but update transactions are serialized.

Although IMDBs rely on main memory only for transaction execution, a transaction log must be kept on disk for recovery. Read-only transactions execute in main memory only; update transactions have to log information on disk before committing. In fact, storing information on disk is the main overhead of update transactions executing in an IMDB. To improve performance, disk writes can be deferred until after the transaction commits. This approach, however, risks losing data in case of database crashes. In replicated environments, though, durability can be ensured outside the database sites, by the termination protocol. Recall that our abstract deferred-update algorithm does not require the number of databases to be finite. Implementations can keep this characteristic and make use of databases without strong durability (or without durability at all). A database that crashes and recovers without transactions can simply assume a new identity and join the system as a new empty database.

The following sections present a low-overhead primary-backup protocol to handle replicated IMDBs that follows our deferred update abstraction. Our solution consists in orchestrating IMDBs without changing their internals. Our protocol exploits two features of IMDBs: simpler (and stronger) concurrency control mechanisms and deferred disk writes. Assuming databases offer a strong concurrency control allowed us to reduce to only two communication steps the latency needed to terminate update transactions, as seen by the application.

Deferred disk writes allowed us to reduce the overhead of committing update transactions. Transaction durability is ensured by the consistency property of the termination protocol. Traditional deferred update algorithms with standard off-the-shelf databases incur two disk accesses for the termination of update transactions: one done by the middleware to ensure correctness despite arbitrary crashes and recoveries, and another done by the database. By disabling durability at the databases, a single disk access is needed.

Our protocol is based on the primary-backup strategy. Read-only transactions can be processed at any replica. Update transactions are executed first by the primary, and then by the backups. From the application's viewpoint, the system behaves as a single-copy serializable database. Primary crashes are handled by electing a new primary whenever the current one fails. False failure suspicion caused by aggressive failure detection is handled by allowing more than one primary to coexist at the same time without violating correctness. Provided that failures and suspicions cease, the protocol ensures that the system converges to a state in which only one primary exists.

4.2.2 Concurrency Control Mechanism

We now explain in detail our assumptions about the concurrency control mechanism inside database replicas. First, in-memory database systems usually rely on lock-based concurrency control (two-phase locking [GR93]) for its simplicity, since IMDBs execute operations and transactions much faster than conventional disk-based databases. Lock-based concurrency control is a stronger form of order-preserving serializability, presented in Section 2.1. Not only does it guarantee that the commit order represents a correct transaction serialization, but it also ensures that a transaction can be committed after the execution of each of its operations. In simpler words, when the result of an operation is given, if it is different from $Aborted$, then the transaction responsible for that operation can have its current history serialized right after the last committed transaction.

The problem of lock-based concurrency control is the possibility of deadlocks. To deal with this, most database systems (whether in-memory or not) make use of timeouts. If a lock is requested but not granted within a certain time interval, the operation is canceled and the transaction responsible for it is aborted. Another property this simple approach gives to the database implementing it is that transactions cannot be blindly aborted, that is, transactions cannot have their locks released and be internally aborted in between operations (just during their execution). Complex databases might set priorities to transactions or use extra timeouts to abort transactions that hold locks for too long, which would invalidate this property. However, most in-memory databases we are aware of either do not provide such capabilities or allow them to be disabled for better performance.

Our complete assumptions about the concurrency control mechanism found in database replicas are given in Figure 4.1 below. The figure shows the required changes in our previous specification of an order-preserving serializable database. The first modified action is $ReplyReq$. Different from our previous specification, now a database can only reply to an operation if it is possible to come up with a complete serialization of all committed and running transactions, taking into consideration the current history of transactions that have not been decided yet. We define $undecidedSet$ to equal the set of all undecided transactions with some history, which may not include t if it is executing its first operation. In the pre-condition of this action, we use the postfix operator $'$ to represent the value of a variable after the action is taken. It is used on $thist$ to represent its value after the transaction history of t is extended with the operation being executed and its result is about to be given to the client.

Action $DoAbort$ expresses our assumption that transactions are not sporadically aborted when they are not executing an operation. Action $DoCommit$ is just simplified in this specification since $ReplyReq$ already guarantees that a running (undecided) transaction can always be committed next.

$ReplyReq(t \in Tid, rep \in Reply)$
Enabled iff:
- $q[t] \in Request$
- **if** $tdec[t] \in Decided$
  then
    $rep = tdec[t]$
  else
    $q[t] \in Op \ \wedge \ rep \in Result \wedge$
    $\forall \, sufix \in Perm(undecidedSet \cup t) \ :$
      $\exists \, st \in DBState \ :$
        $CorrectSerialization(serialSeq \circ sufix, thist', InitialDBState, st)$

Effect:
- $DBResponse(t, rep)$
- $q[t] \leftarrow NoReq$
- **if** $tdec[t] \notin Decided$ **then**
  $thist[t] \leftarrow thist[t] \circ \langle q[t], rep \rangle$

$DoAbort(t \in Tid)$
Enabled iff:
- $tdec[t] \notin Decided$
- $q[t] \in Op$

Effect:
- $tdec[t] \leftarrow Aborted$

$DoCommit(t \in Tid)$
Enabled iff:
- $tdec[t] \notin Decided$
- $q[t] = Commit$

Effect:
- $tdec[t] \leftarrow Committed$
- $serialSeq \leftarrow serialSeq \circ t$

Figure 4.1: Changes in our order-preserving serializability specification.

4.2.3 The Algorithm

Our replication protocol is similar to the protocol in Section 4.1. As for the agents we assume, the only difference is that the current algorithm does not assume a set of leaders. Termination certification is performed using one of the database replicas. The model assumed is basically the same, with the difference that a round now has a single primary database related to it instead of a leader.

The main idea of this protocol is to have active transactions executing on a single database site, the primary, in order to use its concurrency control mechanism to perform their certification for termination. If we disconsider failures, the concurrency control mechanism we assume for databases guarantees that all transactions that are not currently executing operations do not conflict with each other and could be committed or aborted in any order. Therefore, clients can simply propose their active histories to a sequence agreement protocol and databases can apply their updates in the selected order.

As before, we use rounds to overcome the problem of failures. Each round has a single primary responsible for processing active transactions (those requiring certification for

termination). When a client sends the first operation of a transaction to the primary, the response from the primary also brings the round number in which the transaction should be proposed. A round change will prevent the transaction from being chosen (and learned), automatically aborting it. We explain in detail how this mechanism works later. At this point, we want to concentrate on the main idea of the algorithm.

Instead of having clients propose active transactions to a leader responsible for ordering and forwarding them to acceptors, we take a different approach. Since clients have to contact the primary to submit operations of active transactions, their ordering can be calculated during the operation execution phase and the ordering information can be sent to the client together with the operation results. This allows clients to propose active transactions directly to the acceptors and releases the primary from the burden of dealing with "`propose`" messages.

The necessary total ordering of proposed transactions is done by using an infinite number of consensus instances (starting from 0), as explained in Section 3.1. Whenever the primary sends the result of an operation back to the client, it sends also the reference to a "free" instance of consensus the client can use to propose that transaction. The way these instances are selected makes sure that no two transactions are proposed for the same round and the same instance, which automatically avoids the problem of collisions.

Clients send their proposals directly to acceptors with no risk of collisions due to the ordering information provided by the primary. Acceptors accept transactions and forward them to databases and clients. A proposed transaction is chosen and can be safely learned by databases and clients if it has been accepted by a quorum of acceptors, which normally happens after two communication steps only. A transaction is aborted if the round it executed at has been overtaken by a new round without it being chosen in the process. The detailed description of the algorithm is given later in this section.

If instance gaps are formed when a new round starts (due to clients not making their proposals in time to get them chosen), the new primary can fill them with a special transaction $\perp$ that will not be considered by database replicas. In fact, to simplify some actions of the termination protocol, we initialize the system so that $\perp$ is already chosen and learned for instance 0.

Many variables and actions of the algorithm are equal to or resemble those of the previous one. A client c keeps the following variables:

$cthist[c]$, $cq[c]$, $cdreq[c]$, $cpdec[c]$, $cproposed[c]$: The same as in the previous protocol.

$crndof[c]$ A mapping from each transaction under the responsibility of c to the round in which it should be proposed. This variable is initialized during the execution of the algorithm.

$cinstof[c]$ A mapping from each transaction under the responsibility of c to the instance in which it should be proposed. This variable is also initialized during the execution of the algorithm.

$clearned[c]$ A mapping from each consensus instance i to a transaction id or a special value $NoTrans$. Initially, it maps all instances to $NoTrans$.

The variables of a database d are shown in the following. Variables with prefix p are used by d to start a new round coordinated by it.

$ddreply[d], dcnt[d], vers[d], dcom[d]$: The same as in the previous protocol.

$drnd[d]$: The round number database d is currently at, initially 0.

$prnd[d]$**:** The new round number d wants to start, if $prnd[d] \neq drnd[d]$, initially 0.

$pfproposal[d]$: A mapping from instances to values (pairs containing a transaction's id and active history) that might have been chosen in rounds previous to $prnd[d]$, in case $prnd[d] \neq drnd[d]$ (i.e., when d is starting a new round). Initially, it is an empty mapping for all databases but the primary of round 0, in which case $pfproposal[d]$ is a single mapping from instance 0 to special transaction $\perp$.

$dfreeinsts[d]$: A set containing all the instances that are free for round $rnd[d]$, if d is the primary of round $rnd[d]$. The primary of round 0 has it initialized to all instances but instance 0.

$dinstof[d]$: A mapping from each transaction t under the responsibility of d to the current consensus instance that is assigned to t or special value $nothing$. Initially, it maps each transaction to $nothing$.

$dlearned[d]$: A mapping from each instance i to a pair containing a transaction's id and active history or special value $none$. Initially it maps instance 0 to $\perp$ and all the others to $none$.

Acceptor a keeps the following three variables:

$arnd[a]$ The round number a is currently at, initially 0.

$vrnd[a]$ The highest-numbered round a has accepted a value at (in any instance). Different from the previous protocol, we do not keep the whole acceptance history of an acceptor. We do that to show how the protocol can be implemented efficiently.

$vval[a]$ A mapping from each instance i to the pair $\langle tid, acthist \rangle$ a has accepted at round $vrnd[a]$, or special value $none$ if a has not accepted any value at $vrnd[a]$ for instance i. $vval[a]$ can also be $\perp$ if $\perp$ was proposed to fill a gap at instance i. It is initially $none$ for all instances except for instance 0, which is set to $\perp$.

In the following we describe in detail the atomic actions performed by each agent of the algorithm. Instead of forwarding active transactions to the current primary during the execution, we assume $DBof(t)$ is so that it associates active transactions to the current primary. Active transactions executed at non-primary databases will not be able to be proposed. The assumption about $DBof(t)$ is practical because there is no restriction about the way transaction ids are created. One could think of an implementation where the transaction id is defined by a scheduling box outside the replicated database that creates the id right before the transaction starts executing in a way that it is directed to the correct replica. A simple example could have a transaction id as a record that has as one of its fields the database replica it should execute at.

$ReceiveReq(c, t, req)$ This action is very similar to the one described for our previous algorithm. For example, it is enabled by exactly the same pre-conditions:

- $c = ClientOf(t)$,
- $DBRequest(t, req)$ was triggered by the application layer, and
- $cq[c][t] = NoReq$

As the previous action, it sets $cq[c][t]$ to req, but performs a slightly different conditional action, described below.

if $t \notin cproposed[c] \wedge cpdec[c][t] \notin Decided$ **then**
  if $req = Commit \wedge$
    t has executed at least one active operation $\wedge$
    $DBof(t)$ is the primary of $crndof[c][t]$
    then
    - $cproposed[c] \leftarrow cproposed[c] \cup t$
    - Send $\langle$"2a", $cinstof[c][t]$, $crndof[c][t]$, t, $acthist\rangle$, where $acthist$ is the active history of t, to acceptors

  else
    - $cdreq[c][t] \leftarrow req$
    - Send $\langle$"dreq", t, req, $Len(cthist[c][t])\rangle$ to $DBof(t)$

The change makes sure that only active transactions that have executed operations at the correct primary of $crndof[c][t]$ are proposed. Moreover, proposals go directly to the acceptors in "2a" messages containing both the instance and the round for which the proposal is being made.

$ReplyReq(c, t, rep)$ This action is almost identical to the action with the same name in our previous algorithm. Since the variable used by clients to learn about globally committed and aborted transaction is different, it uses the following definition for operator $ctdec(c, t)$:

$ctdec(c,\ t) \triangleq$ **if** $t \notin cproposed[c]$

then $cpdec[c][t]$
else if $clearned[c][cinstof[c][t]] = NoTrans$
then $Unknown$
else if $clearned[c][cinstof[c][t]] = t$
then $Committed$
else $Aborted$

This definition basically says that a proposed transaction is committed if it is the decision of the instance in which it was proposed and is aborted if a different transaction was decided for that instance. The action is enabled iff:

- $c = ClientOf(t)$,
- $cq[c][t] \in Request$, and
- **if** $ctdec(c, t) \in Decided$ **then**
  - $rep = ctdec(c, t)$

  else
  - $cq[c][t] \in Op$,
  - $rep \in Result$, and
  - c received message $\langle$"drep", $t, rep, cnt, instof, rndof\rangle$ with $cnt > Len(cthist[c][t])$

It executes the following operations:

- $DBResponse(t, rep)$,
- $cq[c][t] \leftarrow NoReq$, and
- **if** $ctdec(c, t) \notin Decided$ **then**
  - $cthist[c][t] \leftarrow cthist[c][t] \circ \langle cq[c][t], rep\rangle$
  - $cdreq[c][t] \leftarrow NoReq$
  - $cinstof[c][t] \leftarrow instof$
  - $crndof[c][t] \leftarrow rndof$

The "drep" messages from $DBof(t)$ now carries on the information about the instance and the round at which t should be proposed if it is active. This information is stored on variables $cinstof[c][t]$ and $crndof[c][t]$.

$PrematureAbort(c, t)$ This exaction is exactly the same as in the previous algorithm. However, to avoid building gaps if passive transactions are also executed at the primary, it is recommended to execute it only if a message $\langle$"drep", $t, Aborted\rangle$ is received from $DBof(t)$.

$PassiveCommit(c, t)$ This action is exactly the same as in the previous algorithm.

$DBReq(d, t, req)$ If we define $dlearnedSeq[d]$ as being the sequence built out of $dlearned$ by starting at the first instance and appending every element that differs from $\perp$ until a value $none$ is reached for the first time, this action has the same enabling conditions as the same action in the previous algorithm, that is,

Condition 1 (external operation request)

- $d = DBof(t)$,
- $vers[d][t] = 0$, and
- d received message $\langle$"`dreq`", $t, req, len\rangle$ with $len = dcnt[d][t]$

Condition 2 (operation after termination)

- $\exists i \in 1..Len(dlearnedSeq[d])$:
  - $dlearnedSeq[d][i].trans = t$,
  - $dcnt[d][t] < Len(dlearnedSeq[d][i].acthist)$, and
  - $req = dlearnedSeq[d][i].acthist[dcnt[d][t] + 1]$

Condition 3 (commit after termination)

- $req = Commit$
- $\exists i \in 1..Len(dlearnedSeq[d])$:
  - $dlearnedSeq[d][i].trans = t$
  - $\forall j < i$: $dcom[d][dlearnedSeq[d][j].trans]$
- **either** $d = DBof(t) \wedge vers[d][t] = 0$
  or $dcnt[d][t] = Len(dlearnedSeq[d][i].acthist)$

It executes a few more actions than its counterpart though, since it releases the instance that was previously assigned to transaction t.

- $DB(d)!DBRequest(\langle t, vers[d][t]\rangle, req)$
- $dfreeinsts[d] \leftarrow dfreeinsts[d] \cup \{dinstof[d][t]\}$
- $dinstof[d][t] \leftarrow nothing$

$DBRep(d, t, rep)$ Executed by database d, for transaction t and database reply rep, in response to action $DB(d)!ReplyReq(\langle t, vers[d][t]\rangle, rep)$, triggered by the database replica controlled by d. It performs the following operations:

- **if** $d = DBof(t)$ **then**
  - $ddreply[d][t] = rep$
  - **if** $\nexists i$: $dlearned[d][i].tid = t$ **then**
    if $rep \in Decided$ **then** send message $\langle$"`drep`", $t, rep\rangle$ to $ClientOf(t)$
    else
    - Select instance $inst$ in $dfreeinsts[d]$ (lowest value to avoid gaps)
    - Send $\langle$"`drep`", $t, rep, dcnt[d][t]+1, inst, drnd[d]\rangle$ to $ClientOf(t)$
    - $dfreeinsts[d] \leftarrow dfreeinsts[d] \setminus \{inst\}$

- $dinstof[d][t] \leftarrow inst$

- **if** $rep = Aborted \ \wedge \ \exists i \ : \ dlearned[d][i].tid = t$ **then**
  - $vers[d][t] \leftarrow vers[d][t] + 1$
  - $dcnt[d][t] \leftarrow 0$

  else if $rep \in Result$ **then**
  - $dcnt[d][t] \leftarrow dcnt[d][t] + 1$

  else
  - $dcom[d][t] \leftarrow rep = Committed$

$Phase1a(p, r)$ Executed by primary p, for round r. In this action, p starts a new round r to overcome the possible failure of a previous primary or because it suspects one of its clients has crashed without proposing a transaction, which would leave a gap in the instances and prevent the liveness of termination.
It is enabled iff:

- p is the leader of round r and
- $prnd[p] < r$

As before, these pre-conditions ensures safety but can prevent progress without further constraints. We discuss liveness requirements on the following section. The action performs the following operations:

- $prnd[p] \leftarrow r$
- $pfproposal[p] \leftarrow$ special value $calculating$
- Send message $\langle$"`1a`", $r\rangle$ to acceptors

$Phase1b(a, r)$ Executed by acceptor a to start on round r. It is enabled iff:

- a receives a message $\langle$"`1a`", $r\rangle$ and
- $arnd[a] < r$

The action sets $arnd[a]$ to r and sends message $\langle$"`1b`", $r, a, vrnd[a], vval[a]\rangle$ back to the primary of round r.

$Phase2Start(p, r)$ Executed by primary p, for round r. In this action, p calculates the value of $pfproposal$ based on the "`1b`" messages it received from a quorum of acceptors.
It is enabled iff:

- p is the primary of round r,
- $prnd[p] = r$,
- $drnd[p] < prnd[p]$,
- $pfproposal[p] = calculating$, and

- p has received "1b" messages for round r from a quorum Q of acceptors.

To build $pfproposal[p]$, p first calculates the highest value of $vrnd$ it received in the "1b" messages. For each instance i, if all values $vval[i]$ received equal $none$, $pfproposal[p][i]$ is set to $none$ as well; otherwise it is set to any of the values (which are guaranteed to be the same). After this initial calculation, gaps in $pfproposal[p]$ at instances lower than some instance i to which $pfproposal[p][i]$ was defined are filled with $\perp$. In the end, $pfproposal[p]$ maps each instance i lower than some upper bound to either a value that might have been chosen in a previous round or $\perp$. This will be the initial proposal for round r and primary p cannot start accepting active transactions for round r until it receives an acceptance confirmation for it from the acceptors and it has learned all the values in $pfproposal[p]$. Therefore, p sends a message $\langle$"2aS", r, $pfproposal[p]\rangle$ to all acceptors.

$Phase2b(a, r)$ Executed by acceptor a, for round r.

When a receives a message $\langle$"2aS", r, $pfproposal\rangle$ such that $arnd[a] \leq rnd$ and $vrnd[a] < r$, it does the following:

- $vval[a] \leftarrow pfproposal$
- $vrnd[a] \leftarrow r$
- $arnd[a] \leftarrow r$
- Send $\langle$"2bS", r, a, $inst$, $vval\rangle$ to databases and clients

When a receives a message $\langle$"2a", $inst$, r, t, $acthist\rangle$ such that $arnd[a] \leq r$, $vrnd[a] = r$, and $vval[a][inst] = none$, it takes the following actions:

- $vval[a][inst] \leftarrow \langle t, acthist\rangle$
- $vrnd[a] \leftarrow r$
- $arnd[a] \leftarrow r$
- Send $\langle$"2b", r, a, $inst$, t, $acthist\rangle$ to databases and clients

$ActivateRound(p, r)$ Executed by primary p, for round r. After action $Phase2Start$ is executed, the value of $drnd[p]$ remains the same old value lower than $prnd[p]$. This means that active transactions executing on p will not be proposed under the new round. This is paramount to the correctness of the protocol since p can only allow active transactions to be proposed after it is guaranteed that all transactions committed at previous rounds are committed inside its local database, so that they can be correctly serialized with the currently running transactions. This action is responsible for updating $drnd[p]$, thus allowing new active transactions to be proposed at the new round.

It is enabled iff:

- p is the primary of round r,
- $prnd[p] = r$,

- $drnd[p] < prnd[p]$,
- $pfproposal[p] = calculating$,
- $dlearned[p]$ matches $pfproposal[p]$,
- every transaction t in $pfproposal[p]$ has $dcom[p][t] = \mathbf{true}$
- p has received "2bS" messages for round r from a quorum Q of

These pre-conditions make sure that all previously committed transactions were learned by p and have been committed at its local database. More than that, it guarantees that acceptors have received and acknowledged the "1bS" message sent. At this point p can allow active transactions running on it to be proposed under the new round. The action executes the following operations:

- $drnd[p] \leftarrow prnd[p]$
- $dfreeinsts[p] \leftarrow$ instances not mapped in $pfproposal[p]$
- $\forall t \ : \ dinstof[p][t] \leftarrow nothing$

$ClientLearn(c)$ Executed by client c. The action is enabled if c has received "2b" messages for the same round and instance from a quorum of acceptors, or if c has received "2bS" messages for the same round from a quorum of acceptors. c verifies if any value has been chosen (accepted by a quorum for the same round and instance) and updates $clearned[c]$ accordingly.

$DatabaseLearn(d)$ Executed by database d. It is very similar to $ClientLearn$, with the difference that $dlearned[d][i]$ keeps both the transaction id and the active history of the transaction chosen for instance i, whereas $clearned[c][i]$ keeps only the transaction id.

4.2.4 Correctness and Optimizations

The correctness of our in-memory primary-backup replication protocol can be proved using the same approach as our certification-based algorithm from Section 4.1. In fact, if we define $dlearnedSeq[d]$ the same way we did for action $DBRep$ above, we can practically use the same refinement mapping, except for variable $gdec$. The mapping for $gdec[t]$ is different because transactions are globally committed or aborted differently in the current protocol. To come up with a mapping for the abstract variable $gdec$ we must extend the algorithm with a history variable, that is, variables that keep some history of the protocol execution. In our case, we need only to define a variable $ahist$ to keep the acceptance history of acceptors in the following way.

$ahist[a][r][i]$: The unique value accepted by acceptor a at round r for instance i, or $none$ if nothing was accepted by a for that round and instance.

A proposed transaction t is globally committed if it has been chosen, that is, if has been accepted by a quorum of acceptors for the same round and instance.

$$
\begin{array}{l}
Committed(t) \triangleq \\
\quad \exists Q \in Quorum, r \in \mathbb{N}, i \in \mathbb{N} : \\
\quad\quad \bullet\ ahist[a][r][i] \notin \{none, \bot\} \\
\quad\quad \bullet\ ahist[a][r][i].trans = t
\end{array}
$$

A transaction t is globally aborted if it has been proposed for some instance i and a different transaction is chosen for that instance.

$$
\begin{array}{l}
Aborted(t) \triangleq \\
\quad \bullet\ t \in cproposed[ClientOf(t)] \\
\quad\quad \exists Q \in Quorum, r \in \mathbb{N}, i \in \mathbb{N} : \\
\quad\quad\quad - \ ahist[a][r][i] \notin \{none, \bot\} \\
\quad\quad\quad - \ ahist[a][r][i].trans \neq t
\end{array}
$$

We can now easily create a mapping for termination variable $gdec$ as follows:

$$
gdec[t] \triangleq \begin{cases} Committed & \text{, if } Committed(t) \\ Aborted & \text{, if } Aborted(t) \\ Unknown & \text{, otherwise} \end{cases}
$$

The proof that actions not related to termination implement the actions with the same name in our deferred-update abstraction given this refinement mapping follows exactly the same steps (and uses the same invariants) presented in Section 4.1.4 and there is no need for us to repeat them here.

As for termination, again we are using a simple variant of Paxos based on the optimization presented in [GL06]. The leader is implemented by the primary but it can give up the right to send "`2a`" messages to other processes as long as it is guaranteed that there will be no two such messages for the same instance and round but with different values being proposed. Different from [GL06], in our algorithm transactions can dynamically change the instance and round at which they should be proposed. At each interaction with the primary, an active transaction can be assigned to a different instance and round. Nevertheless, our algorithm guarantees that no instance is assigned to different transactions at the same round.

Paxos ensures that no consensus instance will ever decide on two different values. This solves the sequence agreement part of the termination protocol. It is still necessary to show that all the sequence of committed active transactions is serializable and the serialization order respects the common sequence being learned by databases. When a primary starts a new round, action $ActivateRound$ guarantees that all previously committed transactions have been learned and applied to its local database. More than that, it guarantees that acceptors have completely accepted the initial proposal for the round and no transactions

proposed on previous rounds will ever be chosen. The strong consistency guarantees of the primary database do the rest of the work, ensuring that any subset of the running transaction can be committed at any order.

The liveness guarantees for termination are basically the same of Paxos, with the leader election algorithm being responsible for electing the current primary. The only difference is that now a primary might want to start new rounds to force the abortion of transactions being executed by suspected clients. If the primary does not do that, it risks creating gaps in the learning sequence, preventing committed transactions from being learned by database replicas.

Other important practical liveness issue is the fact that some running transactions can hold locks that will prevent new transactions from being learned. Since we do not constrain the number of databases in the system. It is always possible for replicas to simulate a crash and assume a new identity in the system, thus releasing any locks. In practice this is not necessary, though. There are two conditions that can safely allow a database to simply abort running transactions:

- The transaction is read-only and will not be proposed.
- The transaction has been aborted, because the instance in which it can be proposed has decided a different value.

Databases can keep track of the necessary information in order to check these conditions and abort transactions that might be holding important locks.

As for implementation optimizations, mappings considering all the instances can be easily compressed since there is always only a finite number of instances mapped to a value different from $none$. Moreover, $clearned[c]$ can consider only the instances in which c has proposed something and variable $crndof[c][t]$ can be optimized to keep only one value—the most recent round c has heard of—, since old transactions can be automatically aborted by the client.

Last, as it was the case for the algorithm presented in Section 4.1, clients and databases do not have to keep their state in stable storage. Since their number is not limited by the algorithm, crashed processes can just return to the system with an empty state as long as they assume a different identity. In-memory databases usually offer the option of completely disabling disk access, even for update transactions. This feature can be exploited by this protocol and may result in significant performance gains.

We implemented a simplified version of this algorithm in [CPS06] and obtained some performance results for it. Reducing the latency of the termination protocol from 3 to 2 communication steps represented a gain of of 8%–25% in the response time of update

transactions. By disabling durability at the replicas, a single disk access was performed by acceptors to commit active transactions. In our experimental setup, such an improvement represented a reduction of 51%–64% in the termination latency of update transactions.

4.3 TLA$^+$ Specifications

4.3.1 Module *CertificationBased*

This module presents the complete specification of the Certification-based algorithm presented in 4.1.

MODULE *CertificationBased*

EXTENDS *DatabaseConstants*, *DBInterface*

CONSTANTS *Client*, *Database*, *Acceptor*, *Quorum*, *Leader*, *LeaderOf*(_), *DBof*(_), *ClientOf*(_), *StripPassive*(_)

Required Assumptions

We assume state-deterministic operations

ASSUME $\forall\, op \in Op,\ res1,\ res2 \in Result,\ st,\ st1,\ st2 \in DBState :$
$\land\ CorrectOp(op,\ res1,\ st,\ st1)$
$\land\ CorrectOp(op,\ res2,\ st,\ st2)$
$\Rightarrow\ st1 = st2$

Histories that go through the *StripPassive* operation generate the same final state as the original history.

ASSUME $\forall\, hist \in THist,\ st1,\ st2 \in DBState :$
$\land\ StripPassive(hist) \in THist$
$\land\ CorrectAtomicHist[hist,\ st1,\ st2] \Rightarrow$
$CorrectAtomicHist[StripPassive(hist),\ st1,\ st2]$

Every transaction has a single client and a single database replica associated with.

ASSUME $\forall\, t \in Tid :\ \land\ DBof(t) \in Database$
$\land\ ClientOf(t) \in Client$

Every round r has a single leader, and every leader is the leader of a round greater than r.

ASSUME $\forall\, r \in Nat :\ \land\ LeaderOf(r) \in Leader$
$\land\ \forall\, c \in Leader : \exists\, r2 \in Nat :\ \land\ r < r2$
$\land\ c = LeaderOf(r2)$

We assume that quorums are finite subsets of the acceptors and every pair of quorums has a non-empty intersection.

ASSUME $\forall\, i \in Nat :$

$\land\ Quorum \subseteq \text{SUBSET } Acceptor$
$\land\ \forall\, Q \in Quorum : IsFiniteSet(Q)$
$\land\ \forall\, j \in Nat :$
$\quad \forall\, Q \in Quorum,\ R \in Quorum : Q \cap R \neq \{\}$

Variables

VARIABLES $cthist$, cq, $cdreq$, $cpdec$, $cproposed$, $cgdec$, — Client variables
$ddreply$, $dcnt$, $vers$, $dcom$, $dlearnedSeq$, — External database variables
$ldinter$, $dthist$, $dtdec$, dq, $dserialSeq$, — Internal database variables
$lrnd$, $lval$, — Leader variables
$arnd$, $ahist$, — Acceptor variables
$msgs$, $amLeader$ — Extra variables for message passing and leader election

Auxiliary definitions to help dealing with the declared variables.

$cvars \triangleq \langle cthist,\ cq,\ cdreq,\ cpdec,\ cproposed,\ cgdec\rangle$
$ldvars \triangleq \langle ldinter,\ dthist,\ dtdec,\ dq,\ dserialSeq\rangle$
$gdvars \triangleq \langle ddreply,\ dcnt,\ vers,\ dcom,\ dlearnedSeq\rangle$
$dvars \triangleq \langle gdvars,\ ldvars\rangle$
$lvars \triangleq \langle lrnd,\ lval\rangle$
$avars \triangleq \langle arnd,\ ahist\rangle$

Definitiosn regarding internal database variables and database replicas

Each database accepts transactions with ids in the form $\langle tid,\ version\rangle$ where tid is an element of Tid and version is a Natural. This allows "a single" transaction to be submitted to a database multiple times.

$LocalTid \triangleq Tid \times Nat$

The definition below instantiates each local database used by the general algorithm.

$DBS(d) \triangleq \text{INSTANCE } AOPSerializableDB \text{ WITH } Tid \leftarrow LocalTid,$
$DBinter \leftarrow ldinter[d],$
$thist \leftarrow dthist[d],$
$tdec \leftarrow dtdec[d],$
$q \leftarrow dq[d],$
$serialSeq \leftarrow dserialSeq[d]$

External variables' Types

Client variables

$cthistType \triangleq [Client \to THistVector]$
$cqdreqType \triangleq [Client \to [Tid \to Request \cup \{NoReq\}]]$
$cdecType \triangleq [Client \to [Tid \to Decided \cup \{Unknown\}]]$

$cproposedType \triangleq [Client \to \text{SUBSET } Tid]$

Values propagated in messages and kept by leaders, acceptors, as well as learned by databases.

$UpdatesType \triangleq [trans : Tid,\ acthist : FSeq(Op)]$
$ValType \triangleq [seq : FSeq(UpdatesType),\ aborted : \text{SUBSET } Tid]$

Database variables

$NoRep \triangleq \text{CHOOSE } v : v \notin Reply$ Not a valid reply
$ddreplyType \triangleq [Database \to [Tid \to Reply \cup \{NoRep\}]]$
$dcntversType \triangleq [Database \to [Tid \to Nat]]$
$dcomType \triangleq [Database \to [Tid \to \text{BOOLEAN }]]$
$dlearnedSeqType \triangleq [Database \to Seq(UpdatesType)]$

Coordinator variables

$none \triangleq \text{CHOOSE } v : v \notin ValType$ Not a valid value
$lrndType \triangleq [Leader \to Nat]$
$lvalType \triangleq [Leader \to ValType \cup \{none\}]$

Acceptor variables

$arndType \triangleq [Acceptor \to Nat]$
$ahistType \triangleq [Acceptor \to [Nat \to ValType]]$

Other variables

$amLeaderType \triangleq [Leader \to \text{BOOLEAN }]$
$msgsType \triangleq$
$[type : \{\text{“dreq”}\},\ trans : Tid,\ dreq : Request,\ len : Nat] \cup$
$[type : \{\text{“drep”}\},\ trans : Tid,\ cnt : Nat,\ dreply : Result] \cup$
$[type : \{\text{“drep”}\},\ trans : Tid,\ dreply : Decided] \cup$
$[type : \{\text{“propose”}\},\ trans : Tid,\ hist : THist] \cup$
$[type : \{\text{“1a”}\},\ rnd : Nat] \cup$
$[type : \{\text{“1b”}\},\ rnd : Nat,\ acc : Acceptor,\ vrnd : Nat,\ vval : ValType] \cup$
$[type : \{\text{“2a”}\},\ rnd : Nat,\ val : ValType] \cup$
$[type : \{\text{“2b”}\},\ rnd : Nat,\ acc : Acceptor,\ val : ValType]$

Initialization

$ClientInit \triangleq$
$\land\ cthist = [c \in Client \mapsto [t \in Tid \mapsto \langle\rangle]]$
$\land\ cq = [c \in Client \mapsto [t \in Tid \mapsto NoReq]]$
$\land\ cdreq = [c \in Client \mapsto [t \in Tid \mapsto NoReq]]$
$\land\ cpdec = [c \in Client \mapsto [t \in Tid \mapsto Unknown]]$
$\land\ cproposed = [c \in Client \mapsto \{\}]$
$\land\ cgdec = [c \in Client \mapsto [t \in Tid \mapsto Unknown]]$

$DatabaseInit \triangleq$

$$
\begin{array}{l}
\quad \wedge\ ddreply \quad\ = [d \in Database \mapsto [t \in Tid \mapsto NoRep]] \\
\quad \wedge\ dcnt \quad\ = [d \in Database \mapsto [t \in Tid \mapsto 0]] \\
\quad \wedge\ vers \quad\ = [d \in Database \mapsto [t \in Tid \mapsto 0]] \\
\quad \wedge\ dcom \quad\ = [d \in Database \mapsto [t \in Tid \mapsto \text{FALSE}]] \\
\quad \wedge\ dlearnedSeq = [d \in Database \mapsto \langle\rangle] \\
\quad \wedge \forall\, d \in Database : DBS(d)!AOPInit
\end{array}
$$

$$
\begin{array}{l}
LeaderInit \triangleq \\
\quad \wedge\ lrnd = [l \in Leader \mapsto 0] \\
\quad \wedge\ lval\ = [l \in Leader \mapsto \text{IF } l = LeaderOf(0) \\
\qquad\qquad\qquad\qquad\qquad \text{THEN } [seq \mapsto \langle\rangle,\ aborted \mapsto \{\}] \\
\qquad\qquad\qquad\qquad\qquad \text{ELSE } none]
\end{array}
$$

$$
\begin{array}{l}
AcceptorInit \triangleq \\
\quad \wedge\ arnd = [a \in Acceptor \mapsto 0] \\
\quad \wedge\ ahist = [a \in Acceptor \mapsto \\
\qquad\qquad\qquad [r \in Nat \mapsto \text{IF } r = 0 \\
\qquad\qquad\qquad\qquad\qquad\qquad \text{THEN } [seq \mapsto \langle\rangle,\ aborted \mapsto \{\}] \\
\qquad\qquad\qquad\qquad\qquad\qquad \text{ELSE } none \\
\qquad\qquad\qquad] \\
\qquad\qquad]
\end{array}
$$

$$
\begin{array}{l}
OtherInit \triangleq \\
\quad \wedge\ msgs = \{\} \\
\quad \wedge\ amLeader \in amLeaderType
\end{array}
$$

$$
\begin{array}{l}
Init \triangleq \wedge\ InitInterface \\
\qquad\quad\ \wedge\ ClientInit \\
\qquad\quad\ \wedge\ DatabaseInit \\
\qquad\quad\ \wedge\ LeaderInit \\
\qquad\quad\ \wedge\ AcceptorInit \\
\qquad\quad\ \wedge\ OtherInit
\end{array}
$$

Auxiliary actions and operators

The algorithm below compares whether a value, a record containing both a sequence of committed transactions (with their active histories) and a set of aborted transactions, precedes another value or not, that is, whether the contents of the second value extend those of the first value. Since only compatible values are checked unsing this operator, it suffices to compare the length of both sequences and the cardinality of their sets.

$$
\begin{array}{l}
a \preceq b \triangleq \wedge\ Len(a.seq) \leq Len(b.seq) \\
\qquad\qquad\ \wedge\ Cardinality(a.aborted) \leq Cardinality(b.aborted)
\end{array}
$$

$$a \prec b \triangleq a \preceq b \wedge a \neq b$$

$ActHist(c,\ t)$ returns the current history of transaction t with some of its passive operations taken out of the sequence (according to operator $StripPassive$).

$ActHist(c,\ t) \triangleq StripPassive(cthist[c][t])$

$OpSeq(h)$ returns a sequence containing only the operations of a history (without the operations' results).

$OpSeq(h) \triangleq [i \in \text{DOMAIN } h \mapsto h[i].op]$

$Send(m)$ sends message m

$Send(m) \triangleq msgs' = msgs \cup m$

$DBvars(d)$ returns the internal variables of database d.

$DBvars(d) \triangleq \langle ldinter[d],\ dthist[d],\ dtdec[d],\ dq[d],\ dserialSeq[d]\rangle$

$OtherDBsStutter(d)$ is an action that forces all databases but d to execute a stuttering step, that is, a step in which their internal variables do not change values. For simplicity, our specification does not allow interleaving of database actions. In fact, as we explain in the following, it does not allow interleaving at all.

$$\begin{array}{l} OtherDBsStutter(d) \triangleq \\ \quad \text{LET } dbfn \triangleq [nd \in (Database \setminus \{d\}) \mapsto DBvars(nd)] \\ \quad \text{IN } \ dbfn' = dbfn \end{array}$$

If B is a set of round numbers that contains a maximum element, then $Max(B)$ is defined to equal that maximum. Otherwise, its value is unspecified.

$Max(B) \triangleq \text{CHOOSE } i \in B : \forall j \in B : j \leq i$

Atomic Actions

These are the atomic actions of the algorithm, not including the internal database actions. In order to model check this specification, we had to make it noninterleaving, that is, we had to specify it in terms of actions that cannot occur concurrently (even considering that they are executed by different specification components). This prevented us from using the *DBRequest* and *DBResponse* primitives to interact with the internal databases. Instead, we used the *ReceiveReq* and *ReplyReq* actions directly to submit an operation and get a response from a database.

The *ReceiveReq* action.

$$\begin{array}{l} ReceiveReq(c,\ t,\ req) \triangleq \\ \quad \land\ c = ClientOf(t) \\ \quad \land\ DBRequest(t,\ req) \\ \quad \land\ cq[c][t] \notin Request \\ \quad \land\ cq' = [cq \text{ EXCEPT } ![c][t] = req] \\ \quad \land \text{ IF } t \notin cproposed[c] \land cpdec[c][t] \notin Decided \\ \qquad \text{THEN } \lor\ \land\ req = Commit \\ \qquad\qquad\qquad\ \land\ Len(ActHist(c,\ t)) > 0 \\ \qquad\qquad\qquad\ \land\ cproposed' = [cproposed \text{ EXCEPT } ![c] = @ \cup t] \\ \qquad\qquad\qquad\ \land\ Send([type \mapsto \text{“propose”},\ trans \mapsto t, \\ \qquad\qquad\qquad\qquad\qquad\ hist\ \mapsto cthist[c][t]]) \\ \qquad\qquad\qquad\ \land \text{ UNCHANGED } cdreq \\ \qquad\qquad\ \lor\ \land\ req = Commit \Rightarrow Len(ActHist(c,\ t)) = 0 \\ \qquad\qquad\qquad\ \land\ cdreq' = [cdreq \text{ EXCEPT } ![c][t] = req] \end{array}$$

$$
\begin{array}{l}
\qquad\qquad\qquad \land\ Send([type \mapsto \text{“dreq”},\ trans \mapsto t, \\
\qquad\qquad\qquad\qquad\qquad dreq \mapsto req,\ len \mapsto Len(cthist[c][t])]) \\
\qquad\qquad\qquad \land\ \text{UNCHANGED}\ cproposed \\
\qquad\quad \text{ELSE UNCHANGED}\ \langle cdreq,\ cproposed,\ msgs\rangle \\
\land\ \text{UNCHANGED}\ \langle cthist,\ cpdec,\ cgdec,\ lvars,\ dvars,\ avars,\ amLeader\rangle
\end{array}
$$

Action *ReplyReq* below uses the following definition for $ctdec(c,\ t)$, which calculates $tdec[t]$ based only on variables kept at client c.

$$
\begin{array}{l}
ctdec(c,\ t) \triangleq \text{IF}\ t \notin cproposed[c] \\
\qquad\qquad\qquad\quad \text{THEN}\ cpdec[c][t] \\
\qquad\qquad\qquad\quad \text{ELSE}\ \ cgdec[c][t]
\end{array}
$$

The *ReplyReq* action.

$$
\begin{array}{l}
ReplyReq(c,\ t,\ rep) \triangleq \\
\quad \land\ c = ClientOf(t) \\
\quad \land\ cq[c][t] \in Request \\
\quad \land\ DBResponse(t,\ rep) \\
\quad \land\ cq' = [cq\ \text{EXCEPT}\ ![c][t] = NoReq] \\
\quad \land\ \text{IF}\ ctdec(c,\ t) \in Decided \\
\qquad\quad \text{THEN}\ \land\ rep = ctdec(c,\ t) \\
\qquad\qquad\qquad\ \land\ \text{UNCHANGED}\ \langle cthist,\ cdreq\rangle \\
\qquad\quad \text{ELSE}\ \ \land\ cq[c][t] \in Op \\
\qquad\qquad\qquad\ \land\ rep\ \in Result \\
\qquad\qquad\qquad\ \land\ \exists\, m \in msgs : \\
\qquad\qquad\qquad\qquad\ \land\ m.type = \text{“drep”} \\
\qquad\qquad\qquad\qquad\ \land\ m.trans = t \\
\qquad\qquad\qquad\qquad\ \land\ m.dreply = rep \\
\qquad\qquad\qquad\qquad\ \land\ m.cnt > Len(cthist[t]) \\
\qquad\qquad\qquad\qquad\ \land\ cthist' = [cthist\ \text{EXCEPT}\ ![c][t] = Append(@,\ [op\ \mapsto cq[c][t], \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad res \mapsto rep])] \\
\qquad\qquad\qquad\qquad\ \land\ cdreq' = [cdreq\ \text{EXCEPT}\ ![c][t] = NoReq] \\
\quad \land\ \text{UNCHANGED}\ \langle cproposed,\ cpdec,\ cgdec,\ lvars,\ dvars,\ avars,\ msgs,\ amLeader\rangle
\end{array}
$$

The *PrematureAbort* action.

$$
\begin{array}{l}
PrematureAbort(c,\ t) \triangleq \\
\quad \land\ c = ClientOf(t) \\
\quad \land\ t \notin cproposed[c] \\
\quad \land\ cpdec[c][t] \notin Decided \\
\quad \land\ cpdec' = [cpdec\ \text{EXCEPT}\ ![c][t] = Aborted] \\
\quad \land\ \text{UNCHANGED}\ \langle cthist,\ cq,\ cdreq,\ cproposed,\ cgdec,\ lvars, \\
\qquad\qquad\qquad dvars,\ avars,\ msgs,\ DBinter,\ amLeader\rangle
\end{array}
$$

The *PassiveCommit* action.

$$
\begin{array}{l}
PassiveCommit(c,\ t) \triangleq \\
\quad \land\ c = ClientOf(t)
\end{array}
$$

```
    ∧ t ∉ cproposed[c]
    ∧ cpdec[t] ∉ Decided
    ∧ ∃ m ∈ msgs :
        ∧ m.type = "drep"
        ∧ m.trans = t
        ∧ m.dreply = Committed
        ∧ cpdec' = [cpdec EXCEPT ![c][t] = Committed]
        ∧ UNCHANGED ⟨cthist, cq, cdreq, cproposed, cgdec, dvars,
                     lvars, avars, msgs, DBinter, amLeader⟩
```

The *DBReq* action with its three enabling conditions.

```
DBReq(d, t, req) ≜
  ∧   ∨ ∧ d = DBof(t)                                  Condition 1
        ∧ vers[d][t] = 0
        ∧ ∃ m ∈ msgs :
            ∧ m.type = "dreq"
            ∧ m.trans = t
            ∧ m.dreq = req
            ∧ dcnt[d][t] = m.len
      ∨ ∃ i ∈ DOMAIN dlearnedSeq[d] :                  Condition 2
          ∧ dlearnedSeq[d][i].trans = t
          ∧ dcnt[d][t] < Len(dlearnedSeq[d][i].acthist)
          ∧ req = dlearnedSeq[d][i].acthist[dcnt[d][t] + 1]
      ∨ ∧ req = Commit                                 Condition 3
        ∧ ∃ i ∈ 1 .. Len(dlearnedSeq[d]) :
            ∧ dlearnedSeq[d][i].trans = t
            ∧ ∀ j ∈ 1 .. i − 1 : dcom[d][dlearnedSeq[d][j].trans]
            ∧ ∨ d = DBof(t) ∧ vers[d][t] = 0
              ∨ dcnt[d][t] = Len(dlearnedSeq[d][i].acthist)
  ∧   DBS(d)!ReceiveReq(⟨t, vers[d][t]⟩, req)
  ∧   OtherDBsStutter(d)
  ∧   UNCHANGED ⟨cvars, ddreply, dcnt, vers, dcom, dlearnedSeq, lvars, avars,
                 DBinter, msgs, amLeader⟩
```

The *DBRep* action.

```
DBRep(d, t, rep) ≜
  ∧   DBS(d)!ReplyReq(⟨t, vers[d][t]⟩, rep)
  ∧   OtherDBsStutter(d)
  ∧   IF d = DBof(t)
      THEN ∧ ddreply' = [ddreply EXCEPT ![d][t] = rep]
           ∧ ∨ ∧ ¬∃ i ∈ DOMAIN dlearnedSeq[d] :
                     dlearnedSeq[d][i].trans = t
               ∧ rep ∈ Result
               ∧ Send([type ↦ "drep", trans ↦ t,
```

$$
\begin{array}{l}
\qquad\qquad\qquad\qquad\qquad\qquad cnt \mapsto dcnt[d][t] + 1,\ dreply \mapsto rep]) \\
\qquad\qquad\quad \lor\ \land \neg \exists\, i \in \text{DOMAIN } dlearnedSeq[d] : \\
\qquad\qquad\qquad\qquad\qquad dlearnedSeq[d][i].trans = t \\
\qquad\qquad\qquad\ \land\ rep \in Decided \\
\qquad\qquad\qquad\ \land\ Send([type \mapsto \text{“drep”},\ trans \mapsto t,\ dreply \mapsto rep]) \\
\qquad\qquad\quad \lor\ \land \exists\, i \in \text{DOMAIN } dlearnedSeq[d] : \\
\qquad\qquad\qquad\qquad\qquad dlearnedSeq[d][i].trans = t \\
\qquad\qquad\qquad\ \land\ \text{UNCHANGED } \langle msgs \rangle \\
\qquad\quad \text{ELSE UNCHANGED } \langle ddreply,\ msgs \rangle \\
\land\quad \text{IF } \land\ rep = Aborted \\
\qquad\quad \land \exists\, i \in \text{DOMAIN } dlearnedSeq[d] : \\
\qquad\qquad\quad dlearnedSeq[d][i].trans = t \\
\qquad\quad \text{THEN } \land\ vers' = [vers \text{ EXCEPT } ![d][t] = @ + 1] \\
\qquad\qquad\quad \land\ dcnt' = [dcnt \text{ EXCEPT } ![d][t] = 0] \\
\qquad\qquad\quad \land\ \text{UNCHANGED } dcom \\
\qquad\quad \text{ELSE IF } rep \in Result \\
\qquad\qquad\quad \text{THEN } \land\ dcnt' = [dcnt \text{ EXCEPT } ![d][t] = @ + 1] \\
\qquad\qquad\qquad\qquad \land\ \text{UNCHANGED } \langle dcom,\ vers \rangle \\
\qquad\qquad\quad \text{ELSE } \land\ dcom' = [dcom \text{ EXCEPT } ![d][t] = (rep = Committed)] \\
\qquad\qquad\qquad\qquad \land\ \text{UNCHANGED } \langle dcnt,\ vers \rangle \\
\land\quad \text{UNCHANGED } \langle cvars,\ dlearnedSeq,\ lvars,\ avars,\ DBinter,\ amLeader \rangle
\end{array}
$$

Action $Phase1a(p,\ r)$ is executed by the leader l of round r as specified in the thesis. For progress, l can only execute this action if it believes to be the leader and either l has received a message related to a round between $lrnd[l]$ and r.

$$
\begin{array}{l}
Phase1a(l,\ r) \triangleq \\
\quad \land\ amLeader[l] \\
\quad \land\ l = LeaderOf(r) \\
\quad \land\ lrnd[l] < r \\
\quad \land \exists\, msg \in \{m \in msgs : m.type \in \{\text{“1a”},\ \text{“1b”},\ \text{“2a”},\ \text{“2b”}\}\} : \\
\qquad\quad \land\ lrnd[l] \quad < msg.rnd \\
\qquad\quad \land\ msg.rnd < r \\
\quad \land\ lrnd' = [lrnd \text{ EXCEPT } ![l] = r] \\
\quad \land\ lval' \ = [lval \text{ EXCEPT } ![l] \ = none] \\
\quad \land\ Send([type \mapsto \text{“1a”},\ rnd \mapsto r]) \\
\quad \land\ \text{UNCHANGED } \langle cvars,\ dvars,\ avars,\ DBinter,\ amLeader \rangle
\end{array}
$$

Action $Phase1b(a,\ r)$ is executed by acceptor a, for round r.

$$
\begin{array}{l}
Phase1b(a,\ r) \triangleq \\
\quad \land\ [type \mapsto \text{“1a”},\ rnd \mapsto r] \in msgs \\
\quad \land\ arnd[a] < r \\
\quad \land\ arnd' = [arnd \text{ EXCEPT } ![a] = r] \\
\quad \land \text{LET } vrnd \triangleq \text{CHOOSE } i \in Nat : \\
\qquad\qquad\qquad\qquad \land\ ahist[a][i] \neq none \\
\qquad\qquad\qquad\qquad \land \forall\, j \in \{k \quad \in Nat : k > i\} : ahist[a][j] = none
\end{array}
$$

```
    IN    Send([type ↦ "1b", rnd ↦ r, acc ↦ a, vrnd ↦ vrnd, vval ↦ ahist[a][vrnd]])
  ∧ UNCHANGED ⟨cvars, dvars, lvars, ahist, DBinter, amLeader⟩
```

DistProvedSafe$(Q,\ r,\ 1bMsg)$ below returns a possibly chosen value (extending all chosen ones) based only on the "1b" messages sent by acceptors in Q for round r.

```
DistProvedSafe(Q, r, 1bMsg) ≜
  LET k  ≜ Max({1bMsg[a].vrnd : a ∈ Q})
      AS ≜ {a ∈ Q : ∧ 1bMsg[a].vrnd = k}
      S  ≜ {1bMsg[a].vval : a ∈ AS}
  IN  CHOOSE v ∈ S : ∀ u ∈ S : u ⪯ v
```

Action *Phase2Start*$(l,\ r)$ for leader l and round r.

```
Phase2Start(l, r) ≜
  ∧ amLeader[l]
  ∧ lrnd[l] = r
  ∧ lval[l] = none
  ∧ ∃ Q ∈ Quorum :
      ∧ ∀ a ∈ Q : ∃ msg ∈ msgs : ∧ msg.type = "1b"
                                 ∧ msg.rnd  = r
                                 ∧ msg.acc  = a
      ∧ LET 1bMsg ≜ [a ∈ Q ↦ CHOOSE msg ∈ msgs :
                                   ∧ msg.type = "1b"
                                   ∧ msg.rnd  = r
                                   ∧ msg.acc  = a]
            v     ≜ DistProvedSafe(Q, r, 1bMsg)
        IN  ∧ lval' = [lval EXCEPT ![l] = v]
            ∧ Send([type ↦ "2a", rnd ↦ r, val ↦ v])
  ∧ UNCHANGED ⟨cvars, dvars, lrnd, avars, DBinter, amLeader⟩
```

The certification test just tests if the transaction history is atomically correct with respect to the current database state calculated by the leader and some final database state.

```
CertificationTest(st1, h) ≜ ∃ st2 ∈ DBState : CorrectAtomicHist[h, st1, st2]
```

Function *FinalState* below calculates the final state achieved by a sequence of active transactions.

```
FinalState[seq ∈ Seq([trans : Tid, acthist : Seq(Op)]), initst ∈ DBState] ≜
  IF seq = ⟨⟩
    THEN initst
    ELSE IF Head(seq) = ⟨⟩
           THEN FinalState[Tail(seq), initst]
           ELSE LET st ≜ CHOOSE st ∈ DBState :
                           ∃ res ∈ Result :
                             CorrectOp(Head(Head(seq)), res, initst, st)
                IN  FinalState[⟨Tail(Head(seq))⟩ ∘ Tail(seq), st]
```

Action *Phase2a*$(l,\ r)$ executed by leader l for round r when it wants to certify and propagate a proposed transaction.

$$
\begin{array}{l}
Phase2a(l,\, r) \triangleq \\
\quad \land\ amLeader[l] \\
\quad \land\ lrnd[r] = r \\
\quad \land\ lval[l] \neq none \\
\quad \land\ \text{LET } txSet \triangleq \{lval[l].seq[i].trans : i \in \text{DOMAIN } lval[l].seq\} \\
\qquad\quad indexof(t) \triangleq \text{CHOOSE } i \in \text{DOMAIN } lval[l].seq : lval[l].seq[i].trans = t \\
\qquad\quad xthist \triangleq [t \in Tid \mapsto \text{IF } t \in txSet \\
\qquad\qquad\qquad\qquad\qquad \text{THEN } lval[l].seq[indexof(t)].acthist \\
\qquad\qquad\qquad\qquad\qquad \text{ELSE } \langle\rangle] \\
\qquad\quad xseq \triangleq [i \in \text{DOMAIN } lval[l].seq \mapsto lval[l].seq[i].acthist] \\
\qquad\quad FSTATE \triangleq FinalState[xseq,\, InitialDBState] \\
\qquad \text{IN } \exists\, msg \in msgs : \\
\qquad\quad \land\ msg.type = \text{“propose”} \\
\qquad\quad \land\ msg.trans \notin (txSet \cup lval[l].aborted) \\
\qquad\quad \land\ \lor\ \land\ CertificationTest(FSTATE,\, msg.hist) \\
\qquad\qquad\quad \land\ lval' = [lval \text{ EXCEPT } !.seq = \\
\qquad\qquad\qquad\qquad Append(@,\, [trans \mapsto msg.trans, \\
\qquad\qquad\qquad\qquad\qquad\qquad acthist \mapsto OpSeq(StripPassive(msg.hist)) \\
\qquad\qquad\qquad\qquad\qquad\qquad] \\
\qquad\qquad\qquad\qquad\qquad) \\
\qquad\qquad\qquad\quad] \\
\qquad\qquad\ \lor\ lval' = [lval \text{ EXCEPT } !.aborted = @ \cup msg.trans] \\
\qquad\quad \land\ Send([type \mapsto \text{“2a”},\, rnd \mapsto r,\, val \mapsto lval']) \\
\quad \land\ \text{UNCHANGED } \langle cvars,\, dvars,\, lrnd,\, avars,\, DBinter,\, amLeader\rangle
\end{array}
$$

Action $Phase2b$

$$
\begin{array}{l}
Phase2b(a,\, r) \triangleq \\
\quad \land\ arnd[a] \leq r \\
\quad \land\ \exists\, m \in msgs : \\
\qquad\quad \land\ m.type = \text{“2a”} \\
\qquad\quad \land\ m.rnd = r \\
\qquad\quad \land\ ahist[r] = none \lor ahist[r] \prec m.val \\
\qquad\quad \land\ ahist' = [ahist \text{ EXCEPT } ![a][r] = m.val] \\
\qquad\quad \land\ Send([type \mapsto \text{“2b”},\, rnd \mapsto r,\, acc \mapsto a,\, val \mapsto m.val]) \\
\quad \land\ arnd' = [arnd \text{ EXCEPT } ![a] = r] \\
\quad \land\ \text{UNCHANGED } \langle cvars,\, dvars,\, lvars,\, DBinter,\, amLeader\rangle
\end{array}
$$

Action $ClientLearn(c)$

$$
\begin{array}{l}
ClientLearn(c) \triangleq \\
\quad \land\ \exists\, Q \in Quorum,\, r \in Nat : \\
\qquad \land\ \forall\, a \in Q : \exists\, m \in msgs : \land\ m.type = \text{“2b”} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \land\ m.rnd = r \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \land\ m.acc = a \\
\qquad \land\ \text{LET } 2bmsgs \triangleq \{m \in msgs : \land\ m.type = \text{“2b”}
\end{array}
$$

$$
\begin{array}{l}
\qquad\qquad\qquad\qquad\qquad\qquad \land\ m.rnd\ = r \\
\qquad\qquad\qquad\qquad\qquad\qquad \land\ m.acc\ \in Q\} \\
\qquad S \quad \triangleq\ \{m.val : m \in 2bmsgs\} \\
\qquad v \quad \triangleq\ \text{CHOOSE } v \in S : \forall\, u \in S : v \preceq u \\
\qquad txSet \quad \triangleq\ (\{v.seq[i].trans : i \in \text{DOMAIN } v.seq\} \cup v.aborted) \\
\qquad\qquad\qquad\quad \cap \{t \in Tid : ClientOf(t) = c\} \\
\text{IN} \quad \land \exists\, t \in txSet : cgdec[c][t] = Unknown \\
\qquad\ \land cgdec' = [cgdec \text{ EXCEPT } ![c] = \\
\qquad\qquad\qquad\qquad [t \in Tid \mapsto \text{IF } t \in txSet \\
\qquad\qquad\qquad\qquad\qquad\qquad \text{THEN IF } t \in v.aborted \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \text{THEN } Aborted \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \text{ELSE } Committed \\
\qquad\qquad\qquad\qquad\qquad\qquad \text{ELSE } cgdec[c][t] \\
\qquad\qquad\qquad\qquad] \\
\qquad\qquad\qquad] \\
\land \text{UNCHANGED } \langle cthist,\ cq,\ cdreq,\ cpdec,\ cproposed,\ dvars,\ lvars,\ avars, \\
\qquad\qquad DBinter,\ msgs,\ amLeader \rangle
\end{array}
$$

Action *DatabaseLearn*

$$
\begin{array}{l}
DatabaseLearn(d) \ \triangleq \\
\quad \land \exists\, Q \in Quorum,\ r \in Nat : \\
\qquad \land \forall\, a \in Q : \exists\, m \in msgs :\ \land\ m.type = \text{“2b”} \\
\qquad\qquad\qquad\qquad\qquad\qquad\ \land\ m.rnd\ = r \\
\qquad\qquad\qquad\qquad\qquad\qquad\ \land\ m.acc\ = a \\
\qquad \land \text{LET } 2bmsgs \ \triangleq\ \{m \in msgs :\ \land\ m.type = \text{“2b”} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\ \land\ m.rnd\ = r \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\ \land\ m.acc\ \in Q\} \\
\qquad\qquad S \quad \triangleq\ \{m.val : m \in 2bmsgs\} \\
\qquad\qquad v \quad \triangleq\ \text{CHOOSE } v \in S : \forall\, u \in S : v \preceq u \\
\qquad \text{IN} \quad \land\ Len(dlearnedSeq[d]) < Len(v.seq) \\
\qquad\qquad\ \land\ dlearnedSeq' = [dlearnedSeq \text{ EXCEPT } ![d] = v.seq] \\
\quad \land \text{UNCHANGED } \langle cvars,\ ldvars,\ ddreply,\ dcnt,\ vers,\ dcom,\ lvars,\ avars,\ DBinter, \\
\qquad\qquad msgs,\ amLeader \rangle
\end{array}
$$

Final Specification

The next-state action in terms of the noninterleaving actions.

$$
\begin{array}{l}
ClientAction \ \triangleq \\
\quad \exists\, c \in Client,\ t \in Tid : \\
\qquad \lor \exists\, req \in Request : ReceiveReq(c,\ t,\ req) \\
\qquad \lor \exists\, rep \in Reply\ \ : ReplyReq(c,\ t,\ rep) \\
\qquad \lor PrematureAbort(c,\ t) \\
\qquad \lor PassiveCommit(c,\ t)
\end{array}
$$

$\lor\ ClientLearn(c)$

$DatabaseAction \triangleq$
$\exists\, d \in Database :$
$\lor \exists\, t \in Tid : \lor \exists\, req \in Request : DBReq(d,\, t,\, req)$
$\lor \exists\, rep \in Reply : \quad DBRep(d,\, t,\, rep)$
$\lor\ DatabaseLearn(d)$
$\lor\ \land$ UNCHANGED $ldinter[d] \land DBS(d)!AOPNext$ replica's internal action
$\land\ OtherDBsStutter(d)$
$\land$ UNCHANGED $\langle cvars,\, gdvars,\, lvars,\, avars,\, DBinter,\, msgs,\, amLeader\rangle$

$LeaderAction \triangleq$
$\exists\, l \in Leader,\, r \in Nat :$
$\lor\ Phase1a(l,\, r)$
$\lor\ Phase2Start(l,\, r)$
$\lor\ Phase2a(l,\, r)$

$AcceptorAction \triangleq$
$\exists\, a \in Acceptor,\, r \in Nat :$
$\lor\ Phase1b(a,\, r)$
$\lor\ Phase2b(a,\, r)$

$ChangeLeader \triangleq$
$\land\ amLeader \in amLeaderType$
$\land$ UNCHANGED $\langle cvars,\, dvars,\, lvars,\, avars,\, DBinter,\, msgs\rangle$

$Next \triangleq \lor\ ClientAction$
$\lor\ DatabaseAction$
$\lor\ LeaderAction$
$\lor\ AcceptorAction$
$\lor\ ChangeLeader$

The final specification

$Spec \triangleq\ Init \land \Box[Next]_{\langle cvars,\, dvars,\, lvars,\, avars,\, DBinter,\, msgs,\, amLeader\rangle}$

Theorems

Our specification preserves the following Type Invariant

$TypeInvariant \triangleq$
$\land\ cthist \in cthistType$
$\land\ cq \in cqdreqType$
$\land\ cdreq \in cqdreqType$
$\land\ cpdec \in cdecType$
$\land\ cproposed \in cproposedType$
$\land\ cgdec \in cdecType$

$$
\begin{array}{ll}
\land\ ddreply & \in ddreplyType \\
\land\ dcnt & \in dcntversType \\
\land\ vers & \in dcntversType \\
\land\ dcom & \in dcomType \\
\land\ dlearnedSeq & \in dlearnedSeqType \\
\land\ lrnd & \in lrndType \\
\land\ lval & \in lvalType \\
\land\ arnd & \in arndType \\
\land\ ahist & \in ahistType \\
\land\ amLeader & \in amLeaderType \\
\land\ msgs & \in msgsType
\end{array}
$$

THEOREM $Spec \Rightarrow \Box TypeInvariant$

It also satisfies Serializability based on the following refinement mapping

$$thist \triangleq [t \in Tid \mapsto cthist[ClientOf(t)][t]]$$
$$q \triangleq [t \in Tid \mapsto cq \quad [ClientOf(t)][t]]$$
$$dreq \triangleq [t \in Tid \mapsto cdreq[ClientOf(t)][t]]$$
$$pdec \triangleq [t \in Tid \mapsto cpdec[ClientOf(t)][t]]$$

$$dreply \triangleq [t \in Tid \mapsto ddreply[DBof(t)][t]]$$

$$proposed \triangleq \text{UNION } \{cproposed[c] : c \in Client\}$$

$$
\begin{array}{l}
learnedSeq \triangleq [d \in Database \mapsto \\
\qquad [i \in \text{DOMAIN } dlearnedSeq[d] \mapsto \\
\qquad\qquad dlearnedSeq[d].trans \\
\qquad] \\
\quad]
\end{array}
$$

$$
\begin{array}{l}
IsCommitted(t) \triangleq \\
\quad \exists\, Q \in Quorum,\, r \in Nat : \\
\qquad \forall\, a \in Q : \\
\qquad\quad \land\ ahist[a][r] \neq none \\
\qquad\quad \land\ t \in \{ahist[a][r].seq[i] : i \in \text{DOMAIN } ahist[a][r].seq\}
\end{array}
$$

$$
\begin{array}{l}
IsAborted(t) \triangleq \\
\quad \exists\, Q \in Quorum,\, r \in Nat : \\
\qquad \forall\, a \in Q : \\
\qquad\quad \land\ ahist[a][r] \neq none \\
\qquad\quad \land\ t \in ahist[a][r].aborted
\end{array}
$$

$$
\begin{array}{l}
gdec \triangleq [t \in Tid \mapsto \text{IF } IsCommitted(t) \\
\qquad\qquad\qquad\qquad \text{THEN } Committed \\
\qquad\qquad\qquad\qquad \text{ELSE IF } IsAborted(t) \\
\qquad\qquad\qquad\qquad\qquad\qquad \text{THEN } Aborted
\end{array}
$$

```
                                ELSE Unknown
        ]

Ser ≜ INSTANCE GeneralDeferredUpdate

THEOREM Spec ⇒ Ser!Safety
```

4.3.2 Module *SOPSerializable*

This module specifies our consistency assumptions about in-memory database replicas in Section 4.2.

MODULE *SOPSerializableDB*

EXTENDS *OPSerializableDB*

Set of all running but undecided transactions

$$undecidedSet \triangleq \{t \in Tid : tdec[t] = Unknown \land thist[t] \neq \langle\rangle\}$$

SOPReplyReq(t, *rep*) substitutes *ReplyReq*

$$\begin{aligned}
&SOPReplyReq(t,\ rep) \triangleq \\
&\quad \land\ q[t] \in Request \\
&\quad \land\ DBResponse(t,\ rep) \\
&\quad \land\ q' = [q \text{ EXCEPT } ![t] = NoReq] \\
&\quad \land \text{ IF } tdec[t] \in Decided \\
&\qquad \text{THEN} \quad \land\ rep = tdec[t] \\
&\qquad\qquad\quad \land \text{ UNCHANGED } \langle thist,\ tdec\rangle \\
&\qquad \text{ELSE} \quad \land\ q[t] \in Op \\
&\qquad\qquad\quad \land\ rep \in Result \\
&\qquad\qquad\quad \land\ thist' = [thist \text{ EXCEPT } ![t] = Append(@,\ [op \mapsto q[t], \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad res \mapsto rep])] \\
&\qquad\qquad\quad \land \forall\ sufix \in Perm(undecidedSet \cup t) : \\
&\qquad\qquad\qquad\quad \exists\ st \in DBState : \\
&\qquad\qquad\qquad\qquad\quad CorrectSerialization[serialSeq \circ sufix, \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad thist',\ InitialDBState,\ st] \\
&\qquad\qquad\quad \land \text{ UNCHANGED } \langle tdec\rangle
\end{aligned}$$

SOPDoCommit substitutes *OPDoCommit*

$$\begin{aligned}
&SOPDoCommit(t) \triangleq \land\ tdec[t] = Unknown \\
&\qquad\qquad\qquad\quad \land\ q[t] = Commit \\
&\qquad\qquad\qquad\quad \land\ tdec' = [tdec \text{ EXCEPT } ![t] = Committed] \\
&\qquad\qquad\qquad\quad \land\ serialSeq' = Append(serialSeq,\ t) \\
&\qquad\qquad\qquad\quad \land \text{ UNCHANGED } \langle thist,\ q,\ DBinter\rangle
\end{aligned}$$

SOPDoAbort substitutes *DoAbort*

$$SOPDoAbort(t) \triangleq \land DoAbort(t) \\ \land q[t] \in Op$$

Specification

Next defines the possible "next" steps in a correct execution.

$$SOPNext \triangleq \exists\, t \in Tid : \\ \lor \land \lor \exists\, req \in Request : ReceiveReq(t,\, req) \\ \lor \exists\, rep \in Reply \quad : SOPReplyReq(t,\, rep) \\ \lor SOPDoAbort(t) \\ \land \text{UNCHANGED } serialSeq \\ \lor SOPDoCommit(t)$$

Final specification.

$$SOPSpec \triangleq OPInit \land \Box[SOPNext]_{\langle thist,\, tdec,\, q,\, DBinter,\, serialSeq\rangle}$$

4.3.3 Module *PrimaryBackup*

This module specifies our in-memory primary-backup replication algorithm from Section 4.2.

MODULE *PrimaryBackup*

EXTENDS *DatabaseConstants*, *DBInterface*

CONSTANTS *Client*, *Database*, *Acceptor*, *Quorum*, *PrimaryOf*(_), *DBof*(_), *ClientOf*(_), *StripPassive*(_)

Required Assumptions

We assume state-deterministic operations

$$\text{ASSUME } \forall\, op \in Op,\, res1,\, res2 \in Result,\, st,\, st1,\, st2 \in DBState : \\ \land CorrectOp(op,\, res1,\, st,\, st1) \\ \land CorrectOp(op,\, res2,\, st,\, st2) \\ \Rightarrow st1 = st2$$

Histories that go through the *StripPassive* operation generate the same final state as the original history.

$$\text{ASSUME } \forall\, hist \in THist,\, st1,\, st2 \in DBState : \\ \land StripPassive(hist) \in THist \\ \land CorrectAtomicHist[hist,\, st1,\, st2] \Rightarrow \\ CorrectAtomicHist[StripPassive(hist),\, st1,\, st2]$$

Every transaction has a single client and a single database replica associated with.

ASSUME $\forall\, t \in Tid :\ \land\, DBof(t) \in Database$
$\land\, ClientOf(t) \in Client$

Every round r has a single primary, and every database is the primary of a round greater than r.

ASSUME $\forall\, r \in Nat :\ \land\, PrimaryOf(r) \in Database$
$\land\, \forall\, d \in Database : \exists\, r2 \in Nat :\ \land\, r < r2$
$\land\, d = PrimaryOf(r2)$

We assume that quorums are finite subsets of the acceptors and every pair of quorums has a non-empty intersection.

ASSUME $\forall\, i \in Nat :$
$\land\, Quorum \subseteq \text{SUBSET } Acceptor$
$\land\, \forall\, Q \in Quorum : IsFiniteSet(Q)$
$\land\, \forall\, j \in Nat :$
$\forall\, Q \in Quorum,\ R \in Quorum : Q \cap R \neq \{\}$

Variables

VARIABLES $cthist,\ cq,\ cdreq,\ cpdec,\ cproposed,$ Client variables
$crndof,\ cinstof,\ clearned,$
$ddreply,\ dcnt,\ vers,\ dcom,\ dlearned,$ Database variables
$prnd,\ pfproposal,\ drnd,\ dfreeinsts,\ dinstof,$
$ldinter,\ dthist,\ dtdec,\ dq,\ dserialSeq,$ Internal database variables
$arnd,\ vrnd,\ vval,$ Acceptor variables
$msgs,\ amPrimary,\ dactivec$ Extra variables

Auxiliary definitions to help dealing with the declared variables.

$cvars \triangleq \langle cthist,\ cq,\ cdreq,\ cpdec,\ cproposed,$
$crndof,\ cinstof,\ clearned\rangle$
$ldvars \triangleq \langle ldinter,\ dthist,\ dtdec,\ dq,\ dserialSeq\rangle$
$gdvars \triangleq \langle ddreply,\ dcnt,\ vers,\ dcom,\ dlearned,$
$prnd,\ pfproposal,\ drnd,\ dfreeinsts,\ dinstof\rangle$
$dvars \triangleq \langle gdvars,\ ldvars\rangle$
$avars \triangleq \langle arnd,\ vrnd,\ vval\rangle$

Definitiosn regarding internal database variables and database replicas

Each database accepts transactions with ids in the form $\langle tid,\ version\rangle$ where tid is an element of Tid and version is a Natural. This allows "a single" transaction to be submitted to a database multiple times.

$LocalTid \triangleq Tid \times Nat$

The definition below instantiates each local database used by the general algorithm.

$$DBS(d) \triangleq \text{INSTANCE } SOPSerializableDB \text{ WITH } \begin{array}{ll} Tid & \leftarrow LocalTid, \\ DBinter & \leftarrow ldinter[d], \\ thist & \leftarrow dthist[d], \\ tdec & \leftarrow dtdec[d], \\ q & \leftarrow dq[d], \\ serialSeq & \leftarrow dserialSeq[d] \end{array}$$

External variables' Types

$$\begin{array}{ll} Bottom & \triangleq \text{ CHOOSE } v : v \notin Tid \\ NoTrans & \triangleq \text{ CHOOSE } v : v \notin Tid \cup \{Bottom\} \\ TidExt & \triangleq Tid \cup \{Bottom,\ NoTrans\} \end{array}$$

Client variables

$$\begin{array}{ll} cthistType & \triangleq [Client \to THistVector] \\ cqdreqType & \triangleq [Client \to [Tid \to Request \cup \{NoReq\}]] \\ cdecType & \triangleq [Client \to [Tid \to Decided \cup \{Unknown\}]] \\ cproposedType & \triangleq [Client \to \text{SUBSET } Tid] \\ crndinstofType & \triangleq [Client \to [Tid \to Nat]] \\ clearnedType & \triangleq [Client \to [Nat \to TidExt]] \end{array}$$

Updates accepted by acceptors and learned by databases

$$\begin{array}{ll} UpdatesType & \triangleq [trans : TidExt,\ acthist : FSeq(Op)] \\ none & \triangleq [trans \mapsto NoTrans,\ acthist \mapsto \langle\rangle] \\ bottom & \triangleq [trans \mapsto Bottom,\ \ acthist \mapsto \langle\rangle] \end{array}$$

Database variables

$$\begin{array}{ll} NoRep & \triangleq \text{ CHOOSE } v : v \notin Reply \quad \text{Not a valid reply} \\ ddreplyType & \triangleq [Database \to [Tid \to Reply \cup \{NoRep\}]] \\ dcntversType & \triangleq [Database \to [Tid \to Nat]] \\ dcomType & \triangleq [Database \to [Tid \to \text{BOOLEAN }]] \\ dlearnedType & \triangleq [Database \to [Nat \to UpdatesType \cup \{none\}]] \\ dprndType & \triangleq [Database \to Nat] \\ pfproposalType & \triangleq [Database \to \text{UNION } \{[0\,..\,i \to UpdatesType] : i \in Nat\}] \\ dfreeinstsType & \triangleq [Database \to \text{SUBSET } Nat] \\ dinstofType & \triangleq [Database \to [Tid \to \{\{\}\} \cup \{\{i\} : i \in Nat\}]] \end{array}$$

Acceptor variables

$$\begin{array}{ll} arndType & \triangleq [Acceptor \to Nat] \\ vrndType & \triangleq [Acceptor \to Nat] \\ vvalType & \triangleq [Acceptor \to [Nat \to UpdatesType]] \end{array}$$

Other variables

$$amPrimaryType \triangleq [Database \to \text{BOOLEAN }]$$

$dactivecType \triangleq [Database \to \text{SUBSET } Client]$
$msgsType \triangleq$
$[type : \{\text{“dreq”}\}, trans : Tid, dreq : Request, len : Nat] \cup$
$[type : \{\text{“drep”}\}, trans : Tid, cnt : Nat, dreply : Result, rnd : Nat, inst : Nat] \cup$
$[type : \{\text{“drep”}\}, trans : Tid, dreply : Decided] \cup$
$[type : \{\text{“1a”}\}, rnd : Nat] \cup$
$[type : \{\text{“1b”}\}, rnd : Nat, acc : Acceptor, vrnd : Nat, vval : [Nat \to UpdatesType]] \cup$
$[type : \{\text{“2aS”}\}, rnd : Nat, val : \text{UNION } \{[0 \,..\, i \to UpdatesType] : i \in Nat\}] \cup$
$[type : \{\text{“2a”}\}, inst : Nat, rnd : Nat, trans : Tid, acthist : Seq(Op)] \cup$
$[type : \{\text{“2bS”}\}, rnd : Nat, acc : Acceptor,$
$\quad val : \text{UNION } \{[0 \,..\, i \to UpdatesType] : i \in Nat\}] \cup$
$[type : \{\text{“2b”}\}, rnd : Nat, acc : Acceptor, inst : Nat, trans : Tid, acthist : Seq(Op)]$

Initialization

$ClientInit \triangleq$
$\wedge\ cthist = [c \in Client \mapsto [t \in Tid \mapsto \langle\rangle]]$
$\wedge\ cq = [c \in Client \mapsto [t \in Tid \mapsto NoReq]]$
$\wedge\ cdreq = [c \in Client \mapsto [t \in Tid \mapsto NoReq]]$
$\wedge\ cpdec = [c \in Client \mapsto [t \in Tid \mapsto Unknown]]$
$\wedge\ cproposed = [c \in Client \mapsto \{\}]$
$\wedge\ crndof \in crndinstofType$
$\wedge\ cinstof \in crndinstofType$
$\wedge\ clearned = [c \in Client \mapsto [i \in Nat \mapsto NoTrans]]$

$DatabaseInit \triangleq$
$\wedge\ ddreply = [d \in Database \mapsto [t \in Tid \mapsto NoRep]]$
$\wedge\ dcnt = [d \in Database \mapsto [t \in Tid \mapsto 0]]$
$\wedge\ vers = [d \in Database \mapsto [t \in Tid \mapsto 0]]$
$\wedge\ dcom = [d \in Database \mapsto [t \in Tid \mapsto \text{FALSE}]]$
$\wedge\ dlearned = [d \in Database \mapsto [i \in Nat \mapsto \text{IF } i = 0$
$\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{THEN } bottom$
$\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{ELSE } none]]$
$\wedge\ prnd = [d \in Database \mapsto 0]$
$\wedge\ drnd = [d \in Database \mapsto 0]$
$\wedge\ pfproposal = [d \in Database \mapsto [i \in \{0\} \mapsto bottom]]$
$\wedge\ dfreeinsts = [d \in Database \mapsto Nat \setminus \{0\}]$
$\wedge\ dinstof = [d \in Database \mapsto [t \in Tid \mapsto \{\}]]$
$\wedge \forall\, d \in Database : DBS(d)!OPInit$

$AcceptorInit \triangleq$
$\wedge\ arnd = [a \in Acceptor \mapsto 0]$
$\wedge\ vrnd = [a \in Acceptor \mapsto 0]$
$\wedge\ vval = [a \in Acceptor \mapsto [i \in Nat \mapsto \text{IF } i = 0 \text{ THEN } bottom$

ELSE $none$]]

$OtherInit \triangleq$
$\quad \land msgs = \{\}$
$\quad \land amPrimary \in amPrimaryType$
$\quad \land dactivec \quad \in dactivecType$

$Init \triangleq \land InitInterface$
$\quad \land ClientInit$
$\quad \land DatabaseInit$
$\quad \land AcceptorInit$
$\quad \land OtherInit$

Auxiliary actions and operators

$ActHist(c,\ t)$ returns the current history of transaction t with some of its passive operations taken out of the sequence (according to operator $StripPassive$).

$ActHist(c,\ t) \triangleq StripPassive(cthist[c][t])$

$OpSeq(h)$ returns a sequence containing only the operations of a history (without the operations' results).

$OpSeq(h) \triangleq [i \in \text{DOMAIN } h \mapsto h[i].op]$

$Send(m)$ sends message m

$Send(m) \triangleq msgs' = msgs \cup m$

$DBvars(d)$ returns the internal variables of database d.

$DBvars(d) \triangleq \langle ldinter[d],\ dthist[d],\ dtdec[d],\ dq[d],\ dserialSeq[d]\rangle$

$OtherDBsStutter(d)$ is an action that forces all databases but d to execute a stuttering step, that is, a step in which their internal variables do not change values. For simplicity, our specification does not allow interleaving of database actions. In fact, as we explain in the following, it does not allow interleaving at all.

$OtherDBsStutter(d) \triangleq$
$\quad \text{LET } dbfn \triangleq [nd \in (Database \setminus \{d\}) \mapsto DBvars(nd)]$
$\quad \text{IN } \quad dbfn' = dbfn$

If B is a set of round numbers that contains a maximum element, then $Max(B)$ is defined to equal that maximum. Otherwise, its value is unspecified.

$Max(B) \triangleq \text{CHOOSE } i \in B : \forall j \in B : j \leq i$

Atomic Actions

These are the atomic actions of the algorithm, not including the internal database actions. In order to model check this specification, we had to make it noninterleaving, that is, we had to specify it in terms of actions that cannot occur concurrently (even considering that they are executed by different specification components). This prevented us from using the $DBRequest$ and $DBResponse$ primitives to interact with the internal databases. Instead, we used the $ReceiveReq$ and $ReplyReq$ actions directly to submit an operation and get a response from a database.

The *ReceiveReq* action.

$$
\begin{array}{l}
ReceiveReq(c,\ t,\ req) \triangleq \\
\quad \land\ c = ClientOf(t) \\
\quad \land\ DBRequest(t,\ req) \\
\quad \land\ cq[c][t] \notin Request \\
\quad \land\ cq' = [cq \text{ EXCEPT } ![c][t] = req] \\
\quad \land \text{ IF } t \notin cproposed[c] \land cpdec[c][t] \notin Decided \\
\qquad \text{THEN } \lor\ \land\ req = Commit \\
\qquad\qquad\qquad \land\ Len(ActHist(c,\ t)) > 0 \\
\qquad\qquad\qquad \land\ DBof(t) = PrimaryOf(crndof[c][t]) \\
\qquad\qquad\qquad \land\ cproposed' = [cproposed \text{ EXCEPT } ![c] = @ \cup t] \\
\qquad\qquad\qquad \land\ Send([type \mapsto \text{“2a”},\ inst \mapsto cinstof[c][t], \\
\qquad\qquad\qquad\qquad\qquad rnd \mapsto crndof[c][t],\ trans \mapsto t, \\
\qquad\qquad\qquad\qquad\qquad acthist \mapsto OpSeq(ActHist(c,\ t))]) \\
\qquad\qquad\qquad \land\ \text{UNCHANGED } cdreq \\
\qquad\qquad\ \lor\ \land\ req = Commit \Rightarrow Len(ActHist(c,\ t)) = 0 \\
\qquad\qquad\qquad \land\ cdreq' = [cdreq \text{ EXCEPT } ![c][t] = req] \\
\qquad\qquad\qquad \land\ Send([type \mapsto \text{“dreq”},\ trans \mapsto t, \\
\qquad\qquad\qquad\qquad\qquad dreq \mapsto req,\ len \mapsto Len(cthist[c][t])]) \\
\qquad\qquad\qquad \land\ \text{UNCHANGED } cproposed \\
\qquad \text{ELSE UNCHANGED } \langle cdreq,\ cproposed,\ msgs \rangle \\
\quad \land\ \text{UNCHANGED } \langle cthist,\ cpdec,\ crndof,\ cinstof,\ clearned,\ dvars, \\
\qquad\qquad avars,\ amPrimary,\ dactivec \rangle
\end{array}
$$

Action *ReplyReq* below uses the following definition for $ctdec(c,\ t)$, which calculates $tdec[t]$ based only on variables kept at client c.

$$
\begin{array}{l}
ctdec(c,\ t) \triangleq \text{ IF } t \notin cproposed[c] \\
\qquad\qquad\qquad \text{THEN } cpdec[c][t] \\
\qquad\qquad\qquad \text{ELSE IF } clearned[c][cinstof[c][t]] = NoTrans \\
\qquad\qquad\qquad\qquad\qquad \text{THEN } Unknown \\
\qquad\qquad\qquad\qquad\qquad \text{ELSE IF } clearned[c][cinstof[c][t]] = t \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{THEN } Committed \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{ELSE } Aborted
\end{array}
$$

The *ReplyReq* action.

$$
\begin{array}{l}
ReplyReq(c,\ t,\ rep) \triangleq \\
\quad \land\ c = ClientOf(t) \\
\quad \land\ cq[c][t] \in Request \\
\quad \land\ DBResponse(t,\ rep) \\
\quad \land\ cq' = [cq \text{ EXCEPT } ![c][t] = NoReq] \\
\quad \land \text{ IF } ctdec(c,\ t) \in Decided \\
\qquad \text{THEN } \land\ rep = ctdec(c,\ t) \\
\qquad\qquad\ \land\ \text{UNCHANGED } \langle cthist,\ cdreq,\ crndof,\ cinstof \rangle \\
\qquad \text{ELSE } \land\ cq[c][t] \in Op
\end{array}
$$

$\land\ rep \in Result$
$\land\ \exists\, m \in msgs :$
$\quad \land\ m.type = \text{“drep”}$
$\quad \land\ m.trans = t$
$\quad \land\ m.dreply = rep$
$\quad \land\ m.cnt > Len(cthist[t])$
$\quad \land\ cthist' = [cthist \text{ EXCEPT } ![c][t] = Append(@, [op \mapsto cq[c][t], res \mapsto rep])]$
$\quad \land\ cdreq' = [cdreq \text{ EXCEPT } ![c][t] = NoReq]$
$\quad \land\ crndof' = [crndof \text{ EXCEPT } ![c][t] = m.rnd]$
$\quad \land\ cinstof' = [cinstof \text{ EXCEPT } ![c][t] = m.inst]$
$\land$ UNCHANGED $\langle cproposed, clearned, cpdec, dvars, avars, msgs, amPrimary, dactivec\rangle$

The *PrematureAbort* action.

$PrematureAbort(c, t) \triangleq$
$\land\ c = ClientOf(t)$
$\land\ t \notin cproposed[c]$
$\land\ cpdec[c][t] \notin Decided$
$\land\ \exists\, m \in msgs :$
$\quad \land\ m.type = \text{“drep”}$
$\quad \land\ m.trans = t$
$\quad \land\ m.dreply = Aborted$
$\quad \land\ cpdec' = [cpdec \text{ EXCEPT } ![c][t] = Aborted]$
$\quad \land$ UNCHANGED $\langle cthist, cq, cdreq, cproposed, crndof, cinstof, clearned, dvars, avars, msgs, DBinter, amPrimary, dactivec\rangle$

The *PassiveCommit* action.

$PassiveCommit(c, t) \triangleq$
$\land\ c = ClientOf(t)$
$\land\ t \notin cproposed[c]$
$\land\ cpdec[t] \notin Decided$
$\land\ \exists\, m \in msgs :$
$\quad \land\ m.type = \text{“drep”}$
$\quad \land\ m.trans = t$
$\quad \land\ m.dreply = Committed$
$\quad \land\ cpdec' = [cpdec \text{ EXCEPT } ![c][t] = Committed]$
$\quad \land$ UNCHANGED $\langle cthist, cq, cdreq, cproposed, crndof, cinstof, clearned, dvars, avars, msgs, DBinter, amPrimary, dactivec\rangle$

$learnedSeq \triangleq$

LET $recgenseq[d \in Database, i \in Nat, s \in Seq(Tid)] \triangleq$
IF $dlearned[d][i] = none$
THEN s end of recursion
ELSE IF $dlearned[d][i] = bottom$

```
                THEN recgenseq[d, i + 1, s]   skip this instance
                ELSE recgenseq[d, i + 1, Append(s, dlearned[d][i].trans)]   adds t
    IN  [d ∈ Database ↦ recgenseq[d, 0, ⟨⟩]]
```

The *DBReq* action with its three enabling conditions.

```
DBReq(d, t, req) ≜
  ∧  ∨ ∧ d = DBof(t)                                   Condition 1
     ∨ ∧ d = DBof(t)
       ∧ vers[d][t] = 0
       ∧ ∃ m ∈ msgs :
          ∧ m.type = "dreq"
          ∧ m.trans = t
          ∧ m.dreq = req
          ∧ dcnt[d][t] = m.len                         Condition 2
       ∨ ∧ ∃ i ∈ 1 .. Len(learnedSeq[d]) :
            ∧ learnedSeq[d][i] = t
            ∧ ∀ j ∈ 1 .. i − 1 : dcom[d][learnedSeq[d][j]]
         ∧ LET i ≜ CHOOSE i ∈ Nat : dlearned[d][i].trans = t
           IN  ∧ dcnt[d][t]  < Len(dlearned[d][i].acthist)
               ∧ req = dlearned[d][i].acthist[dcnt[d][t] + 1]
       ∨ ∧ req = Commit                                Condition 3
         ∧ ∃ i ∈ 1 .. Len(learnedSeq[d]) :
            ∧ learnedSeq[d][i] = t
            ∧ ∀ j ∈ 1 .. i − 1 : dcom[d][learnedSeq[d][j]]
         ∧ ∨ d = DBof(t) ∧ vers[d][t] = 0
           ∨ LET i ≜ CHOOSE i ∈ Nat : dlearned[d][i].trans = t
             IN  dcnt[d][t] = Len(dlearned[d][i].acthist)
  ∧  dfreeinsts′ = [dfreeinsts EXCEPT ![d] = @ ∪ dinstof[d][t]]
  ∧  dinstof′    = [dinstof    EXCEPT ![d][t] = {}]
  ∧  DBS(d)!ReceiveReq(⟨t, vers[d][t]⟩, req)
  ∧  OtherDBsStutter(d)
  ∧  UNCHANGED ⟨cvars, ddreply, dcnt, vers, dcom, dlearned, drnd, prnd,
                pfproposal, avars, DBinter, msgs, amPrimary, dactivec⟩
```

The *DBRep* action.

```
DBRep(d, t, rep) ≜
  ∧  DBS(d)!ReplyReq(⟨t, vers[d][t]⟩, rep)
  ∧  OtherDBsStutter(d)
  ∧  IF d = DBof(t)
     THEN ∧ ddreply′ = [ddreply EXCEPT ![d][t] = rep]
          ∧ ∨ ∧ ¬∃ i ∈ Nat :
                   dlearned[d][i].trans = t
            ∧ rep ∈ Result
            ∧ ∃ i ∈ dfreeinsts[d] :
```

```
                                    ∧ Send([type  ↦ "drep", trans ↦ t,
                                            cnt   ↦ dcnt[d][t] + 1, dreply ↦ rep,
                                            rnd   ↦ drnd[d], inst ↦ i])
                                    ∧ dfreeinsts' = [dfreeinsts EXCEPT ![d] = @ \ {i}]
                                    ∧ dinstof'    = [dinstof    EXCEPT ![d][t] = {i}]
                              ∨ ∧ ¬∃ i ∈ Nat :
                                      dlearned[d][i].trans = t
                                ∧ rep ∈ Decided
                                ∧ Send([type ↦ "drep", trans ↦ t, dreply ↦ rep])
                                ∧ UNCHANGED ⟨dfreeinsts, dinstof⟩
                              ∨ ∧ ∃ i ∈ Nat :
                                    dlearned[d][i].trans = t
                                ∧ UNCHANGED ⟨msgs, dfreeinsts, dinstof⟩
                        ELSE UNCHANGED ⟨ddreply, msgs, dfreeinsts, dinstof⟩
    ∧   IF ∧ rep = Aborted
           ∧ ∃ i ∈ Nat :
               dlearned[d][i].trans = t
           THEN ∧ vers' = [vers EXCEPT ![d][t] = @ + 1]
                ∧ dcnt' = [dcnt EXCEPT ![d][t] = 0]
                ∧ UNCHANGED dcom
           ELSE IF rep ∈ Result
                THEN ∧ dcnt' = [dcnt EXCEPT ![d][t] = @ + 1]
                     ∧ UNCHANGED ⟨dcom, vers⟩
                ELSE ∧ dcom' = [dcom EXCEPT ![d][t] = (rep = Committed)]
                     ∧ UNCHANGED ⟨dcnt, vers⟩
    ∧   UNCHANGED ⟨cvars, dlearned, drnd, prnd, pfproposal, avars,
                   DBinter, amPrimary, dactivec⟩
```

Action $Phase1a(p, r)$ is executed by the primary p of round r. To ensure Liveness, p can only execute this action if it believes to be the primary and either p has received a message related to a round between $prnd[p]$ and r, or it suspects one of the clients currently running transactions to have failed.

```
Phase1a(p, r) ≜
  ∧ amPrimary[p]
  ∧ p = PrimaryOf(r)
  ∧ prnd[p] < r
  ∧ ∨ ∃ msg ∈ {m ∈ msgs : m.type ∈ {"1a", "1b", "2a", "2aS", "2b", "2bs"}} :
        ∧ prnd[p]  < msg.rnd
        ∧ msg.rnd < r
    ∨ ∃ c ∈ Client :
        ∧ c ∉ dactivec[p]
        ∧ ∃ t ∈ Tid :
            ∧ p = DBof(t)
            ∧ c = ClientOf(t)
            ∧ dinstof[p][t] ≠ {}
```

$\land\ prnd' = [drnd \text{ EXCEPT } ![p] = r]$
$\land\ pfproposal' = [pfproposal \text{ EXCEPT } ![p] = [i \in Nat \mapsto none]]$
$\land\ Send([type \mapsto \text{“1a”},\ rnd \mapsto r])$
$\land$ UNCHANGED $\langle cvars,\ ldvars,\ ddreply,\ dcnt,\ vers,\ dcom,\ dlearned,\ drnd,$
$dfreeinsts,\ dinstof,\ avars,\ DBinter,\ amPrimary,\ dactivec\rangle$

Action $Phase1b(a,\ r)$ is executed by acceptor a, for round r.

$Phase1b(a,\ r) \triangleq$
$\land\ [type \mapsto \text{“1a”},\ rnd \mapsto r] \in msgs$
$\land\ arnd[a] < r$
$\land\ arnd' = [arnd \text{ EXCEPT } ![a] = r]$
$\land\ Send([type \mapsto \text{“1b”},\ rnd \mapsto r,\ acc \mapsto a,\ vrnd \mapsto vrnd[a],\ vval \mapsto vval[a]])$
$\land$ UNCHANGED $\langle cvars,\ dvars,\ vrnd,\ vval,\ DBinter,\ amPrimary,\ dactivec\rangle$

Let $MaxN$ be the maximum instance with some value different from none accepted based on the “$1b$” messages received from a quorum Q for round r. $DistProvedSafe(Q,\ r,\ 1bMsg)$ below returns a mapping from the integers in $0\ ..\ MaxN$ to a value that might have been chosen in that instance for rounds lower than r. If no transaction might have been chosen for some instance in this interval, $DistProvedSafe$ maps it to bottom.

$DistProvedSafe(Q,\ r,\ 1bMsg) \triangleq$
LET $MaxN \triangleq$ CHOOSE $MaxN \in Nat$:
$\land\ \exists\, a \in Q : 1bMsg[a].vval[MaxN] \neq none$
$\land\ \forall\, a \in Q,\ j \in \{k \in Nat : k > MaxN\} :$
$1bMsg[a].vval[j] \neq none$
IN $[i \in 0\ ..\ MaxN \mapsto$
IF $\forall\, a \in Q : 1bMsg[a].vval[i] = none$
THEN $bottom$
ELSE LET $k \triangleq Max(\{1bMsg[a].vrnd : a \in Q\})$
$AS \triangleq \{a \in Q : \land\ 1bMsg[a].vrnd = k$
$\land\ 1bMsg[a].vval[i] \neq none\}$
$S \triangleq \{1bMsg[a].vval[i] : a \in AS\}$
IN CHOOSE $v : v \in S\ ]$

Action $Phase2Start(p,\ r)$ executed by primary p of round r when it wants to start a round based on “$1b$” messages received from a quorum of acceptors.

$Phase2Start(p,\ r) \triangleq$
$\land\ amPrimary[p]$
$\land\ prnd[p] = r$
$\land$ DOMAIN $pfproposal[p] = Nat$
$\land\ \exists\, Q \in Quorum :$
$\land\ \forall\, a \in Q : \exists\, msg \in msgs : \land\ msg.type = \text{“1b”}$
$\land\ msg.rnd = r$
$\land\ msg.acc = a$
$\land$ LET $1bMsg \triangleq [a \in Q \mapsto$ CHOOSE $msg \in msgs :$
$\land\ msg.type = \text{“1b”}$
$\land\ msg.rnd = r$

$$
\begin{array}{l}
\qquad\qquad\qquad\qquad\qquad\qquad \land\ msg.acc\ = a] \\
\qquad\quad v \quad \triangleq\ DistProvedSafe(Q,\ r,\ 1bMsg) \\
\quad \text{IN} \quad \land\ pfproposal' = [pfproposal \text{ EXCEPT } ![p] = v] \\
\qquad\qquad \land\ Send([type \mapsto \text{“2aS”},\ rnd \mapsto r,\ val \mapsto v]) \\
\land \text{ UNCHANGED } \langle cvars,\ ldvars,\ ddreply,\ dcnt,\ vers,\ dcom,\ dlearned,\ prnd, \\
\qquad\qquad\qquad drnd,\ dfreeinsts,\ dinstof,\ avars,\ DBinter,\ amPrimary,\ dactivec\rangle
\end{array}
$$

Action $Phase2b$

$$
\begin{array}{l}
Phase2b(a,\ r)\ \triangleq \\
\quad \land\ arnd[a] \leq r \\
\quad \land\ \lor \exists\, m \in msgs : \\
\qquad\qquad \land\ m.type = \text{“2aS”} \\
\qquad\qquad \land\ m.rnd\ = r \\
\qquad\qquad \land\ vrnd[a] < r \\
\qquad\qquad \land\ vval' = [vval \text{ EXCEPT } ![a] = \\
\qquad\qquad\qquad\qquad [i \in Nat \mapsto \text{IF } i \in \text{DOMAIN } m.val \\
\qquad\qquad\qquad\qquad\qquad\qquad \text{THEN } m.val[i] \\
\qquad\qquad\qquad\qquad\qquad\qquad \text{ELSE } none \\
\qquad\qquad\qquad\qquad] \\
\qquad\qquad\qquad] \\
\qquad\qquad \land\ Send([type \mapsto \text{“2bS”},\ rnd \mapsto r,\ acc \mapsto a,\ val \mapsto m.val]) \\
\qquad \lor \exists\, m \in msgs : \\
\qquad\qquad \land\ m.type = \text{“2a”} \\
\qquad\qquad \land\ m.rnd\ = r \\
\qquad\qquad \land\ vrnd[a] = r \land vval[a][m.inst] = none \\
\qquad\qquad \land\ vval' = [vval \text{ EXCEPT } ![a][m.inst] = [trans\ \ \mapsto m.trans, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad acthist \mapsto m.acthist]] \\
\qquad\qquad \land\ Send([type \mapsto \text{“2b”},\ rnd \mapsto r,\ acc \mapsto a,\ inst \mapsto m.inst, \\
\qquad\qquad\qquad\qquad\qquad\qquad trans \mapsto m.trans,\ acthist \mapsto m.acthist]) \\
\quad \land\ arnd' = [arnd \text{ EXCEPT } ![a] = r] \\
\quad \land\ vrnd' = [vrnd \text{ EXCEPT } ![a] = r] \\
\quad \land \text{ UNCHANGED } \langle cvars,\ dvars,\ DBinter,\ amPrimary,\ dactivec\rangle
\end{array}
$$

Action $Phase2Start(p,\ r)$ executed by primary p of round r when it wants to activate a round for independent proposals coming from clients.

$$
\begin{array}{l}
ActivateRound(p,\ r)\ \triangleq \\
\quad \land\ amPrimary[p] \\
\quad \land\ prnd[p] = r \\
\quad \land\ drnd[p] < prnd[p] \\
\quad \land \text{ DOMAIN } pfproposal[p] \neq Nat \\
\quad \land \forall\, i \in \text{DOMAIN } pfproposal[p] : dlearned[p][i] = pfproposal[p][i] \\
\quad \land \forall\, t \in \{pfproposal[p][i].trans : i \in \text{DOMAIN } pfproposal[p]\} \setminus \{Bottom\} : \\
\qquad dcom[p][t] \\
\quad \land \exists\, Q \in Quorum :
\end{array}
$$

$$
\begin{array}{l}
\quad \land \forall\, a \in Q : \exists\, m \in msgs : \land\, m.type = \text{“2bS”} \\
\qquad\qquad\qquad\qquad\qquad\qquad \land\, m.rnd = r \\
\qquad\qquad\qquad\qquad\qquad\qquad \land\, m.acc = a \\
\land\, drnd' = [drnd \ \text{EXCEPT}\ ![p] = prnd[p]] \\
\land\, dfreeinsts' = [dfreeinsts \ \text{EXCEPT}\ ![p] = Nat \setminus \text{DOMAIN}\ pfproposal[p]] \\
\land\, dinstof' = [dinstof \ \text{EXCEPT}\ ![p] = [t \in Tid \mapsto \{\}]] \\
\land\, \text{UNCHANGED}\ \langle cvars,\, ldvars,\, ddreply,\, dcnt,\, vers,\, dcom,\, dlearned,\, prnd,\, pfproposal, \\
\qquad\qquad avars,\, DBinter,\, msgs,\, amPrimary,\, dactivec \rangle
\end{array}
$$

Action $ClientLearn(c)$

$$
\begin{array}{l}
ClientLearn(c) \triangleq \\
\quad \land\, \exists\, Q \in Quorum,\, r \in Nat : \\
\qquad \lor\ \land\, \forall\, a \in Q : \exists\, m \in msgs : \land\, m.type = \text{“2bS”} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \land\, m.rnd = r \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \land\, m.acc = a \\
\qquad\qquad \land\, \text{LET}\ 2bSmsgs \triangleq \{m \in msgs : \land\, m.type = \text{“2bS”} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \land\, m.rnd = r \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \land\, m.acc \in Q\} \\
\qquad\qquad\qquad val \triangleq \text{CHOOSE}\ val \in \{m.val : m \in 2bSmsgs\} : \text{TRUE} \\
\qquad\qquad\ \text{IN}\quad \land\, \exists\, i \in \text{DOMAIN}\ val : clearned[c][i] = NoTrans \\
\qquad\qquad\qquad\quad \land\, clearned' = [clearned \ \text{EXCEPT}\ ![c] = \\
\qquad\qquad\qquad\qquad\qquad [i \in Nat \mapsto \text{IF}\ \land\, clearned[c][i] = NoTrans \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \land\, i \in \text{DOMAIN}\ val \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \text{THEN}\ val[i].trans \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \text{ELSE}\ clearned[c][i] \\
\qquad\qquad\qquad\qquad\qquad] \\
\qquad\qquad\qquad\qquad] \\
\qquad \lor\, \exists\, inst \in Nat : \\
\qquad\qquad \land\, clearned[c][inst] = NoTrans \\
\qquad\qquad \land\, \forall\, a \in Q : \exists\, m \in msgs : \land\, m.type = \text{“2b”} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \land\, m.rnd = r \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \land\, m.acc = a \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \land\, m.inst = inst \\
\qquad\qquad \land\, \text{LET}\ 2bmsgs \triangleq \{m \in msgs : \land\, m.type = \text{“2b”} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \land\, m.rnd = r \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \land\, m.acc \in Q \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \land\, m.inst = inst\} \\
\qquad\qquad\qquad trans \triangleq \text{CHOOSE}\ trans \in \{m.trans : m \in 2bmsgs\} : \text{TRUE} \\
\qquad\qquad\ \text{IN}\quad clearned' = [clearned \ \text{EXCEPT}\ ![c][inst] = trans] \\
\quad \land\, \text{UNCHANGED}\ \langle cthist,\, cq,\, cdreq,\, cpdec,\, cproposed,\, crndof,\, cinstof, \\
\qquad\qquad dvars,\, avars,\, DBinter,\, msgs,\, amPrimary,\, dactivec \rangle
\end{array}
$$

Action $DatabaseLearn(d)$

$$DatabaseLearn(d) \triangleq$$

$$
\begin{array}{l}
\land \exists\, Q \in Quorum,\, r \in Nat : \\
\quad \lor \land \forall\, a \in Q : \exists\, m \in msgs : \land\, m.type = \text{“2bS”} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \land\, m.rnd\ = r \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \land\, m.acc\ = a \\
\qquad\quad \land \text{LET } 2bSmsgs \triangleq \{m \in msgs : \land\, m.type = \text{“2bS”} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \land\, m.rnd\ = r \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \land\, m.acc\ \in Q\} \\
\qquad\qquad val \triangleq \text{CHOOSE } val \in \{m.val : m \in 2bSmsgs\} : \text{TRUE} \\
\qquad\quad \text{IN} \quad \land \exists\, i \in \text{DOMAIN } val : dlearned[d][i] = none \\
\qquad\qquad\quad \land\, dlearned' = [dlearned \text{ EXCEPT } ![d] = \\
\qquad\qquad\qquad\qquad\qquad [i \in Nat \mapsto \text{IF } \land\, dlearned[d][i] = none \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \land\, i \in \text{DOMAIN } val \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \text{THEN } val[i] \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \text{ELSE } dlearned[d][i] \\
\qquad\qquad\qquad\qquad\qquad\quad] \\
\qquad\qquad\qquad\qquad\quad] \\
\quad \lor \exists\, inst \in Nat : \\
\qquad \land\, dlearned[d][inst] = none \\
\qquad \land \forall\, a \in Q : \exists\, m \in msgs : \land\, m.type = \text{“2b”} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \land\, m.rnd\ = r \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \land\, m.acc\ = a \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \land\, m.inst\ = inst \\
\qquad \land \text{LET } 2bmsgs \triangleq \{m \in msgs : \land\, m.type = \text{“2b”} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \land\, m.rnd\ = r \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \land\, m.acc\ \in Q \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \land\, m.inst = inst\} \\
\qquad\qquad msg \triangleq \text{CHOOSE } msg \in 2bmsgs : \text{TRUE} \\
\qquad\quad \text{IN} \quad dlearned' = [dlearned \text{ EXCEPT } ![d][inst] = \\
\qquad\qquad\qquad\qquad\qquad [trans\ \mapsto msg.trans, \\
\qquad\qquad\qquad\qquad\qquad\ acthist \mapsto msg.acthist \\
\qquad\qquad\qquad\qquad\qquad] \\
\qquad\qquad\qquad\qquad] \\
\land \text{UNCHANGED } \langle cvars,\, ldvars,\, ddreply,\, dcnt,\, vers,\, dcom,\, prnd,\, pfproposal,\, drnd, \\
\qquad\qquad dfreeinsts,\, dinstof,\, avars,\, DBinter,\, msgs,\, amPrimary,\, dactivec\rangle
\end{array}
$$

$DBAbortT(d,\, t)$ allows the database to abort transactions that are blocking liveness.

$$
\begin{array}{l}
DBAbortT(d,\, t) \triangleq \\
\quad \land\, d = DBof(t) \\
\quad \land\, vers[d][t] = 0 \\
\quad \land\, dcnt[d][t] > 0 \\
\quad \land \lor \land\, d \neq PrimaryOf(drnd[d]) \quad \text{read-only transaction} \\
\qquad\ \land\, dinstof[d][t] \neq \{\} \\
\qquad\ \land\, dinstof' = [dinstof \text{ EXCEPT } ![d][t] = \{\}] \\
\qquad \lor \land\, dinstof[d][t] = \{\} \qquad \text{will be aborted, if proposed}
\end{array}
$$

$\land\ dtdec[d][t] = Unknown$ and has not been locally decided yet
$\land$ UNCHANGED $\langle dinstof \rangle$
$\land\ DBS(d)!ReceiveReq(\langle t, 0\rangle, Abort)$
$\land\ OtherDBsStutter(d)$
$\land$ UNCHANGED $\langle cvars, ddreply, dcnt, vers, dcom, dlearned, prnd, pfproposal, drnd, dfreeinsts, avars, DBinter, msgs, amPrimary, dactivec\rangle$

$DBChaneRound(d, r)$ is an extra action that allows databases to advance their round numbers. This can be used in practice for databases to know more easily if they are the current primary or not. A database in a round it is not the primary of is necessarily a replica responsible for passive transactions only.

$DBChangeRound(d, r) \triangleq$
$\land \neg amPrimary[d]$
$\land\ prnd[d] < r$
$\land\ \lor \exists\, msg \in \{m \in msgs : m.type \in \{$"1a", "1b", "2a", "2aS", "2b", "2bs"$\}\} :$
$\land\ prnd[d] < msg.rnd$
$\land\ msg.rnd = r$
$\land\ prnd' = [prnd$ EXCEPT $![d] = r]$
$\land\ drnd' = [drnd$ EXCEPT $![d] = r]$
$\land\ pfproposal' = [pfproposal$ EXCEPT $![d] = [i \in \{\} \mapsto none]]$
$\land\ dfreeinsts' = [dfreeinsts$ EXCEPT $![d] = Nat]$
$\land\ dinstof' = [dinstof$ EXCEPT $![d] = [t \in Tid \mapsto \{\}]]$
$\land$ UNCHANGED $\langle cvars, ldvars, ddreply, dcnt, vers, dcom, dlearned, avars, DBinter, msgs, amPrimary, dactivec\rangle$

Final Specification

The next-state action in terms of the noninterleaving actions.

$ClientAction \triangleq$
$\exists\, c \in Client, t \in Tid :$
$\lor \exists\, req \in Request : ReceiveReq(c, t, req)$
$\lor \exists\, rep \in Reply : ReplyReq(c, t, rep)$
$\lor\ PrematureAbort(c, t)$
$\lor\ PassiveCommit(c, t)$
$\lor\ ClientLearn(c)$

$DatabaseAction \triangleq$
$\exists\, d \in Database :$
$\lor \exists\, t \in Tid : \lor \exists\, req \in Request : DBReq(d, t, req)$
$\lor \exists\, rep \in Reply : DBRep(d, t, rep)$
$\lor\ DBAbortT(d, t)$
$\lor\ DatabaseLearn(d)$
$\lor\ \land$ UNCHANGED $ldinter[d] \land DBS(d)!SOPNext$ replica's internal action
$\land\ OtherDBsStutter(d)$
$\land$ UNCHANGED $\langle cvars, gdvars, avars, DBinter, msgs, amPrimary, dactivec\rangle$

$$\lor \exists\, r \in Nat : \lor Phase1a(d,\, r) \\ \lor Phase2Start(d,\, r) \\ \lor ActivateRound(d,\, r)$$

$$AcceptorAction \triangleq \\ \exists\, a \in Acceptor,\, r \in Nat : \\ \lor Phase1b(a,\, r) \\ \lor Phase2b(a,\, r)$$

$$ChangeOtherVariables \triangleq \\ \land amPrimary \in amPrimaryType \\ \land dactivec \in dactivecType \\ \land \text{UNCHANGED } \langle cvars,\, dvars,\, avars,\, DBinter,\, msgs \rangle$$

$$Next \triangleq \lor ClientAction \\ \lor DatabaseAction \\ \lor AcceptorAction \\ \lor ChangeOtherVariables$$

The final specification

$$Spec \triangleq Init \land \Box[Next]_{\langle cvars,\, dvars,\, avars,\, DBinter,\, msgs,\, amPrimary,\, dactivec \rangle}$$

4.4 Related Work and Final Remarks

The algorithm presented in Section 4.1.4 is inspired by the certified termination of the Database State Machine (DBSM) protocol described in [PGS03]. However, we use our understanding of the termination abstraction to improve performance. If we look at the termination properties stated in Section 2.2.3, there is nothing forcing certification to happen after totally ordering transactions. In fact, what the consistency property states is that committed transactions must be submitted to replicas at the same order, implying that certification should actually happen before propagation. This rationale allowed us to come up with an algorithm that reduces to burden of certification to a single process and manages to propagate only update operations to replicas, without increasing the termination latency or reducing the degree of fault tolerance. Since the idea behind the certification test remains the same as in the original DBSM protocol, optimizations like reordering [PGS97] or the use of version numbers [ZP05, ZP08] can still be made.

Interestingly, some deferred-update algorithms do not use any direct implementation of sequence agreement or atomic broadcast for active transaction termination. Instead, they use some variant of two-phase-commit [AAS97]. One might wonder if this contradicts our result that termination implies sequence agreement. The answer is no, it does

not. These algorithms use two-phase-commit as an optimistic implementation of atomic broadcast[PS03, PSUC02]. When two-phase-commit starts, each database votes for its local ordering of transactions. If all orderings are the same, transactions commit; otherwise, transactions simply abort. In the end, transactions are only committed if all database replicas agreed on the same commit order.

Our second algorithm, presented in Section 4.2 optimizes certification further by having it done by one of the replicas. A similar approach is taken in [PF00]. However, ours manages to get a significant lower latency for termination (3 communication steps less) due to the use of a tailor-made propagation protocol.

The idea of disabling disk writes at database replicas to obtain better performance has also been explored in the context of database replication protocols with weaker consistency guarantees [EDP06]. As one can easily imagine, performance gains due to this technique are significant.

Chapter 5

Conclusion

A whole is that which has beginning, middle, and end.
Aristotle

Replication is a good way to improve both performance and dependability of database systems. If several database replicas are available, performance can be improved by distributing the load among them. Moreover, if one of the replicas cannot be accessed due to failures, users can still rely on the other ones. However, providing a consistent database interface out of several replicas is not an easy task and requires replica synchronization. Although we would like to have replicas as independent of each other as possible for performance and dependability reasons, we must keep them synchronized if we want to provide a consistent interface to users. In this work, we studied how we can balance this trade-off to provide good performance and fault-tolerance without compromising consistency. Our basis is a widely used technique for database replication known as the deferred update technique. In this technique, transactions are initially executed in a single replica. After this initial execution, passive transactions, which do not change the state of the database, can commit locally to the replica they execute. Active transactions, which change the database state, must be synchronized with the transactions running on other replicas.

This thesis discusses the details involved in the design of deferred-update database replication protocols, and proposes a new model to design and analyze such protocols as well as novel algorithms of practical motivation and application.

5.1 Research Assessment

This thesis research has led to four major contributions, summarized below.

Abstraction of the Deferred Update Technique. We have presented a formal abstraction of the deferred update technique for database replication. Although this replication technique has been widely used in practical and theoretical works, there was no formalization of it up to now, forcing the design, analysis, and correctness proof of such protocols to be done by non-standard mechanisms, adapted from the analysis of centralized databases. Our abstraction allows one to come up with general results concerning this family of protocols (lower and upper bounds), prove their correctness, and easily design novel correct-by-construction protocols. As an example of the usefulness of our abstraction we have shown that, contrary to the assumptions made by previous works on deferred-update replication, the technique can cope with a concurrency control mechanism weaker than strict order-preserving serializability on database replicas. In fact, this motivated us to introduce the concept of active order-preserving serializability, which generalizes the behavior of some optimistic concurrency control algorithms and can be safely employed on general deferred-update algorithms. We have also used our abstraction to prove some limitations regarding the termination protocol for committing and propagating active transactions. Specifically, we have shown that it necessarily solves a sequence agreement problem among databases.

M-Consensus and Collision-fast Paxos. Using consensus to solve sequence agreement can lead to collisions, that is, concurrent proposals made to the same instance will conflict and, therefore, prevent at least one of them from being learned at that instance. Even collision-fast consensus protocols do not implement collision-fast sequence agreement using the standard implementation of atomic broadcast based on consensus. We introduced a variant of consensus called M-Consensus, more general than the original problem and more suitable as a building block for efficient sequence agreement implementations. Collision-fast Paxos, our solution to M-Consensus, allows for a very efficient and fault-tolerant solution to the problem of sequence agreement. Our latency-optimal algorithm, derived from the Paxos consensus protocol [Lam98], is very dynamic and can quickly reconfigure and adapt to failures, which distinguishes it from previous approaches achieving similar bounds. Our extensive correctness proofs not only increase our confidence in the algorithm, but they also carry some general structure that can be used to prove similar algorithms.

Certification-based Deferred-Update Algorithm. Our novel certification-based deferred-update algorithm has very little overhead associated with the termination of active transactions, even though it requires no extra assumptions about the database engines. Its most important feature is that it propagates only strictly necessary information to replicas. It also does not incur any extra work on the database replicas besides running local transactions and applying the required updates. Our termination protocol borrows from Paxos, ensuring the same latency and degree of fault tolerance as the original algorithm. It can certify and propagate active transactions to replicas within three communication steps as seen from the client. To the best of our knowledge, no previous protocol can ensure this latency without propagating certification information to replicas and requiring them to perform the certification test.

In-memory Primary-Backup Replication Algorithm. The previous algorithm has termination certification performed by a single agent, the current termination leader, and the test simply verifies the serializability of proposed active transactions. Therefore, one might think of using a database engine to perform this test. This is possible, in principle, but it does not guarantee that this database engine will be synchronized with the other replicas, which prevents it from being used to handle passive transactions and invalidates its state on the event of a leader change. A transaction that passes such a certification test will be internally committed at the leader's database even though it has not yet been chosen by acceptors and propagated to the other replicas. A failure suspicion can trigger a new round and a leader change that could certify this transaction differently and propagate a different outcome to the replicas, making them diverge from the state in the database of the previous leader. In order to allow the certification to be done by one of the database replicas, one has to assume stronger properties from the concurrency control mechanism they provide. Our in-memory primary-backup replication algorithm uses strong assumptions found in in-memory databases to reduce even more the latency and the burden associated with transaction termination, requiring only two communication steps and no extra certification procedure to commit proposed transactions. This algorithm can be nicely coupled with in-memory databases to provide very good performance in practice. Besides its practical relevance, it also shows how our deferred-update abstraction can help the design and analysis of protocols even if termination depends on stronger assumptions about the consistency guarantees of database replicas.

5.2 Future Directions and Open Questions

Weak Consistency. We have studied the context of deferred-update replication algorithms ensuring Serializability. The search for good performance has made many researchers exploit weaker consistency guarantees in replicated settings (e.g., [EZP05, LKMPJP05, EDP06]). Many of these algorithms rely on the main principle of deferred-update replication, that is, initially executing transactions on a single replica and proposing active transactions for termination. We conjecture that our deferred-update abstraction can be adapted to weaker consistency guarantees by just weakening the Consistency termination property and the assumptions about the replicas' concurrency control mechanism.

Strong Consistency. Some people might argue that Serializability is too weak as a database consistency guarantee [Lam92]. The main criticism against Serializability is that it can delay the serialization of write-only transactions forever. Nothing forces such transactions to be applied to the internal database state. Stronger properties might, for example, force a new transaction to always see the result of the last committed transactions. It would be interesting to analyze which modifications would be necessary to make our abstraction provide consistency guarantees stronger than serializability.

Transactional Memories. Our deferred-update abstraction is very general and could be possibly applied to shared memory systems as well as distributed settings. In shared memory systems, software transactional memories [ST97] have become an active research area and one could think of using our abstraction to design and analyze high-performance fault-tolerant transactional memory interfaces.

Wide-area Group Communication. In the classic Paxos algorithm, only a single leader is allowed to have its proposals accepted and learned in two communication steps. If the protocol is applied to a wide-area setting where clusters of machines are separated by long distances with large communication delays, proposals originated in a cluster different from the leader's will take three wide-area communication delays to be learned. Collision-fast Paxos allows a set of proposers to have their proposals learned in two communication steps and seems to be a good algorithm for wide area networks.

Collision-fast Paxos and Generic Broadcast. A good characteristic of Collision-fast Paxos, inherited from M-Consensus, is that it allows incomplete mappings to be learned. Gaps in the mapping would prevent learned values to be delivered through a sequence agreement interface. Applying Collision-fast Paxos to solve Generic Broadcast [PS02, ADGFT00] can allow values after the gap to be delivered if the learner knows that the proposer responsible for the gap has fast-proposed a value that does not conflict with the value after the gap. Recall that it takes two communication steps to have a proposed value learned, but the collision-fast proposer can send this notification directly to learners just to allow them to learn other values faster. Whether this optimization can have a practical application seems to be an interesting problem.

Partial Replication. Our deferred-update abstraction copes with full replication only. It can be used in the design and analysis of simple partial replication protocols such as [SSP06], but we do not know to what extent it can be applied or what modifications would be necessary to deal with more complicated algorithms.

Certification-free Termination. Some deferred-update protocols such as [KA00b] allow active transactions to execute on any database site and manage to use database replicas to perform termination certification, without propagating more information about the transaction history than the active history. These protocols, however, have strong failure detection assumptions and do not allow wrong failure suspicions. We think we can use some of the ideas of the algorithms presented in Chapter 4 to create a protocol that achieves such a certification-free termination and can cope with false failure suspicions.

Bibliography

[AAAS97] D. Agrawal, G. Alonso, A. E. Abbadi, and I. Stanoi. Exploiting atomic broadcast in replicated databases. In *Proceedings of EuroPar (EuroPar'97)*, Passau (Germany), September 1997.

[AAS97] D. Agrawal, A. E. Abbadi, and R. Steinke. Epidemic algorithms in replicated databases. In *Proceedings of the 16th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, Tucson (USA), May 1997.

[ADGFT00] M. K. Aguilera, C. Delporte-Gallet, H. Fauconnier, and S. Toueg. Thrifty generic broadcast. In *Proc. of the 14th Intl. Conference on Distributed Computing*, pages 268–282, Toledo, Spain, 2000.

[Agu04] M. K. Aguilera. A pleasant stroll through the land of infinitely many creatures. *SIGACT News*, 35(2):36–59, 2004.

[AL91] M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, May 1991.

[AT02] Y. Amir and C. Tutu. From total order to database replication. In *International Conference on Distributed Computing Systems (ICDCS)*, July 2002.

[BBG89] C. Beeri, P. A. Bernstein, and N. Goodman. A model for concurrency in nested transaction systems. *Journal of the ACM*, 36(2):230–269, April 1989.

[BBG+95] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ANSI SQL isolation levels. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 24(2):1–10, June 1995.

[BHG87] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[BK02] S. Blott and H. F. Korth. An almost-serial protocol for transaction execution in main-memory database systems. In *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB)*, pages 706–717, 2002.

[BN97] P. Bernstein and E. Newcomer. *Principles of Transaction Processing*. Morgan Kaufmann Publishers, 1997.

[Bro92] M. Broy. Algebraic and functional specification of an interactive serializable database interface. *Distributed Computing*, 6(1):5–18, 1992.

[CBS06] B. Charron-Bost and A. Schiper. Improving fast paxos: being optimistic with no overhead. *prdc*, 0:287–295, 2006.

[CPS06] L. Camargos, F. Pedone, and R. Schmidt. A Primary-Backup Protocol for In-Memory Database Replication. In *5th IEEE International Symposium on Network Computing and Applications (NCA'2006)*, pages 204–211, 2006.

[CSP06] L. Camargos, R. Schmidt, and F. Pedone. Multicoordinated paxos. Technical report, EPFL, 2006.

[CT96] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Communications of the ACM*, 43(2):225–267, 1996.

[DSU04] X. Défago, A. Schiper, and P. Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comput. Surv.*, 36(4):372–421, 2004.

[EDP06] S. Elnikety, S. Dropsho, and F. Pedone. Tashkent: uniting durability with transaction ordering for high-performance scalable database replication. In *EuroSys '06: Proceedings of the ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, pages 117–130, New York, NY, USA, 2006. ACM.

[EZP05] S. Elnikety, W. Zwaenepoel, and F. Pedone. Database replication using generalized snapshot isolation. In *SRDS '05: Proceedings of the 24th IEEE Symposium on Reliable Distributed Systems (SRDS'05)*, pages 73–84, Washington, DC, USA, 2005. IEEE Computer Society.

[FLP85] M. J. Fischer, N. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, 1985.

[GHOS96] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, Montreal (Canada), June 1996.

[Gif79] D. K. Gifford. Weighted voting for replicated data. In *Proceedings of the 7th Symposium on Operating System Principles SOSP 7*, pages 150–162, Pacific Grove (USA), December 1979.

[GL03] E. Gafni and L. Lamport. Disk paxos. *Distrib. Comput.*, 16(1):1–20, 2003.

[GL06] J. Gray and L. Lamport. Consensus on transaction commit. *ACM Transactions on Database Systems*, 31(1):133–160, 2006.

[GMS92] H. Garcia-Molina and K. Salem. Main memory database systems: An overview. *IEEE Transactions on Knowledge and Data Engineering*, 4(6):509–516, 1992.

[GR93] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[HAA99] J. Holliday, D. Agrawal, and A. E. Abbadi. The performance of database replication with group multicast. In *Proceedings of International Symposium on Fault Tolerant Computing (FTCS29)*, pages 158–165. IEEE Computer Society, 1999.

[HT93] V. Hadzilacos and S. Toueg. *Fault-tolerant broadcasts and related problems*, pages 97–145. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2 edition, 1993.

[KA00a] B. Kemme and G. Alonso. Don't be lazy, be consistent: Postgres-r, a new way to implement database replication. In *Proceedings of 26th International Conference on Very Large Data Bases*, September 2000.

[KA00b] B. Kemme and G. Alonso. A new approach to developing and implementing eager database replication protocols. *ACM Transactions on Database Systems*, 25(3):333–379, 2000.

[KD96] I. Keidar and D. Dolev. Efficient message ordering in dynamic networks. In *PODC '96: Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, pages 68–76, New York, NY, USA, 1996. ACM.

[KS92] R. Kurki-Suonio. Operational specification with joint actions: serializable databases. *Distributed Computing*, 6(1):19–37, 1992.

[Lam78] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[Lam89] L. Lamport. A simple approach to specifying concurrent systems. *Communications of the ACM*, 32(1):32–45, January 1989.

[Lam92] L. Lamport. Critique of the lake arrowhead three. *Distributed Computing*, 6(1):65–71, 1992.

[Lam95] L. Lamport. How to write a proof. *American Mathematical Monthly*, 102(7):600–608, 1995.

[Lam98] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998.

[Lam02] L. Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[Lam04] L. Lamport. Generalized consensus and paxos. Technical Report MSR-TR-2005-33, Microsoft Research, 2004.

[Lam06a] L. Lamport. Fast paxos. *Distributed Computing*, 19(2):79–103, October 2006.

[Lam06b] L. Lamport. Lower bounds for asynchronous consensus. *Distributed Computing*, 19(2):104–125, 2006.

[LKMPJP05] Y. Lin, B. Kemme, n.-M. Marta Pati and R. Jiménez-Peris. Middleware based data replication providing snapshot isolation. In *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 419–430, New York, NY, USA, 2005. ACM.

[LLOR99] P. Ladkin, L. Lamport, B. Olivier, and D. Roegel. Lazy caching in tla. *Distributed Computing*, 12(2/3):151–174, 1999.

[LMWF94] N. Lynch, M. Merrit, W. Weihl, and A. Fekete. *Atomic Transactions*. Morgan Kaufmann Publishers, Inc., San Mateo, CA, USA, 1994.

[LS92] S. S. Lam and A. U. Shankar. Specifying modules to satisfy interfaces: a state transition system approach. *Distrib. Comput.*, 6(1):39–63, 1992.

[Lyn96] N. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, Inc., San Mateo, CA, USA, 1996.

[ML04] M. Massa and L. Lamport. Cheap paxos. In *Proc. of the 2004 Intl. Conference on Dependable Systems and Networks*, June 2004.

[MLP79] R. A. D. Millo, R. J. Lipton, and A. J. Perlis. Social processes and proofs of theorems and programs. *Communications of the ACM*, 22(5):271–280, 1979.

[Pap79] C. H. Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM*, 26(4):631–653, October 1979.

[PF00] F. Pedone and S. Frølund. Pronto: A fast failover protocol for off-the-shelf commercial databases. In *Proceedings of the 19th IEEE Symposium on Reliable Distributed Systems (SRDS'2000)*, pages 176–185, 2000.

[Pfi98] G. F. Pfister. *In search of clusters*. Prentice Hall, 1998.

[PGS97] F. Pedone, R. Guerraoui, and A. Schiper. Transaction reordering in replicated databases. In *Proceedings of the 16th IEEE Symposium on Reliable Distributed Systems*, Durham (USA), October 1997.

[PGS03] F. Pedone, R. Guerraoui, and A. Schiper. The database state machine approach. *Journal of Distributed and Parallel Databases and Technology*, 14(1):71–98, 2003.

[PMJPKA00] M. Patino-Martínez, R. Jiménez-Peris, B. Kemme, and G. Alonso. Scalable replication in database clusters. In *Proc. of the* 14^{th} *International Symposium on Distributed Computing (DISC'2000)*, 2000.

[PS02] F. Pedone and A. Schiper. Handling message semantics with generic broadcast protocols. *Distributed Computing*, 15(2):97–107, April 2002.

[PS03] F. Pedone and A. Schiper. Optimistic atomic broadcast: a pragmatic viewpoint. *Theoretical Computer Science*, 291(1):79–101, January 2003.

[PSL80] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *J. ACM*, 27(2):228–234, 1980.

[PSUC02] F. Pedone, A. Schiper, P. Urban, and D. Cavin. Solving agreement problems with weak ordering oracles. In *Proc. of the 4th European Dependable Computing Conference*, pages 44–61, 2002.

[RMA+02] L. Rodrigues, H. Miranda, R. Almeida, J. Martins, and P. Vicente. Strong replication in the GLOBDATA middleware. In *Workshop on Dependable Middleware-Based Systems*, 2002.

[SS99] D. F. Savarese and T. Sterling. Beowulf. In *High Performance Cluster Computing*, volume 1, Architectures and Systems, pages 625–645. Prentice Hall PTR, 1999.

[SSP06] N. Schiper, R. Schmidt, and F. Pedone. Optimistic algorithms for partial database replication. In *Proceedings of the 10th International Conference on Principles of Distributed Systems (OPODIS'2006)*, pages 81–93, 2006.

[ST97] N. Shavit and D. Touitou. Software transactional memory. *Distributed Computing*, 2(2):99–116, 1997.

[Sto79] M. Stonebraker. Concurrency control and consistency of multiple copies of data in distributed Ingres. *IEEE Transactions on Software Engineering*, SE-5:188–194, May 1979.

[Tho79] R. H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Transactions on Database Systems*, 4(2):180–209, June 1979.

[VBLM07] B. Vandiver, H. Balakrishnan, B. Liskov, and S. Madden. Tolerating byzantine faults in database systems using commit barrier scheduling. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP)*, Stevenson, Washington, USA, October 2007.

[VR02] P. Vicente and L. Rodrigues. An indulgent uniform total order algorithm with optimistic delivery. In *Proc. of the 21th IEEE Symp. on Reliable Distributed Systems (SRDS'02)*, pages 92–101, Osaka, Japan, October 2002.

[WPS+00a] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso. Database replication techniques: a three parameter classification. In *Proceedings of 19th Symposium on Reliable Distributed Systems (SRDS'2000)*, pages 206–215, Nürnberg, Germany, 2000. IEEE Computer Society.

[WPS+00b] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso. Understanding replication in databases and distributed systems. In *Proceedings of 20th International Conference on Distributed Computing Systems (ICDCS'2000)*, pages 264–274, Taipei, Taiwan, April 2000.

[Zie06] P. Zielinski. Low-latency atomic broadcast in the presence of contention. In *Proc. of the* 20^{th} *Intl. Symposium on Distributed Computing, DISC'2006*, pages 505–519, 2006.

[ZP05] V. Zuikeviciute and F. Pedone. Revisiting the database state machine. In *VLDB Workshop on Design, Implementation and Deployment of Database Replication*, 2005.

[ZP08] V. Zuikeviciute and F. Pedone. Conflict-aware load-balancing techniques for database replication. In *Proceedings of the 23rd ACM Symposium on Applied Computing*, 2008.

Curriculum Vitae

I was born in Rio Grande, Rio Grande do Sul, south of Brazil, in 1978. From 1983 to 1992 I attended primary school. By the end of it, I was starting to get interested in computer systems and wanted to understand better how they worked. From 1993 to 1996 I attended a technical high school in Rio Grande called Colégio Técnico Industrial (CTI). There, I received regular secondary education together with technical courses in Computer Science. I do not know exactly when it happened, but by the time I graduated from CTI, I was already in love with Computer Science. From 1996 to 2000 I studied Computer Engineering at the University of Rio Grande (FURG).

From 2001 to 2003 I moved to Campinas, in the state of São Paulo, to do Masters in Computer Science at the University of Campinas (Unicamp). In my last semester at Unicamp, I taught Computer Science at the Catholic University of Campinas. By the end of 2003, I moved to Lausanne, Switzerland, to start my PhD studies at EPFL. I started as a pre-doctoral student at the Computer Networking Laboratory, and later moved to the Laboratory of Operating Systems. During the summer of 2005, I spent three months in an internship at HP Labs in Bristol, England. In the context of my PhD, a collaboration between EPFL and the University of Lugano, I had the opportunity to work with several people, give research talks, and interact with industrial partners.