Multicoordinated Agreement Protocols and the Log Service

Doctoral Dissertation submitted to the
Faculty of Informatics of the University of Lugano
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

presented by
Lásaro Jonas Camargos

under the supervision of
Prof. Fernando Pedone and Prof. Edmundo R.M. Madeira

April 2008

Dissertation Committee

Prof. Walter Binder	University of Lugano, Switzerland
Prof. Antonio Carzaniga	University of Lugano, Switzerland
Prof. Christof Fetzer	Technische Universität Dresden, Germany
Prof. Rodrigo Rodrigues	Max Plank Institute for Software Systems, Germany

Dissertation accepted on 29 April 2008

Supervisor
Prof. Fernando Pedone

Co-Supervisor
Prof. Edmundo R.M. Madeira

PhD program director
Prof. Dr. Fabio Crestani

I certify that except where due acknowledgement has been given, the work presented in this thesis is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; and the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program.

Lásaro Jonas Camargos
Lugano, 29 April 2008

Abstract

Agreement problems are a common abstraction in distributed systems. They appear when the components of the system must concur on reconfigurations, changes of state, or in lines of action in general. Examples of agreement problems are Consensus, Atomic Commitment, and Atomic Broadcast. In this thesis we investigate these abstractions in the context of the environment in which they will run and the applications that they will serve; in general, we consider the asynchronous crash-recovery model. The goal is to devise protocols that explore the contextual information to deliver improved availability. The correctness of our protocols holds even when the extra assumptions do not.

In the first part of this thesis we explore the following property: messages broadcast in small networks tend to be delivered in order and reliably. We make three contributions in this part. The first contribution is to turn known Consensus algorithms that harness this ordering property to reach agreement in the crash-stop model into practical protocols. That is, protocols that tolerate message losses and recovery after crashes, efficiently. Our protocols ensure progress even in the presence of failures, if spontaneous ordering holds frequently. In the absence of spontaneous ordering, some other assumption is required to cope with failures. The second contribution of this thesis is to generalize one of our crash-recovery consensus protocols as a "multicoordinated" mode of a hybrid Consensus protocol, that may use spontaneous ordering or failure detection to progress. Compared to other protocols, ours provide improved availability with no price in resilience. The third contribution is to employ this new mode to solve Generalized Consensus, a problem that generalizes a series of other agreement problems and, hence, is of much practical interest. Moreover, we considered several aspects of solving this problem in practice, which had not been considered before. As a result, our Generalized Consensus protocol features graceful degradation, load balancing, and is parsimonious in accessing stable storage.

In the second part of this thesis we have considered agreement problems in wide area networks organized hierarchically. More specifically, we considered a topology that is commonplace in the data centers of large corporations: groups of nodes, with large-bandwidth low-latency links connecting the nodes in the same group, and

slow and limited links connecting nodes in different groups. In such environments, latency is clearly a major concern and reconfiguration procedures that render the agreement protocol momentarily unavailable must be avoided as much as possible. Our contribution here is in avoiding reconfigurations and improving the availability of a collision fast agreement protocol. That is, a protocol that can reach agreement in two intergroup communication steps, irrespectively to concurrent proposals. Besides the use of a multicoordinated approach, we employed multicast primitives and consensus to restrict some reconfigurations to within groups, where they are less expensive.

In the last part of this thesis we study the problem of terminating distributed transactions. The problem consists of enforcing agreement among the parties on whether to commit or rollback the transaction and ensuring the durability of committed transactions. Our contribution in this topic is an abstract log service that detaches the termination problem from the processes actually performing the transactions. The service works as a black box and abstracts its implementation details from the application utilizing it. Moreover, it allows slow and failed resource managers be re-started on different hosts without relying on the stable storage of the previous host. We provide two implementations of the service, which we evaluated experimentally.

Resumo

Problemas de acordo, como Consenso, Terminação Atômica e Difusão Atômica, são abstrações comuns em sistemas distribuídos. Eles ocorrem quando os componentes do sistema precisam concordar em reconfigurações, mudanças de estado ou em linhas de ação em geral. Nesta tese, investigamos estes problemas no contexto do ambiente e aplicações em que serão utilizados. O modelo geral é o assíncrono sujeito a quebras com possível posterior recuperação. Nossa meta é desenvolver protocolos que explorem esta informação contextual para prover maior disponibilidade, e que se mantenham corretos mesmo que algumas das prerrogativas do contexto tornem-se inválidas.

Na primeira parte da tese, exploramos a seguinte propriedade: mensagens difundidas em pequenas redes tendem a ser entregues ordenada e confiavelmente. Nós fazemos três contribuições nesta parte da tese. A primeira é a transformação de algoritmos conhecidos para o modelo quebra-e-pára, que utilizam a propriedade de ordenação mencionada, em protocolos práticos. Isto é, protocolos que toleram perda de mensagens e recuperação após a quebra. Nossos protocolos garantem progresso na presença de falhas, contanto que mensagens sejam espontaneamente ordenadas freqüentemente. Na ausência de ordenação expontânea, outras prerrogativas são necessárias para contornar falhas. A segunda contribuição é a generalização de um dos algoritmos citados acima em um modo de execução "multi-coordenado" em um protocolo híbrido de consenso, que usa ou ordenação expontânea ou detecção de falhas para progredir. Em comparação a outros protocolos, o nosso provê maior disponibilidade sem comprometer resiliência. A terceira contribuição é a utilização do modo multi-coordenado para resolver Consenso Generalizado, um problema que generaliza uma série de outros e que, portanto, é de grande interesse prático. Além disso, fizemos diversas considerações sobre aspectos práticos da utilização deste protocolo. Como resultado, nosso protocolo perde desempenho gradualmente no caso de condições desfavoráveis, permite o balanceamento de carga sobre os coordenadores, e acessa a memória estável parcimoniosamente.

Na segunda parte da tese, consideramos problemas de acordo no contexto de redes organizadas hierarquicamente. Em específico, nós consideramos uma topologia usada nos *data centers* de grandes cooporações: grupos de máquinas conectadas in-

ternamente por *links* de baixa latência, mas por *links* mais lentos entre grupos. Em tais cenários, latência é claramente um fator importante e reconfigurações, onerosas aos protocolos, devem ser evitadas tanto quanto possível. Nossa contribuição neste tópico está em evitar reconfigurações e melhorar a disponibilidade de um protocolo de acordo que é *rápido* a despeito de colisões. Isto é, um protocolo que consegue chegar a uma decisão em dois passos *inter-grupos* mesmo quando várias propostas são feitas concorrentemente. Além do uso da técnica de multicoordenação, nós usamos primitivas de *multicast* e consenso para conter algumas reconfigurações dentro dos grupos, onde seus custos são menores.

Na última parte da tese nós estudamos o problema de terminação de transações distribuídas. O problema consiste em garantir que os vários participantes da transação concordem em aplicar ou cancelar de forma consistente as suas operações no contexto da transação. Além disso, é necessário garantir a durabilidade das alterações feitas por transações terminadas com sucesso. Nossa contribuição neste tópico é um *serviço de log* que abstrai e desassocia a terminação de transações dos processos que executam tais transações. O serviço funciona como uma caixa preta e permite que *resource managers* lentos ou falhos sejam reiniciados em servidores diferentes, sem dependências na memória estável do servidor em que executava anteriormente. Nós apresentamos e avaliamos experimentalmente duas implementações do serviço.

Preface

This thesis describes my PhD work, initiated in 2003 at the State University of Campinas, under the supervision of Prof. Edmundo Madeira, and finished in 2008 at the University of Lugano, under the supervision Prof. Fernando Pedone. During this period I have worked on different problems, with different approaches, and in collaboration with other PhD students. The work, however, has always been on agreement problems.

In the early "middleware" stage of my studies I implemented an agreement library featuring the weak ordering based consensus algorithms presented in the third chapter of this thesis. In modularizing the library and making the algorithms interchangeable with a Paxos implementation, I began to understand what Mike Burrows allegedly meant with "In my experience, all distributed consensus algorithms are either 1: Paxos, 2: Paxos with unnecessary extra crust, or 3: broken." While I do not completely agree with him, I believe that the formalism in which Paxos was specified is an invaluable tool to understand and compare agreement algorithms. Mainly, the separation of concerns in it. The work in [Camargos et al., 2006a] presented my "paxonized" weak ordering based protocols and compared them with Paxos, the classic one.

Database replication came into play in my studies soon after I moved to Switzerland and joined the work presented in [Camargos et al., 2006b]. Our transaction processing protocol was very efficient, as long as the database using it could afford serializing the execution of its transactions. The killer application for the protocol, which we then called "sprint", was an in-memory database.

When the new Sprint appeared, it was a cluster data management system, featuring replication and partitioning of data, and my agreement library was an important part of it. I rewrote the library to better fit the project, and it became a lightweight group communication library, later used in two other projects within our research group. Sprint is described in [Camargos et al., 2007a], and the group communication within it was abstracted as the log service presented in the last part of this thesis [Camargos et al., 2008b].

Given these good experiences, my work focused on optimizing agreement protocols to specific scenarios, but in a more abstract way. First improving resilience with

multicoordination [Camargos et al., 2007b, Camargos et al., 2008a], then minimizing the effects of concurrent proposals [Schmidt et al., 2007], and finally mixing both approaches to solve agreement among groups.

To conclude, part of my work was done in collaboration with other researchers. In this thesis, I focus on the work in which I have been the primary contributor. These are the works presented in [Camargos et al., 2006a, Camargos et al., 2008a, Camargos et al., 2008b] and the agreement protocol for groups introduced in this thesis.

[Camargos et al., 2006a] Camargos, L., Madeira, E. R. M., and Pedone, F. (2006a). Optimal and practical wab-based consensus algorithms. In *Euro-Par 2006 Parallel Processing*, volume 4128 of *Lecture Notes in Computer Science*, pages 549–558, Berlin / Heidelberg. Springer.

[Camargos et al., 2006b] Camargos, L., Pedone, F., and Schmidt, R. (2006b). A primary-backup protocol for in-memory database replication. In *NCA '06: Proceedings of the Fifth IEEE International Symposium on Network Computing and Applications*, pages 204–211, Washington, DC, USA. IEEE Computer Society.

[Camargos et al., 2007a] Camargos, L., Pedone, F., and Wieloch, M. (2007a). Sprint: a middleware for high-performance transaction processing. In *EuroSys '07: Proceedings of the 2007 conference on EuroSys*, pages 385–398, New York, NY, USA. ACM Press.

[Camargos et al., 2007b] Camargos, L. J., Schmidt, R. M., and Pedone, F. (2007b). Multicoordinated paxos: Brief announcement. In *PODC '07: Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, pages 316–317, New York, NY, USA. ACM Press.

[Camargos et al., 2008a] Camargos, L., Schmidt, R., and Pedone, F. (2008a). Multicoordinated agreement protocols for higher availabilty. In *NCA '08: Proceedings of the Seventh IEEE International Symposium on Network Computing and Applications*, Washington, DC, USA. IEEE Computer Society.

[Camargos et al., 2008b] Camargos, L., Wieloch, M., Pedone, F., and Madeira, E. (2008b). A highly available log service for transaction termination. In *Proceedings of the seventh International Symposium on Parallel and Distributed Computing (ISPDC 2008)*.

Acknowledgements

This PhD is the culmination of a long sequence of steps started several years ago. For supporting me in the decision of starting each of these steps and helping me getting to the next, I would like to thank my family. Dona Ida, Ewe, Sebastião, "Seu" Eurípedes, Dona Neuza: esta conquista também é sua.

Everybody says that the most important outcome of a Ph.D. is what you learn in the process. For helping me learn, I would like to thank my advisors, Profs. Edmundo Madeira and Fernando Pedone.

I would like to also thank everybody who in a way or another contributed and supported me in finishing this PhD. For the discussions, feedback on my work, chats and laughs, and for priceless dinners, I would like to thank all my lab-mates, colleagues, and friends that I have made in Campinas, Lugano, and Seattle. The following is a list of some of them in order of "appearance": Herr, Pará, Bezerra, Cláudio, Rodrigo, Bianca, Vaide, Marcin, Nicholas, Marija, Dan, Avinash, Pino, Deniz, Avi, Alex, Aliaksey, and Amir.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

In order to make an apple pie from scratch,
you must first create the universe.
Carl Sagan

A distributed application is a composite of agents that perform local actions and exchange information to cooperatively perform some global task. These agents—machines, processors, or processes—are often required to synchronize their actions to meet some consistency criteria: only one agent may access a given resource at any point in time; all must commit the effect of their actions or rollback to a previous state; a certain agent must be excluded from any future interactions. Synchronizing actions, here, is short for reaching agreement.

Research on agreement problems has been an active field for at least thirty years and resulted in many algorithms for different computational models, lower and upper bounds, and an incredible amount of publications. Nonetheless, we feel that there is still room for improvement, mainly concerning the practicality of algorithms.

The focus of this thesis is on practical protocols for agreement problems in distributed asynchronous systems. That is, protocols that cope with failures of participants and their posterior recover in a graceful way; protocols that explore synchronism and optimistic assumptions to progress, but do not rely on them to remain correct and, more importantly, that adapt when assumptions no longer hold; protocols that explore building blocks readily available in todays infrastructures, properties of the environments in which they run, and the characteristics of the application that they serve.

1.1 About Rounds, Consensus, and Agreement Problems

Most agreement problems can be subsumed by the Consensus problem, in which agents must agree on one out of a set of proposed values. Consensus algorithms perform in stages that reflect the very nature of the problem: proposals precede a deliberation phase, that precedes the learning of a decision. To be fault-tolerant, algorithms must be ready to retry this cycle, guaranteeing that a decision in a previous cycle is honored in the next ones. We call each of these tries a round.

The fastest general round has the agents collecting the proposals, selecting one as their proposals, and exchanging their selections. Agents then decide for the proposal selected by at least a quorum of agents. In an asynchronous system, we can rely on overlapping quorums to ensure that no two different proposals are decided. Breaking a tie when starting a new round and searching for previous decisions, however, requires larger overlappings—in general, every three quorums must intersect—or more time—by searching in two phases instead of one, simply overlapping quorums are enough.

While asynchronism may prevent the progress of any fault-tolerant consensus algorithm [Fischer et al., 1985], the approach described the in the previous paragraph may not decide even when the environment behaves synchronously and no failure occurs. If multiple proposals are selected, but no one is ever selected by a full quorum in any round, then no decision is reached. Agents may circumvent this problem and eventually select the same value with the aid of random oracles [Rabin, 1983, Ben-Or, 1983] or based on the order in which the options are presented to them [Pedone et al., 2002b, Pedone et al., 2002a, Camargos et al., 2006a]. Another way is to avoid multiple possibilities by having one of the agents filter all but one proposal in the round [Chandra et al., 1996, Lamport, 1998]. The problem then becomes selecting this special agent [Chandra et al., 1996].

Although in the literature most algorithms are specified with a single set of agents, in actual systems some agents are responsible for proposing values, some only care for learning the decision, and just a subset of them actually work in reaching the agreement. From the previous paragraph, there is yet another set of agents that coordinate rounds, the coordinators.

Coordinators are of special interest in that they are tightly related to the algorithms' availability. If multiple coordinators start their rounds in parallel, none may succeed in reaching agreement. Hence, a single coordinator should be picked. On the one hand, a decision can only be reached with such a coordinator's participation, making it extremely important to replace it as soon as it fails. On the other hand, aggressive failure detection is resource consuming and error prone, and replacing a coordinator is not done without a price: the algorithm will be unavailable while the new coordinator contacts the other agents to determine any previously decided value. Therefore, replacement should be avoided as much as possible.

While discussing availability issues due to coordinator replacement in a consensus instance may seem frivolous at first, the impression fades when considered the way consensus is used in practice. A typical example is the implementation of a replicated state machine [Lamport, 1978, Lampson, 2001]. This well known technique consists of implementing reliable services by replicating simpler instances of the services on failure-independent processors. Replicas consistently change their states by applying deterministic commands from an agreed sequence. A consensus instance can be used to decide on each command of the sequence. To make the implementation efficient, all instances may run in parallel, share the same elected coordinator, and overlap parts of their rounds. Hence, being unavailable means delaying not one, but all ongoing decisions.

In many replicated state machines and similar applications, some commands submitted to the servers are commutable and do not have to be ordered. The straightforward use of consensus, which does not capture this commutability property, would unnecessarily order them. Generalized consensus [Lamport, 2004] , which in fact generalizes many agreement problems, captures the notion of commutability and may be used to agree not on a sequence, but on a partial order of commands. This particular instance of generalized consensus is known as the generic broadcast problem [Pedone and Schiper, 2002]. The only algorithm for generalized consensus, that we are aware of, has the same availability problems of consensus algorithms that we have pointed in the previous paragraphs: it either uses large quorums or relies on a single round coordinator to ensure progress.

The power of generalized consensus and generic broadcast comes from identifying conflicting proposals, e.g., non-commutable commands, so that they do not have to be unnecessarily ordered. A different approach is to ignore conflicts *a priori* to deliver them as fast as possible and use the conflict information to order delivered proposals *a posteriori*. For example, in M-Consensus [Schmidt et al., 2007] agents agree not on a single value per instance, but on a mapping from each proposer to its proposal or a nil value. On the one hand, if no two proposals are made by the same proposer, then decisions can be reached very fast and, after a full mapping is decided, a deterministic function may flatten the mapping into a sequence. On the other hand, because in general the deterministic function requires a full mapping and because only a proposer can propose for itself, the availability of M-Consensus protocols are bound to the availability of these proposers.

1.2 Transaction Termination

Many distributed systems' agreement requirements may be seen as the more abstract atomic commitment problem, most notably data management systems. In essence, to terminate a distributed transaction, each participating resource manager votes

to either commit or abort the transaction. If all participants vote to commit the transaction, then this must be the final outcome; if any participant votes for the abortion, then the transaction is aborted. For most real systems, this definition of the problem is not practical since it requires the vote of every participant to be accounted. This restriction is relaxed in *non-blocking atomic commitment*, in which the transaction may be aborted if any participant is suspected of having crashed. Commit must be guaranteed only if all participants vote to commit the transaction and none is suspected of failure. In this thesis we consider this weaker problem.

Widely used in practice, the *Two-Phase Commit* protocol (2PC) is non-blocking in the presence of failures of all but one process, the transaction manager. Gray and Lamport's Paxos Commit [Gray and Lamport, 2006] is a non-blocking atomic commitment protocol in which the casting of a vote is reduced to a Paxos consensus instance. Failures are naturally handled by letting participants enforce termination of each other's instances, and the transaction manager is required only to trigger the termination upon completion of a transaction; if it fails, resource managers simply abort the transaction. Hence, such a reduction to consensus is very powerful.

Even when *atomicity* is enforced by the commit protocol, if resource managers are allowed to forget the updates of a committed transactions, for example due to a temporary failure, then the system may still get to an inconsistent state. Hence, besides ensuring atomicity, a transaction termination protocol may be required to enforce the *durability* of committed transactions. That is, to ensure that all changes done by committed transaction are reflected by any future state of the database, in spite of any failures. In conventional protocols, durability is achieved by having each resource manager store its updates in a local stable media before voting. Should it fail, at recovery time the resource manager can read the committed updates from the local storage and replay them to recover its previous state. A drawback of this approach is that it couples the availability of the resource manager with the availability of the server hosting it.

1.3 Contributions

This thesis makes the following contributions.

Multicoordinated Consensus Regarding the consensus problem, we have made two contributions. The first contribution is a pair of protocols, namely B*-Consensus and R*-Consensus, which explore spontaneous ordering to solve consensus in the crash-recovery model. Comparatively, these protocols trade resilience for latency: B*-Consensus takes three communication steps to finish and tolerates the permanent failure of any minority of the agents. R*-Consensus decides in two steps, but tolerates less than one third of permanent failures.

The second contribution is the generalization of B*-Consensus as a multicoordinated mode of execution for agreement protocols. In this mode, the coordinator of each round is replaced by a set of overlapping quorums of coordinators. As long as at least one quorum of coordinators is alive, there is no need to change rounds and incur in the temporary unavailability resulting from it. We present a multicoordinated consensus protocol that extends Fast Paxos [Lamport, 2004], with its fast and classic modes. The protocol can switch to different types of round and adapt to environment changes: slower network, more message losses, higher or lower workload, and spontaneous message ordering.

Multicoordinated Generalized Consensus The third contribution is the use of multicoordination to solve generalized consensus. While the use of semantic information minimizes the chances of the coordinators in the same round disagreeing, multicoordination minimizes unwanted round changes. Hence, our protocol, Multicoordinated Paxos, is a synergism of these techniques. We present the protocol along with a discussion of various aspects of using it in practice, and present a simplified instantiation of the protocol that solves the Generic Broadcast problem [Pedone and Schiper, 2002].

Multicoordinated Agreement for Groups The fourth contribution regards the use of multicoordination in hierarchically organized networks. In these scenarios, coordinator quorums will increase availability, but will still require proposals to travel from all ends of the network to reach the quorum. Spreading the quorums will not improve the situation since quorums must be overlapping. We present a Multicoordinated Consensus protocol that solves this problem by splitting coordinator quorums in partially independent systems, and use the M-Consensus approach to reach agreement. We also show how this protocol can be improved by a recursive use of consensus and multicast technology.

Log Service Our last contribution is to abstract the atomicity and durability problems in transaction termination in terms of a *Log Service*. The service collects the updates performed by each participant as well as their votes (commit and abort) for each transaction they take part. When enough votes are collected, the service contacts the participants and informs them the transaction outcome, based on the votes received. Defining transaction termination in terms of our log service has two advantages. First, transaction termination becomes oblivious to particularities of the system, taken care of or explored by the services implementation transparently. Second, the overall availability of resource managers is improved by using a highly available implementation of the service. As mentioned earlier, the service can be

used to migrate crashed or slow resource managers to functional and more dependable hosts. As a consequence, resource managers may choose to asynchronously store their state locally for later recovery or rely solely on the state kept at the log service.

Besides the abstract specification, we present two implementations of the log service. Both implementations rely on consensus to achieve high availability in different ways, with performance implications on both the termination of transactions and on the recovery of resource managers. In the first implementation, *uncoordinated*, voting is completely distributed, abstracting Paxos Commit and, by extension, 2PC. In the second, *coordinated*, voting is managed by an easily replaceable coordinator agent. The two approaches abstract the trade-off "message complexity versus number of communication steps" between atomic commit protocols. We compare both approaches analytically and experimentally to previous work in the area and show that reducing the number of communication steps at the expense of increasing the message complexity not always leads to better performance.

1.4 Algorithmic Notation

In this thesis we present algorithms as state machines, specified in terms of atomic actions, state functions, and operators. The notation we use to define each of them should to be less ambiguous than the normally used pseudo-code, but more easily readable than the TLA$^+$ [Lamport, 2002a] specifications in which we have specified our main algorithms.

An action can only be executed if all of its pre-conditions are satisfied, in which case we say that the action is enabled. An enabled action executes *atomically* and changes the state machine accordingly. State functions evaluate some condition over the state of the state machine and are useful to express properties; a state function that evaluates to a boolean value is called a state predicate. Operators are functions that operate on a set of parameters.

Agents are divided in sets according to their roles. Hence, checking if an agent plays a given role is the same as checking its inclusion in the respective set. Agents keep their state in variables indexed by their names. In the following specification, for example, the action *FlipIfZero* is enabled for an agent a of type *flipper* that has its variable $var[a]$ equal to 0. If the action is executed, $var[a]$ is flipped to 1.

Most keywords that we use have obvious meanings. For example, the $If(a, b, c)$ operator defined above evaluates to b if a is true and to c otherwise. Another example is the pair LET and IN, which specifies the scope of a definition. In the definition of $SumSeq(seq)$ above, for example, *Sum* is defined as a recursive function that iterates over the elements of a sequence summing them, but only inside *SumSeq*. For

Algorithm 1 Notation example.

FlipIfZero$(a) \triangleq$
  pre-conditions: · $a \in$ *flipper*
  · *var*$[a] = 0$
  actions: · *var*$[a] \leftarrow 1$

If$(a, b, c) \triangleq$ IF a THEN b ELSE c

SumSeq$(seq) \triangleq$
  LET *Sum*$(i) \triangleq$ IF $i = 0$ THEN *seq*$[i]$ ELSE *seq*$[i] +$ *Sum*$(i - 1)$
  IN *Sum*(*Len*(*seq*))

RandInt$(min, max) \triangleq$ CHOOSE $i : i \in \mathbb{I} \land min \leq i \leq max$

the sake of simplicity, we collapse nested LET IN pairs into a single one and let each definition be in the scope of the next ones.

We represent sequences as tuples. We denote by LEN(s) the number of elements of sequence s and by $s[i]$ the i-th element of s. The sequence $\langle\rangle$ is the empty sequence and $\langle a, b, c\rangle$ is the sequence of elements a, b and c, in this order. When comparing sequences, the symbol "_" matches any element. For example, $\langle a, b\rangle = \langle \_, b\rangle$ and $\langle \_, b\rangle = \langle c, b\rangle$ even if $c \neq a$.

The CHOOSE $v : C$ operator is used to select some value v that satisfies condition C. In the definition of *RandInt*, for example, CHOOSE is used to select a random integer between the min and max values.

1.5 System Model

A distributed system is composed of a set of *agents* with well defined roles that cooperate to achieve a common goal. In practice, an agent can be implemented by a process or collection of them, by a processor, or any computation enabled entity. Moreover, any single entity that implements one agent could also implement multiple of them. Reasoning in terms of agents allows us to specify problems and algorithms more concisely and in terms of heterogeneous agents.

Distributed systems can be classified in different axis according to the way agents exchange information, the way they fail and recover, and the relative speeds at which they perform computation. In this work we address asynchronous distributed systems in which agents can crash and recover, and use unreliable communication channels to exchange messages.

In asynchronous distributed systems there are no bounds on the time it takes an agent to execute any action or for a message to be transmitted. We show that if

such bounds exist, then the protocols we present in this thesis ensure some liveness properties, if the number of failures can be limited in time. Our liveness proofs require the bounds to exist but do not require them to be known by any agent.

Even though we assume that agents may recover, they are not obliged to do so once they have failed. For simplicity, an agent is considered to be *nonfaulty* iff it never fails. Agents are assumed to have access to local stable storage which they can use to keep their state in between failures. State not kept in stable storage is reset after a crash. Lastly, we assume that agents do not execute any arbitrary step, *i.e.*, we do not consider byzantine failures.

Although channels are unreliable, we assume that if agents keep retransmitting their messages, then they eventually succeed in communicating with each other. We also assume that messages are not duplicated and cannot be undetectably corrupted.

1.6 Thesis Outline

The remainder of this thesis is organized as follows. In Chapter 2 we present our wab based consensus protocols, namely B*-Consensus and R*-Consensus, in Section 2.2. Also in Chapter 2, we present our multicoordinated consensus algorithm, in Section 2.3. In Chapter 3 we focus on the generalized consensus problem. In specific, in Section 3.4 we present our multicoordinated generalized consensus protocol, Multicoordinated Paxos. In Section 3.5 we show how to instantiate Multicoordinated Paxos to solve the generic broadcast problem. In Chapter 4 we deal with agreement among agents organized in groups and present our basic and extended protocols for such a scenario. We present our last contribution, the log service specification and implementations, in Chapter 5. Finally, in Chapter 6, we conclude this thesis and point some directions for future works.

Chapter 2

Multicoordinated Consensus

2.1 Consensus and the FLP Impossibility Result

Distributed problems and algorithms are commonly described in terms of a task to be attained by homogeneous agents. The standard specification of the consensus problem, for example, states that "a set of agents must eventually agree on a value, in spite of a maximum number of failures". As a result, algorithms for such problems are also given in terms of homogeneous agents with similar behavior. Because processes play different roles in real systems, we use a different approach and specify problems and algorithms in terms of the roles which agents play. For example, in a client/server architecture, we refer to *client* agents and *server* agents, or to *sender* and *receiver* in a mailing system.

In the case of consensus, we use three kinds of agents: *proposer*, *learner*, and *acceptor*. Because consensus subsumes many agreement problems, the same sets of agents may also make sense it their specifications and, hence, we will adopt the same terminology. To the best of our knowledge, specifications based on roles was introduced by Lamport [Lamport, 1998].

In the consensus problem, agents must agree on a single value out of a given set of proposals. Proposer agents issue proposals out of which one will become the decision. Once a decision is reached, learners must become aware of its value. In the context of a common distributed application, state machine replication, proposers can be thought of as clients issuing commands and learners as the application servers that execute the decided commands. For this reason, we interchangeably refer to proposals also as commands. Clients might also be learners to know whether their issued commands were accepted by the system to be executed.

Formally, the safety requirements of consensus are three [Lamport, 2006b]:

Nontriviality: Any value learned must have been proposed.

Stability: A learner can learn at most one value.

Consistency: Two different learners cannot learn different values.

While the safety requirements are stated in terms of proposers and learners, the liveness requirement is stated over the set of *acceptors*. This happens because, as we pointed out, proposer and learner roles are associated with clients of applications, and it would be unreasonable requiring them not to fail. Acceptors, conversely, are part of the application infrastructure, and it is more reasonable to make reliability assumptions about them. We call a *quorum* any finite set of acceptors that is large enough not to forbid liveness and define the liveness requirement of consensus as follows:

Liveness: For any proposer p and learner l, if p, l, and a quorum Q of acceptors are nonfaulty and p proposes a value, then l eventually learns some value.

Under the assumed asynchronous crash-recovery model, it is well known that no fault-tolerant consensus algorithm can ensure termination in the presence of failures [Fischer et al., 1985]. Phrased in terms of acceptors, this result implies that quorums must equal the set of all acceptors. Hence, algorithms must make extra assumptions about the system to ensure liveness if they must be fault-tolerant.

2.1.1 Randomization and Spontaneous Ordering

One possible way of circumventing the impossibility result is through the use of randomization. The algorithm of Bracha and Toueg [Bracha and Toueg, 1983], for example, rely on the fact that, if agents keep exchanging messages in rounds, then there is a non-zero probability that they will all eventually receive the same set of messages in some round. The authors have called this property *fair scheduling*.

The algorithms of Rabin [Rabin, 1983] and Ben-Or [Ben-Or, 1983] employ randomization in a different way. In their algorithms, if agents have no reason to opt for some proposal or another in some round, then they use a random bit generator to chose one. Given that there is a non zero probability that all chose the same random bit, agreement is reached with probability 1.

Pedone *et al.* [Pedone et al., 2002a] later replaced the selection based on the random bit generator used by Rabin [Rabin, 1983] and Ben-Or [Ben-Or, 1983] for a selection based on the order in which messages are received. This is possible if, for every round, there is a non zero probability that messages will be received in the same order. These properties, somehow related to fair scheduling, are abstracted by *weak ordering oracles* [Pedone et al., 2002b]. In specific, the algorithms of Pedone *et al.* [Pedone et al., 2002a], namely B-Consensus and R-Consensus in reference to Ben-Or and Rabin, use the *weak atomic broadcast (WAB)* oracle, which ensures that if processes keep exchanging broadcast messages then, in some rounds, the first

message received by all running agents is the same. In the same work, the authors show that, with high probability, Ethernet broadcast satisfies the WAB specification.

WAB based protocols are interesting from a practical perspective since they do not make any synchrony assumption. Nonetheless, B-Consensus and R-Consensus are of more theoretical than practical interest for their assumed failure and communication models: crash-stop and reliable links. From a pragmatic perspective, agents should be capable of reintegrating the system after a crash and tolerate message losses, being able to can make better use of highly-efficient communication means (e.g., UDP messages).

Crash-Recovery WAB-Based Consensus

We have extended the protocols of Pedone *et al.* [Pedone et al., 2002a] to the crash-recovery model with fairly lossy channels. Moreover, we defined these extended protocols using roles and relaxed the constraint that all agents run the same protocol as in the original algorithms [Camargos et al., 2006a]. These protocols, which we have named B*-Consnesus and R*-Consensus in reference to the algorithms in which they were inspired, are the first contribution of this chapter.

2.1.2 Synchronism and Failure Detection

Several works have considered circumventing the impossibility of solving consensus by assuming a partially synchronous model. Dolev *et al.* [Dolev et al., 1987] and Dwork *et al.* [Dwork et al., 1988] have studied, classified, and determined minimal synchrony assumptions needed to solve consensus. Christian and Fetzer [Cristian and Fetzer, 1999] have shown that several synchronism assumptions are realistic to some distributed systems.

Chandra and Toueg [Chandra and Toueg, 1996] introduced the concept of unreliable failure detectors, or UFD, which encapsulate the synchrony assumptions needed to solve consensus as abstract properties. These oracles provide possibly wrong information about the failures of agents. The weakest UFD that can be used to solve consensus, $\diamond\mathscr{W}$ [Chandra et al., 1996], ensures two properties:

Eventual Weak Completeness all agents that permanently crash are eventually suspected by a nonfaulty agent;

Eventual Weak Accuracy eventually, at least one nonfaulty agent will stop being suspected by the other nonfaulty agents.

Another interesting failure detection abstraction is the Ω leader election oracle, introduced and shown equivalent to $\diamond\mathscr{W}$ by Chandra *et al.* [Chandra et al., 1996]. Briefly, Ω ensures that nonfaulty agents eventually agree on the identity of some

nonfaulty agent, the leader. Failure detectors have been used to solve consensus in crash-stop [Chandra and Toueg, 1996, Schiper, 1997, Hurfin and Raynal, 1999, Hurfin et al., 2002, Dutta and Guerraoui, 2002], in crash-recovery [Lamport, 1998, Aguilera et al., 1998, Hurfin et al., 1998, Lamport, 2001], and byzantine settings [Castro and Liskov, 1999, Lampson, 2001, Zielinski, 2004, Martin and Alvisi, 2006].

In each of these algorithms, computation progresses as a sequence of rounds, each of which is managed by a *coordinator* agent. The coordinators are the only agents that send proposals to acceptors to get them decided; proposers resort to coordinators to have their proposals considered. When a new round is started, its coordinator must determine possibly previously decided values and use such a value, if existent, in the new round. Moreover, a new round must prevent previous ones from deciding if they have not done so yet. Hence, if rounds are indiscriminately started, no one will succeed in deciding any value. To avoid such a scenario, a *leader* coordinator is selected at a time to start new rounds using a leader election oracle, like Ω. Because failure detectors can make mistakes, rounds may be started even in absence of failures, what can prevent progress unless these mistakes cease to happen.

2.1.3 Availability Issues of Leader Based Protocols

A proposal issued by the leader takes two steps to be decided and learned, in the best scenario. One step to be propagated to the acceptors and one more from the acceptors to the learners. From the point of view of regular proposers, one step more is required to send their proposals to the leader. Hence, the leader becomes a single point of failure for every round. When it fails, its failure must be detected, a new leader must be elected, and a new round must be started. As we mentioned before, starting a round the new leader to synchronize with acceptors to block previously started rounds; while this synchronization is done, no decision can be reached.

Fast Paxos [Lamport, 2006a] extends the leader-based Paxos [Lamport, 1998], mentioned above, to allow proposers send their proposals directly to acceptors in some rounds, reducing in one step the latency of the protocol in synchronous periods of the system and in absence of failures. The price for these *fast* rounds, as opposed to the *classic* rounds in the original protocol, is the possibility of different proposals being accepted in the same round; we call this situation a *collision*. Collisions have two bad effects: first, they may force the execution of a new round, with all the overhead of starting it, even in the absence of failures; second, the simple possibility of collisions forces the leaders to synchronize with more acceptors in starting each round and, therefore, fast rounds are less resilient. An execution of Fast Paxos with only fast rounds is equivalent to an execution of the R*-Consensus protocol [Camargos et al., 2006a].

Multicoordinated Rounds

We have introduced a *multicoordinated* type of rounds that minimize the dependence on the coordinator and the availability problems resulting of this dependence [Camargos et al., 2007b, Camargos et al., 2008a]. Multicoordinated rounds have multiple coordinators, to which proposers send their proposals in parallel. As in the original Paxos protocol, coordinators forward the proposals to acceptors; acceptors, however, only take into consideration proposals forwarded by a quorum of the round coordinators. By requiring these coordinator quorums to intersect, the protocol ensures that no two different values will be considered on the same round and, hence, no collision happens on the acceptors. Collisions, however, may occur at the coordinators if different proposals are sent in parallel. These collisions are inherently less expensive than collisions on the acceptors in that they can be handled without any stable storage access. We further discuss this aspect of multicoordinated rounds in Section 3.4.4.

As with fast rounds and R*-Consensus, multicoordinated rounds and the B*-Consensus protocol are somewhat equivalent. Our multicoordinated protocol, however, extends Fast Paxos to have all three round types and allow coordinators to start rounds of either type at any point during the execution. An implementation of the protocol can harness this property to adapt to changes in the execution environment at runtime. The multicoordinated consensus protocol is the second contribution of this chapter.

2.2 WAB-Based Consensus

In this section we present the WAB-based consensus algorithms B*-Consensus and R*-Consensus [Camargos et al., 2006a]. These protocols execute as a sequence of rounds, identified by natural numbers. In each round, proposers can make any number of proposals. Acceptors can *accept* only a single proposal per round, and the goal of each round is to have a value accepted by a quorum of acceptors, in which case we say that the value has been *chosen*. The algorithms ensure that if a value is chosen at some round, then no other round will chose a different value. Hence, a learner can safely *learn* a value that has been chosen. To ensure that no two different values are chosen in any round, the algorithms require that quorums intersect. This requirement is formalized by the following assumption.

Assumption 1 (Quorum Requirement) *If Q and R are quorums, then $Q \cap R \neq \emptyset$.*

In fact, any general algorithm for asynchronous consensus must satisfy a similar requirement, as shown by the Accepting Lemma in [Lamport, 2006b]. A simple way

to satisfy this requirement is define quorums as any majority of the acceptors. We use this approach in this section for its simplicity.

B*-Consensus explores Assumption 1 to terminate in three communication steps in good runs. By making the following stronger assumption, R*-Consensus can terminate in just two steps. The assumption is easily satisfied by defining quorums with more than two thirds of the acceptors.

Assumption 2 (Simple Fast Quorum Requirement) *If Q, R, and S are quorums, then $Q \cap R \cap S \neq \emptyset$.*

We will explain how Assumption 2 is used once we have presented the algorithm. In fact, a relaxed version of it would be enough, but that would unnecessarily complicate the algorithm. We relax the assumption later, when reviewing the Fast Paxos protocol in Section 2.3.2, from which point on all algorithms will assume the relaxed version.

Before presenting B*-Consensus and R*-Consensus, respectively in Sections 2.2.2 and 2.2.3, we describe Weak Ordering Oracles and, more specifically, the WAB abstraction used in both algorithms.

2.2.1 Weak Atomic Broadcast

Weak ordering oracles [Pedone et al., 2002b] provide best-effort message ordering guarantees, that is, they try to deliver messages in some total order to all processes but cannot be trusted to always perform this task. A WAB, or Weak Atomic Broadcast, is a weak ordering oracle defined by the primitives w-broadcast(k, m) and w-deliver(k, m), where $k \in \mathbb{N}$ defines a w-broadcast instance and m is a message. The invocation of "w-broadcast(k, m) to S" broadcasts message m in instance k to the agents in S; w-deliver(k, m) w-delivers a message m w-broadcast in instance k. WAB satisfies the following property:

WAB If agents w-broadcast in an infinite number of instances to the agents in some set S, then for every instance k there is an instance $k' \geq k$ in which

- **Fairness** every nonfaulty agent in S w-delivers a message, and
- **Spontaneous Order** the first message w-delivered in instance k' is the same for every agent that w-delivers a message in k'.

Consider, for example, the use of WAB depicted in Figure 2.1, where w-broadcast and w-deliver are noted as `wb` and `wd`, respectively. The two agents depicted by dashed lines w-broadcast messages to the other processes. No agent crashes and no message is lost. In instance 1, message $m1$ is the first w-delivered by one of the three

receivers; the spontaneous order property is clearly not satisfied in this instance. In instance 2, however, the first message w-delivered by all receivers is the same, $m4$, satisfying spontaneous ordering. Last, in instance 3, even though just one message is w-broadcast and spontaneous ordering is attained, not all nonfaulty processes deliver the message. As long as this bad scenario does not happen infinitely often in the subsequent instances, the WAB properties may still be satisfied.

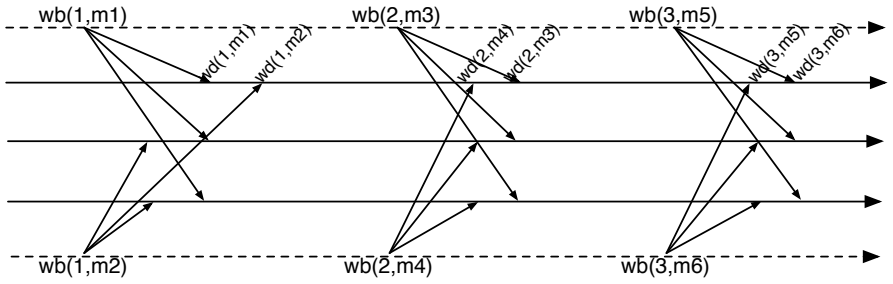

Figure 2.1: Spontaneous Order in WAB.

From a practical perspective, the behavior of IP-multicast in some local-area networks (e.g., Ethernet), under reasonable loads, matches the WAB specification. In such environments, IP-multicast ensures that most broadcast messages are delivered in the same order by all addressed nodes [Pedone et al., 2002b].

2.2.2 B*-Consensus

B*-Consensus (See Algorithm 2) has seven actions out of which the first three are performed by all agents, the fourth is performed only by proposers, fifth and sixth only by acceptors, and the last one by learners. Agents are divided in three sets, according to their roles: *acceptors*, *proposers*, and *learners*, and checking whether an agent may execute an action or not is performed by checking its inclusion into these sets. Remember that proposers are assumed to be also learners.

The algorithm is defined in terms of four variables: rnd, $prop$, $preAcc$, and acc. Even though some variables are particular to specific roles, to simplify the presentation we let all agents have all variables. Variables not associated to an agent will remain in its initial state throughout the execution of the protocol. We explain the function of each variable as needed while explaining the actions.

Bootstrap and Recovery

Action *Start*(a) executes when a is started for the first time. It sets the data structures to their initial values. The second action, *Recovery*(a) executes upon recovery,

i.e., when the agent is recovering from a crash. Initialization and recovery are easily discernible by using stable storage. Agents write to and read from stable storage through *Log* and *Retrieve* functions. The other actions in the algorithm run in response to the arrivals of messages. However, they only run after the agent has been initialized or recovered.

For every agent a, variable $rnd[a]$ has the highest-numbered round in which a has taken part; initially, $rnd[a]$ equals zero. If a is a proposer, $prop[a]$ has the value a has proposed in round $rnd[a]$; if a has not proposed any value, then $prop[a]$ has its initial value, $none$.

Algorithm 2 B*-Consensus

```
1:  Common Actions:
2:  Start(a) ≜                                   ◁ Run at boot, but not at recovery.
3:    actions:
4:      rnd[a] ← 0
5:      prop[a] ← none
6:      preAcc[a] ← none
7:      acc[a] ← none
8:  Recover(a) ≜                                 ◁ Run at recovery, but not at boot.
9:    actions:
10:     Retrieve(rnd[a], prop[a], preAcc[a], acc[a])
11: Skip(a) ≜
12:   pre-conditions:
13:     received ⟨_, q, rnd, ..., prop⟩ or w-delivered ⟨"first", q, rnd, prop⟩
14:     rnd[a] ≠ rnd
15:   actions:
16:     IF rnd[a] > rnd THEN
17:       send ⟨"skip", a, rnd[a], prop[a]⟩ to q
18:     IF rnd[a] < rnd THEN
19:       rnd[a] ← rnd
20:       prop[a] ← prop
21:       preAcc[a] ← none
22:       acc[a] ← none
23: Proposer Actions:
24: Propose(a, v) ≜
25:   pre-conditions:
26:     a ∈ proposers
27:     prop[a] = none
28:   actions:
29:     prop[a] ← v
30:     w-broadcast ⟨"proposal", a, rnd[a], prop[a]⟩ to acceptors
```

Skipping rounds

Rounds are started in sequence; only when a learner sees the end of a round, a new one is started. If an agent has been down for some time or is too slow, it might be delayed in the round progression. When an agent a in round $rnd[a]$ receives a message sent in a round $rnd > rnd[a]$, it immediately jumps to *rnd* skipping rounds $rnd[a]+1 \ldots rnd-1$, rapidly catching up with the more advanced agents. However, not every value can be proposed in every round, an agent skipping rounds must pick a value that is proposable in this new round. After a round in which a value is decided, for example, only this value can be proposed. Only agents that finished round $rnd-1$ initially know which values can be proposed in round *rnd*. Hence, processes skipping round $rnd-1$ to *rnd* must learn from the processes already in r about which values are valid proposals in r. In the algorithm, each message carries a proposal valid in the round in which it was sent, so that other agents can use it to skip round (this is the last field of each message).

The actual round skipping is performed by action *Skip* in the algorithm. This action runs before the other actions have also been triggered by the receipt of messages. This way, the process is able to jump to a higher round and then use the message to proceed in the round. Although not specified to simplify the algorithm, this action runs only once for every message received. The algorithms in [Aguilera et al., 1998] can also skip rounds, but the procedure is considerably more complicated than the one we present.

Proposing a Value

Proposers propose a value by executing the fourth action. Due to message losses and process crashes, a consensus instance may not terminate in the first attempt, and may have to be retried. At any time, proposers can retry a consensus instance if they believe that the previous attempt has failed; consistency is kept even if the previous attempt is still running. Since we assume that proposers are also learners, they are able to learn that a round of the algorithm has terminated. So, if a proposer does not learn the decision of the consensus it has initiated after some time, it re-starts its execution by proposing in its current round.

Accepting a Proposal

Acceptors in B*-Consensus accept proposed values in two steps. First, the value is pre-accepted and then exchanged and compared with pre-accepted values of other acceptors. Only then a value may be accepted. Actions *PreAccept* and *Accept* show these two steps. Variables *preAcc* and *acc* store the acceptors' currently pre-accepted and accepted values, respectively.

Algorithm 2 B*-Consensus(continued)

Acceptor Actions:
PreAccept(a) $\triangleq$
  pre-conditions:
    $a \in$ *acceptors*
    w-delivered $\langle$"`proposal`", _, $rnd[a]$, *prop*$\rangle$
    preAcc$[a] =$ *none*
  actions:
    preAcc$[a] \leftarrow$ *prop*
    prop$[a] \leftarrow$ *prop*
    Log(*preAcc*$[a]$,*rnd*$[a]$,*prop*$[a]$)
    send $\langle$"`pre-accepted`", $a, rnd[a], preAcc[a], prop\rangle$ to *acceptors*
Accept(a) $\triangleq$
  pre-conditions:
    $a \in$ *acceptors*
    $\forall q \in Q, |Q| = \lceil (n+1)/2 \rceil$, a received $\langle$"`pre-accepted`", $q, rnd[a], preAcc, prop\rangle$
  actions:
    LET $M = \{\langle$"`pre-accepted`", $q, rnd[a], preAcc, prop\rangle : q \in Q\}$
    IN
      IF $\forall m \in M : m = \langle$"`pre-accepted`", _, $rnd[a]$, $preAcc$, _$\rangle$
       THEN $acc[a] \leftarrow preAcc$ ◁ *Accepted preAcc.*
       ELSE $acc[a] \leftarrow \top$ ◁ *Did not accept anything.*
      $prop[a] \leftarrow prop$
      log $(acc[a], rnd[a], prop)$
      send $\langle$"`accepted`", $a, rnd[a], acc[a], prop[a]\rangle$ to *learners*
Learner Actions:
Learn(a) $\triangleq$
  pre-conditions:
    $a \in$ *learners*
    $\forall q \in Q, |Q| = \lceil (n+1)/2 \rceil$, a received $\langle$"`accepted`", $q, rnd[a], acc$, _$\rangle$
  actions:
    LET $M = \{\langle$"`accepted`", $q, rnd[a], acc$, _$\rangle : q \in Q\}$
    IN
      IF $\forall m \in M : m = \langle$"`accepted`", $q, rnda[a], acc$, _$\rangle$ **and** $acc \neq \top$
       THEN decide acc
      IF $\exists \langle$"`accepted`", $q, rnd[a], acc$, _$\rangle \in M : acc \neq \top$
       THEN $prop[a] \leftarrow acc$
      $rnd[a] \leftarrow rnd[a] + 1$

Acceptors pre-accept values when they first w-deliver a "`proposal`" message in their current round. Remember that agents may update their current rounds just before w-delivering, through the *Skip* action. On pre-accepting a proposal *prop* received in a message $\langle$"`proposal`", $q, rnd, prop\rangle$, an acceptor a sets *preAcc*$[a]$ to

prop and logs it along with *rnd*[a] and some proposal valid for round *rnd*[a], as for example the proposal just received. These values are used to recover the process in case of a crash. In Section 3.4.5 we show how this stable storage access can be avoided.

After pre-accepting a value, acceptors exchange these values with one another in "`pre-accepted`" messages. The goal is to determine if enough processes have pre-accepted the same proposal. Each acceptor a collects $\lceil (n+1)/2 \rceil$ pre-acceptances, including its own, where n is the number of acceptors in the system. This is shown in action *Accept*. If all the $\lceil (n+1)/2 \rceil$ pre-accepted values are equal to the same value *preAcc*, then a accepts the value by setting *acc*[a] to *preAcc*. Otherwise, if the values differ, a sets *acc*[a] to $\top$ to indicate that no value has been accepted in the given round. The update of *acc*[a] is logged on stable storage as well as *rnd*[a] and a valid proposal, as done with pre-acceptances. Finally, acceptors send their accepted values or $\top$, if they did not accept any value, to all learners using "`accepted`" messages.

Learning the Decision

Once learners have received $\lceil (n+1)/2 \rceil$ "`accepted`" messages, they execute action *Learn*. The first step in the action is to check whether all received "`accepted`" messages carry the same accepted value *acc*. If that is the case, then *acc* has been chosen as the decision and should be learned. The decision is also used as the proposal for the next round. If no value has been decided, then learners look for any accepted value $v \neq \top$ to be used as a proposal in the next round. If none is found, any value is used. The learner then increments its round number so that its proposer counterpart starts the new round.

2.2.3 R*-Consensus

R*-Consensus (See Algorithm 3) has a structure very similar to B*-Consensus. The most notable difference is that acceptors do not pre-accept proposals and check against each other. Hence, there is one fewer action in the algorithms as well as one fewer variable, $preAcc$. In fact, except for the removal of $preAcc$, the first four actions are the same as in B*-Consensus. We therefore omit the explanation of these actions.

Accepting a Proposal

In the R*-Consensus algorithm, an acceptor a directly accepts the first proposal it w-delivers in its current round. As in B*-Consensus, the accepted value is logged

Algorithm 3 R*-Consensus

1: Common Actions:
2: *Start*(a) $\triangleq$ ◁ *Run at boot, but not at recovery.*
3: **actions:**
4: $rnd[a] \leftarrow 0$
5: $prop[a] \leftarrow none$
6: $acc[a] \leftarrow none$
7: *Recover*(a) $\triangleq$ ◁ *Run at recovery, but not at boot.*
8: **actions:**
9: *Retrieve*$(rnd[a], prop[a], acc[a])$
10: *Skip*(a) $\triangleq$
11: **pre-conditions:**
12: received $\langle \_, q, rnd, \ldots, prop \rangle$ **or** w-delivered $\langle$"`first`", $q, rnd, prop\rangle$
13: $rnd[a] \neq rnd$
14: **actions:**
15: IF $rnd[a] > rnd$ THEN
16: send $\langle$"`skip`", $a, rnd[a], prop[a]\rangle$ to q
17: IF $rnd[a] < rnd$ THEN
18: $rnd[a] \leftarrow rnd$
19: $prop[a] \leftarrow prop$
20: $acc[a] \leftarrow none$
21: Proposer Actions:
22: *Propose*(a, v) $\triangleq$
23: **pre-conditions:**
24: $a \in proposers$
25: $prop[a] = none$
26: **actions:**
27: $prop[a] \leftarrow v$
28: w-broadcast $\langle$"`proposal`", $a, rnd[a], prop[a]\rangle$ to *acceptors*

along with $rnd[a]$ and $prop[a]$, so that these values will not be forgotten in case of a crash. Acceptors then send the accepted values to all learners in "`accepted`" messages.

Learning the Decision

Instead of a simple majority of "`accepted`" messages as in B*-Consensus, learners in R*-Consensus gather $\lceil (2n+1)/3 \rceil$ messages for the same round as a precondition to action *Learn*. As in the other algorithm, a value is learned (or decided) if all received messages show the same value as accepted. If that is not the case, then a new round is started. If more than half of the "`accepted`" messages indicate the same value, then this value is chosen as the proposal for the next round and stored in the $prop$ variable. If no value satisfies such a condition, then any value can be used as proposal. In any case, $rnd[a]$ is incremented.

Algorithm 3 R*-Consensus (continued)

29: Acceptor Actions:
30: *Accept*(*a*) $\triangleq$
31: **pre-conditions:**
32: $a \in acceptors$
33: w-delivered $\langle$"`proposal`", $q, rnd[a], prop\rangle$
34: $acc[a] = none$
35: **actions:**
36: $acc[a] \leftarrow prop$
37: $prop[a] \leftarrow prop$
38: *Log*($acc[a], rnd[a], prop[a]$)
39: send $\langle$"`accepted`", $a, rnd[a], acc[a], prop\rangle$ to *learners*
40: Learner Actions:
41: *Learn*(*a*) $\triangleq$
42: **pre-conditions:**
43: $a \in learners$
44: $\forall q \in Q, |Q| = \lceil (2n+1)/3 \rceil$, a received $\langle$"`accepted`", $q, rnd[a], acc, prop\rangle$
45: **actions:**
46: LET $M = \{\langle$"`accepted`", $q, rnd[a], acc, prop\rangle : q \in Q\}$
47: IN
48: IF $\forall m \in M : m = \langle$"`accepted`", $\_, rnda[a], acc, \_\rangle$ THEN
49: decide acc
50: IF $\exists v_{maj}$: for $\lceil (n+1)/3 \rceil$ $m \in M : m = \langle$"`accepted`", $\_, rnd[a], v_{maj}, \_\rangle$ THEN
51: $prop[a] \leftarrow v_{maj}$
52: $rnd[a] \leftarrow rnd[a] + 1$

2.2.4 Correctness and Liveness

In this section we sketch the correctness and liveness proofs or R*-Consensus and B*-Consensus. B*-Consensus and R*-Consensus are simplified versions of the Multicoordinated Consensus, presented in Section 2.3, and of Multicoordinated Paxos, presented in Chapter 3. Hence, the formal proof of correctness and liveness for Multicoordinated Consensus and Multicoordinated Paxos, in the Appendix A, also serve as formal proofs for B*-Consensus and R*-Consensus.

From the algorithms, B*-Consensus and R*-Consensus ensure the Nontriviality criterium in the consensus specification since no value besides the proposals are introduced in the algorithms. Consistency is ensured in B*-Consensus for the following reason:

- Due to Assumption 1, only a single value may be accepted in action *Accept* of any round by all agents in some quorum.

- If some quorum of acceptors accept the same proposal v in the same round, then every learner that executes *Learn* sees at least one "`accepted`" message with v in B*-Consensus. Hence, every learner that increments its round number does so after setting v as the new proposal, and v becomes the only proposable value in the next rounds.

- If there is a single value v proposed in some round, then all acceptors that accept some value must accept v. Hence, by the end of the round, v will be the only decided by any learner that decides, and the only proposal valid for the next rounds.

For a similar reason, Consistency also holds for R*-Consensus. That is

- Due to Assumption 1, only a single value may be accepted in action *Accept* of any round by all agents in some quorum.

- If some quorum of acceptor accepts the same proposal v in the same round, then every learner that executes *Learn* sees at least $n/3 + 1$ in R*-Consensus, due to Assumption 2. Hence, every learner that increments its round number does so after setting v as the new proposal, and v becomes the only proposable value in the next rounds.

- If there is a single value v proposed in some round, then all acceptors that accept some value must accept v. Hence, by the end of the round, v will be the only decided by any learner that decides, and the only proposal valid for the next rounds.

The same reasoning implies the Stability property. That is, after deciding on some value, each learner can only decide again if for the same value.

By the definition of the Liveness property, no algorithm can ensure progress unless all acceptors in some quorum are nonfaulty. If this condition is satisfied and actions are fairly executed (i.e., they are eventually executed if they are frequently enabled.), then as soon as the WAB properties hold in some round r a value v will be chosen. Once v is chosen, any learner that finishes r choses v as its new proposal and any agents moving to a round $r' \geq r$ only considers v as a valid proposal. Hence, even if the WAB properties do not hold in rounds bigger than r, every learner that decides in r' decides for v.

2.3 Multi-Coordinated Consensus

The Paxos consensus protocol [Lamport, 1998, Lamport, 2001] is a well known and largely studied leader based consensus protocol. When starting each round of Paxos, the leader checks if any decision was or may still be decided in any previously started round. If this is the case, then the leader forwards this value to be accepted by the acceptors in the new round. This inductively ensures that new rounds are always in accordance with the decisions of previously started rounds. Fast Paxos [Lamport, 2006a] is an extension of Paxos in which the leader, after determining that no value could have been decided in previous rounds, delegates to the proposers the task of sending proposals directly to the acceptors. Hence, in Fast Paxos, the leader can decide to switch back and forth from a classic Paxos round to a Fast Paxos round. (Henceforth, we refer to Paxos and its rounds as Classic Paxos and classic rounds.)

In the same way that Fast Paxos extends Classic Paxos with fast rounds, which are roughly equivalent to R*-Consensus rounds, we can extend Fast Paxos with B*-Consensus-like rounds. In doing so, we create rounds in which the task of the leader is shared by multiple coordinators, which never write on disk and therefore can be easily replaced by another coordinator. Moreover, these new rounds, which we call *multicoordinated*, do not require larger acceptor quorums as Fast Paxos does. Hence, we claim that our extended protocol provides greater availability than Fast Paxos.

The best way to explain our protocol is by starting with Classic Paxos and adding support to fast rounds, obtaining Fast Paxos, and then multicoordinated rounds. Hence, we start by explaining Classic Paxos.

2.3.1 Classic Paxos

The rounds of Paxos and Fast Paxos are identified by round numbers (sometimes also called a *ballot numbers* [Lamport, 1998]) which are totally ordered by a relation $<$

and in an infinite number. Although there is a total order among round numbers, the execution of rounds need not follow this order, and actions referring to different rounds may even interleave. This is in opposition to our WAB-based consensus protocols, in which a round i is always followed by round $i+1$ (although delayed agents may skip some rounds). For now, we assume that round numbers correspond to the set of natural numbers. In Section 3.4.5 we discuss another type of round numbers and their interesting properties in Section 3.4.5.

We say that a value is *chosen* in a round r if a quorum of acceptors has accepted the value in round r. Because the protocol assumes that Quorum Requirement (Assumption 1, on page 13) is satisfied, it is ensured that only a single value can be chosen in any round. In Classic Paxos, a round is divided into two phases: the first phase serves to identify previously chosen values and the second phase tries to get some value chosen in the current round. Each phase involves two actions: *Phase1a*, *Phase1b*, *Phase2a*, and *Phase2b*. Actions *Phase1a* and *Phase2a* initiate each phase, while the actions *Phase1b* and *Phase2b* may be seen as replies to the previous actions.

To orchestrate round executions, Paxos assumes a set of *coordinator* processes, besides proposers, acceptors, and learners. Every round has a single coordinator, responsible for starting each phase of the round, by executing actions *Phase1a* and *Phase1b*.[1]

Two other actions complete the algorithm. The first, *Propose*, is executed by proposers to propose a value. The second, *Learn*, is executed by learners to learn the decision of a consensus instance.

Figure 2.2 presents the dependencies among actions in each round. It shows that a learner l can only learn the decision of an instance after the execution of the two phases of some round i. Besides, a learner l can only learn some value if some proposer p has proposed it. Observe that the *Propose* action must happen before the second phase, but not necessarily before the first phase. The importance of this property will be explained later.

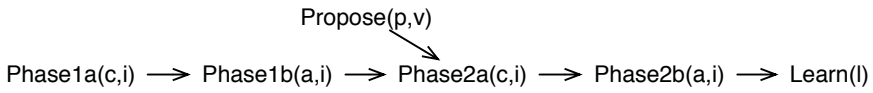

Figure 2.2: Dependencies among actions of proposer p, coordinator c, acceptor a and learner l, for some round i. $\rightarrow$ is the regular happens-before relation.

We now present an abridged version of the Classic Paxos specification along with a description of each phase of the protocol. Actions are defined in terms of the set

[1] In the original protocol, coordinators were drawn from the set of acceptors, not a different set of agents.

of variables listed below. The function of each variable becomes clear from the explanation of the actions. A coordinator c keeps the following variable:

crnd[c] The current round of c, which is initially 0.

An acceptor a keeps three variables:

rnd[a] The current round of a, that is, the highest-numbered round a has heard of; initially 0.

vrnd[a] The round at which a has accepted the latest value, initially 0.

vval[a] The value a has accepted at *vrnd*[a], initially $none$.

Each learner l keeps only the value it has learned so far.

learned[l] The value learned by l, initially $none$.

Proposing a Value

A proposer agent a proposes a value v by executing action *Propose*(a, v). The action consists simply of sending a $\langle$"`propose`", $v\rangle$ message to all coordinators.

Phase One

The coordinator c of round i executes action *Phase1a*(c, i) to start the phase one of i. The action consists of c sending a message $\langle$"`1a`", $c, i\rangle$ to each acceptor a asking a to take part in round i.

After receipt of message $\langle$"`1a`", $c, i\rangle$, acceptor a executes action *Phase1b*(a, i) to join round i. a executes the action only if i is greater than any other round a has ever heard of, where a has heard of j if it already executed actions *Phase1b*(a,j) or *Phase2b*(a,j). In this case, a sends an $\langle$"`1b`", $i, a,$ *vrnd*[a], *vval*[a]$\rangle$ message to c, where *vrnd*[a] is the highest-numbered round in which a has accepted a value (or the initial state, if no value has been accepted by a) and *vval*[a] is the value it accepted in *vrnd*[a] (or an invalid value $none$, if no value has been accepted yet). The pre-condition of this action ensures that after it is executed for round i, acceptor a will not execute it for a round j such that $j < i$. As we show in action *Phase2b*, this action also prevents a from accepting a value for a round j lower than i. This is a guarantee to c that a will not change its mind about the latest value accepted in a round number lower than i.

Algorithm 4 Classic Paxos

1: Proposer Actions:
2: *Propose*$(a, v) \triangleq$
3: **pre-conditions:**
4: $a \in$ *proposers*
5: **actions:**
6: send ⟨"`propose`", v⟩ to *coordinators*
7: Phase One:
8: *Phase1a*$(a, i) \triangleq$
9: **pre-conditions:**
10: a is the coordinator of round i
11: $crnd[a] < i$
12: **actions:**
13: send ⟨"`1a`", a, i⟩ to *acceptors*
14: *Phase1b*$(a, i) \triangleq$
15: **pre-conditions:**
16: $a \in$ *acceptors*
17: $rnd[a] < i$
18: received message ⟨"`1a`", c, i⟩ from coordinator c
19: **actions:**
20: $rnd[a] \leftarrow i$
21: send ⟨"`1b`", $a, i, vrnd[a], vval[a]$⟩ to c

Phase Two

The second phase starts once the coordinator c receives ⟨"`1b`", $a, i, vrnd, vval$⟩ messages for the same round i from all acceptors a in a quorum Q and executes action *Phase2a*(c, i). In the action, c sends message ⟨"`2a`", c, i, val⟩ to the acceptors, where *val* is c's selected proposal defined as follows. If no "`1b`" message informed of a previously accepted value, then c is free to select *val* among the proposals received directly from proposers. Otherwise, c must pick a value that has been or might be chosen in a previous round to make sure that no two different values will end up being chosen. This procedure ensures that if a value is chosen in round j, then no acceptor will ever accept a value other than v in a round bigger than j. Since all coordinators have done the same for previously started rounds, c must consider only the "`1b`" messages with the highest value of *vrnd*. Moreover, since Paxos ensures that no two acceptors can accept different values at the same round, all such messages are guaranteed to have the same value *vval*, which c picks up. Hereinafter we use *pick a value* when referring to the proposal selection performed by coordinator c during a *Phase2a* action, and say that a value is *pickable* in round i if no other value was or can still be chosen at any round $j < i$. In Algorithm 4, the picking of a value is executed by function *PickValue*(c, Q, i).

The second part of phase two of round i is executed by the acceptors in reply to the $\langle$"2a", $c, i, val\rangle$ message received from c. An acceptor a accepts the proposal *val* from c if it has not heard of a round j greater than i. The acceptor a then sends a message $\langle$"2b", $a, i, val\rangle$ to all learners. As we can notice, the fact that an acceptor only accepts values at round i sent by the coordinator of i in a phase "2a" message (which is the same sent to all acceptors) ensures that no two acceptors accept different values at the same round.

Learning a Value

If a learner l receives $\langle$"2b", $a, i, val\rangle$ from each acceptor a in a quorum, then it knows that *val* has been chosen and can be safely learnt. That is, *val* has been accepted by a quorum of acceptors and any coordinator that starts a round bigger than i will pick *val* as its proposal. l learns *val* by storing it on its *learned*[l] variable.

Ensuring Liveness

As described in the definition of *Phase1b*, executing that action for a given round prevents the same acceptor from executing action *Phase2b* for any smaller round (See Figure 2.2.). If different coordinators keep starting new rounds, it may happen that acceptors continuously execute *Phase1b* for the new rounds before executing *Phase2b* for the smaller rounds, and no value is ever chosen. To ensure liveness, a single coordinator must be entitled to start new rounds; we call such coordinator the *leader*. When there is just one agent in the system that believes itself to be the leader, then it will be able to start a round that is high enough to overcome all previously started rounds and make it succeed. However, having a single leader is just a liveness condition; safety is never violated no matter how many coordinators incorrectly believe themselves to be the leader. (See the complete description of Paxos for a formal proof of its liveness guarantees [Lamport, 1998].)

Skipping Phase One

Since a coordinator sends the value to be accepted only at the beginning of phase 2, the first phase of the algorithm can be executed before receiving any proposal. On a real application, probably many consensus instances will be needed, and the leader can execute phase 1 *a priori* for all consensus instances. Thus, the amortized latency for solving each instance becomes only three messages steps if there are no failures and no other coordinator interferes by starting a higher-numbered round: one step for the proposal to reach the leader and two more for the second phase of the leader's current round.

Algorithm 4 Classic Paxos (Continued)

Phase Two:
Phase2a$(a, i) \triangleq$
  pre-conditions:
    a is the coordinator of round i
    $crnd[a] \leq i$
    $\exists Q : Q$ is a quorum **and** $\forall ac \in Q$, a received $\langle$"1b", $ac, i, rnd, val\rangle$
  actions:
    $crnd[a] \leftarrow i$
    $cval[a] \leftarrow PickValue(c, Q, i)$
    send $\langle$"2a", $a, crnd[a], PickValue(c, Q)\rangle$ to *acceptors*
PickValue$(c, Q, i) \triangleq$
  LET $v \triangleq$ CHOOSE $val : c$ received $\langle$"1b", $ac, i, k, val\rangle$ from some $ac \in Q$:
    $\forall ac' \in Q, m' = \langle$"1b", $ac', i, k', \_\rangle$ c received from ac': $k' \leq k$
  IN IF $v \neq none$ THEN val ELSE CHOOSE v: c received $\langle$"proposal", $v\rangle$ from some proposer
Phase2b$(a, i) \triangleq$
  pre-conditions:
    $a \in acceptors$
    $rnd[a] \leq i$
    $vnd[a] < i$
    received message $\langle$"2a", $c, i, val\rangle$ from coordinator c
  actions:
    $rnd[a] \leftarrow i$
    $vrnd[a] \leftarrow i$
    $vval[a] \leftarrow val$
    send $\langle$"2b", $a, i, val\rangle$ to *learners*
Learn$(a) \triangleq$
  pre-conditions:
    $a \in learners$
    $\exists Q : Q$ is a quorum **and** $\forall ac \in Q$, a received $\langle$"2b", $ac, i, val\rangle$
  actions:
    $learned[a] \leftarrow val$

2.3.2 Fast Paxos

Fast Paxos [Lamport, 2006a] is an extension of the classic algorithm in which acceptors may accept proposals directly from proposers in some rounds, reducing the latency of reaching a decision in one communication step in good scenarios. There are two main differences between Fast and Classic Paxos that enact this shortcut. First, in Fast Paxos, each round has its own set of quorums; we call a quorum for round i an i-quorum. The reason for this is made clear later in this section (see Assumption 3, on page 31). Second, Fast Paxos has two sorts of round: *classic* and *fast*. Classic rounds have the same structure as rounds in Classic Paxos. Fast rounds

share the first phase with classic rounds, but their second phase differs slightly as we now explain.

Picking a Value in Fast Rounds

In a fast round i, after receiving the "1b" messages from an i-quorum, if the coordinator is free to pick any value, it can delegate this task to acceptors; it does so by picking as placeholder the special value *Any* and sending it in a "2a" message to the acceptors.

After receiving the "2a" message with the value *Any*, acceptor a waits for a message $\langle$"propose", $v\rangle$, in case it has not received one yet, and behaves as if it had received a message $\langle$"2a", $c, i, v\rangle$ from the coordinator c of round i. Hence, for Fast Paxos to work properly, proposers should send their "propose" messages to both coordinators and acceptors.

In Fast Paxos, acceptors are still bound to accept just one value per round but, differently from Classic Paxos, different acceptors can accept different values in the same fast round. Hence, to ensure the property that if a value v is chosen at round i, then no acceptor will ever accept a value different from v at any round j such that $j > i$, we must revisit the rule used by the coordinator of round i to find pickable values after receiving the "1b" messages from an i-quorum. That is, we must redefine the *PickValue* function.

Ensuring Safety with Fast Rounds

To understand how a coordinator picks a value in a fast round i, consider a run of the algorithm in which the coordinator of i has just received the "1b" messages for i from an i-quorum Q. There are three cases to analyze. First, if none of the received messages has a valid proposal in the *val* field, then no value has been or might be chosen at lower-numbered rounds, since there is no quorum that could have chosen such a value that does not intersect with Q. Therefore, any proposed value is pickable and it is up to c to decide whether to pick one or to let proposers propose directly.

If the first case does not apply, then let k be the greatest value for *rnd* received amongst the phase "1b" messages. If all messages in which *rnd* = *k* report the same value v as *val*, it might be the case that v was or will be chosen at a round $j < k$ Because the coordinator of k was also aware of this fact, v is the only value that it could have sent to the acceptors. Moreover, since any k-quorum must intersect Q and acceptors a in Q are sure to have executed action *Phase1b*(a, i) for round i ($i > k$), no value different from v can be chosen at k. Therefore, the coordinator can safely pick v in i.

Now consider the third case, in which more than one value has been reported in the phase "1b" messages with *rnd* $= k$ (k still being the greatest value for *vrnd* reported in the "1b" messages). This implies that the coordinator of k could pick any value and, hence, no value was chosen or will be chosen at a round lower than k. As a result, the coordinator must only figure out which of the values has been or might yet be chosen in k. There are three subcases to consider with respect to the "1b" messages received with *rnd* $= k$:

1. There is no value v and k-quorum R such that, for every acceptor a in $Q \cap R$, a message $\langle$"1b", $a, i, k, v\rangle$ was received from a. This implies that no value v has been or might be chosen at k since no k-quorum has even partially agreed on a value v at k. In this case, any proposed value is pickable.

2. There is only one value v such that, for some k-quorum R, a message of the form $\langle$"1b", $a, i, k, v\rangle$ has been received from every acceptor a in $Q \cap R$. This means that only value v has been or might be chosen at k depending on what the other acceptors have accepted or might still accept. In this case, only v is pickable.

3. There are two different values v and w and two k-quorums R and S such that, for every acceptor a in $Q \cap R$, a message $\langle$"1b", $a, i, k, v\rangle$ was received from a, and, for every acceptor b in $Q \cap S$, a message $\langle$"1b", $b, i, k, w\rangle$ was received from b. This means that either one of the values has been chosen or might still be chosen depending on what the other acceptors in R and S accept at k. By the quorum requirement, R and S have a non-empty intersection, which prevents both values from being chosen, but if this intersection does not intersects Q, then i's coordinator cannot decide which value is pickable. This case is depicted by Figure 2.3,

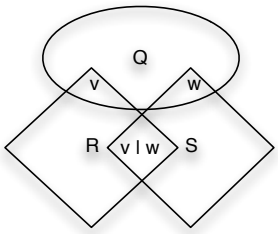

Figure 2.3: Q is an i-quorum, R and S are k-quorums, and v and w are the values accepted by acceptors in $Q \cap S$ and $Q \cap R$.

The way to avoid the third case is by strengthening the assumption made on the intersection of quorums and making sure that the intersection of any two quorums

R and S as shown in case 3 above also intersects Q. If this is ensured, the situation discussed in case 3 will never happen. The Simple Quorum Requirement in Assumption 2 would suffice here, but as we mentioned when defining it, it is stronger than really needed. The reason is that it forces any three quorums to intersect. It was reasonable to make this requirement for R*-Consensus, since it only has "fast" rounds. In Fast Paxos, which has classic and fast rounds, the following requirement is enough:

Assumption 3 (Fast Quorum Requirement) *For any rounds i and j:*

- *If Q is an i-quorum and S is a j-quorum, then $Q \cap S \neq \emptyset$.*
- *If Q is an i-quorum, R and S are j-quorums, and j is fast, then $Q \cap R \cap S \neq \emptyset$.*

In the general case, this stronger assumption requires *bigger quorums*. If every set of $n-E$ acceptors is a quorum for a fast round (*fast quorum*, for short) and every set of $n-F$ acceptors is a quorum for a classic round (*classic quorum*), where n is the total number of acceptors, then n must be greater than $2E+F$ as well as greater than $2F$. These constraints are achieved, for example, if every set of $\left\lfloor \frac{2n}{3} \right\rfloor + 1$ acceptors is a fast and classic quorum. If classic quorums are defined to be any majority of acceptors, fast quorums must be as big as $\left\lfloor \frac{3n}{4} \right\rfloor + 1$ acceptors. It has been shown that any asynchronous consensus protocol that allows a decision to be reached in two communication steps must satisfy similar quorum requirements [Lamport, 2006b] (Fast Learning Theorem).

Collisions and Collision Recovery

If two different values v and w are proposed in the same fast round i, it may be that none gets chosen, even in the absence of failures or suspicions, due to *collisions* [Lamport, 2006a]. A collision happens when the acceptors of a fast quorum accept different values. In pessimistic scenarios, collisions will prevent any value from being chosen in a fast round. If part of the acceptors accept v, part accept w, and the remaining never accept anything. There are three ways to break the tie and recover from a collision, and all reduce to executing a higher-numbered round. However, depending on how this new round is chosen, latency can be reduced considerably.

Let us assume a collision has happened at round i. The simplest approach has i's coordinator c to monitor the acceptors' phase "2b" messages and start a new round from the beginning after it learns the collision has happened. This approach is expensive as it takes four communication steps to recover.

Assume now that numbers are discrete, that is, for any round i it is possible to determine the round l such that there is no round j, $i < j < l$. For simplicity, we refer to the round subsequent to i simply as $i+1$. If, besides being the coordinator of i, c is also the coordinator of $i+1$, then c can exploit the fact that the information in "2b" messages sent for round i is essentially the same that would be sent in "1b" messages for round $i+1$ if c sends "1a" message for i. Hence, c may skip the first phase and proceed directly to the second phase of round $i+1$, incurring only two communication steps for collision recovery. This second approach is called *coordinated recovery*.

As an extension of coordinated recovery, if round i's "2b" messages are also sent to the set of acceptors, they can try to guess the coordinator's "2a" message for round $i+1$. They do that by interpreting these "2b" messages for round i as "1b" messages for round $i+1$ and applying the same algorithm that the coordinator uses to pick a value for a phase "2a" message; as seen before, this algorithm is guaranteed to return a pickable value. The acceptors then simulate the reception of a "2a" message with such a value. The advantage of this method is that it takes only one communication step to recover from collisions. However, because there is no guarantee that the acceptors will pick the same value, round $i+1$ is required to be fast, allowing acceptors to accept different values but risking in a new conflict. This third approach is called *uncoordinated recovery*. As explained in [Lamport, 2006a], some strategies can be used to try to make them accept the same value.

Observe that a run of Fast Paxos with only fast rounds that succeed each other by uncoordinated recovery is equivalent to running the R*-Consensus algorithm.

2.3.3 Multi-Coordinated Rounds and Coord-Quorums

While the expected latency of Classic Paxos is just three communication steps, in good runs, this number jumps when the coordinator must be replaced. Although at a first glance, the costs in replacing it consists only in the two communication steps needed to execute the first phase of another round of the protocol, this is not necessarily true. In large distributed systems, it is likely that just some of the nodes take part in the failure detection to minimize its cost. Moreover, efforts have been made to make failure detection as quiescent as possible [Aguilera et al., 2001]. These and other factors may impact on how fast a new leader can be elected and new decisions can be reached. Therefore, relying on a leader can significantly impact on the availability of a system implementing this approach. In the Chubby lock service, for example, nodes run Classic Paxos to agree on leases to other services; even though all nodes running the service are inside the same cluster, failures of the leader account for most of the system downtime [Burrows, 2006].

On the one hand, the fast rounds of Fast Paxos seem to solve the problem: they

do not depend on a leader during normal execution and even reduce the expected latency to learn a decision in good runs in one communication step. On the other hand, their stricter requirements on quorum sizes may limit the overall resilience of the system and hurt performance. Remember that with fast rounds the number of acceptors must be bigger than $2E + F$ (c.f. Assumption 3).

We propose a different approach to minimize the dependency on the leader: to extend classic rounds to have multiple coordinators, making them more reliable while maintaining their latency and the same acceptor quorum requirements.

This *multicoordination* approach stems from the realization that the tasks of pre-accepting and of accepting values in the B*-Consensus protocol, presented in Section 2.2, serve very different purposes. Pre-acceptances are meant to filter proposals only, but not choosing them; choosing happens at the acceptance stage. These two tasks are exactly what coordinators and acceptors do in Classic Paxos.

In the next sections we present a multicoordinated consensus protocol that extents the Fast Paxos protocol with multi-coordinated rounds. The resulting protocol can switch from classic, to fast, to multi-coordinated rounds to adapt to changes in the environment. Hence, our protocol can be seen as a synergy of B*-Consensus and Fast Paxos.

In Chapter 3 we present a similar multi-mode protocol that solves the more complex Generalized Consensus Problem, in which processes agree not on a single value but on a continuously growing sequence of them, and give an instantiation of the protocol to solve the Generic Broadcast problem [Pedone and Schiper, 2002].

Coordinator Quorums

In our protocol, rounds are still classified as in Fast Paxos, according to their minimum latency: classic rounds, if they require three messages steps to finish, or as fast rounds, if they may finish in two steps. However, in our extended protocol we relax the assumption that each round has a single coordinator. All coordinators of a round perform the same tasks. We divide the set of coordinators in coordinator quorums for rounds i, or i-coordquorums for short. We say that c is a coordinator of round i if it belongs to any i-coordquorum.

We change the behavior of the acceptors to accept a value v in a round i only if such value has been received from all coordinators in some i-coordquorum. Hence, the original Classic Paxos rounds, with a single coordinator, are simply multicoordinated rounds with a single one-element quorum of coordinators. Moreover, as discussed in Sections 2.3.1 and 2.3.2, in Classic and Fast Paxos, coordinators rely only on one value being accepted in any classic round when they pick values. This invariant must also be kept in our multicoordinated rounds in spite of coordinators being allowed to send different values to be accepted by acceptors. We do so by requiring all coordinator quorums of a round to intersect, as stated by the assumption

below.

Assumption 4 (Coord-quorum requirement) *For any two quorums of coordinators P and Q for the same classic round, $P \cap Q \neq \emptyset$.*

Fast rounds can have multiple coordinators and Assumption 4 would place no restriction upon them. However, since fast rounds are meant to avoid coordinators during normal execution, we see no reason to have something different from a single coordinator for them except in some very specific scenarios which we discuss in Section 3.4.4.

2.3.4 Algorithm

We now present the complete multicoordinated consensus algorithm. It is defined in terms of the same variables as Fast Paxos, which we repeat below for completeness.

A coordinator c keeps the following variable:

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

An acceptor a keeps three variables:

rnd[a] The current round of a, that is, the highest-numbered round a has heard of. Initially 0.

vrnd[a] The round at which a has accepted the latest value. Initially 0.

vval[a] The value a has accepted at *vrnd*[a]. Initially $none$.

Each learner l keeps only the value it has learned so far.

learned[l] The value learned by l. Initially $none$.

Proposing a Value

As in Fast Paxos, proposals are sent to both coordinators and acceptors to allow the execution of fast rounds. To minimize communication overhead, proposals may be initially sent only to a quorum of coordinators and a quorum of acceptors. Observe that this is only possible if multiple rounds share the same coordquorums and acceptor quorums, otherwise proposers would not know where to send proposals, since they are oblivious to rounds.

Phase One

In the multicoordinated consensus protocol, any coordinator c of a round i may start the round by sending a ⟨"1a", c, i⟩ message to the acceptors, asking them to take part in round i. This happens in action *Phase1a*(c, i).

In reply, an acceptor a executes *Phase1b*(a, i), as in the other Paxos protocols, if it has not heard of any round bigger than i, as stated in the pre-conditions of the action. In this case, a joins round i by setting *rnd*$[a]$ to i, and sends a message ⟨"1b", $a, i,$ *vrnd*$[a]$, *vval*$[a]$⟩ to all the coordinators of round i, where *vrnd*$[a]$ is the highest-numbered round in which a has accepted a value, and *vval*$[a]$ is the value it accepted in *vrnd*$[a]$. If no value has been accepted by a yet, then *vrnd*$[a]$ equals its initial state and *vval*$[a]$ equals an invalid proposal, $none$.

By the pre-condition that *rnd*$[a] < a$ and because the action sets *rnd*$[a]$ to i, once executed for i, this action can only execute again for a bigger round number. Moreover, it also forbids the execution of action *Phase2b* for a round smaller than i.

Algorithm 5 Multicoordinated Consensus

```
1:  Proposer Actions:
2:  Propose(a, v) ≜
3:    pre-conditions:
4:      a ∈ proposers
5:    actions:
6:      send ⟨"propose", v⟩ to coordinators ∪ acceptors
7:  Phase One:
8:  Phase1a(a, i) ≜
9:    pre-conditions:
10:     a ∈ ⋃ i-coordquorum
11:     crnd[a] < i
12:   actions:
13:     send ⟨"1a", a, i⟩ to acceptors
14: Phase1b(a, i) ≜
15:   pre-conditions:
16:     a ∈ acceptors
17:     rnd[a] < i
18:     received message ⟨"1a", c, i⟩ from coordinator c
19:   actions:
20:     rnd[a] ← i
21:     send ⟨"1b", a, i, vrnd[a], vval[a]⟩ to c
```

Algorithm 5 Multicoordinated Consensus (Continued)

22: Phase Two:
23: *Phase2a*$(c, i) \triangleq$
24: **pre-conditions:**
25: $c \in \bigcup i$-coordquorum
26: $crnd[c] \leq i$
27: $\exists Q : Q$ is a quorum **and** $\forall a \in Q$, c received $\langle$"1b", $a, i, rnd, val\rangle$
28: **actions:**
29: $crnd[c] \leftarrow i$
30: $cval[c] \leftarrow PickValue(c, Q, i)$
31: send $\langle$"2a", $c, crnd[c], PickValue(c, Q, i)\rangle$ to *acceptors*
32: *PickValue*(c, Q, i) is defined in Algorithm 6.
33: *Phase2b*$(a, i) \triangleq$
34: **pre-conditions:**
35: $a \in acceptors$
36: $rnd[a] \leq i$
37: $vrnd[a] < i$
38: $\exists C, val : C$ is an i-coordquorum **and** $\forall c \in C$ a received $\langle$"2a", $c, i, val\rangle$ from c
39: **actions:**
40: $rnd[a] \leftarrow i$
41: $vrnd[a] \leftarrow i$
42: IF $v \neq Any$ THEN $vval[a] \leftarrow val$
43: ELSE $vval[a] \leftarrow$ CHOOSE v: a received $\langle$"propose", $v\rangle$ form some proposer
44: send $\langle$"2b", $a, i, val\rangle$ to *learners*
45: *Learn*$(l) \triangleq$
46: **pre-conditions:**
47: $l \in learners$
48: $\exists Q : Q$ is a quorum **and** $\forall ac \in Q$, a received $\langle$"2b", $ac, i, val\rangle$
49: **actions:**
50: $learned[l] \leftarrow val$

Phase Two

To start the second phase of round i, coordinator c executes action *Phase2a*(c, i) once it has received $\langle$"1b", $a, i, vrnd, vval\rangle$ messages from every acceptor a in an i-quorum Q and only if it has not executed *Phase2a*(c, i) already. In this action, c sends a $\langle$"2a", $c, i, val\rangle$ message to the acceptors, where *val* is the value picked by c to be accepted by the acceptors. *val* is picked according to the "1b" messages c received from acceptors in Q in round i. This procedure is defined as function *PickValue*(c, Q, i) in Algorithm 6, and explained below:

- If none of the "1b" messages received from acceptors in Q has a value differ-

ent from the invalid proposal $none$, then they have not and will not accept any value for any round $j < i$. Since any j-quorum R must intersect Q, by Assumption 1, no such a quorum has or will succeed in choosing any value in j. Hence, c can pick any proposed value.

Otherwise, if some valid value was received by c, let k be the greatest round number $vrnd$ received amongst the "1b" messages received by c.

- If there exists a value v such that, for some k-quorum R, c received message $\langle$"1b", $a, i, k, v\rangle$ from every acceptor in $a \in R \cap Q$, then c picks v, since it may have been chosen by the acceptors of R.
- If no such a value exists, then c can pick any proposed value.

In the first and third cases, in which any value may be picked, if i is a fast round then c can pick the special value Any, which tells acceptors to accept proposals directly from proposers.

Algorithm 6 *PickValue*(Q) in Multicoordinated Consensus

```
1: PickValue(c, Q, i) ≜
2:   LET k ≜ CHOOSE r : c received ⟨"1b", a, i, r, _⟩ from some a ∈ Q and
                        ∀ a' ∈ Q, m' = ⟨"1b", a', i, r', _⟩ c received from a': r' ≤ r}
3:       rAcceptors(S, r) ≜ {a : a ∈ S and c received ⟨"1b", a, i, r, _⟩ from a
4:       rVals(S, r) ≜ {v : c received ⟨"1b", a, i, r, v⟩ from a ∈ S}
5:   IN
6:   IF ∃R, v ≠ none : R is an i-quorum and Q ∩ R ∈ rAcceptors(Q, k) and rVals(Q ∩ R, k) = {v}
7:    THEN v
8:    ELSE IF i is a fast round THEN Any
9:    ELSE CHOOSE v: received ⟨"proposal", v⟩ from some proposer
```

Once a round starts, it is on the interest of all agents to have it succeed. Hence, any coordinator of round i can start this round's second phase even if it did not start phase one and as long as it has received the "1b" messages required to execute action *Phase2a*. This approach allows a round to finish even if the coordinator that just started it becomes unavailable.

Acceptors accept values by executing action *Phase2b*(a, i), as in Fast Paxos. The only difference is that in our multicoordinated consensus this action is enabled if a has not heard of a round greater than i and has received a message $\langle$"2a", $c, i, val\rangle$ coming from all coordinators in some i-coordquorum with the same value *val*. If i is a fast round and *val* = *Any*, then a can accept any value sent in a "propose" message. If i is classic and, therefore, *val* ≠ *Any*, then a accepts *val*. After accepting value v, a sends the message $\langle$"2b", $i, v\rangle$ to all learners.

Learning a Value

Learning a value happens exactly like in Fast and Classic Paxos. That is, a learner l executes action *Learn*(l) when it receives a $\langle$"2b", $a, i, val\rangle$ message from each acceptor a in an i-quorum. The messages imply that *val* has been chosen and l can learn it by attributing it to variable *learned*[l].

2.3.5 Correctness and Liveness

From the explanation of our multicoordinated consensus algorithm, it should be clear that it satisfies the safety properties of consensus mainly because, as Fast and Classic Paxos, it ensures that rounds are consistent with previous decisions. It may seem that this is harder to ensure in a multicoordinated round, since any of its coordinators can start the round, but, in fact, the complexity is exactly the same. In a multicoordinated round, a proposal will be accepted by an acceptor only if a quorum of coordinators has forwarded such a value. Since all coordinator quorums intersect, only one value is accepted in such rounds. Coordinators of rounds executed afterwards only contact the acceptors, as in the other versions of the protocol, and are not influenced by the multiple coordinators of lower rounds.

Regarding liveness, since our protocol is an extension of Fast Paxos and, as such, can switch to single-coordinated rounds at any moment, it can ensure liveness under the same conditions of Classic Paxos. That is, if a quorum of acceptors, a coordinator, a proposer, and learner do not crash, then the learner will eventually learn the decision if there is a single coordinator that believes itself to be the leader and the proposer proposes some value. Nonetheless, it is interesting to understand what conditions allow a multicoordinated round to finish.

Roughly speaking, a multicoordinated round i will finish if its coordinator choses coordquorums such that at least one coordquorum is composed of non-crashed coordinators, no other round is started, and some proposal is made.

Because our consensus protocol is, in fact, a simplified instance of a more complex protocol which we present in Chapter 3, satisfying the liveness conditions of the latter will also satisfy the first's. Hence, we defer a formal discussion about liveness when using multicoordinated rounds to Section 3.4.7, where we formalize the liveness requirements of the more complex protocol.

2.4 Final Remarks and Related Work

The literature on consensus algorithms is very extensive and some of which has been discussed in the first section of this chapter. For these reasons, here we focus on the

work that is closely related to ours, specially with regards to using spontaneous ordering.

Spontaneous ordering (or weak ordering oracles) and consensus algorithms using it were introduced by Pedone *et al.* [Pedone et al., 2002b, Pedone et al., 2002a]. Their algorithms, however, assumed crash-stop failures and reliable channels. In this chapter we have presented improved versions of their algorithms that allow agents to recover after crashes and messages to be lost, namely B*-Consensus and R*-Consensus. B*-Consensus takes three communication steps to reach a decision and requires a majority of stable processes to ensure progress. R*-Consensus can decide in two communication steps, but requires more than two thirds of stable processes for progress. Both algorithms are optimal in terms of communication steps for the resilience they provide.

The problem of consensus in the crash-recovery model was previously studied in various other works [Lamport, 1998, Aguilera et al., 1998, Dolev et al., 1987, Hurfin et al., 1998, Oki and Liskov, 1988, Oliveira et al., 1997]. In all of these approaches, the asynchronous model was extended with unreliable failure detection or leader election oracles.

Fast Paxos [Lamport and Massa, 2004] is a hybrid algorithm with two modes, one that uses a leader election oracle (classic) and another that uses spontaneous ordering (fast), being the last one roughly equivalent to R*-Consensus. In the last parts of this chapter we have introduced an extension to Paxos with another mode that also uses spontaneous ordering, namely, the multicoordinated mode. In the same way that fast mode is related to R*-Consensus, the multicoordinated mode is related to B*-Consensus. That is, it is possible to simulate a run of B*-Consensus with the multicoordinated mode.

Aguilera *et al.* [Aguilera et al., 1998] have shown that, if the number of agents that never crash ("always-up processes") is bigger than the number of processes that eventually remain crashed or that crash and recover infinitely many times, then consensus is solvable without stable storage; without this assumption stable storage is required. As we do not bound the number of processes that are allowed to crash, our algorithms must use stable storage. Nonetheless, B-Consensus and R-Consensus use stable storage sparingly, not keeping all state in it. The algorithms of Dolev *et al.*[Dolev et al., 1997] and Oliveira *et al.*[Oliveira et al., 1997] do exactly the opposite: they keep all their variables in stable storage. Aguilera *et al.*'s algorithm that relies on stable storage [Aguilera et al., 1998] keeps just part of the state in memory and also writes on stable storage twice per round. Although the fast and multicoordinated modes may be seen as generalizations of R*-Consensus and B*-Consensus, respectively, the stable storage usage of our multicoordinated consensus protocol is wiser. First, coordinators never write on disk, since the number of coordinators is not limited in the protocol and, hence, a failed coordinator can completely forget its

state and recover as a new one. Second, acceptors seldom have to write on disk in the first phase, as we will show in the next chapter, in Section 3.4.5.

Lamport [Lamport, 2006b] has summarized several lower bounds on how fast a fault-tolerant consensus algorithm for asynchronous systems can be in terms of communication steps. Roughly, if any value proposed by two or more proposers can be decided within two communication steps, then less than a third of the acceptors can be unstable. To be able to decide in three communication steps less than half of the acceptors can be unstable. B*-Consensus, R*-Consensus, and consequently the fast and multicoordinated modes meet these bounds. The classic mode of Classic Paxos may decide in two communication steps and still tolerate permanent failures of any minority. However, only the value proposed by the round's coordinator can be decided in two steps; deciding on a value proposed by other processes requires at least one message step more. This becomes obvious when considering that the classic mode is, in fact, a particular instance of the multicoordinated mode.

Hurfin et al. [Hurfin et al., 1998] presented an algorithm that has the same message pattern as Paxos when phase one is skipped. Because it uses round robin to select the coordinators (the rotating coordinator paradigm) the decision may be delayed when a coordinator crashes and the successive one, according the round robin, is already crashed. Compared to our multicoordinated consensus, the protocol also lacks the ability to dynamically switch between execution modes. Aguilera *et al.*'s algorithm [Aguilera et al., 1998] is also $f < n/2$ resilient and reaches decision within three communication steps, from the point of view of the coordinator.

Finally, Aguilera and Toueg [Aguilera and Toueg, 1998] have proposed a hybrid binary algorithm (proposals are 0 or 1) that mixes failure detection and randomization to reach consensus. The algorithm works as a series of classic rounds but in which the coordinator selects a random bit when it is possible to pick any value. It is not clear how such an approach would improve over the selection of a proposed value in non-binary consensus.

In summary, in this chapter we presented practical crash recovery WAB-based algorithms and generalized one of them into a multicoordinated mode for a hybrid consensus protocol. The multicoordinated mode increases the availability of the protocol by replicating round coordinators. Although it replicates coordinators, the multicoordinated mode does not diminishes the maximum number of permanent failures in the protocol, differently from the fast mode; replicated coordinators do not need to be recovered, and are discarded if suspected to have failed.

While improving the availability of a single consensus instance may be a minor improvement, it is just a logical step to improving the availability of long living protocols for atomic and generic broadcast, or generalized consensus. In the next chapter we show how to apply the technique into protocols for such problems and discuss practical aspects of the multicoordinated mode.

Chapter 3

Multicoordinated Generalized Consensus and Generic Broadcast

3.1 One Problem to Rule Them All

Many applications can be implemented on top of total order (or atomic) broadcast [Hadzilacos and Toueg, 1993], which enforces agreement on an ever growing sequence of values. One approach to solving this problem is to use infinitely many consensus instances, one for each position in the sequence. Using a fast algorithm in each instance is prohibitive due to collisions, which may happen even during the stable periods of executions—*i.e.*, without failures or suspicions. In a state machine replication scenario, for example, a collision may happen if two commands are proposed concurrently to the same instance of consensus. In many systems, however, commands may commute and there is no need for totally ordering them since the final state is the same independently of the order in which they are applied. Consensus, as applied to state machine replication, is too strong to capture this notion and a collision may happen even if the two concurrent proposals are commutable.

Generic Broadcast [Pedone and Schiper, 1999, Pedone and Schiper, 2002] is a generalization of atomic broadcast in which agents agree not on a sequence, but on a partial order of commands. The partial order must be such that any pair of conflicting commands are ordered, where two commands are conflicting according to some meaningful conflict relation (*e.g.*, if they do not commute). A generic broadcast protocol may harness such a conflict relation to mitigate the effects of collisions and efficiently implement state machine replication. By defining the conflict relation such that any or no pair of commands conflict, generic broadcast becomes atomic broadcast or reliable broadcast, respectively.

An even more general problem, Generalized Consensus, is defined in terms of a data structure called *command structure*, or simply *c-struct*. The basic operation

performed on a c-struct is appending commands to it. Depending on the c-struct instantiated, generalized consensus solves a different agreement problem. Reliable broadcast, for example, is an instance of Generalized Consensus whose c-structs are simple sets. Limit the set's cardinality to one and the problem degenerates to consensus; add ordering to the set and the problem becomes generic or atomic broadcast.

Multicoordinated Paxos

We are aware of only one generalized consensus protocol in the literature, by Lamport [Lamport, 2004]. This protocol, namely Generalized Paxos, is based on Fast Paxos and inherits its two execution modes. As a result, Generalized Paxos suffers from the same availability limitations of Fast Paxos when compared to our multicoordinated consensus: it either relies on a leader or requires bigger acceptor quorums. Again as Fast Paxos, Generalized Paxos is amenable to multicoordinated execution or, equivalently, our multicoordinated consensus is amenable to "generalization". In the next section we present our multicoordinated version of Generalized Paxos, namely Multicoordinated Paxos.

Besides its generality, Multicoordinated Paxos has some interesting features from a practical point of view:

- it allows an infinite number of coordinators to be used in a run, what completely eliminates the need to recover these agents;
- it allows very efficient usage of stable storage, as coordinators do not write on disk (since they are not recovered), and acceptors only write once per round in the absence of failures, even if wrong failure suspicion happens; and,
- it allows load balancing at the coordinators and acceptors.

We discuss these and other topics in Sections 3.4.3–3.4.6.

Since Multicoordinated Paxos solves Generalized Consensus, with an appropriate c-struct it also solves Generic Broadcast. In Section 3.5, we introduce one simple but powerful appropriate c-struct. To the best of our knowledge, there are three extensions of the original generic broadcasts algorithm [Pedone and Schiper, 1999] in the literature [Pedone and Schiper, 2002, Aguilera et al., 2000, Zielinski, 2005] . We compare Multicoordinated Paxos with these works in Section 3.6.

3.2 Generalized Consensus

3.2.1 C-Structs

Before presenting Multicoordinated Paxos, we formally define c-structs and the Generalized Consensus problem. The definitions and notation used here are borrowed from Lamport's work [Lamport, 2004].

A c-struct set *CStruct* is defined by a tuple $\langle \mathit{Cmd}, \bot, \bullet \rangle$, where *Cmd* is the set of commands that compose the c-structs of *CStruct*, $\bot$ is a "null" c-struct, and $\bullet$ is an operator that appends a command from *Cmd* to a c-struct of *CStruct*, and by a set of five axioms listed later. More generally, a c-struct v is in *CStruct* if $v = \bot \bullet C$ and $C \in \mathit{Cmd}$, or if $v = w \bullet C$, $w \in \mathit{CStruct}$ and $C \in \mathit{Cmd}$. For example, one could create a c-struct set where c-structs are subsets of *Cmd*, $\bot$ is the empty set, and $v \bullet C$ simply adds element C to the current value of v. Another c-struct set could have c-structs as partially ordered sets, $\bot$ as the empty set, and $v \bullet C$ as an operation that extends partially ordered set v with command C by making C succeed (with respect to the partial order) any conflicting element of v, given an external conflicting relation over *Cmd*—a c-struct set that could capture the notion of commutable commands.

Before we present the five axioms of a c-struct set, some definitions are necessary. A finite sequence with elements C_i is represented by $\langle C_1, C_2, \ldots, C_m \rangle$. *Seq*(*S*) is defined to be the set of all (finite) sequences whose elements are in the set S (with possible repetitions of elements in the sequence). Moreover, we use the term c-seq when referring to a finite sequence of commands—that is, an element of *Seq*(*Cmd*). We can now extend the operator $\bullet$ to a c-struct v and a c-seq $\langle C_1, \ldots, C_m \rangle$ as follows:

$$v \bullet \langle C_1, \ldots, C_m \rangle = \begin{cases} v & \text{if } m = 0, \\ (v \bullet C_1) \bullet \langle C_2, \ldots, C_m \rangle & \text{otherwise} \end{cases}$$

We say that c-struct w *extends* c-struct v or v *is a prefix* of w, and note as $v \sqsubseteq w$, iff there exists a c-seq σ such that $w = v \bullet \sigma$. Given a set T of c-structs, we say that c-struct 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, and we represent it by $\sqcap T$. 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, and we represent it by $\sqcup T$. If $\sqsubseteq$ is a reflexive partial order on the set of c-structs and a glb or lub of T exists, then it is unique. For simplicity of notation, we use $v \sqcap w$ and $v \sqcup w$ to represent $\sqcap\{v, w\}$ and $\sqcup\{v, w\}$, respectively. Two c-structs v and w are defined to be *compatible* iff they have a common upper bound, and a set S of c-structs is compatible iff its elements are pairwise compatible.

We say that c-struct v is *constructible from* a set P of commands if $v = \bot \bullet \sigma$, for some c-seq σ containing all elements of P. Moreover, we say that v *contains*

command C if v is constructible from some set P of commands such that $C \in P$. We define *Str*(*P*) to be the set of all c-structs constructible from subsets of P for some set P of commands—that is, $\mathit{Str}(P) \triangleq \{\bot \bullet \sigma : \sigma \in \mathit{Seq}(P)\}$.

A c-struct set *CStruct* must satisfy the five axioms below.

CS0. $\forall C \in \mathit{Cmd}, w \in \mathit{CStruct} : w \bullet C \in \mathit{CStruct}$

CS1. $\mathit{CStruct} = \mathit{Str}(\mathit{Cmd})$

CS2. $\sqsubseteq$ is a reflexive partial order on *CStruct*.

CS3. For any set $P \subseteq \mathit{Cmd}$ and any c-structs u,v, and w in *Str*(*P*):

- $v \sqcap w$ exists and is in *Str*(*P*).
- If v and w are compatible, then $v \sqcup w$ exists and is in *Str*(*P*).
- If $\{u, v, w\}$ is compatible, then u and $v \sqcup w$ are compatible.

CS4. For any command $C \in \mathit{Cmd}$ and compatible c-structs v and w in *CStruct*, if v and w both contain C then $v \sqcap w$ contains C.

CS0-CS2 are basic requirements to satisfy the properties just described. CS0 says that by extending any c-struct in *CStruct* with any command in the respective *Cmd* set, the obtained result is a c-struct in *CStruct*; CS1 requires c-structs to be well formed; and CS2 forces any *CStruct* set to have a glb and a lub. CS3 and CS4 are necessary for Generalized Paxos and similar algorithms to ensure the safety and liveness properties of Generalized Consensus, described next. Roughly, CS3 says that the lower bound of any two c-structs only contains commands contained in the two c-structs; the same is valid for the upper bound, if it exists. CS4 states that the upper bound of two c-structs contains all their commands in common.

3.2.2 Problem Definition

We can now generalize the original definition of consensus to deal with a c-struct set instead of single absolute values. The problem is defined in terms of a c-struct set *CStruct* which, as shown in the previous section, is based on a null value $\bot$, a set *Cmd* of commands, and an operator $\bullet$. Proposers propose commands in *Cmd* and we let *learned*[l] be the c-struct learner l has learned (initially $\bot$). Generalized Consensus is defined by the following properties:

Nontriviality: For any learner l, *learned*[l] is always a c-struct constructible from some subset of the proposed commands.

Stability: For any learner l, if the value of *learned*[l] at any time is v, then $v \sqsubseteq$ *learned*[l] at all later times.

Consistency: The set {*learned*[l] : l is a learner} is always compatible.

Liveness: For any proposer p and learner l, if p, l, and a quorum Q of acceptors are nonfaulty and p proposes a command C, then *learned*[l] eventually contains C.

3.3 Lamport's Generalized Paxos

Generalized Paxos is an extension of Fast Paxos to solve Generalized Consensus. The algorithm has the advantage that, by the problem definition, a collision is not characterized if two acceptors accept different but compatible c-structs. In such a case, both acceptors can later extend their accepted c-structs so that they converge to the same one (since compatible c-structs have a common upper bound). C-struct sets like command histories with commutable commands, explained better in Section 3.5, might have very few incompatible c-structs, which reduces the chances of a collision to happen and favors the use of fast rounds.

Generalized Paxos relies on the Fast Quorum Requirement (Assumption 3, on page 31). It assumes that acceptors have initially accepted $\bot$ at round 0, lower than any other round. Since Multicoordinated Paxos extends Generalized Paxos, we just overview its actions here, and explain them in detail when discussing Multicoordinated Paxos. Generalized Paxos has the following actions.

Propose(p, C) Executed by proposer p to propose command $C \in$ *Cmd*. In the action, p sends a ⟨"`propose`", C⟩ message to all coordinators and acceptors. (The same as in Classic Paxos.)

Phase1a(c, i) Executed by *any* coordinator c of round i to start round i. In the action, c sends a message ⟨"`1a`", c, i⟩ to each acceptor a asking a to take part in round i. (The same as in Classic Paxos, except that values are now c-structs.)

Phase1b(a, i) Executed by acceptor a to join round i upon reception of message ⟨"`1a`", i⟩, if i is greater than any other round a has ever heard of. In this case, a sends a message ⟨"`1b`", $a, i,$ *vrnd*, *vval*⟩ to the coordinator of round i, where *vrnd* is the highest-numbered round in which a has accepted a c-struct and *vval* is the c-struct it accepted in *vrnd*. As before, this action also prevents a from executing the same action for a round smaller than or equal to i, and from accepting a c-struct for a round smaller than i. (The same as in Classic Paxos.)

Phase2Start(c, i) In this action, coordinator c of round i, after receiving a "1b" message for round i coming from each acceptor in an i-quorum Q, picks a c-struct and sends it to acceptors to be extended during the round. Picking a c-struct in Generalized Consensus is like picking a value in Fast-Paxos or in our multicoordinated consensus algorithm: if the c-struct was possibly chosen in some round $j < i$, the c-struct must be picked. As before, a c-struct is said to be chosen if it was accepted by all acceptors of some quorum of any round. The difference is on what accepted means. In the Classic Paxos, acceptors accept single values, but in Generalized Paxos, when an acceptor accepts a c-struct, it automatically accepts all of its prefixes. Hence, when picking a value, the coordinator must consider not only the complete c-structures accepted, but also their prefixes.

Suppose for example that c has received $\langle$"1b", $a_k, i, j, v_k\rangle$ messages from a quorum of three acceptors $a_k, 1 \leq k \leq 3$. Also, assume for simplicity that the c-struct used is a total order of commands, easily represented by a sequence, and that the messages had the following c-structs: $v_1 = \langle a, b, c, e, f\rangle, v_2 = \langle a, b, c, d\rangle$ and $v_3 = \langle a, b, c, e, d\rangle$. Observe that, since the three acceptors form a quorum, both the sequences $\langle a\rangle$, $\langle a, b\rangle$ and $\langle a, b, c\rangle$ were chosen and might have been learned. Hence, c is obliged to pick the three c-structs to use in the second phase of the algorithm. Hence, c picks the longest c-struct, whose the others are prefixes of; in fact, $\langle a, b, c\rangle$ is the longest prefix of all received c-structs.

Suppose now that there is another quorum in the system, formed by acceptors a_1, a_3, and a_4. The two quorums are represented graphically in Figure 3.1, below. If a_4 has accepted any sequence in round k that extends $\langle a, b, c, e\rangle$, then such a sequence was also chosen and should be picked by c. Since c never heard from a_4, it has no option but to behave safely and pick sequence $\langle a, b, c, e\rangle$. Since a_1 and a_3 disagree about which command should be the fifth in the sequence, $\langle a, b, c, e\rangle$ is the shortest sequence that extends all possibly chosen sequences.

Generalizing from the previous two paragraphs, once the coordinator receives the "1b" messages from a quorum of acceptors, it picks the c-struct that extends the largest prefix of the c-structs seen (*i.e.*, its glb), which was already chosen, and their smallest possibly chosen extensions (lub). This c-struct is then sent to the acceptors to ensure that if a c-struct v has been chosen at some round j and w is accepted by some acceptor at a higher-numbered round, then $v \sqsubseteq w$. As before, for coordinator c to gather the set of all possibly chosen c-structs in previous rounds, it suffices to look at the "1b" messages with the highest-numbered *vrnd* value. More formally, let k be such a round number.

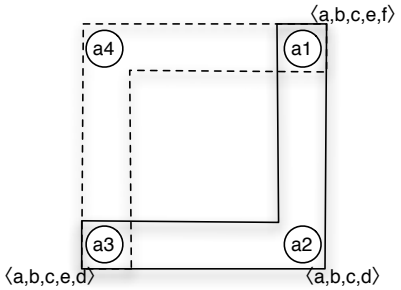

Figure 3.1: A system of four acceptors and their accepted values. The two marked sets of acceptors constitute quorums.

There are only two cases to consider.

First, if there is no k-quorum R such that, for every acceptor a in $R \cap Q$, c has received a "1b" message from a with *vrnd* $= k$, then c is assured that no c-struct has been or might be chosen at k. Moreover, the algorithm ensures that if a c-struct v has been chosen at a round $j < k$, then any value w accepted at k satisfies $v \sqsubseteq w$. Therefore, c can pick any c-struct received in one of the "1b" messages in which *vrnd* $= k$.

If the first case does not apply, then, for every k-quorum R such that c has received a "1b" message with *vrnd* $= k$ from every acceptor in $R \cap Q$, c calculates the glb of the c-structs *vval* received in such messages and adds it to a set Γ initially empty. After that, Γ will contain all c-structs that have been or might be chosen at lower-numbered rounds. The second condition of the Fast Quorum Requirement ensures that Γ is compatible and, therefore, has a least upper bound $\sqcup\Gamma$ that can be safely picked by c.

After picking a c-struct *val* based on the previous two cases, c sends a message $\langle$"2a", $c, i,$ *val*$\rangle$ to all acceptors.

Phase2bClassic(a, i) Upon receiving a $\langle$"2a", $c, i,$ *val*$\rangle$ message from the coordinator, acceptor a accepts the forwarded c-struct like in Fast Paxos. That is, if k is highest-numbered round a has heard of and $k < i$, then a accepts *val* and sends message $\langle$"2b", $i, val\rangle$ to every learner.

In Generalized Paxos, however, coordinators may send a "2a" message to acceptors with some c-struct, and later request the acceptors to accept an extension of such c-struct. This is done in action *Phase2aClassic*, explained below. To cope with this extension requests, the pre-conditions of action *Phase2bClassic* are extended as follows. Upon receiving a $\langle$"2a", $c, i,$ *val*$\rangle$ message from coordinator c, a accepts *val* iff: $k < i \vee (k = i \wedge v \sqsubseteq$ *val*$)$, where k is the

highest-numbered round a has heard of and v is the latest c-struct accepted by a.

Phase2aClassic(c, i) This action is executed by coordinator c of round i to request acceptors to accept an extension of a previously forwarded c-struct. Hence, it is executed by coordinator c only if it has already sent a phase "2a" message for round i to the acceptors. This action is performed only for a classic round i; for fast rounds, acceptors can extend their accepted c-structs themselves, as shown in action *Phase2bFast*, below.

Let $\langle$"2a", i, *val*$\rangle$ be the latest phase "2a" message c has sent for round i and let *newval* be *val* $\bullet\ \sigma$ for some c-seq σ of proposed values received in "propose" messages. In this action, c simply sends a message $\langle$"2a", i, *newval*$\rangle$ to all acceptors, requesting them to extend what they had previously accepted with σ.

Phase2bFast(a, i) By executing this action, acceptor a appends a command to its previously accepted c-struct. This action is enabled iff i is fast, i is the highest-numbered round a has heard of, a has already accepted a value in i, and a has received a $\langle$"propose", $C\rangle$ message. Let v be the latest value a has accepted in i; a accepts $v \bullet C$ and sends message $\langle$"2b", $i, v \bullet C\rangle$ to every learner.

Learn(l) Leaner l executes this action to extend its previously learned c-struct. The action is executed by l after it receives a phase "2b" message for some round i from each acceptor in an i-quorum. Let v be the glb of the values received in such messages; l sets *learned*$[l]$ to *learned*$[l] \sqcup v$.

In Generalized Paxos, a round i starts by the round coordinator executing action *Phase1a*(c, i). Acceptors then should execute action *Phase1b*(a, i), followed by the execution of *Phase2Start*(c, i) by c and *Phase2aClassic*(a, i) by the acceptors. After this point, the execution depends on whether i is fast or classic. If i is classic and proposers keep proposing new commands to the coordinator, then the coordinator continuously executes *Phase2aClassic*(c, i) with longer c-structs, followed by acceptors executing *Phase2bClassic*(a, i). If i is fast and proposers keep proposing, then acceptors execute *Phase2bFast*(a, i) to append the proposals to their accepted c-structs. In any case, learners continuously execute *Learn*(l) to learn the c-struct accepted and its extensions.

3.4 Multicoordinated Paxos

We now explain Multicoordinated Paxos, our multicoordinated generalized consensus algorithm. Round numbers are defined as in the multicoordinated consensus

protocol. The algorithm assumes a c-struct set *CStruct*, the Fast Quorum Requirement (Assumption 3) for quorums of acceptors, and the Coord-quorum Requirement (Assumption 4) for quorums of coordinators.

As in Generalized Paxos, we ensure that if a c-struct v is chosen at a round i, then any c-struct w that is accepted by any acceptor at some round $j > i$ extends v ($v \sqsubseteq w$). This is guaranteed by the coordinators of a round due to the rule used by the coordinators to pick a value based on the phase "1b" messages received from a quorum of acceptors (explained in Section 3.3, action *Phase2Start*(c, i)). In Multicoordinated Paxos, we embody the rule in function *ProvedSafe*$(Q, 1bMsg)$. Before explaining the function in Section 3.4.2, we present the Multicoordinated Paxos algorithm.

3.4.1 The Algorithm

Algorithm 7 shows the basic atomic actions that compose Multicoordinated Paxos, which always ensure safety. To ensure liveness as well, some extra pre-conditions must be imposed on the actions. The required changes in the algorithm are presented in Section 3.4.7.

Variables

The algorithm manipulates a set of six variables associated to the roles that agents play. The variables are initiated as if the first phase of the smaller round available, which we assume to be 0, has already been executed. Proposers have no special variable. The variables of a coordinator c are the following:

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

cval$[c]$ The latest c-struct c has sent in a phase "2a" message for round *crnd*$[c]$. Initially $\bot$.

An acceptor a keeps three variables:

rnd$[a]$ The current round of a, that is, the highest-numbered round a has heard of. Initially 0.

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

vval$[a]$ The c-struct a has accepted at *vrnd*$[a]$. Initially $\bot$.

Each learner l keeps only the c-struct it has learned so far.

Algorithm 7 Multicoordinated Paxos

1: Proposer Actions:
2: *Propose*$(p, C) \triangleq$
3: **pre-conditions:**
4: $a \in$ *proposers*
5: **actions:**
6: send $\langle$"propose", $C\rangle$ to *coordinators* $\cup$ *acceptors*
7: Phase One:
8: *Phase1a*$(c, i) \triangleq$
9: **pre-conditions:**
10: $c \in \bigcup i$-coordquorum
11: *crnd*$[c] < i$
12: **actions:**
13: send $\langle$"1a", $c, i\rangle$ to *acceptors*
14: *Phase1b*$(a, i) \triangleq$
15: **pre-conditions:**
16: $a \in$ *acceptors*
17: *rnd*$[a] < i$
18: received $\langle$"1a", $c, i\rangle$ from coordinator c
19: **actions:**
20: *rnd*$[a] \leftarrow i$
21: send $\langle$"1b", $a, i, vrnd[a], vval[a]\rangle$ to $\bigcup i$-coordquorum
22: Phase Two:
23: *Phase2Start*$(c, i) \triangleq$
24: **pre-conditions:**
25: $c \in \bigcup i$-coordquorum
26: *crnd*$[c] < i$
27: $\exists Q : Q$ is a quorum **and** $\forall a \in Q$, c received $\langle$"1b", $a, i, rnd, val\rangle$
28: **actions:**
29: *crnd*$[c] \leftarrow i$
30: *cval*$[c] \leftarrow$ *PickValue*(c, Q, i)
31: send $\langle$"2a", $c, crnd[c], cval[c]\rangle$ to *acceptors*
32: *PickValue*$(c, Q, i) \triangleq$
33: LET *1bMsg* $\triangleq f(x) = m : c$ received $m = \langle$"1b", $a, i, \_, \_, \_\rangle$, $a \in Q$
34: IN CHOOSE $v : v \in$ *ProvedSafe*$(Q,$ *1bMsg*$)$

learned$[l]$ The c-struct currently learned by l. Initially $\bot$.

As the other Paxos protocols, Multicoordinated Paxos defines actions for agents to propose, interact to chose c-structs, and learn the chosen c-structs, and the actions are executed in phases. The proposition of a command, however, is not associated to any round and can be executed at any time.

Algorithm 7 Multicoordinated Paxos (Continued)

35: *Phase2aClassic*$(c, i) \triangleq$
36: **pre-conditions:**
37: $c \in \bigcup i$-coordquorum
38: *crnd*$[c] = i$
39: i is a classic round
40: c received ⟨"propose", C⟩
41: **actions:**
42: *cval*$[c] \leftarrow$ *cval*$[c] \bullet C$
43: send ⟨"2a", c, $crnd[c]$, $cval[c]$⟩ to *acceptors*
44: *Phase2bClassic*$(a, i) \triangleq$
45: **pre-conditions:**
46: $a \in$ *acceptors*
47: *rnd*$[a] \leq i$
48: $\exists L : L$ is an i-coordquorum **and** $\forall c \in L$ a received ⟨"2a", $c, i, \_$⟩
49: *vrnd*$[a] < i$ **or** *vval*$[a]$ is compatible with $\sqcap\{v : a$ received ⟨"2a", c, i, v⟩, $c \in L\}$
50: **actions:**
51: LET *LVals* $= \{v$: a received ⟨"2a", c, i, v⟩, $c \in L\}$
52: IN
53: IF *vrnd*$[a] = i$
54: THEN *vval*$[a] \leftarrow$ *vval*$[a] \sqcup (\sqcap$*LVals*$)$
55: ELSE *vval*$[a] \leftarrow \sqcap$*L2aVals*
56: *vrnd*$[a] \leftarrow i$
57: *rnd*$[a] \leftarrow i$
58: send ⟨"2b", a, i, val⟩ to *learners*
59: *Phase2bFast*(a)
60: **pre-conditions:**
61: $a \in$ *acceptors*
62: *rnd*$[a]$ is a fast round
63: *rnd*$[a] =$ *vrnd*$[a]$, and
64: a has received a ⟨"propose", C⟩ message
65: **actions:**
66: *vval*$[a] \leftarrow$ *vval*$[a] \bullet C$
67: send ⟨"2b", $vrnd[a]$, $vval[a] \bullet C$⟩ to learners
68: Learning:
69: *Learn*$(l) \triangleq$
70: **pre-conditions:**
71: $l \in$ *learners*
72: $\exists Q : Q$ is a quorum **and** $\forall ac \in Q$, a received ⟨"2b", ac, i, val⟩
73: **actions:**
74: LET *Q2Vals* $\triangleq \{v : l$ received $m =$⟨"2b", a, i, v⟩, $a \in Q\}$
75: IN *learned*$[p] \leftarrow learned[p] \sqcup (\sqcap Q2bVals)$

Phase One

The first phase of Multicoordinated Paxos is similar to the other protocols: a coordinator willing to start a round sends a "1a" message to the acceptors and waits for their replies. The only difference is that in our protocol multiple coordinators may start the same round simultaneously. Acceptors reply to "1a" messages with "1b" messages, if they have not joined any bigger round.

The goal of this phase is to notify acceptors about the beginning of a new round, so that they do not accept values for smaller ones, and to gather information about accepted c-structs to execute the second phase.

Phase Two

After receiving replies from a quorum of acceptors, coordinators can start the second phase of the protocol: irrespectively to whether the round is classic or fast, a coordinator of the round picks a c-struct in action *Phase2Start*() and sends it to the acceptors. The c-struct picked extends all c-structs already chosen and contains as many commands as possible; the procedure is formalized in the next section.

If the round is classic, then the coordinator will append other commands to the initially picked c-struct and send it to the acceptors. If the round is fast, the acceptors extend the c-structs themselves. What differentiates Multicoordinated Paxos from Generalized Paxos is that, in the first, multiple coordinators may execute this phase in parallel in the same round.

Learning a Value

Learning is performed in action *Learn*. It consists simply of extending the already learned c-struct with suffixes from the accepted c-structs. As the learned c-struct is extended, its commands can be processed by the application according to its semantics.

Hence, in a general execution scenario, the following sequence of actions is expected to happen only when a new round starts, due to failures or collisions, which is supposed to be seldom:

1. One (or more) coordinators will execute action *Phase1a*(c, i) for some high enough round i.

2. Acceptors will acknowledge it by executing action *Phase1b*(a, i).

3. All coordinators in i-coordquorums will then execute *Phase2Start*(c, i), which will trigger the execution of *Phase2bClassic*(a, i) by the acceptors.

During the rest of the round, a simpler execution pattern takes place. If the round is fast, then

1. proposers execute *Propose*(p, C), and
2. acceptors execute *Phase2bFast*(a).

If the round is classic (multicoordinated or not), then

1. proposers execute *Propose*(p, C),
2. the round coordinators execute *Phase2aClassic*(c), and
3. acceptors execute *Phase2bClassic*(a, i).

In any case, learners repeatedly execute *Learn*(l).

3.4.2 The *ProvedSafe* Function

In this section we define the *ProvedSafe* function through which coordinators pick c-structs on the *Phase2Start* of new rounds. The function has two parameters: Q, a set of acceptors, and *1bMsg*, a mapping from every acceptor a in Q to a "1b" message sent by a. If Q is an i-quorum and every acceptor a in Q has sent "1b" message *1bMsg*$[a]$ with field rnd equal to i, which is mapped from a in *1bMsg*, then *ProvedSafe*$(Q, 1bMsg)$ returns a set of c-structs that are pickable for round i.

The function is formally defined as follows. For convenience and simplicity of the explanation, we assign aliases to the fields of "1b" messages: given a message $m = \langle$"1b", $a, i, vval, vrnd\rangle$, we refer to the five fields of m as *m.type*, *m.sndr*, *m.rnd*, *m.vval*, and *m.vrnd*, respectively.

Definition 1 (Proved Safe) *For any set of acceptors Q, and mapping 1bMsg from each acceptor in Q to a phase "1b" message, let:*

- $vals(S) \triangleq \{1bMsg[a].vval : a \in S\}$
  Set of vval values sent by acceptors in $S \subseteq Q$.
- $vrnds \triangleq \{1bMsg[a].vrnd : a \in Q\}$
  Set of vrnd values sent in all "1b" messages.
- $k \triangleq Max(vrnds)$
  Highest-numbered round in vrnds.
- $kacceptors \triangleq \{a \in Q : 1bMsg[a].vrnd = k\}$
  Set of acceptors that sent "1b" messages with $vrnd$ equal to k.

- $QinterR \triangleq \{Q \cap R : R \text{ is a } k\text{-quorum }\}$
  Set of intersections between Q and every k-quorum R.
- $QinterRAtk \triangleq \{S \in QinterR : S \subseteq kacceptors\}$
  Intersections of interest: those in which all elements sent "1b" messages with $vrnd$ equal to k.
- $\Gamma \triangleq \{\sqcap(vals(inter)) : inter \in QinterRAtk\}$
  glb's of the values sent in the "1b" messages, for every intersection of interest.

Then ProvedSafe is defined as follows:

$$ProvedSafe(Q, 1bMsg) \triangleq \text{IF } QinterRAtk = \{\} \text{ THEN } vals(kacceptors) \text{ ELSE } \{\sqcup\Gamma\}$$

The function implements the rule explained in Section 3.3, action *Phase2Start*. That is, if there is no k-quorum R for which all acceptors in $R \cap Q$ have sent "1b" messages for round i with field *vrnd* equal to k, then any value that has been reported in "1b" messages with *vrnd* $= k$ is pickable. Otherwise, the Fast Quorum Requirement ensures that all elements of the set Γ are extensions of any chosen c-struct. Even more, due to the properties of c-structs, Γ is compatible. As a result, its lub exists and is pickable. Hence, the lub, which contains as many commands as possible, should be picked and sent on "2a" messages to the acceptors.

In [Lamport, 2004], Lamport discusses how to deal efficiently with large c-structs and all the ideas presented there can be directly applied to our algorithm. The complexity of calculating lubs, glbs, and verifying the compatibility of c-structs will depend on the c-struct set being used. Most c-struct sets we are aware of (e.g., those presented in [Lamport, 2004]) admit relatively simple implementations of these operations. The complexity of calculating function *ProvedSafe* also depends on how quorums are defined and it can be simplified if quorums are defined as any set of processes of a certain size (e.g., majority sets).

3.4.3 Availability and Load-Balancing with Multiple Coordinators

The main problem of single-coordinated rounds as compared to multicoordinated ones has to do with availability. If the coordinator of a single-coordinated round crashes, time must be spent with the identification of the failure (usually done through timeouts), the election of a new coordinator, and the execution of the first phase of a higher-numbered round, before normal execution can be resumed. The optimization of these tasks may also effect performance or availability. For example, aggressive failure detection may trigger false suspicions, and simple leader election algorithms can elect a crashed process or more than a single leader at a time. If a

round has multiple quorums of coordinators, a single failure will not require immediate round change. Hence, we have introduced multicoordinated rounds as a way to circumvent the aforementioned availability problems.

A simple implementation of Multicoordinated Paxos would have a fixed number of coordinator processes in every round and define coordinator quorums of multicoordinated rounds as any majority of them so that the Coordquorum Requirement is satisfied. In such a case, the failure of any minority of the coordinators leaves at least one quorum of coordinators still available and, therefore, able to forward proposals to the acceptors.

One could argue that fast rounds also do not rely on a single coordinator during normal execution, since acceptors can accept proposals directly from proposers. However, the Fast Quorum Requirement imposes stricter restrictions on how fast quorums are defined, which also affects availability since fewer failures are tolerated.

The existence of multiple quorums of both coordinators and acceptors also enables implementations with better load balance than Classic Paxos. Recall that in Classic Paxos all commands must go through the current leader (round coordinator) and, depending on the system load, this might be a performance bottleneck. In Multicoordinated Paxos, for a command C to be learned in multicoordinated rounds, it must be forwarded by a coordinator quorum and accepted by an acceptor quorum only. If there are multiple coordinator and acceptor quorums, no acceptor or coordinator needs to process all commands proposed. A simple way to distribute the load has the proposer p of a command C choose (randomly or through some uniformly distributed function) a quorum of coordinators and a quorum of acceptors for C. p sends the "`propose`" message only to the chosen coordinator quorum, with the chosen quorum of acceptors piggybacked in the message since all coordinators in the quorum must forward C to the right acceptors. The coordinators send their "`2a`" message with C only to the indicated quorum of acceptors, which accept C and send phase "`2b`" messages to the learners. If not all learners need to learn about C, the same approach can be used, forwarding with C the set of learners to which the phase "`2b`" messages should be sent.

Fast rounds also allow distributing the load over the set of acceptors. As before, however, the stricter quorum requirement implies a worse distribution. If fast rounds are composed of $\lceil (3n+1)/4 \rceil$ acceptors, where n is the total number of acceptors (a necessary condition if any majority of the acceptors is a quorum for a classic round), then it is not hard to verify that every acceptor will have to process more than 3/4 of the proposed commands. In multicoordinated rounds, if any majority of the coordinators of a round i is an i-coordquorum and any majority of acceptors is an i-quorum, then the load can be distributed so that each coordinator processes at most $(1/2 + 1/nc)$ of the proposed commands, where nc is the total

number of coordinators for round i, and each acceptor accepts at most $(1/2 + 1/n)$ of the proposed commands. It is true that in this scenario, each command must be dealt twice (first by the coordinators and then by the acceptors), but the coordinators' action is much cheaper since it does not involve disk writes, as we show later.

Doing this sort of load balancing does not jeopardize availability. The optimistic use of a single quorum only does not mean that the other quorums cannot be used. A clever implementation would resort initially to a single quorum of coordinators and acceptors. If the command is not learned after some time has elapsed (triggered by a timeout or a failure suspicion), then other quorums might be used. This wait time can be set to a minimum since they will never trigger a round change as discussed in the beginning of this section.

Finally, it is clear from the algorithm that the sets of coordinators and acceptors need not have the same number of elements. Actually, in many cases it might be better to have more acceptors than coordinators in a round. Note that the set of coordinators for round i can be completely different from the set of coordinators of round $j \neq i$, but this is not the case for acceptors since they must be queried in every new round to check whether a value has already been chosen at some previous round. Moreover, the acceptors' task of accepting a value is more expensive than the coordinators' task of forwarding it, since the former requires a disk write but the latter does not. As a result, implementations might use a high number of acceptors to improve the system's resilience or performance (due to load balancing). But an equally high number of coordinators for a round increases only the availability of that round, and the load balancing will not be as effective since the coordinators' task is cheaper. For a small system, a configuration with 5 acceptors in total and 3 coordinators for multicoordinated rounds (with different sets of coordinators for different rounds) sounds plausible, since it tolerates the failure of any two processes and does not introduce temporary unavailability if a single coordinator crashes.

3.4.4 Collisions

Multicoordinated rounds have a drawback that does not exist in single-coordinated ones—collisions. In multicoordinated rounds, a collision happens when commands proposed concurrently arrive at the coordinators in different orders and this leads to their forwarding of incompatible c-structs. If no coordinator quorum forwards c-structs whose glb can extend the values previously accepted by the acceptors, the round is stuck since no new command can get accepted.

This is a different type of collision than the one that may occur in fast rounds, explained in Section 2.3.2. In fast rounds, a collision happens when acceptors accept incompatible c-structs that cannot further extend the values learned by learners so

far. In this case, however, acceptors pay the price of accepting commands that will never be learned, which does not happen in collisions of multicoordinated rounds. This is a major difference between the two kinds of collisions since acceptors must write on stable storage every time they accept a value but coordinators do not have to, as we explain in Section 3.4.5.

The mechanisms to solve collisions in the original Fast Paxos algorithm presented in Section 2.3.2, which guessed the values of "1b" messages for round $i+1$ out of "2b" messages for round i, cannot be directly applied to Generalized Paxos. This happens because acceptors are allowed to append commands to their c-structs at will in fast rounds and, hence, unless the coordinator asks the acceptors to move to the next round, it could wrongly guess the latest acceptances of acceptors and pick a bad c-struct for the next round. If no other algorithm exists, the techniques we present below for multicoordinated rounds can be used at the cost of one extra communication step for the acceptors to identify the collision in the c-structs they have accepted. Another possibility consists of explicitly starting a new higher-numbered classic single-coordinated round from the beginning after its coordinator identifies the collision.

In multicoordinated rounds, collision identification can be done by the acceptors when they receive the phase "2a" messages from the coordinators of a classic round i. If two coordinators of the same i-coordquorum send "2a" messages for round i with incompatible c-structs, acceptors execute action *Phase1b*$(a, i+1)$ as if they had received a phase "1a" message for round $i+1$. What comes next will depend on whether round $i+1$ is classic or fast.

If round $i+1$ is classic and enough acceptors identify the collision, which will normally happen if messages are not lost and processes do not crash, then the coordinators of round $i+1$ will execute action *Phase2Start*$(c, i+1)$ based on the received messages, followed by one or more executions of action *Phase2aClassic*(c). Thus, the collision in round i will be resolved with only two extra communication steps (as compared to the usual three of a classic round). Clearly, to avoid that another collision happens when the coordinators start round $i+1$, it is advisable to have it as a single-coordinated round. After some time of normal execution, if conflicting commands stop being proposed, the coordinator of round $i+1$ can start a multicoordinated round again. This approach is a variation of the *coordinated recovery* presented in [Lamport, 2006a].

If round $i+1$ is fast, performance can be improved by setting $i+1$-coordquorums wisely. Since the Coord-quorum Requirement does not place any restriction on fast rounds, we can define that any single acceptor by itself constitutes a coordinator quorum for a fast round (playing both roles—acceptor and coordinator). When a coordinator of round $i+1$ (which is also an acceptor) executes actions *Phase2Start*$(c, i+1)$ and *Phase2aClassic*(c), it can locally accept the values suppos-

edly sent in the "2a" messages, without actually sending them. This approach can resolve collisions with only one extra communication step. However, new collisions might happen when the round $i + 1$ is started. This is a variation of the *uncoordinated recovery* presented in [Lamport, 2006a], which also presents some interesting ideas to avoid having collisions when round $i + 1$ is started. We do not cover them here because the use of a fast round to recover from a collision in a classic one does not seem to be of practical use. We just present the idea of the uncoordinated recovery mechanism for completeness.

One might ask herself if there can be some kind of round that does not rely on a single coordinator but in a coordquorum and still avoid collisions. In the most general case, its existence would contradict the FLP result since quorums could be defined to tolerate a single failure and the absence of collisions would mean that liveness can be achieved in such rounds.

3.4.5 Reducing disk writes

Assumption 4, on page 34 imposes no restriction on coordinator quorums of different rounds. If it is always possible to start new rounds with any set of coordinator quorums, coordinators are not required to write on stable storage. A coordinator that crashes and later recovers could just be seen as a new coordinator in the system, which is easily implemented by having an "incarnation" counter associated with its identifier. In the following we explain how new rounds can be created with any set of coordinator quorums.

Consider round numbers as records of the form $\langle Count, Id, RType, S\rangle$, where $Count$ is a natural number, Id is a coordinator's unique identifier, $RType$ is a natural number, and S is a set of coordinator quorums. Rounds are uniquely identified by the first three fields of the record and totally ordered by comparing these fields lexicographically. Field S is merely informative and is not taken into consideration when comparing two rounds. Using this approach, when the current leader wants to start a new round, it can simply define the four fields according to its current knowledge. By setting $Count$ properly, it can always create a round higher-numbered than any other it has seen before. Id identifies the coordinator that created the round and possibly the one responsible for coordinated recovery. $RType$ tells the round type, being 0 for fast and bigger integers for classic. Finally, S identifies all valid coordinator quorums for the new round.

Since Assumption 3 requires that quorums of different rounds intersect, acceptors cannot lose their state after a crash and assume a different identity upon recovery. This happens because the values accepted by acceptors cannot be forgotten, or the algorithm's safety would be compromised. Therefore, these values must always be stored on stable storage, incurring a disk write (or equivalent op-

eration) whenever an acceptor executes a *Phase2b* action. As a result, acceptors are not as easily replaceable as coordinators and more complex strategies must be used [Lamport and Massa, 2004, Lorch et al., 2006].

Action *Phase1b* also changes the internal state of an acceptor and, at a first sight, this seems to imply that *Phase1b* must also write on disk. However, an acceptor a may store $rnd[a]$ only in main memory as long as, after recovering from a crash, it manages to initialize $rnd[a]$ with a higher value than the previous one. We propose this to be done as follows: field $Count$, previously described in this section, can be composed of a major and a minor component, *MCount* and *mCount*. When an acceptor executes *Phase1b* for some round, if $MCount$ equals the previous value in $rnd[a]$, it changes $rnd[a]$ in volatile memory only; otherwise, it writes it on disk. During recovery, the acceptor simulates the reception of a "`1a`" message with an $MCount$ higher than the one it has on disk. To get values accepted by the recovered acceptor, coordinators will be forced to use higher rounds. In the normal case, acceptors write on disk only once, when they are started. In the presence of failures, this strategy results in one extra disk write at each acceptor, per recovery.

3.4.6 Setting rounds and quorums

The schema used to define round numbers presented in the previous section, that is, as a vector of the form $\langle MCount{:}mCount, Id, RType, S\rangle$, should fit most of application scenarios. However, there are some specific cases in which this schema should be adapted for better performance. There are two main points on doing these adaptations: the likeliness of collisions and how they are recovered, and what type of round follows each other.

For example, if collisions are frequent in the system, then fast rounds should not be used because the recovery cost could outweigh the economy provided by fast rounds. If fast rounds are used in conjunction with coordinated recovery, then they should be followed by single-coordinated rounds. In the case of uncoordinated recovery, fast rounds should be followed by multicoordinated fast rounds. (See Section 3.4.4.)

In the case of the example, where some fast rounds should be followed by other fast rounds, using the record described above would force the use of rounds with different Id or $Count$ field. Because recovery relies on a process knowing exactly what is the next round number, this schema would not work. A possible solution is to change how the field $RType$ is interpreted. For example, letting all $RType$ in the range 0 to 5 be interpreted as fast, instead of simply 0, and allowing a round number to be incremented without changing any but the $RType$ field.

The set of coord-quorums can be defined at run-time, considering the status of the system when a new round is created. To ensure liveness, multicoordinated

rounds should be followed by single-coordinated rounds (See Section 3.4.7). However, this transition does not have to be abrupt and could be done through a series of multicoordinated rounds with smaller quorums, minimizing the risk of collisions while still allowing the benefits of multicoordination.

Below we present a few general scenarios and discuss how round numbers and quorums could be defined for them. These scenarios are not necessarily disjoint, and real world systems would probably share the characteristics of both of them, as well as other relevant ones. Hence, a per-case analysis is needed to find the best solution. Notice that none of the Paxos algorithms requires the ordered use of ballot numbers and that coordinators are allowed to skip rounds—possibly based on the the dynamics of the environment it is in. However, because collision recovery explores the sequential execution of rounds, skipping rounds could prevent efficient recovery and therefore the rounds' configuration should be defined *a priori* as precisely as possible.

Highly reliable clustered systems

Clustered systems connected through high-speed networks have a large probability of spontaneously ordering the messages sent to the same destinations. In such systems, acceptors executing a fast round are likely to accept the values in the same order. Therefore, values can be learned in two communication steps, even when conflicting proposals are made.

In such a scenario, a large sequence of fast rounds followed by single-coordinated rounds seem the best configuration: most of the time values are fast learned and, in the rare case of conflicts, they are solved by the variant of uncoordinated recovery presented in Section 3.4.4. Coordinators can always resort to the next single-coordinated round to ensure liveness if uncoordinated recovery does not succeed.

For this scenario, the values of the $RType$ field in the basic approach can be mapped to a range of many fast rounds followed by classic rounds, as we mentioned before. By dividing $RType$ in a major and minor component, as we did with $Count$, the number of successive fast rounds is bounded only by the memory size.

If conflicts are rare but tend to be persistent and require coordinated recovery, then a solution is to map $RType$'s even values to fast rounds and odd values to single-coordinated classic rounds.

Crash prone clustered systems

Remember that for a fast round to progress, more than two thirds of all acceptors must be functional and reachable. Not even resorting to classic rounds would allow the protocol to progress if this condition is not met. As we explained in Section 2.3.2, to progress with fewer acceptors in classic rounds, the number of acceptors needed

for fast rounds to terminate may increase to up to three quarters of their total number. In a scenario where more than one third of acceptors can crash, fast rounds should not be used.

In such a scenario, even though spontaneous ordering cannot be used to achieve fast termination, it can be used to improve availability; with multicoordinated rounds, progress can be achieved in the absence of any minority of coordinators for a given round. If acceptors and coordinators are collocated, for example, then the round can tolerate any minority of acceptor-coordinator crashes.

Using the same round number definition as in the previous sections, $RType$ is permanently set to classic round. The leader starts new rounds whenever the number of functional and reachable coordinators falls below a certain threshold. The extra availability given by multicoordinated rounds can be used to relax failure detection requirements.

Highly loaded and time constrained systems

If most proposed commands commute, either multicoordinated or fast rounds can be used, with the second option having the advantage of smaller latency. However, even though commands do not semantically conflict, they do contend for stable storage when accepted at the acceptors. To mitigate the performance loss due to I/O, the load of accepting values can be distributed over the acceptors by randomly selecting a quorum to whom send proposals. Since fast rounds require quorums of more than two thirds of the acceptors, each acceptor must bare the load of at least two thirds of the proposals. Using classic rounds, the load over each acceptor may be lowered to roughly half of the acceptances. Since coordinators do not need to access stable storage, the fewer stable storage accesses at the acceptors fairly makes up for the extra communication delay with state-of-the-art networks.

Under particularly high load or if other tasks must be performed for each proposal, just forwarding proposals may become a too demanding task for a single coordinator. For example, if coordinators deal with proposers in an open network and must authenticate or decode their proposals before forwarding them to acceptors. In such a scenario, multicoordination can be used to alleviate the load on each coordinator to roughly half. Round numbers can be specified as in the previous scenario.

Conflict prone

In widely distributed systems or under high load, spontaneous ordering cannot be expected and non-commutable proposals often result in conflicts. In such environments, if non-commutable commands prevail, then the algorithms will always end up resorting to single-coordinated classic rounds to finish.

In such an environment, it does not make sense to have fast nor multicoordinated rounds, and round numbers can be defined as the set of integer and partitioned among a finite set of coordinators by some module function. For an infinite set of coordinators or for having each coordinator as the leader of infinitely many consecutive rounds, a simplified round number of the form $\langle MCount{:}mCount, Id\rangle$ could be used.

3.4.7 Ensuring Liveness

The Multicoordinated Paxos algorithm, as presented in Algorithm 7, always satisfies the safety properties of Generalized Consensus. Regarding liveness, on the one hand, the possibility of all coordinators always starting new rounds allows the algorithm to progress if a round does not succeed due to coordinator crashes or proposal collisions. On the other hand, starting new rounds carelessly can also prevent liveness, since by starting a new round a coordinator may prevent smaller rounds from succeeding. In Classic and Fast Paxos, this problem is avoided by using some unreliable leader election mechanism to elect a single coordinator as responsible for starting higher-numbered rounds under its coordination.

In Multicoordinated Paxos, we can use the same strategy to prevent the continuous initialization of new rounds. If the current leader starts a new classic single-coordinated round (of which it is the only coordinator), liveness is ensured as long as the leader does not crash and other coordinators do not wrongly think they are the current leader and try to start a higher-numbered round; such a property is ensured by the Ω leader election oracle [Chandra et al., 1996]. If other coordinators interfere, the leader starts an even bigger round. Below we change the algorithm to include negative replies to "`1a`" or "`2a`" messages with a round number lower than the acceptors' current one.

If the leader starts a fast round, its progress is ensured as long as the leader does not crash during the execution of phase 1 and collisions do not happen during the rest of the round's execution. If there are no failures, collisions can be resolved by adapting the collision recovery mechanisms presented in [Lamport, 2006a] to Generalized Paxos; acceptors identify collisions and use the approach we explained in Section 3.4.4 to solve them, or simply the leader identifies the collision and starts a new classic single-coordinated round.

In classic multicoordinated rounds, progress is ensured if the leader does not crash during the execution of phase 1, collisions do not happen, and at least one coordquorum remains available during the rest of the round execution. The failure of the leader is not a problem since another correct leader is eventually selected which will make sure that a new round gets started. As for collisions, the mechanism presented in Section 3.4.4 can be used—the only restriction we make is that

the leader must be one of the coordinators for the following round, otherwise the leader might think of the round change as an interference and try an even higher-numbered round. Last, to cope with the failure of coordinators, the leader must start a new round if it believes that too many of the other coordinators have failed. Their possible failure can be assessed by monitoring their "2a" messages or through some external failure detection mechanism. While there are still enough coordinators to form coordquorums, the leader can decide on coordinator quorums for the next round. When the leader notices that there are not enough coordinators in the current round to ensure progress, it starts the new higher-numbered round.

We now present a list of modifications to Algorithm 7 to incorporate the mechanisms discussed in the previous paragraphs.

- Add variable *amLeader*$[c]$ to every coordinator c, with any boolean as initial value.
  The variable contains a boolean stating whether coordinator c believes itself to be the current leader (TRUE) or not (FALSE).

- Add variable *activec*$[c]$ to every coordinator c, with initial $\emptyset$.
  The variable maintains the set of all coordinators whom c believes to be functional. The contents of $activec[c]$ is used both to monitor if the number of coordinators of a round has fallen too short and to determine the coordinators that will be in the coord-quorums for the next rounds.

- Add a *Nack*(a,i) action, enabled when an acceptor a receives a "1a" or "2a" message for a round $i < rnd[a]$ and the coordinators of i and $rnd[a]$ are different. If these pre-conditions are satisfied, then a sends a $\langle$"skip", $rnd[a]\rangle$ message to the coordinator of i.
  This action serves to inform the coordinator of i that a round $rnd[a]$, bigger than i was already initiated, and that it should start an even bigger round.

- Add "*amLeader*$[c]$" to action $Phase1a(c,i)$ as an extra pre-condition.
  This condition prevents coordinators that do not believe themselves to be the leader from creating new rounds *ad infinitum*.

- Add "(c received a message $\langle$"skip", $j\rangle$ such that *crnd*$[c] < j < i$ **or** there is no coord-quorum for round $crnd[c]$ that is a subset of $activec[c]$) **and** there is i-coordquorum that is a subset of $activec[c]$" as an extra pre-condition to action *Phase1a*(c,i).
  This change lets c start round i as a reaction to the "skip" message sent by some acceptor, or if c believes that no coord-quorum in its current round is still alive. Moreover, it requires that at least one coord-quorum of round i be active.

- Add an action "*Resend*(a)", whose only sub-action is to resend the last message sent by agent a.
  This action will ensure that messages required for progress are eventually delivered.

Since these changes only restrict the pre-conditions of the algorithm's original actions or add new actions already allowed by the protocols behavior, the resulting algorithm is an implementation of the original. Therefore, it maintains the safety properties of Multicoordinated Paxos.

Let p be a proposer, c a coordinator, l a learner, Q a quorum of acceptors, and C a non-empty set of coordinators. The liveness condition of Multicoordinated Paxos, *MCLiv*(p, l, c, Q, C), is the conjunction of the following conditions.

- $\{p, l, c\} \cup Q \cup C$ are not crashed.
- p has proposed a command v.
- c is the only coordinator that believes itself to be the leader.
- $activec[c]$ is a subset of C.
- For every round $i > crnd[c]$, there exists a round $j > i$ such that some subset of $activec[c]$ is a j-coordquorum.
- There is an upper bound on the number of uncoordinated recovery tries that the algorithm makes or messages are delivered in the same order by all coordinators and functions are deterministically executed as soon as their preconditions are satisfied.

If *MCLiv*(p, l, c, Q, C) holds eventually forever and there is an upper bound on the number of uncoordinated recovery tries that the algorithm makes, then l eventually learns a c-struct that contains the command proposed by p. This implies that, as we mentioned before, Multicoordinated Paxos can ensure liveness under the same assumptions as Classic Paxos since it can always resort to a single-coordinated round to ensure progress, which is possible when using round numbers as specified in the previous section. If no conflicting commands are proposed, then the algorithm ensures progress if *MCLiv*(p, l, c, Q, C) holds eventually forever even if only uncoordinated recovery is used.

3.5 Solving Generic Broadcast with Multicoordinated Paxos

In the Generic Broadcast problem [Pedone and Schiper, 2002], agents must agree on a partially ordered set, or poset, of proposed commands; we refer to these posets

in the generic broadcast problem as c-hists, short for command histories. The partial order in a c-hist must order non-commutable commands, where commutable is defined in terms of a conflict relation. Read-only operations are common examples of commutable commands. Operations changing the same piece of data, as a file in a file system or a row in a database, may be commutable or not, depending on the application.

3.5.1 Command Histories and Formal Definition

Formally, a command history is defined as follows, for some Generic Broadcast instance.

Definition 2 (Command History) *Let Cmd be the set of commands that may be broadcast and $\asymp$ be a reflexive and symmetric conflict relation over the commands of Cmd. Then the partially ordered set $(S, \prec)$ is a command history iff:*

- $S \subseteq Cmd$
- $\forall C, D \in S,\ C \asymp D \Rightarrow (C \prec D \text{ or } D \prec C)$

A c-struct set whose c-structs are c-hists is defined by the tuple $\langle Cmd, \bot, \bullet \rangle$ where $\bot = (\emptyset, \prec)$ and $\bullet$ is defined as follows:

- For all $C \in Cmd$

$$(S, \prec) \bullet C = \begin{cases} (S, \prec) & \text{if } C \in S, \\ (S \cup \{C\}, \prec_\bullet): \ \forall a, b \in S, a \prec b \Leftrightarrow a \prec_\bullet b, & \\ \qquad\qquad \forall a \in S, a \asymp C \Rightarrow a \prec_\bullet C & \text{otherwise} \end{cases}$$

- For any sequence of commands $\langle C_1, \ldots, C_m \rangle,\ C_1, \ldots, C_m \in Cmd$

$$(S, \prec) \bullet \langle C_1, \ldots, C_m \rangle = \begin{cases} (S, \prec) & \text{if } m = 0, \\ ((S, \prec) \bullet C_1) \bullet \langle C_2, \ldots, C_m \rangle & \text{otherwise} \end{cases}$$

The properties that define c-structs are useful when defining practical generalized consensus protocols and it is not clear how useful c-hists are without such properties. Hence, hereafter, c-hists refer to the c-structs defined from c-hists. What is more, definitions on Section 3.2.1 defined for c-structs are automatically defined for command histories as well. In special, we say that a command history g extends a command history h ($h \sqsubseteq g$) if there exists a sequence σ of commands such that $g = h \bullet \sigma$. As an example of the meaning of such definitions, consider the following set of c-hists, where $a \leftarrow b$ means $a \prec b$.

$$T = \left\{ \begin{array}{l} a \leftarrow c \quad b \leftarrow d \\ a \leftarrow \quad b \leftarrow d \\ a \leftarrow c \leftarrow b \quad e \end{array} \right\}$$

The lower bounds of T are $\{\bot, a, a \leftarrow b\}$, being $\sqcap T = a \leftarrow b$ (the greatest lower bound of T) and $\sqcup T = a \leftarrow c \leftarrow b \leftarrow d \quad e$ (the least of upper bound of T).

With command histories formally defined, we can now properly state the Generic Broadcast problem. Let $learned[l]$ be the c-hist learner l has learned. Generic Broadcast is defined by the following properties.

Non-triviality For any learner l, $learned[l]$ only contains proposed commands.

Stability For any learner l, the value of $learned[l]$ at any time is an extension of $learned[l]$ at any previous time.

Consistency The set $\{learned[l] : l \text{ is a learner}\}$ is always compatible.

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

3.5.2 A Simple Command History

Using Multicoordinated Paxos to solve Generic Broadcast is straightforward, and is more related to choosing the right c-struct than changing the protocol. In fact, the only change is restricting the use of c-structs in the protocol to c-hists. In this section we define a simple representation of c-hists and the operations performed on it.

Command histories can be represented as sequences of commands. The command history $\bot$, for example, may be represented simply as $\langle\rangle$, while the command history $\bot \leftarrow \begin{array}{l} a \leftarrow c \\ b \leftarrow d \end{array}$, where the arrows point to the previous elements in the partial order, may be represented as $\langle a, b, c, d\rangle$, $\langle a, c, b, d\rangle$, $\langle a, b, d, c\rangle$, $\langle b, a, d, c\rangle$, or $\langle b, a, c, d\rangle$.

This representation, which is in fact a topological sorting of the c-hist, ensures that, given a command history $(S, \preceq)$, integers i and j, $i < j < |S|$, and the sequence s that represents the command history, $s[i] \preceq s[j]$.

New commands can be added to a sequence by simply appending it at the sequence's end, if it is not in the sequence yet. Formally, the $\bullet$ operator can be defined as follows:

$$\langle C_1, \ldots, C_m\rangle \bullet C = \left\{ \begin{array}{ll} \langle C_1, \ldots, C_m\rangle & \text{if } \exists i, C = C_i \\ \langle C_1, \ldots, C_m, C\rangle & \text{otherwise.} \end{array} \right.$$

Because these sequences do not represent all the ordering information, the conflict relation ($\asymp$) is still needed to assess the order of commands within sequences when, for example, calculating the lub or checking the compatibility.

Algorithm 8 below determines the longest common prefix between two command histories H and I, that is $H \sqcap I$. It does so by checking if the head h of H exists on I before any conflicting command, in which case it is part of their common prefix. Otherwise, the algorithm recursively proceeds on the tail of H stripped of the descendants of h, since none of them can be part of the prefix. Observe that in this definition we use the set minus operator, $\setminus$, to remove some element from a sequence.

Algorithm 8 Longest common prefix of two c-hists H and I.

```
1: H ⊓ I ≜
2:   IF H = ⟨⟩ ∨ I = ⟨⟩
3:    THEN ⟨⟩
4:    ELSE IF ∃j : Head(H) = I[j] ∧ ¬∃k < j : Head(H) ≍ I[k]
5:      THEN ⟨Head(H)⟩ ∘ (Tail(H) ⊓ (I \ Head(H)))
6:      ELSE (Tail(H) \ Descendants(Head(H), Tail(H)) ⊓ I)
```

To calculate the glb of a set of command histories, instead of a simple pair, the search on the sequence I could be performed in parallel for many sequences. Nonetheless, here we stick to an iterative approach of simpler understanding, presented in Algorithm 9.

Algorithm 9 Longest common prefix of a set of c-hists S.

```
1: ⊓S ≜
2:   IF S = {e}
3:     THEN e
4:     ELSE LET e,f ∈ S, e ≠ f
5:       IN   ⊓((e ⊓ f) ∪ S \ {e,f})
```

Determining if two sequences are compatible is more complicated. The procedure presented below iterates over the first sequence, H, looking for the first element e of H that does not appear in I. If, during this search, some conflicting ordering is identified among the sequences, in a procedure similar to the one on the glb operator, then the sequences are not compatible. If no incompatibility is found, then the procedure searches for descendants of e in I. If any exists, then it also indicates incompatibility, as e would have to appear before its descendant also in I. If none is found, the operator recursively proceeds on the rest of H, but keeping the list of removed elements in a set of ancestors A, so that new descendants can

be identified in the next steps. Because the conflict relation is reflexive, there is no need to reverse the roles of H and I and repeat the procedure.

Algorithm 10 Determines if two c-hists H and I are compatible.

```
1:  AreCompatible(H,I,A) ≜
2:    IF H = ⟨⟩ ∨ I = ⟨⟩
3:     THEN TRUE
4:     ELSE  IF ∃j : Head(H) ≍ I[j] ∧ ¬∃k < j : Head(H) = I[k]
5:       THEN FALSE
6:       ELSE  IF ∃j : Head(H) = I[j]
7:         THEN IF ∃f : f ∈ A ∧ Head(H) ≍ I[f]
8:           THEN FALSE
9:           ELSE  AreCompatible(Tail(H), I \ Head(H), A)
10:        ELSE  AreCompatible(Tail(H), I, A ∪ {Head(H)})
```

AreCompatible can be rewritten to generate the lub of two sequences as it goes on verifying their compatibility. Because such operator would be more complex, we opted for a simplified version, which assumes that the sequences are compatible.

Algorithm 11 Shortest common extension of two c-hists H and I.

```
1:  H ⊔ I ≜
2:    IF H = ⟨⟩
3:     THEN I
4:     ELSE  IF ∃j : Head(H) = I[j]
5:     THEN ⟨Head(H)⟩ ∘ (Tail(H) ⊔ (I \ Head(H)))
6:     ELSE  ⟨Head(H)⟩ ∘ (Tail(H) ⊔ I)
```

Finally, the following operator calculates the lub of a set of compatible sequences.

Algorithm 12 Shortest common extension of a set of c-hists S.

```
1:  ⊔S ≜
2:    IF S = {e}
3:     THEN e
4:     ELSE  LET e,f ∈ S, e ≠ f IN   ⊔((e ⊔ f) ∪ S \ {e,f})
```

3.5.3 A Run of Generic Broadcast

Figure 3.2 shows a run of our Generic Broadcast algorithm. In order not to clutter the picture, we have represented messages addressed to multiple agents as a single

line pointing to the different addressees. Commands broadcast are represented as simple geometric shapes. The conflict relation is defined such that commands with the same shape *or* color commute. Time flows left to right.

In the run, the first three broadcast commands (□, ▽,○) are chosen and learned without conflicts. The fourth command, ▼, conflicts with the first and second ones (▼ ≍○ and ▼ ≍ □). Because both the first and second coordinators had seen □, they append ▼ after □ in their c-hists (□▽▼ and ○□▼, respectively). Hence, because the first coordinator has not seen ○, its c-hist will not be compatible with the c-hist sent by the second coordinator to the acceptors, which has ○ before ▼. (The ▽ command is not important here, since it commutes with ▼.) Had both coordinators heard of ○, their c-hists would be compatible and accepted by the acceptors. Since that is not the case, a new round to solve the conflict is required. In the example, we have shown a single-coordinated round being started and solving the conflict.

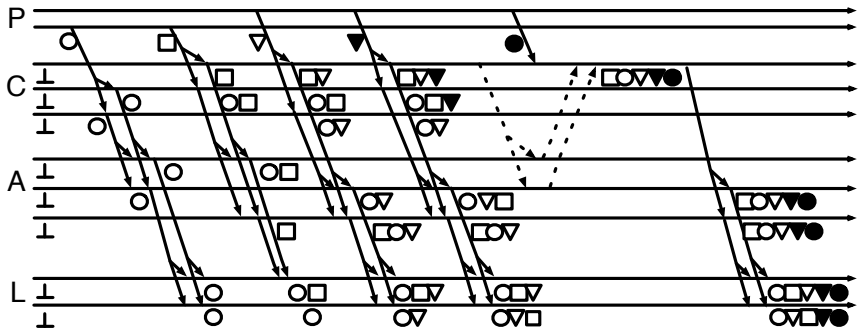

Figure 3.2: Multicoordinated round followed by a single-coordinated round: After a series of successful appends to the accepted c-hist, one conflict happens and one coordinator changes to the single-coordinated mode to solve the conflict. Observe how learners learn different but compatible prefixes. The dashed arrows show the leader polling the acceptors for their accepted c-hists.

3.6 Final Remarks and Related Work

Multicoordination is a technique that improves the availability of leader-based agreement protocols by replacing the leader by multiple coordinators working concurrently. Because the use of multicoordination introduces the risk of collisions of commands at the coordinators, we believe it is better employed by applications that can use semantics to minimize the cost of conflicts. For this reason, we have used multicoordination to solve Generalized Consensus, in which semantics may be captured by defining the problem's command structures. We have done so by extending

Generalized Paxos, the only Generalized Consensus algorithm that we are aware of. We have named the resulting protocol Multicoordinated Paxos.

Generic Broadcast [Pedone and Schiper, 1999, Pedone and Schiper, 2002] is an instantiation of Generalized Consensus using c-hists, c-structs in the form of partial orders that do not order commutable commands. In Section 3.5 we have presented one such c-hist which, in spite of its power, has a very simple implementation.

There are three generic broadcasts algorithms in the literature to which we can compare Multicoordinated Paxos using our c-hist. GB+ [Pedone and Schiper, 2002] by Pedone and Schiper, an improved version of their first algorithm, introduced along the definition of generic broadcast [Pedone and Schiper, 1999]. In GB+, a proposal v is learned in two steps if no conflict arises or if previously accepted conflicting proposals have already been chosen. Otherwise the protocol resorts to a fast consensus instance, amounting to a total of four steps if the consensus works well. The algorithm was developed for crash-stop environments and requires more than two thirds of the acceptors to be non-faulty. Hence, if compared to Generalized and Multicoordinated Paxos and ignoring the difference in the failure model, GB+ is roughly equivalent to the fast mode with uncoordinated conflict recovery. We say roughly because the fast mode always ignores conflicts between messages that have been spontaneously ordered, while GB+ only ignores conflicts if the messages have been delivered distant in time. What is more, Generalized and Multicoordinated Paxos may resort to coordinated recovery and, in the case of Multicoordinated Paxos, an uncoordinated classic recovery which may gradually lower the requirement for spontaneous ordering until the instance is decided.

The thrifty generic broadcast of Aguilera *et al.* [Aguilera et al., 2000], which we refer to as AGB, requires three steps to learn a proposal in the absence of collisions and failures. As GB+, AGB may be able to ignore some conflicts but, in general it falls back to atomic broadcast when conflicting proposals happen, and requires three more steps to have proposals learned. While AGB tolerates any minority of crashed acceptors , the authors have also described a variation of the algorithm that is equivalent to GB+. That is, it tolerates less than one third of failures but decides in two or four steps.

In the same way that GB+ compares to the fast mode, AGB compares to the multicoordinated mode of Multicoordinated Paxos: it may identify a collision even if conflicting commands are received in the same order, and cannot change the execution mode during execution.

While GB+ and AGB do not resort to consensus or atomic broadcast before at least two communication steps have been used, Zielinski's optimistic generic broadcast [Zielinski, 2005], OptGB, starts using consensus from the moment a value is proposed. With respect to latency, OptGB is more efficient than the other two protocols: it lets a proposal be learned after two steps if there are no conflicts or pro-

posals are orderly received by the acceptors, and in three steps otherwise. Hence, OptGB and fast rounds offer the same latency in the absence of conflicts, actual or perceived. OptGB solves conflicts in one more step, but only if the leader election oracle it relies upon works. Fast rounds solve conflicts in one step if the uncoordinated recovery works; using uncoordinated recovery takes two steps under the same conditions of OptGB. Compared to the multicoordinated mode, OptGB is always faster by one step, at the expense of tolerating less failures—OptGB tolerates less than one third of acceptor crashes. Moreover, it is not clear how it could be used to solve Generalized Consensus in general.

Another work that is closely related to ours is the decomposition of consensus algorithms by Song *et al.*[Song et al., 2008]. In this work, the authors break consensus algorithms in two quorums systems, one of selectors (coordinators), that pick proposals from proposers, and another of archivers (acceptors), that store the selected proposals and forward them to the deciders (learners). According to the way these quorums are configured, they can instantiate different protocols. The same happens with the multicoordinated consensus protocol from Chapter 2. For example, one can instantiate Rabin [Rabin, 1983] with fast rounds and uncoordinated recovery, Ben-Or [Ben-Or, 1983] with multicoordinated rounds and uncoordinated recovery, or Paxos [Lamport, 1998] with single coordinated rounds. The main difference between their work and ours is that they have not applied their deconstruction to the generalized consensus problem, nor to agreement among groups of processes, which we discuss in the next chapter.

Chapter 4

Fast Agreement for Groups

The ideal situation occurs when the things that we regard as beautiful are also regarded by other people as useful.
Donald Knuth

4.1 Agreement in Networks of Groups

With data centers spread across the globe and distributed systems that span the Internet, the problem of ensuring consistency in distributed environments has gained new dimensions. Online retailers and communities, e.g., Facebook and Amazon.com, have hundreds of thousands of clients around the globe accessing different data centers, depending on their location. In some cases, although clients may access their data on any data centers, updates are directed to a single location to be applied and later propagated to the other locations. In such applications, allowing updates be performed in any data center such that updates are readily seen after they are performed, without having to wait for a propagation time, would probably improve the user experience. Implementing such an update-everywhere database replication may be done atop of agreement protocols (e.g., Pedone *et al.* [Pedone et al., 2003]). To be advantageous, such approach must be based on agreement protocols that are efficient and resilient in this networking scenario, which we refer to as corporative networks.

Corporative networks may be abstracted as groups of agents and, typically, are characterized by large differences in terms of latency of communication between two agents: while agents within the same group use low latency communication channels to exchange messages, those in different groups may experience latencies that are orders of magnitude higher. More specifically, we consider networks abstracted by Figure 4.1, on page 74, in which agents are organized in a set of m subsets or groups $\Gamma = \{G_1, \ldots, G_m\}$. Although groups may be seen as data centers,

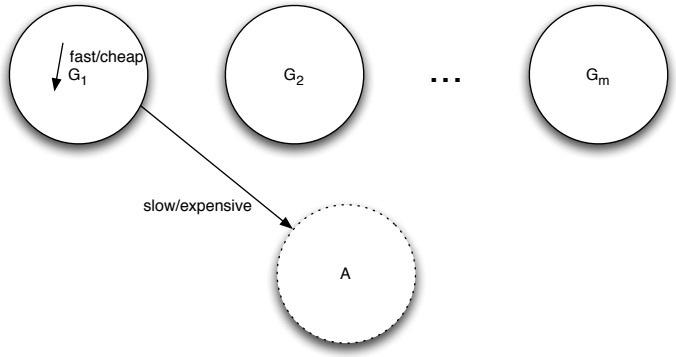

Figure 4.1: Agents distributed in groups in a wide area network: a common setup for corporative networks. Agents in a group $G_i, 0 \leq i \leq m$, are physically close to each other. Agents in A are spread geographically.

the same abstraction also works for smaller setups as, for example, the internals of a single data center, where groups are racks of nodes.

Besides the sets $G_1, \ldots, G_m$ of agents that effectively propose and learn the agreed values, Figure 4.1 depicts another set A, which stands for the acceptors in the system. To ensure fault tolerance, A should be composed by agents executing in the same physical locations as the agents in Γ. Hence, we consider that the cost of exchanging information between any group and A and between the elements of A is as expensive as between two agents in different groups.

Fast x Multicoordinated Rounds

Consider the price of solving standard consensus in such an environment using a leader based protocol such as Classic Paxos. Let G_l be the group to which the leader belongs. Because the protocol is so dependent on the leader, it has the following clear drawbacks: (i) agents in $G_m, m \neq l$, must monitor the leader through intergroup links; (ii) in the case of a leader failure, the reconfiguration takes two intergroup delays, plus the time to detect the failure; and, (iii) while for every agent in G_l the time between proposal and decision is of two intergroup delays in good runs, for any other agent not in G_l the latency is of at least three intergroup delays.

Using the multicoordinated approach that we have described in the previous chapters with all coordinators within G_l will minimize the effects of the first two drawbacks. That is, failure detection may be less aggressive since rounds are more resilient and it is unlikely that rounds will need to be changed due to conflicts, since groups are probably within a single area network where spontaneously is more likely. Multicoordination, however, does not help with the third drawback.

To allow a two-step latency from any group of agents, fast rounds would be more appropriate, but they have their own drawbacks: collisions, a large number of messages, and lesser resilience. One way to cope with collisions is to rely on spontaneous ordering of proposals but that may not be reasonable over intergroup links for any minimally loaded system. The effects of collisions can be minimized by using protocols that are conflict aware, as Generalized and Multicoordinated Paxos. More than simply avoiding conflicts, this approach has the extra benefit of letting multiple values be decided in each round.

To minimize the number of messages exchanged, instead of letting every agent propose in every fast round, one agent may be selected from each group to aggregate the groups' proposals and forward them to the acceptors. Such a fast round with aggregation is, in fact, a multicoordinated round in which each agent aggregating proposals is a coordquorum. Observe that, in this case, the coordquorum requirement (Assumption 4, on page 34) is not satisfied.

When comparing single-coordinated and fast rounds, we see that the cost for letting possibly conflicting values be proposed in a round is to require more acceptors be available. That is, we require Assumption 3 to be true. The same is valid for multicoordinated classic and multicoordinated fast rounds, as the one described in the previous paragraph. That is, either the coordquorum or the fast-quorum requirement can be relaxed, but not both at once, unless the absence of conflicts is guaranteed in some other way.

Collision-Fast Rounds

Guaranteeing that proposals do not conflict is exactly what the Collision-Fast Paxos protocol [Schmidt et al., 2007] does to ensure that, in the absence of failures and message loss, agents reach agreement and learn the decision in two communication steps. In Collision-Fast Paxos, CFPaxos for short, agents agree not on a single value but on mapping from each proposer to its proposed value, allowing the protocol to void any conflicts since each proposal is mapped from a different proposer. If commands are not commutable, then once the mapping is learned, a deterministic function may be applied to transform it into a sequence. Another very interesting aspect of CFPaxos is that, in the absence of failures, the more parallel proposals there are, the smaller the relative cost for each one to be learned because all proposals are learned in parallel.

Disjoint Coordquorums

In this chapter we redefine coordquorums to be overlapping inside a group but disjoint across them, and apply these disjoint coordquorums to obtain an efficient an highly available agreement protocol for groups of agents. Inside each group,

proposers to send their proposals to local coordinators that forward proposals to the acceptors in A. To minimize the overhead of having multiple coordinators in each group, we improve our protocol to harness multicast technologies to let agents in one group address other groups as a whole, with single messages, and become oblivious to their contents. In order to avoid unnecessary round changes, we further improve our protocol to exchange coordinators inside a group without starting new rounds, as long as a majority of the coordinators inside a group are alive. Our protocols are based on CFPaxos and their rounds are roughly equivalent to the multicoordinated fast rounds that we described above, but in which conflicts do not happen.

In the next sections we first overview the Collision-Fast Paxos protocol and then present our extended protocols. In Section 4.4 we show how CFPaxos rounds, and of our extended protocols, may be added as another mode into Multicoordinated Paxos and present the respective c-struct set.

4.2 Collision-Fast Paxos

In CFPaxos learners must eventually learn a mapping from all proposers to the values in that they have proposed or to a special value Nil. Ideally, a proposer would be mapped to Nil only if it does not have any proposal to make. Nonetheless, due to the asynchrony of the system and false failure suspicions, proposers can be mapped to Nil also if suspected to have crashed during the execution of the protocol. The mapping is a special data structure called value mappings [Schmidt et al., 2007], or v-maps. These mappings are also what names the problem solved by CFPaxos, that is, the Mapping Consensus problem [Schmidt et al., 2007], or M-Consensus for short. We explain the v-maps and the M-Consensus problem in the next two sections.

4.2.1 Value Mapping Sets

A *Value Mapping Set VMap* is defined in terms of sets $Domain$ and $Value$ as the set of all surjective mappings from subsets of $Domain$ to values on $Value$ or to $Nil \notin Value$. That is, a value mapping $(v : D \rightarrow R) \in$ *VMap* has as domain $D \subseteq Domain$ and as range $R \subseteq Value \cup \{Nil\}$, such that all values in the range are mapped from some value in the domain. Let $\perp$ be a mapping with empty domain and range. $\perp$ is clearly an element of every value mapping set.

We represent a v-map $v : \{a\} \rightarrow \{b\}$, with domain and range of cardinality one, simply as $\{a \mapsto b\}$. We refer to such v-maps as s-maps, short for "single maps", and define the $\bullet$ operator that extends a v-map with an s-map. Formally, $\bullet$ is defined as follows, where v is a v-map, s is an s-map, and $Dom(v)$ is the domain of v-map v:

$v \bullet s = w$, such that

- $Dom(w) = Dom(v) \cup Dom(s)$,
- $\forall e \in Dom(v), w(e) = v(e)$, and
- $\forall e \in Dom(s) \setminus Dom(v), w(e) = s(e)$.

Although described for a v-map and an s-map, the $\bullet$ operator naturally works for any two v-maps. One can think of extending a v-map v with another v-map w as the recursive extension of v with the s-maps that form w. We say that v-map v is a prefix of v-map w and that w extends v ($v \sqsubseteq w$) iff there exists a v-map σ such that $v \bullet \sigma = w$. Hence, $\sqsubseteq$ is a partial order relation on v-map sets.

Other than the aforementioned single maps, two other v-maps are of special interest: *complete* and *trivial*. Complete v-maps are those whose domain equals the respective $Domain$ set; complete v-maps cannot be extended, hence the name. Trivial v-maps are complete v-maps whose ranges equal $\{Nil\}$.

A v-map set defined by $Domain$ and $Value$ is, in fact, a c-struct set defined by the tuple $\langle Cmd, \bot, \bullet \rangle$ where Cmd is the set of all s-maps in $Domain \times (Value \cup Nil)$. That is: $Cmd = \{\{d \mapsto v\} : d \in Domain \wedge v \in Value \cup \{Nil\}\}$

Hence, lower and upper bounds as well as least upper bounds and greatest lower bounds are naturally defined for v-map sets. As for any set of c-structs, the existence of the lub $\sqcup V$ of a set of v-maps V depends on V being compatible. A set V of v-maps is compatible iff for every pair of v-maps $v, w \in V$, for all elements $e \in Dom(v) \cap Dom(w), v(e) = w(e)$.

4.2.2 M-Consensus

The M-Consensus problem is formalized in terms of proposers, acceptors, learners, and a v-map set whose $Domain$ and $Value$ sets equal the set of proposers and the set of proposable commands, respectively.

As proposers propose commands, learners learn v-maps from proposers to commands or to Nil. Learners may learn different v-maps, but they must be always compatible. A learner can only learn another v-map if it is an extension of the previously learned one. Eventually, all nonfaulty learners must learn the same complete non-trivial v-map. Formally, the properties of M-Consensus are the following, where $learned[l]$ is the v-map learned by learner l, initially $\bot$ [Schmidt et al., 2007]:

Nontriviality For any learner l, $learned[l]$ is always a nontrivial v-map and the range of $learned[l]$ contains only proposed commands.

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-maps is always compatible and has a nontrivial least upper bound.

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.

The specification of M-Consensus is similar to the one given for consensus, in Chapter 2. This happens because the two problems are equivalent: one can solve consensus by deterministically choosing a proposal from the M-Consensus solution, and build a v-map whose range equals the decision of consensus to solve M-Consensus. As a result, all lower-bounds of consensus are valid for M-Consensus, as for example, the Quorum Requirement for asynchronous algorithms (Assumption 1).

4.2.3 Collision-Fast Paxos

CFPaxos [Schmidt et al., 2007] is a Paxos-like M-Consensus protocol. As with other Paxos-like protocols, CFPaxos runs in rounds totally ordered by a relation $\leq$ and associated to a single coordinator agents that may start them. Each round is associated to a subset of the proposers, which are the only ones allowed to send their proposals to the acceptors in that round. Hence, they may have their proposals decided in two communication steps in such a round. We call these proposers the collision-fast proposers of the round. The other proposers use the collision-fast ones as their proxies. As we explain later, making all proposers collision-fast for all rounds would restrict the algorithm's resilience.

To propose a value v in some round, a collision-fast proposer p builds the s-map $\{p \mapsto v\}$ and sends it to the acceptors and the other collision-fast proposers of the round. This is a request to the acceptors to accept the s-map and to inform the other collision-fast proposers that a proposal has been made. When informed of a proposal, a collision-fast proposer q may decide to send its own s-map to the acceptors and the other collision-fast proposers or to abstain from proposing in that round. To abstain, q sends the s-map $\{q \mapsto Nil\}$ directly to the learners.

Acceptors accept s-maps to extend their accepted v-maps. Since they only receive s-maps with non-Nil values from collision-fast proposers, their accepted v-maps always map to a range other than $\{Nil\}$. Every time acceptors extend their accepted v-maps, they notify the learners about the newly accepted v-map.

As learners receive notifications from the acceptors and the Nil valued s-maps from the collision-fast proposers, they can identify which proposers must be mapped to Nil and which have had their proposals already accepted by a quorum of acceptors satisfying Assumption 1. We say that these mappings are *chosen* and are bound to compose the complete v-map which learners will eventually learn. For-

mally, a v-map v has been chosen in a round r iff, for every collision-fast proposer $p \in Dom(v)$, of round r, either

- there is a quorum of acceptors which accepted v-maps mapping p to $v(p)$, or
- p has proposed $\{p \mapsto Nil\}$ in r and $v(p) = Nil$.

In a good run of CFPaxos in which a single collision-fast proposer p proposes a value v, a round executes as follows:

- p proposes $\{p \mapsto v\}$;
- after one communication step, the acceptors and each other collision-fast proposers cp learn about $\{p \mapsto v\}$ and, respectively, accept the value and send the s-map $\{cp \mapsto Nil\}$ to learners.
- Learners receive the messages from the acceptors and collision-fast proposers and learn the v-map decided.

If other collision-fast proposers also have values to propose, then instead of sending $\{cp \mapsto Nil\}$ to learners, they send their valued s-maps to the acceptors and the other collision-fast proposers, as p did. Hence, if collision-fast proposers are synchronized, all values will be learned within two steps. If there are no failures or message losses but they are nor synchronized, then the protocol takes three steps to terminate.

To cope with failures, an elected leader will start a new round with a different set of collision-fast proposers when suspecting that one of the current ones has crashed. To ensure consistency, the leader starts the new round by identifying possibly chosen v-maps and making sure that they are the only possibly chosen v-map in the new round. The procedure is similar to the other Paxos algorithms and, in special, to Generalized and Multicoordinated Paxos: if no v-map may be possibly chosen at a lower-numbered round, the collision-fast proposers of the new round are notified so that they can fast-propose.

For a full description of the algorithm, the interested reader is referred to the original work [Schmidt et al., 2007], in which the authors prove the correctness of the algorithm, show how to implement Atomic Broadcast on top of CFPaxos, and state the assumptions needed to ensure liveness.

4.3 Multicoordination and Collision-Fast Paxos

Collision-Fast Paxos may be used to reach agreement between processes in various scenarios, but it is specially suited for systems that can be organized in groups of

processes. More specifically, consider the scenario depicted by Figure 4.1, on page 74. By assigning one collision-fast proposer to each group, to whom the other proposers in the group send their proposals, CFPaxos can deliver messages from each group to each other in two intergroup communication steps in the absence of failures. In case some collision-fast proposer crashes, the leader simply starts a new round replacing the crashed collision-fast proposer with another proposer in the same group. If a group G is disconnected or its proposers are too unstable to assume the role of collision-fast proposer, then the leader can start rounds without a collision-fast proposer local to G, forcing its regular proposers to send their proposals to coordinators in the other groups or stop proposing.

We extend CFPaxos to tolerate the failure of collision-fast proposers in a group without changing rounds or leaving the proposers "orphan" in situations in which CFPaxos would do so. Our extension consists in using multicoordination inside each group. Because collision-fast proposers execute the tasks that we have associated to coordinators in the previous chapters, namely, forwarding messages from proposers to acceptors, hereinafter we refer to them also as coordinators.

There are two main differences between the original CFPaxos and our multicoordinated version. First, instead of a single coordinator per group each round defines a set of coordinators per group. The set of coordinators of a group implement the collision-fast proposer in the original protocol. To propose in some round, a proposer sends its proposal to all coordinators in its group. The coordinators of the current round then forward the proposals to the acceptors. Second, in the extension, agents agree on a map from groups to proposed values, as opposed to maps from proposers to values as in CFPaxos. That is, coordinators send s-maps of the form $\{G \mapsto v\}$ to the acceptors, where G is a group and v the proposal of some proposer in G (or possibly an aggregation of them).

On the acceptors side the only difference is that acceptors do not accept a proposal unless it has been received from a quorum of coordinators of the respective group for the respective round. More precisely, let G_i be the set of quorums of coordinators of group G in round i, or the G_i-coordquorums. An acceptor a accepts proposal $\{G \mapsto v\}$ in round i if it has received the same proposal from every coordinator in some set $Q \in G_i$-coordquorum. To ensure that no two acceptors accept different mappings for the same group, we require that every two quorums intersect. This requirement is formally stated by Assumption 5.

Assumption 5 (Group Quorum Requirement) *For any round i, if P and Q are G_i-coordquorums, then $P \cap Q \neq \emptyset$.*

In Section 4.3.1 we detail a basic algorithm, BasicMCF, which implements our multicoordinated extension of CFPaxos. This basic algorithm improves the resilience of CFPaxos but may be further enhanced to allow reconfiguration internally to a

group before resorting to a round change. We present this algorithm, which we call ExtendedMCF, in Section 4.3.2.

4.3.1 Basic Algorithm

In BasicMCF, we assume that every proposer and coordinator a knows to which group it belongs. This information is stored in the variable $group[a]$. Proposers keep no other information. A coordinator c has two other variables:

crnd[*c*] The current round of c, initially 0.

cval[*c*] The s-map that c is proposing in round *crnd*[*c*], if already defined, or the special value $none$, otherwise. The value is defined and informed by the coordinator who started the round or left to be defined once a proposal is received from some proposer. Initially, it equals $none$.

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-map 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-map it has learned so far.

learned[l] The v-map currently learned by l; initially $\perp$.

We assume that round numbers are partitioned among the possible coordinators in the protocol, for example, by including the coordinator's unique identifier as part of the round number. Moreover, every round is associated to a set of coordinator quorums. This scheme may be defined as in Section 3.4.5 by having each round number as the sequence $\langle Count, Id, S\rangle$, where $Count$ is an integer, Id is the coordinator of the round, and S is the set of coordinator quorums. Round numbers are compared lexicographically taking only the two first fields into account to define the required total order among them.

Algorithm 13, on the next page, presents the actions of BasicMCF.

Proposing a Command

To propose a value v, proposer p executes action *Propose*(p,v). In the action, p simply sends its proposal to all coordinators that belong to its own group for them to forward the proposal to the acceptors. The proposal is executed out of the context of the phases one and two of the algorithm, which effectively choose a v-map.

Algorithm 13 Basic Implementation Multicoordinated Collision-Fast Paxos

Proposer Actions:

Propose$(p, v) \triangleq$
  pre-conditions:
    $p \in$ *proposers*
  actions:
    send ⟨"propose", v⟩ to all coordinators of group *group*[p]

Phase One:

Phase1a$(c, i) \triangleq$
  pre-conditions:
    c is the coordinator of i
    crnd[c] $< i$
  actions:
    send ⟨"1a", c, i⟩ to *acceptors*

Phase1b$(a, i) \triangleq$
  pre-conditions:
    $a \in$ *acceptors*
    rnd[a] $< i$
    received ⟨"1a", c, i⟩ from coordinator c
  actions:
    rnd[a] $\leftarrow i$
    send ⟨"1b", $a, i, vrnd[a], vval[a]$⟩ to c

Phase Two:

Phase2Start$(c, i) \triangleq$
  pre-conditions:
    c is the coordinator of i
    crnd[c] $< i$
    $\exists Q : Q$ is a quorum **and** $\forall a \in Q$, c received ⟨"1b", a, i, rnd, val⟩
  actions:
    LET $v \triangleq$ *PickValue*(c, Q, i)
    IN
      crnd[c] $\leftarrow i$
      IF $v = \bot$ THEN send ⟨"2S", c, i, v⟩ to $\bigcup_{G \in \Gamma} G_i$-coordquorum
                    IF $c \in \bigcup_{G \in \Gamma} G_i$-coordquorum THEN *cval*[c] $\leftarrow$ *none*
                ELSE send ⟨"2S", c, i, v⟩ to $\bigcup_{G \in \Gamma} G_i$-coordquorum $\cup$ *acceptors*
                    IF $c \in \bigcup_{G \in \Gamma} G_i$-coordquorum THEN *cval*[c] $\leftarrow$ *v*(*group*[c])

PickValue$(c, Q, i) \triangleq$
  LET $k \triangleq$ CHOOSE $r : c$ received ⟨"1b", $a, i, r, \_$⟩ from some $a \in Q$ **and**
                    $\forall a' \in Q, m' =$⟨"1b", $a', i, r', \_$⟩ c received from a': $r' \leq r\}$
      $S \triangleq \{v : c$ received ⟨"1b", a, i, k, v⟩ from $a \in Q\} \setminus \{none\}$
  IN IF $S = \{\}$ THEN $\bot$ ELSE $\sqcup S \bullet \{G \in \Gamma \mapsto Nil\}$

Phase One

To start the phase one of round i, the coordinator c of i executes action *Phase1a*(c, i). In the action, c queries the acceptors about previously accepted v-maps by sending them a $\langle$"1a", $c, i\rangle$ message.

An acceptor a reacts to a $\langle$"1a", $c, i\rangle$ message as follows: if its current round $rnd[a]$ is smaller than i, then a sets $rnd[a]$ to i and replies to c with message $\langle$"1b", $a, i, vrnd[a], vval[a]\rangle$. Through this "1b" message, a promises to c that it will not accept any v-map in any round smaller than i.

Phase Two

The second phase of round i starts when its coordinator c receives "1b" messages from a quorum Q of acceptors for round i. It then executes action *Phase2Start*(c, i) as follows. Initially, c picks a v-map v that must extend any v-map possibly chosen in a round smaller than i; the v-map is picked based on the "1b" messages received from the acceptors in Q through function *PickValue*(c, Q, i), which we explain later. Next, c sends a $\langle$"2S", $c, i, v\rangle$ message to all the coordinators in the round i. The purpose of this message is twofold: first, it lets the coordinators that form the coordquorums of round i know whether they are allowed to propose something (if $v = \bot$) or not; second, it lets acceptors know whether the coordinator who started i has defined a single mapping that they can accept in the round ($v \neq \bot$) or not. If $v \neq \bot$, the message is also sent to the acceptors. If c is also one of the coordinators of i, it sets its $cval[c]$ variable as the other coordinators will do after receiving the "2S" message: to *none* if $v = \bot$ or to $v(group[c])$ otherwise.

When a coordinator c receives message $\langle$"2S", $d, i, v\rangle$ for the first time it executes action *Phase2Prepare*. In the action, c first sets $crnd[c]$ to i and then checks whether v equals $\bot$ or not. If $v \neq \bot$, then it sets $cval[c]$ to $v(group[c])$, which has been determined by the coordinator who started the round. Otherwise, it sets $cval[c]$ to *none*, to indicate that it can still send some s-map to the acceptors by executing action *Phase2a*(c, i).

After having executed action *Phase2Prepare*(c, i), if it is in some coordinator quorum of round i for group $group[c]$, coordinator c may execute action *Phase2a*(c, i) in two situations. The first situation is after receipt of a $\langle$"propose", $w\rangle$ message, in which case c sets $cval[c]$ to w and then sends a $\langle$"2a", $c, i, \{group[c] \mapsto cval[c]\}\rangle$ message to the acceptors and all the other coordinators of round i. The second is after receipt of a $\langle$"2a", _, i, _, $\{G \mapsto w\}\rangle$ message where $\{G \mapsto W\}$ is an s-map from group $G \neq group[p]$ to a non-Nil value w. In this case, c sets $cval[c]$ to Nil and sends message $\langle$"2a", $c, i, \{group[c] \mapsto cval[c]\}\rangle$ directly to the learners.

An acceptor a executes action *Phase2b*(a, i) upon receipt of a $\langle$"2S", $c, i, v\rangle$ mes-

Algorithm 13 BasicMCF (Continued)

40: *Phase2Prepare*$(c, i) \triangleq$
41: **pre-conditions:**
42: $c \in \bigcup_{G \in \Gamma} G_i$-coordquorum
43: *crnd*$[c] < i$
44: c received $\langle$"2S", $c, i, v\rangle$
45: **actions:**
46: *crnd*$[c] \leftarrow i$
47: IF $v = \bot$ THEN *cval*$[c] \leftarrow$ *none* ELSE *cval*$[c] \leftarrow v$(*group*$[c]$)
48: *Phase2a*$(c, i) \triangleq$
49: **pre-conditions:**
50: $c \in \bigcup_{G \in \Gamma} G_i$-coordquorum
51: *crnd*$[c] = i \wedge$ *cval*$[c] =$ *none*
52: $\exists v : (c$ received $\langle$"propose", $v\rangle) \vee$
(received $\langle$"2a", $d, i, \{G \mapsto w\}\rangle$: $G \neq group[c] \wedge w \neq Nil = v$)
53: **actions:**
54: *cval*$[c] \leftarrow \{G \mapsto v\}$
55: IF $v = Nil$ THEN send $\langle$"2a", $c, crnd[c], cval[c]\rangle$ to *learners*
56: ELSE send $\langle$"2a", $c, crnd[c], cval[c]\rangle$ to $\bigcup_{G \in \Gamma} G_i$-coordinators $\cup acceptors$
57: *Phase2b*$(a, i) \triangleq$
58: **pre-conditions:**
59: $a \in$ *acceptors*
60: *rnd*$[a] \leq i$
61: (a received $\langle$"2S", $c, i, v\rangle$: $v \neq \bot \wedge (vnrd[a] < i \vee vval[a] = none)) \vee$
($\exists L, v \neq Nil : L$ is an G_i-coordquorum $\wedge \forall c \in L$: a received $\langle$"2a", $c, i, \{G \mapsto v\}\rangle$)
62: **actions:**
63: IF a received $\langle$"2S", $c, i, v\rangle$: $v \neq \bot \wedge (vnrd[a] < i \vee vval[a] = none)$
64: THEN *vval*$[a] \leftarrow v$
65: ELSE IF $\exists L, v \neq Nil : (L$ is an G_i-coordquorum) $\wedge$ ($\forall c \in L$: a received $\langle$"2a", $c, i, \{G \mapsto v\}\rangle$)
$\wedge$ ($vnrd[a] < i \vee vval[a] = none$)
66: THEN $vval[a] \leftarrow \{G \mapsto v\} \bullet \{G \in \Gamma \mapsto Nil\}$
67: ELSE $vval[a] \leftarrow vval[a] \bullet \{G \mapsto v\}$
68: $rnd[a] \leftarrow vrnd[a] \leftarrow i$
69: send $\langle$"2b", $a, i, vval[a]\rangle$ to *learners*
70: *Learn*$(l) \triangleq$
71: **pre-conditions:**
72: $l \in$ *learners*
73: $\exists Q : Q$ is a quorum **and** $\forall a \in Q$, l received $\langle$"2b", $a, i, val\rangle$
74: $\exists \gamma \in \Gamma : \forall G \in \gamma : \exists P : P$ is a G_i-coordquorum **and** $\forall c \in P$, l received $\langle$"2b", $c, i, \{G \mapsto Nil\}\rangle$
75: **actions:**
76: LET *Q2Vals* $\triangleq \{v : l$ received $m = \langle$"2b", $a, i, v\rangle$, $a \in Q\}$
77: IN $learned[l] \leftarrow learned[l] \sqcup (\sqcap Q2bVals \bullet \{G \in \gamma \mapsto Nil\})$

sage for round i from coordinator c, if i is bigger than its current round, $rnd[a]$ or if it equals its current round but a has never accepted any v-map, *i.e.*, $vval[a] = none$. If this is the case, then a accepts the v-map v picked by c. It does so by setting $rnd[a]$ and $vrnd[a]$ to i and $vval[a]$ to v.

Acceptors may also execute action *Phase2b*(a, i) after receiving a $\langle$"2a", $c, i, \{G \mapsto v\}\rangle$ message from every coordinator c in some G_i-coordquorum L, where $v \neq Nil$. Observe that such a condition implies that the coordinator who started round i has picked an empty v-map in action *Phase2Prepare* and informed the coordinators of i with "2S" messages. Hence, the "2a" messages received from the coordinators in L actually convey the information in the "2S" not received. When first executing action *Phase2b* in this case, a has not accepted any v-map yet. Hence, besides setting $rnd[a]$ and $vrnd[a]$ to i, a it sets $vval[a]$ to $\{G \mapsto v\}$ extended with $\{H \mapsto Nil\}$ for every group H with no H_i-coordquorum. That is, it records the fact that it will not receive any "2a" messages from coordinators in H during round i. On the following times that a executes *Phase2b*(a, i), it simply extends $vval[a]$ with $\{G \mapsto v\}$, that is, it sets $vval[a]$ to $vval[a] \bullet \{G \mapsto v\}$. The action finishes when a sends message $\langle$"2b", $a, i, vval[a]\rangle$ to all learners, with the updated value of $vval[a]$.

Because a coordinator can only pick a single v-map when starting the phase two in action *Phase2Start*(c, i), the two conditions to execute action *Phase2b* are mutually exclusive. That is, there are no two acceptors such that one executes action *Phase2b* due to receipt of "2S" messages and another that executes it due to receipt of "2a" messages for the same round i. Either both accept the same complete v-map v sent by the coordinator, or they accept s-maps received from quorums of coordinators. Since no coordinator can send different "2a" messages for the same round and all quorums of the same group for round i intersect, by Assumption 5, there is only one s-map per group that may be accepted. Since no accepted s-map conflicts, all v-maps accepted by must be compatible. This property is explored in picking a v-map in function $PickValue(c, Q, i)$, explained next.

Picking a v-map

Let k be the highest $vrnd$ in the "1b" messages that c received from acceptors in Q in round i, and let S be the set of all v-maps received in the "1b" messages with field $vrnd$ equal to k, not including *none*. If S is empty, then no v-map has been or might be chosen at a lower-numbered round and the function returns $\bot$.

If S is not empty, then some v-map has been or might still be chosen at a round lower than or equal to k. In this case, the function evaluates to $\sqcup S$ extended with $\{d \mapsto Nil\}$ for every coordinator d. Because the coordinator of round k must have picked a v-map that extended v-maps possibly chosen on rounds smaller than k and because any v-map chosen in k must be in S due to Assumption 1, the v-maps in S are extensions of any v-map possibly chosen in a round smaller then k. Moreover,

as we explained in the previous paragraphs, acceptors do not accept conflicting s-maps in the same round and, therefore, S is compatible and $\sqcup S$ extends any v-map possibly chosen at k. In addition, because acceptors only accept non-Nil v-maps, $\sqcup S$ is also non-Nil.

Learning a Value

Learning a v-map happens in action *Learn*. The action is enabled for a learner l once it has received "2b" messages for some round i from a quorum Q of acceptors and message $\langle$"2a", $c, r, \{G \mapsto Nil\}\rangle$ from every coordinator c in some G_i-coordquorum P for every group G in a (possibly empty) subset $\gamma \in \Gamma$ of the groups that have coordinators in round i. In this case, l calculates the lub of the chosen v-maps based on the received information in order to update its currently learned v-map. Let *Q2bVals* be the set of all v-maps received in the "2b" messages for round i from acceptors in Q. The glb of *Q2bVals* is the chosen v-map identifiable from the messages in *Q2bVals*. Hence, the action sets $learned[l]$ to $learned[l] \sqcup (\sqcap(Q2bVals \bullet \{G \in \gamma \mapsto Nil\}))$.

4.3.2 Adding Intra-group Reconfiguration

Although reconfiguration happens less often in BasicMCF than in the original CF-Paxos, it may still happen if coordinators inside groups are unreliable. In the following we discuss how to extend BasicMCF in such a way that, in certain situations, failures are handled inside groups without changing rounds. This extended algorithm, that we call ExtendedMCF, makes two extra assumptions.

The first assumption is that messages may be addressed to the coordinators inside a group obliviously to the group's actual membership. In practice, this feature is available through multicast protocols such as IP multicast. In IP multicast, processes subscribe to a group by registering at the closest multicast router. The router then informs other routers that it has subscribers for the group, but does not have to inform which or how many. The second assumption is that the coordinators of a group have access to a consensus oracle running inside that group.

In general lines, the idea in ExtendedMCF is that inter-group message exchange addresses all coordinators of a group instead of specific ones. Inside each group, agents agree on exactly which coordinators should actually form coord-quorums for the group. As the protocol proceeds, the selected coordinators may be replaced to cope with alleged failures, as long as care is taken to avoid inconsistent decisions. That is, if a mapping from the group to some value has been possibly learned, the group can only confirm the value; they cannot propose another. Acceptors and learners, which receive messages from the coordinators, can be informed on-the-fly about the composition of coord-quorums. To allow them to distinct different

coord-quorum versions for the same round, coordinators add a version number to their messages; acceptors and learners only consider the messages with the highest version numbers. The following paragraphs details these procedures.

When the leader starts the second phase of a round i, it multicasts the "2S" message to the coordinators of each group containing coordquorums. When the coordinators of some group G hear about round i, by receiving the respective "2S" message, they run consensus to decide on the G_i-coordquorums. A deterministic function over the decision assigns unique identifiers to each coordinator in a quorum, which are used in the "2a" messages sent outside the group.

Once coordinators agree on the G_i-coordquorums, they start monitoring the selected coordinators for failures. If they suspect that more failures than some configurable threshold have happened, then they try an internal reconfiguration. That is, the coordinators execute the following steps:

- The suspicious coordinators broadcast a request to all coordinators in G_i-coordquorum not to send any other "2a" message outside the group while a new set of coordquorums is agreed upon.
- Coordinators reply to this request positively if they have not sent any "2a" message yet. If they have already sent such a message, then they reply negatively. Both replies are sent to all coordinators in the group.
- All coordinators gather the replies for some time and then propose them in a consensus instance.

The outcome of the consensus instance is then used to identify one of the following situations and define the appropriate line of action.

- For every current coordquorum, there is at least one coordinator that has not sent any "2a" messages in the current round and that abides by the request for not doing so. In this case, no mapping from the group to a value was possibly chosen in the round. Coordinators then deterministically choose a new set of coordquorums and continue with the round.
- There is at least one G_i-coordquorum P for which every coordinator in P has already sent a "2a" message in round i with the same s-map. Therefore, the s-map may have been already chosen by the acceptors. Moreover, due to Assumption 5, this is the only s-map possibly chosen. In this case, the coordinators deterministically choose replacements for the suspected ones, but with the condition that the already proposed s-map will be their proposal.
- Coordinators cannot determine whether any of the previous situations hold. In this case they cannot do anything but wait for an external reconfiguration (round change).

In the first alternative, in which the set of coordquorums of the group is changed and could propose different s-maps, it might happen that "2S" messages from before and after the change combine to have different s-maps accepted, for the same group. To avoid such a case, "2S" messages are extended with a coordquorum version number. Acceptors and learners are then modified to consider messages only from coordquorums with the same version number.

In the third case, if no coordquorum is functional to ensure liveness, the coordinator of the round will eventually give up on waiting for a decision and start a higher-numbered round, in which the problematic group will be initially mapped to Nil. This is equivalent to suspecting a coordinator in the original CFPaxos protocol.

4.3.3 Correctness and Liveness

The correctness and liveness properties of CFPaxos were formally proven when the protocol was introduced [Schmidt et al., 2007]. Proving the same properties for BasicMCF and ExtendedMCF is a matter of reducing these protocols to CFPaxos. The reduction is rather simple, as we now show.

To understand the reduction, observe that the idea behind BasicMCF and ExtendedMCF is to use a set of coordinators forming overlapping quorums to implement each collision-fast proposer. Hence, an action of a collision-fast proposer in the original protocol is implemented by executing the same action in at least a quorum of the correspondent coordinators in our protocols. In the following discussion, we refer to each action A of BasicMCF as $\overline{A}$, to differentiate it from its homonym in CFPaxos. ExtendecMCF is discussed afterwards.

The original *Propose* action and our extended version $\overline{Propose}$ differ only in the number of messages sent by the proposer. Instead of sending a single message to a collision-fast proposer, the proposer sends the message to all coordinators in its group. In fact, even in the original protocol, it is up to the proposer to choose to which coordinators to send its proposals and this choice does not affect the correctness of the protocol, although it could affect liveness. Hence, $\overline{Propose}$ implements *Propose*. The other actions of phase one, *Phase1a* and *Phase1b*, are exactly the same in both protocols and, hence, implementations of each other.

Action $\overline{Phase2Start(c,i)}$ implements action $Phase2Start(c,i)$ by performing the same sub-actions under the same pre-conditions. More than that, $\overline{Phase2Start(c,i)}$ also implements a special case of action $Phase2Prepare(c,i)$ if c is in some coordquorum for round i. To avoid having a flag variable to indicate this special condition in $\overline{Phase2Prepare(c,i)}$ as in CFPaxos, we added the sub-actions to $\overline{Phase2Start(c,i)}$.

$\overline{Phase2Prepare(c,i)}$, where c is not the creator of i, differs from $Phase2Prepare(c,i)$ only in that in the first c must be in some G_i-coordquorum for some group G, while in the last c must be in the list of collision-fast proposers of round i. As we men-

tioned above, for each collision-fast proposer g in CFPaxos, there is a corresponding set of coordinators in G forming coordquorums to implement g. Hence, action *Phase2Prepare*(g,i) is implemented by the compositions of actions $\overline{\textit{Phase2Prepare}(c,i)}$ for every coordinator c in some G_i-coordquorum and due to the reception of the same "2S" message (and $\overline{\textit{Phase2Start}(c,i)}$, if c is the creator of round i). On the one hand, since all G_i-coordquorums overlap (Assumption 5), no two coordquorums will succeed in executing the action with different "2S" messages and, therefore, action *Phase2Prepare*(c,i) will be simulated only once. On the other hand, if no G_i-coordquorum execute the action, then no *Phase2Prepare*(c,i) will be simulated, which does not violate the safety of the protocol. Exactly the same analysis is valid for actions *Phase2a*(c,i) and $\overline{\textit{Phase2a}(c,i)}$.

Action $\overline{\textit{Phase2b}(a,i)}$ differs from *Phase2b*(a,i) in the sense that the latter requires a single message $\langle$"2a", $c, i, \{c \mapsto v\}\rangle$ from a collision-fast proposers c to accept an s-map $\{c \mapsto v\}$, while the first requires similar messages from a quorum of the coordinators implementing c. Because coordquorums overlap, the concatenation of the actions of receiving a similar message from each coordinator in a coordquorum implements the reception of a single message with the same contents from a collision-fast proposer in the original protocol. Hence, $\overline{\textit{Phase2b}(a,i)}$ in fact implements action *Phase2a*(a,i). With a similar argument we conclude that action $\overline{\textit{Learn}(l)}$ implements *Learn*(l).

It is straightforward to see that ExtendedMCF implements BasicMCF since each action in BasicMCF is executed in the same way in ExtendendMCF or under more pre-conditions. In particular, the actions that use "2a" messages, *Phase2b*(a,i) and *Learn*(l) require all such messages to have the same version number in order do be executed. Since version numbers are only incremented if there is no possibility that *Phase2b* and *Learn* had been executed, it does not happen that the actions execute twice for the same round, due to the messages sent by the same coordquorum.

To ensure liveness, CFPaxos must be modified in the same way that we did for Multicoordinated Consensus in Section 3.4.7. That is, the algorithm must have round creation limited to a leader coordinator, elected through some leader election oracle. Moreover, to ensure that a learner l eventually learns a complete v-map, CFPaxos requires that (i)l, a proposer p, a coordinator c, a quorum of acceptors Q, and a set S of collision-fast proposers remain alive from some point on, (ii) a subset of S is trusted by c from some point on, and (iii) p issues a proposal. These conditions are easily adapted to BasicMCF by replacing the set S by a set of coordquorums. ExtendedMCF requires the extra condition that coordinators in the groups with coordquorums in S stop suspecting these coordquorums and, therefore, do not change their versions from some point in time on.

4.4 Generalizing Collision-Fast Rounds

As we have previously noted in this chapter, v-maps are a special kind of c-struct and, therefore, could be used in any Generalized Consensus Protocol. What makes v-maps special is the way they are used in CFPaxos and its extensions. That is, by guaranteeing that no conflicting s-maps are proposed in the same round, it is possible to agree on a v-map in a fast round without Assumption 3. The same property may be explored in a Generalized Consensus protocol. In fact, collision-fast rounds could be integrated into Multicoordinated Paxos as an adaptation of multicoordinated rounds. To fit in Multicoordinated Paxos, which may be used to agree on an ever growing c-struct while trying to avoid round changes, the collision-fast round would have to allow agreeing not simply on a mapping from coordinator/group to a command, but to a growing sequence of commands and/or Nil.

To incorporate collision-fast rounds into Multicoordinated Paxos, the following behavior is required from the agents.

- When starting a collision-fast round, the coordinator of the round picks a c-struct and appends a special command $chkpt$ to its end. The command must conflict with all the other commands. The extended c-struct is then sent to the acceptors and the coordinators of the round. (If the $\bullet$ operator does not accept appending the $chkpt$ command, then collision-fast rounds cannot be used.)

- After receiving a c-struct v terminated with $chkpt$, a coordinator c can propose extensions to v of the form $v \bullet \{G \mapsto \sigma\}$, where G is the group that c belongs to, σ has the form $\langle\langle 1, C_x\rangle, \ldots, \langle n, C_x\rangle\rangle$, and C_x are commands received from proposers or Nil. The extensions are forwarded to acceptors, learners, and other coordinators in the round. Whenever c receives a c-struct $w \bullet chkpt \bullet \{H \mapsto \pi\}$, from coordinator in group $H \neq G$ during a collision-fast round, it extends its own proposal to have at least as many commands as π and proposes it again.

- When accepting a c-struct, acceptors follow the rules presented next. Two mappings $\{G \mapsto \sigma\}$ and $\{H \mapsto \pi\}$ do not commute iff $G = H$. Hence, given two c-structs $w \bullet chkpt \bullet \{G \mapsto \sigma\}$ and $w \bullet chkpt \bullet \{H \mapsto \pi\}$:
  - If $G \neq H$, then their lub equals $w \bullet chkpt \bullet \{G \mapsto \sigma\} \bullet \{H \mapsto \pi\}$;
  - If $G = H$, then if σ is a prefix of π or vice versa, then the lub equals $w \bullet chkpt \bullet \{H \mapsto \delta\}$ where δ is the longest of the sequences. If they are not prefixes of each other, then the two c-structs are not compatible;
  - If $G \neq H$, then their glb equals $w \bullet chkpt$.
  - If $G = H$, then the glb equals $w \bullet chkpt \bullet \{H \mapsto \delta\}$, where δ is the longest common prefix of σ and π.

- Learners learn both from the c-structs received from acceptors and directly from coordinators. C-structs received from coordinators are used only do identify Nil commands in the mapped sequence without having to wait for it to be accepted by an acceptor.

4.5 Final Remarks and Related Work

In this chapter we have discussed the issue of efficiently reaching agreement in a network organized as groups of agents. This organization reflects, for example, spread apart data centers connected through dedicated links or the internet, or grid networks formed by several clusters at different locations. In this scenario, one would like to minimize the need for a node in one group (cluster, or data center) to communicate with nodes in other groups.

We have presented an improved version of the Collision-Fast Paxos protocol of Schmidt *et al.* [Schmidt et al., 2007], which lets agents in all groups learn the agreed proposals from all the other groups in two communication steps. Our improvements to the protocol make it resilient to the failures of coordinators inside each group and make agents oblivious to the membership of other groups.

Structuring agents in groups has also been considered in the work of Kooh and Haddad [Kooh and Haddad, 1999]. Their hierarchical consensus algorithm recursively agrees on proposals over a multilevel tree of agents. More specifically, in their protocol each set of agents in a given level of the same branch of the tree constitute a group. From the leaves to the root, agents in the same group agree on a value to be their proposal on the upper level, until they have agreed on a single value at the root of the tree.

There are two main differences between Kooh and Haddad's work and ours. First, our protocol is better suited for applications that need to agree on *all* proposals, not just a single one. This characteristic is inherited from CFPaxos. Second, in Kooh and Haddad's protocol, agents are replicated using consensus: an instance is used to agree on each state change. In our approach, we let the coordinators in each quorum diverge and only use consensus to recover from failures. The price we pay is in terms of messages sent from the group—since each coordinator may be in a different state, we must have all of them communicating with the acceptors.

Other works have considered reaching agreement over wide area networks but focusing on the delay aspect, not on the topology of the network [Sousa et al., 2002, Vicente and Rodrigues, 2002]. That is, they considered a flat group of agents connected by heterogeneous links, some to nearby and some to far nodes. These protocols may reach agreement in two long link message steps, but because they rely on spontaneous ordering on these links, they will be inefficient when this assumption does not hold. CFPaxos ensures agreement in two steps in the absence of failures,

irrespective to the ordering of messages. Although our extension to CFPaxos does rely on spontaneous ordering, it does so only inside groups, where we expect this property to hold often [Pedone and Schiper, 1999].

Chapter 5

Log Service for Transaction Termination

5.1 Log Service

The need to atomically commit transactions in distributed management systems is recurrent. Briefly, to terminate a distributed transaction, each participating resource manager votes to either commit or abort it. The transaction outcome is then determined based on these votes: commit, if all resource managers vote to commit the transaction, or abort, otherwise. If the vote from each participant is always necessary, however, the procedure may block in the absence of some resource manager. To avoid this scenario, a weaker version of atomic commitment allows the outcome to be abort if some participant is suspected to have failed. Commit will be guaranteed only if all participants vote to commit the transaction and none is suspected. This weaker problem is known as *non-blocking atomic commitment*.

Besides ensuring atomicity, a transaction termination protocol should also guarantee the durability of committed transactions. Once committed, the changes done by a transaction should not be forgotten despite failures. In conventional protocols this is achieved by having each resource manager stores its updates in a local stable media before voting. Should a resource manager fail, it can, at recovery time, read the updates from the local storage and replay them to recover its previous state. A drawback of this approach is that it couples the availability of the resource manager with the availability of the server hosting it. In a clustered environment, for example, one could recover a resource manager on a different node, should its current host fail. But obviously, this can only be done if the state of the resource manager is not stored on the crashed node.

In this chapter, we propose abstracting transaction termination in terms of a simple but powerful *log service*. Once the termination is triggered, the transaction participants submit their votes to the log service using a simple interface. Then they simply wait until they learn a decision or until they suspect that some other resource

manager will not vote. In this case, they simply vote to abort the transaction on the behalf of the suspected resource manager; the service will ignore all but the first vote received for each resource manager to determine the transaction's outcome, properly ensuring consistency. In addition to storing votes, the service may also durably store transaction updates, allowing resource managers to be restored after a crash without relying on their local storage. Besides specifying the service, we present two implementations of the log service, namely *coordinated* and *uncoordinated*, both relying on consensus to provide high availability, although in different ways. The two approaches abstract the tradeoff "message complexity versus number of communication steps" in atomic commit protocols, as we discuss in the next section.

Transaction Termination

Gray and Lamport proposed in a recent work an interesting non-blocking atomic commitment protocol called Paxos Commit [Gray and Lamport, 2006]. In Paxos Commit, each resource manager uses an instance of Classic Paxos [Lamport, 1998] to cast its vote, hence its name. By using consensus to vote, Paxos Commit detaches the termination protocol from the availability of resource managers and transaction manager (the process that triggers and coordinates the protocol). Hence, failures are handled by conservatively voting abort on behalf of (supposedly) crashed resource managers. Any suspicious resource manager can do so by proposing "abort" in the consensus in which the suspected resource manager was supposed to cast its vote, since consensus ensures that all involved parts agree on a single vote per resource manager in spite of possibly different proposals. Paxos Commit has the same expected latency as the well-known *Two-Phase Commit* (2PC) protocol. In fact, it is a generalization of 2PC that tolerates failures not only of resource managers, but also of the transaction manager. Compared to *Three-Phase Commit* (3PC) [Bernstein et al., 1987], Paxos Commit has fewer communication steps and is simpler in case of a coordinator failure.

Besides atomicity, transactions must also be durable if they are used to enforce strong consistency of applications. That is, once a transaction is committed, its updates must always be reflected subsequent states of the system, irrespectively to failures. In the same way that Paxos Commit detached the termination procedure from the resource managers, we propose to detach the durability. That is, by moving the information needed for recovery out of resource managers and into the log service, whose availability can be tuned according to the application needs.

Defining transaction termination in terms of our log service has two strong advantages. First, transaction termination becomes oblivious to particularities of the system, taken care of or explored by the log service implementation in a transparent way. For example, the service could be implemented using message passing, as we

illustrate in this thesis, or shared memory, possibly better suited for a multiprocessor environment. Moreover, if enough nodes can be relied upon not to crash simultaneously, then the service could be implemented in main memory only, removing disk access times from the termination of transactions. Second, the overall availability of resource managers is improved by using a highly available implementation of the log service. As mentioned earlier, the service can be used to migrate crashed or slow resource managers to functional and more dependable hosts. As a consequence, resource managers may choose to asynchronously store their state locally for later recovery or rely solely on the state kept at the log service.

Service Implementations

The two implementations of the log service that we present here rely on consensus to terminate transactions atomically and to durably store the updates of committed transactions. In the first, *uncoordinated*, voting is completely distributed (this approach abstracts Paxos Commit). In the second, *coordinated*, voting is centralized, managed by a coordinating process. This difference has performance implications on both the termination of transactions and on the recovery of resource managers. In the following sections we provide and in-depth presentation of both approaches. As we mentioned before, the two approaches abstract the tradeoff "message complexity versus number of communication steps" in atomic commit protocols. Our coordinated implementation has linear message complexity, similarly to 3PC, but needs 5 steps to terminate a transaction; the uncoordinated approach, similarly to Paxos Commit and other proposals (for example, [Guerraoui et al., 1996, Jiménez-Peris et al., 2001]), reduces the number of steps to 3, at the expense of a quadratic message complexity.

We have experimentally evaluated the two implementations and verified, in the tested scenarios, that the coordinated solution outperforms the uncoordinated one by 8x when transactions are short, what typically happens in performance-critical systems. The performance gain is mainly due to batching of concurrently issued votes by the coordinator, saving on network and disk utilization.

In the following sections we formally define non-blocking atomic commitment and the durability property. In Section 5.3 we present the log service and show how it is used to terminate transactions. In Section 5.4 we discuss the building blocks used in the implementations, presented in Sections 5.5 and 5.6. We compare the two implementations among them and to other solutions in Section 5.7, and conclude the chapter with some final remarks in Section 5.8. Complete specs and correctness proofs can be found in the Appendix B.

5.2 Problem statement

A *resource manager* (RM) is the owner of some resource that can be read or written, such as a file, a region of memory, or a table in a relational database. We define RMs not as agents, but as roles that can be "incarnated" by different processes at different points in time. We assume the availability of infinitely many agents and, therefore, that there is always an agent to incarnate an RM. This model allows an RM to be incarnated at an operational host, should its previous host crash, without waiting for the crashed node to recover. Hence, RMs have crash-recovery failure pattern.

A distributed transaction is a partially ordered set of read and write operations executed by RMs on their resources. An RM is a *participant* of transactions for which it has been requested to execute operations. Each transaction is managed by a *transaction manager* (TM), another role in our model. To terminate a transaction, the TM asks its participants to agree on committing or aborting the transaction through a non-blocking atomic commit (NBAC) protocol. An RM can only vote to commit a transaction after receiving a request from the TM along with the complete list of participants of the transaction. Abort votes, however, may be sent by RMs at any time, even before receiving the request and the participant's list.

Formally, NBAC is defined by the following properties.

Validity If a participant decides to commit a transaction, then all participants voted to commit the transaction.

Agreement No two participants decide differently.

Nontriviality If all participants vote to commit the transaction and none is suspected of failing throughout the execution of the protocol, then the decision is commit.

Termination All non-faulty participants eventually decide.

Nontriviality implies that RMs can be "suspected" of failing. The only assumption we make about failure detection is that if an RM fails (actually, the agent incarnating it), then it will eventually be suspected by the other agents. This property is similar to the *eventual weak completeness* property of Chandra and Toueg's failure detectors [Chandra and Toueg, 1996], discussed in Section 2.1. Therefore, RMs may be incorrectly suspected to have failed and transactions unnecessarily aborted. TM failures are handled with RM unilateral aborts.

NBAC defines proper transaction termination, but says nothing about durability, *i.e.*, on making the effects of committed transaction last in spite of crashes of RMs. We define *durability* as follows [Silberschatz et al., 2001]:

Durability After a transaction commits, the changes it has made to the database persist, even in the presence of agent failures.

Durability can be ensured, for example, by reinitializing the database and replaying the updates of committed transaction in the commit order. RMs can store updates on a local stable storage, as traditionally done in database systems, or on an external and possibly replicated media to improve availability.

5.3 The Log Service

In this section we specify the log service and show how resource managers interact with it to atomically terminate transactions and recover crashed resource managers. We show how these behaviors can be implemented in a shared-nothing asynchronous distributed system in Sections 5.4–5.6. For simplicity we assume that, except for action *Terminate* of Algorithm 14, all actions presented in the chapter are performed atomically. The extended specification that we give in Appendix B is defined with atomic actions only.

5.3.1 The Log Service Specification

It is easier to understand the log service specification by seeing it as an always available service, accessed by the RMs through remote calls. The service manipulates the following five data structures:

V The set of all votes received by the service. Votes are sequences of the form $\langle rm, t, tset, vote, update\rangle$, read as "$rm$ voted $vote$ on transaction t, with updates $update$, if any (*i.e.*, the empty set, if $vote = \text{ABORT}$)". $tset$ is the possibly incomplete list of RMs involved in t. Initially, $V = \{\}$.

T The partially ordered set $(S, \preceq)$, where S is the set of committed transactions and $\preceq$ is a partial order on S. For any two transactions $t_1, t_2 \in S$ whose commits are not seen as overlapping by the log service, either $t_1 \prec t_2$ or $t_2 \prec t_1$; we say that such transactions are "non-concurrent". For simplicity we represent T as a sequence of sets of committed transactions such that, given two sets $T[i]$ and $T[j]$ in T, $i < j$ implies that for all transactions $t \in T[i]$ and $u \in T[j]$, $t \preceq u$. Initially, $T = \langle\rangle$.

C The set of committing transactions, that is, the set of non-terminated transactions for which votes have been issued. Initially, $C = \{\}$.

$LastC$ Given the last transaction to have been committed, t, $LastC$ contains the subset of C will all transactions whose votes were received while t was being terminated (their terminations were concurrent to t). Initially, $LastC = \{\}$.

R A map from RMs to the agents currently “incarnating" them. An RM is mapped to $\perp$ (not a valid agent) if it has never been incarnated. Initially, all RMs are mapped to $\perp$.

The first part of Algorithm 14 initializes the data structures and defines the operators *Outcome*, *IsInvolved*, and *Updates*, and the action *Vote*.

Outcome(t) evaluates to the outcome of transaction t: ABORT, if at least one vote for t is ABORT; COMMIT, if all votes are present and equal COMMIT; and UNDEFINED, if all known votes for t are COMMIT but there are missing votes, which could turn out to be of either kind.

IsInvolved(t,*rm*) evaluates to TRUE if rm is a participant of t, *i.e.*, if rm is in the list of participants ($tset$) of any vote for t already received by the service. Because ABORT votes may not carry the complete participants' list, they are not enough to give negative answers. That is, if only ABORT votes are known and none has rm in the $tset$ field, then *IsInvolved*(t,*rm*) evaluates to UNKNOWN. On the contrary, if a COMMIT vote is known and rm is not in the $tset$ field, then the function evaluates to FALSE.

Vote(v) adds vote v to V. A vote is added only if no other vote for the same resource manager and transaction is already in V. This ensures that, if conflicting votes are issued by mistake for the same participant, then only one vote per participant is considered.

Updates(*rm*) evaluates to the sequence of sets of updates performed by resource manager rm, partially ordered accordingly to T. The evaluation is done by recursively iterating over the sets of committed transactions to find the ones in which rm took part and with which updates.

5.3.2 Termination and Recovery

The second part of Algorithm 14 defines the actions *Incarnate* and *Terminate*.

Incarnate(*rm*,*pid*) is used by agent pid to incarnate resource manager rm. First, the agent sets $R[rm]$ to its own identifier, pid. Second, it evaluates *Updates*, described above, to get the updates executed by the previous incarnations of rm. Third, pid scans the updates from the first to the last set, applying all updates in one set before those in the next set to its state; updates in the same set do not have to be ordered. The action ends with pid incarnating rm, with a state equal to the

Algorithm 14 Log service specification

```
1:  Initially:
2:    V ← ∅                                          ◁ The history of votes.
3:    T ← ⟨⟩                                         ◁ Sequence of sets of committed trans.
4:    C ← ∅                                          ◁ Set of concurrent trans.
5:    LastC ← ∅                                      ◁ Set of trans. concurrent to the last committed.
6:    ∀r ∈ RM, R[r] ← ⊥                              ◁ Processes incarnating RMs.
7:  Outcome(t) ≜
8:    IF ∃⟨_, t, _, ABORT, _⟩ ∈ V                    ◁ Any ABORTs?
9:     THEN ABORT
10:    ELSE IF ∃⟨_, t, tset, COMMIT, _⟩ ∈ V : ∀p ∈ tset : ⟨p, t, _, COMMIT, _⟩ ∈ V   ◁ All COMMITs?
11:    THEN COMMIT
12:    ELSE UNDEFINED                                 ◁ Not enough commits?
13: IsInvolved(t, rm) ≜
14:   IF ∃⟨_, t, tset, _, _⟩ ∈ V : rm ∈ tset          ◁ Is rm in any list?
15:    THEN TRUE
16:    ELSE IF ∃⟨_, t, _, v, _⟩ ∈ V : v = COMMIT      ◁ Is tset a complete list?
17:    THEN FALSE
18:    ELSE UNKNOWN                                   ◁ Not enough information.
19: Vote(⟨rm, t, tset, vote, update⟩) ≜
20:   IF Outcome(t) = UNDEFINED                       ◁ If t has not terminated yet
21:    THEN C ← C ∪ {t}                               ◁ add it to C.
22:   IF ¬∃⟨rm, t, _, _, _⟩ ∈ V THEN                  ◁ If rm has not voted for t yet
23:     currState ← Outcome(t)                        ◁ The current state (ABORT or UNDEFINED).
24:     V ← V ∪ {⟨rm, t, tset, vote, update⟩}         ◁ Store this vote and check...
25:     IF (currState = UNDEFINED) ∧ (Outcome(t) = COMMIT) THEN   ◁ ... if the state changed.
26:       IF t ∈ LastC                                ◁ If t can be added to the last set...
27:        THEN T ← T ⊕ t                             ◁ ...do it;
28:        ELSE
29:          T ← T • {t}                              ◁ otherwise, add it to a new set
30:          LastC ← C                                ◁ with a new LastC.
31:       C ← C\{t}
32: Updates(rm) ≜
33:   LET Upd(i) ≜
34:     IF i = 0 THEN ⟨⟩                             ◁ Recursion end.
35:      ELSE                                        ◁ Get updates and recurse.
36:       Upd(i − 1) •{upd : ⟨rm, t, _, COMMIT, upd⟩ ∈ V : ∧ t ∈ T[i]   ◁ t committed and
                                                     ∧IsInvolved(rm, t)}   ◁ involves rm.
37:   in Upd(Len(T))                                 ◁ Return updates for rm
```

previous incarnation. pid will accept and process new transactions as rm until it crashes or another agent incarnates rm (*i.e.*, pid no longer equals $R[rm]$). If more than one agent try to incarnate rm, a quick succession of incarnations will happen,

Algorithm 14 Log service specification (continued)

```
38: Incarnate(rm, pid) ≜
39:   R[rm] ← pid
40:   updates ← Updates(rm)                          ◁ Get committed state.
41:   FOR i = 1 to Len(updates)                      ◁ For each set of committed transactions...
42:     apply updates in updates[i]                  ◁ ...apply it to the database.
43: Terminate(rm, t, tset, vote, upd) ≜
44:   Vote(⟨rm, t, tset, vote, upd⟩)                 ◁ Vote for t and wait...
45:   WHILE Outcome(t, rm) = UNDEFINED
46:     WAIT (Outcome(t, rm) ≠ UNDEFINED) ∨ (suspect r ∈ tset)
47:     IF suspected r ∈ tset                        ◁ If suspects that r has crashed...
48:       THEN Vote(⟨r, t, tset, Abort, ∅⟩)          ◁ ... vote on its behalf.
49:   IF Outcome(t, rm) = ABORT THEN abort t in the database
50:     ELSE apply upd to database
```

but only one will remain incarnated.

Terminate is used by resource managers (the process incarnating them) to trigger or join the termination of transactions in which they participate. To terminate t, a resource manager rm executes *Terminate* ($\langle rm, t, tset, vote, upd\rangle$), where $tset$ is the set of resource managers known by rm to be participants of t, and $vote$ is either ABORT or COMMIT, depending on whether rm is willing to commit the transaction or not. If $vote$ equals COMMIT, then upd contains the updates performed by rm in t. If $vote$ equals ABORT, upd is the empty set.

After casting its vote, rm waits until it learns t's outcome. While waiting, rm monitors the other resource managers in $tset$, also involved in t. If rm suspects that some participant crashed, it votes ABORT on its behalf. After learning that t committed, rm will apply its updates and release the related locks, if locking is used. If rm learns that t aborted, it locally aborts the transaction. Updates are made durable by the log service. We assume that the execution of *Terminate* is not necessarily an atomic operation, allowing multiple resource managers to vote in parallel and the same resource manager to terminate distinct transactions in parallel, if its scheduling model allows it.

5.3.3 Correctness

Termination using our log service provably solves NBAC and recovery through the *Incarnate* action provably ensures the durability property. Below we present a sketch of these proofs. In Appendix B we present detailed proofs of these properties.

Fact 1 For any resource manager rm and transaction t, there is at most one element $\langle rm, t, tset, v, u\rangle$ in V, and only if rm is a participant of transaction t.

The first part of Fact 1 is implied by lines 22 and 24 of the algorithm, which for-

bid the inclusion of votes in V for the same pair (transaction,RM). The second part is implied by (i)the definition of *Terminate*, (ii)the assumption that RMs get accurate information, although possibly incomplete, of which RMs participate in a given transaction, and (iii) the fact that RMs only vote in transactions they participate.

Fact 2 If *Outcome*(t) evaluates to COMMIT (respectively, ABORT) at any given time, then it will evaluate to COMMIT (respectively, ABORT) at any later time.

Because votes are not removed from V, once the condition of line 8 is true, it remains true no matter what other votes are added to V. For the same reason, once votes from all participants are received and are all for committing the transaction, the condition of line 10 becomes true and remains like that. Recall that the $tset$ of any commit vote has the complete set of participants of the respective transaction.

Fact 2 implies AC-Validity and Facts 1 and 2 imply AC-Agreement.

If no participant of a transaction is suspected and all vote COMMIT, then only COMMIT votes can be added to V. In this case, the condition to abort the transaction, on line 8, will never be true and COMMIT is the only possible outcome for such a transaction, satisfying the AC-Non-Triviality property.

Finally, once asked to vote, each participant either votes or is suspected and has a vote issued on its behalf by a non-faulty participant. Hence, one vote for each participant is eventually cast in the log service, and eventually either the abort or the commit condition becomes true. The protocol therefore satisfies the AC-Termination property.

The state of resource managers is changed by applying deterministic updates from write transactions. Hence, given two copies of a resource manager in the same initial state, applying the same sequence of updates leads them to the same final state. If two transactions committed concurrently, then they do not interfere with each other and, even if commuting the order in which they are applied to the database, the final state will be still the same. The recovery procedure builds a sequence with all updates performed by the recovering resource manager in its previous incarnations. Updates are then replayed in an order compatible with the commit order of transactions. That is, for any two transactions t_1 and t_2, if t_2 had its termination requested after t_1 had terminated, then the updates of t_1 are applied before those of t_2. If the termination of t_2 started before t_1 had terminated, then their updates might be applied in any order; in any case, the transactions are commutable and the same final state will be reached, hence, ensuring the durability property.

5.4 From the specification to implementations

We have defined our log service abstractly as a state machine in terms of some variables and atomic actions. To implement this abstraction in a distributed way, we assume that agents communicate via message passing over unreliable but fair

communication channels. We also assume that agents have access to a consensus oracle. Consensus instances are uniquely identified by a natural number. We use *propose(*i, v*)* to denote the proposition of value v in the consensus instance i, and *decide(*i, v*)* to denote the learning of decision v of instance i.

In special, the coordinated implementation of the log service assumes a leader-election oracle like Ω [Chandra et al., 1996]. Participants use this oracle to determine the current coordinator. The leader-election oracle guarantees that eventually all participants will elect the same non-faulty agent as the leader. Obviously, this can only be ensured if there is at least one agent that eventually remains operational "forever". In practical terms, "forever" is reduced to "long enough to accomplish some useful computation", *e.g.*, deciding on a transaction's outcome.

5.5 Coordinated Implementation

5.5.1 Overview

The coordinated implementation is named after a coordinator agent that serves as the interface for the log service to RMs. Instead of simply accessing the service, RMs exchange messages with the coordinator to implement each action in the service's specification. To vote in a transactions, RMs send a $vote$ message to the coordinator, which ensures that votes become durable by proposing them in a sequence of consensus instances until they are decided in some instance, at which point they are durable. The coordinator analyzes durable votes to determine transactions' outcomes and inform the RMs, which then complete the implementation of action VOTE. To amortize the cost of termination of each transaction, the coordinator batches as many votes as possible in the proposal of each consensus instance. If an RM has enough local information to implement some action without waiting for the coordinator's reply, then it does so to improve performance. One such example is the ability to abort a transaction locally when the RMs' vote itself is ABORT.

Multiple processes capable of coordinating the implementation run in parallel and, along with the RMs, they use the leader-election oracle to elect the effective coordinator. The elected coordinator is the one to which RMs send their votes and to which non-elected coordinators forward all votes that they may receive from mistaken RMs. Because all coordinators decide on the transactions' outcome based on the same sequence of consensus instances, they all take the same decisions. Hence, safety is not violated even if several processes become coordinator simultaneously. Liveness, on the other hand, may be violated in this case, because coordinators could jeopardize each other's attempts to reach agreement. Eventually the leader-election oracle ensures that a single coordinator exists, and liveness is ensured.

Incarnating a given resource manager happens in a similar way: the process try-

ing to incarnate the RM sends a special message to the coordinator, which proposes the change in a consensus instance. Once an instance decides on an incarnation change, all next instances consider the newly incarnated resource manager. The decision is also informed to all processes incarnating some RM so that they can identify if they have been replaced.

5.5.2 The Algorithm

The coordinated implementation is decomposed in two parts. The first part is a set of "stubs" for the actions and operators used by RMs as defined in Algorithm 14. From the RMs point of view, these replacement stubs implement the original service definitions, but using local data structures and exchanging messages with the coordinator to simulate their specified behavior. The second part is coordinator's protocol, which the stubs in the first part interact to implement the log service specification.

RM Stubs

Algorithm 15 defines the stubs. The first part of the algorithm initializes the three data structures kept by an agent pid incarnating rm.

outcome the local view of function *Outcome*;

myInc the number of rm's incarnation.

rm2pid a flag indicating if pid is currently incarnating rm. Along with the previous variable, this is used to implement implement $R[rm]$ locally.

Observe that *Terminate* does not have a stub, and executes as in the service specification.

The *Incarnate* stub sends a "`Recover`" message to the coordinator requesting it to change the agent incarnating RM rm for agent pid, and waits for a confirmation that the change was performed. The confirmation carries the updates executed by previous incarnations and the incarnation number of rm. $R[rm] = pid$ evaluates to TRUE until the stub learns a bigger incarnation number, indicating that another agent took over the the role of rm. As we show below, the incarnation number is also sent to the coordinator along with the votes, allowing the coordinator to cope with multiple agents believing to be incarnating the same RM. To account for coordinator crashes, every time a process executes *Incarnate*, it does so with a different pid.

Vote forwards the vote to the coordinator. As an optimization, if the vote is for ABORT then the stub updates the transaction outcome before sending the vote to the

Algorithm 15 Stubs to implement Algorithm 14

```
1:Initially:
2:   ∀t, outcome[t] ← UNDEFINED                    ◁ All transactions are undecided.
3:   myInc ← ⊥                                     ◁ Have not incarnated yet.
4:   rm2pid ← ⊥                                    ◁ Ditto.
5:Incarnate(rm, pid) ≜
6:   send ⟨"Recover", pid, rm⟩ to coordinator
7:   WAIT received ⟨"Recovered", rm, upd, inc⟩
8:   myInc ← inc
9:   rm2pid ← pid                                  ◁ pid has incarnated rm.
10:  FOR i = 1 to Len(upd)                         ◁ For each set of committed transactions...
11:     apply updates in upd[i]                    ◁ ...apply it to the database.
12:R[rm]
13:  return rm2pid                                 ◁ Either my own pid or ⊥.
14:Vote(⟨rm, t, tset, vote, upd⟩) ≜
15:  IF vote = ABORT THEN outcome[t] ← ABORT       ◁ Quickly abort.
16:  send ⟨"Vote", rm, myInc, t, tset, vote, upd⟩ to coordinator
17:Outcome(t) ≜
18:  return outcome[t]
19:when received ⟨"Terminated", t, out⟩             ◁ Learn decision.
20:  outcome[t] ← out                              ◁ Learn t's outcome.
21:when receive ⟨"Incarnate", rm, newInc⟩           ◁ Was replaced.
22:  IF newInc > myInc THEN rm2pid ← ⊥              ◁ No longer incarnates rm.
```

coordinator. This is possible because either the vote will be seen by the coordinator or another resource manager votes ABORT on its behalf and, in either case, the transaction is indeed aborted, or the resource manager has been reincarnated and this local abortion has no influence on future transactions.

The **when** clauses at the end of the algorithm update the local data structures when the outcome of a transaction becomes known and when another agent has taken over the RM being incarnated. These actions execute fairly and atomically once their conditions become TRUE.

Coordinator

Algorithm 16 is a set of handlers for message delivery and consensus decision events. All agents that could become the elected coordinator run the algorithm and update their data structures accordingly. However, to minimize network usage, only agents that believe themselves to be the elected coordinator actually send messages. The data structures for a coordinator c are the following:

$T[c]$ Implements T in c. Besides the votes of committed transactions, it also keeps incarnation requests sent by resource managers. Initially $\langle\rangle$.

$V[c]$ Implements V in c, as in the specification.

B The set of votes received but not yet treated by the coordinator. Initially empty.

i The coordinator executes a series of consensus instances, and i is the identifier of the instance the coordinator is waiting to terminate.

$recSet$ The set of reincarnation requests awaiting to be decided.

$R[rm]$ is not explicitly kept in any data structure, but lies implicit in the subsequence of "`Incarnate`" votes in $T[c]$: $R[rm]$ equals the agent in the last vote of type "`Incarnate`" in $T[c]$ (lines 7-9).

When "`Vote`" messages are received by the coordinator, they are added to B to be proposed in the next consensus instance (lines 10-12). Once B is no longer empty, the coordinator proposes it on instance i (lines 13-14). For simplicity, we assume that B always fits in a consensus proposal. In a consensus implementation in which proposals have bounded sizes, B would have to be split and proposed in several instances.

The coordinator waits for the decision of instance i and postpones the learning of decisions of later instances. Once the coordinator learns the decision D (line 15), it first removes it from B (line 16) and then processes each of its elements (lines 17-32). If the element is a vote then the coordinator determines its final meaning before adding it to $V[c]$. The final meaning of a vote is ABORT if its issuer no longer equals $R[rm]$; otherwise, it is the issuer's proposed vote. If the vote caused the transaction to be terminated, that is, it turned an *Undefined* outcome into either COMMIT or ABORT, then the coordinator adds the transaction to $T[c]$ (lines 25-27), which ensures that upon recovery the new rm will learn all committed transactions. Next, the coordinator warns all known participants with a "`Terminated`" message (lines 28 and 30). The "`Incarnate`" special votes, explained in the next paragraph, are handled after all the other votes (lines 31-32). The last step in the action moves the coordinator to the next consensus instance (line 33).

⟨"`Recover`", pid, rm⟩ messages asking the coordinator to change the agent incarnating rm to pid have a simple handling (lines 34-36): upon the arrival of such a message, the coordinator creates a special vote ⟨"`Incarnate`", pid, rm⟩ and adds it to B (line 36). As mentioned in the previous paragraph, ⟨"`Incarnate`", pid, rm⟩ is appended to $T[c]$ after all the normal votes decided have been processed, meaning that $R[rm]$ has been set to pid until another "`Incarnate`" vote overwrites it. The coordinator monitors $T[c]$ to learn when $R[rm]$ changes to pid (lines 35, 37-38). When it happens, the coordinator gathers all the updates performed by rm (lines 39-40) and the number of its new incarnation (line 41), and sends this information

Algorithm 16 Coordinator's protocol

```
1:  Initialization:
2:    T[c] ← ∅                                  ◁ The sequence of terminated transactions.
3:    V[c] ← ∅                                  ◁ The set of durable votes.
4:    B ← ∅                                     ◁ Votes to be broadcast.
5:    i ← 0                                     ◁ Current consensus instance.
6:    recSet ← 0                                ◁ Reincarnation transactions.
7:  R[rm]
8:    rrm ← r = ⟨"Incarnate", pid, rm⟩ ∈ T[c] :   ◁ Pick the reincarnation request for rm
        ∀r' = ⟨"Incarnate", pid', rm⟩ ∈ T[c], r >_{T[c]} r'   ◁ that was last received.
9:    return rrm[2]
10: when receive ⟨"Vote", rm, pid, t, tset, vote, upd⟩
11:   if¬∃⟨rm, _, t, _, _, _⟩ ∈ B
12:     B ← B ∪ {⟨rm, pid, t, tset, vote, upd⟩}   ◁ Only the first vote is considered.
13: when B ≠ ∅
14:   propose(i, B)                             ◁ Propose B on instance i.
15: when decide(i, D)                           ◁ Decided D on instance i.
16:   B ← B \ D                                 ◁ Remove from next proposals.
17:   T[c] ← T[c] • ∅                           ◁ New set of transactions.
18:   for all⟨rm, pid, t, tset, vote, upd⟩ ∈ D
19:     IF ¬∃⟨rm, _, t, _, _, _⟩ ∈ V[c]
20:       THEN
21:         prevState ← Outcome(t)
22:         IF R[rm] ≠ pid                      ◁ If rm has been reincarnated
23:           THEN V[c] ← V[c] ∪ ⟨rm, t, tset, ABORT, ∅⟩   ◁ turn the vote into ABORT
24:           ELSE V[c] ← V[c] ∪ {⟨rm, t, tset, vote, upd⟩}   ◁ else, add it to the set of votes.
25:         IF (prevState = UNDEFINED) ∧ (Outcome(t) = COMMIT)   ◁ If vote lead to commit of t
26:           THEN
27:             T[c] ← T[c] ⊕ t                 ◁ add it to T[c]
28:             ∀p ∈ tset, send message ⟨"Terminated", t, Outcome(t)⟩ to p   ◁ and warn participants.
29:         ELSE IF vote = ABORT                ◁ If lead to abortion of t
30:           THEN ∀p ∈ tset, send message ⟨"Terminated", t, ABORT⟩ to p ◁ warn known participants.
31:   FOR ALL d = ⟨"Incarnate", _, _⟩ ∈ D        ◁ Process "Incarnate" votes.
32:     T[c] ← T[c] • d
33:   i ← i + 1                                 ◁ Process the next batch.
```

to pid, so it can start playing rm (line 42); the other resource managers are warned about the change in the last step of the algorithm (line 43).

Traditionally, RMs write their logs on disk before voting. Even when using the log service, RMs may still write locally for a number of reasons, such as (i)recovery speedup, because reading locally is faster than reading remotely, and (ii)minimizing network usage, by sending just empty updates to the service. If the RM does not write its updates locally, then it can resort to the coordinator to obtain them. In this

Algorithm 16 Coordinator's protocol (Continued)

34: **when** receive ⟨"Recover", pid, rm⟩ ◁ *Upon request to recover*
35: $recSet \leftarrow recSet \bigcup$⟨"Incarnate", pid, rm⟩ ◁ *remember the request*
36: $B \leftarrow B \bigcup$⟨"Incarnate", pid, rm⟩ ◁ *and add it to* B.
37: **when** $\exists\ t_{inc} =$⟨"Incarnate", pid, rm⟩ $\in recSet : t_{inc} \in T[c]$ ◁ *Once a reincarnation terminates*
38: $recSet \leftarrow recSet \setminus \{t_{inc}\}$
39: $V^{rm} \leftarrow \{e = \langle rm, t, \_, \_, \_\rangle \in V[c] :$ ◁ *determine* rm*'s previous updates*
$(Outcome(t) = \text{COMMIT}) \wedge (t <_{T[c]} t_{inc})\}$
40: $U^{rm} \leftarrow \langle u : \langle rm, \_, \_, u\rangle \in V^{rm}\rangle,$ ◁ *and order them as in* T.
$\forall \langle rm, t_1, \_, u_1\rangle, \langle rm, t_2, \_, u_2\rangle \in V^{rm},$
$t_1 <_{T[c]} t_2 \Rightarrow u_1 <_{U^{rm}} u_2\}$
41: $inc \leftarrow |\ Old\ |$: ◁ *Determine the incarnation number,*
$Old = \{r =$⟨"Incarnate", $\_, rm\rangle \in T : r <_T$⟨"Incarnate", $pid, rm\rangle\}$
42: send ⟨"Recovered", $rm, U^{rm}, inc\rangle$ to pid ◁ *tell process* pid
43: send ⟨"Incarnate", $rm, inc\rangle$ to all resource managers ◁ *and the other RMs.*

case, the coordinator scans the decisions of all consensus instances to determine the updates of committed transactions concerning the RM when recovering. Since all coordinators maintain the same state, any can be safely contacted.

5.6 Uncoordinated Implementation

5.6.1 Overview

The uncoordinated implementation is based on the Paxos Commit protocol. The main purpose of this implementation is to save time by not having the votes sent to the service through a coordinator. Instead, each vote is cast in a distinct consensus instance.

When an RM is first contacted by a TM in the context of a transaction, besides executing the requested operation the RM also names the consensus instance in which it will vote to terminate the transaction. The instances are sequentially taken from a per-RM pool and informed to the TM along with the reply to the first operation. The TM informs the named instances to all participants along with its commit request so that they know which instance to use to learn votes and to vote on behalf of each other, should they need.

Besides voting, RMs also use consensus instances to propose changes of incarnations. Because RMs cannot rely on a total order of INCARNATE requests, as in the coordinated implementation, the protocol must rely on some other assumption to limit the scope of each incarnation. The assumption we make is that at most k transactions are executed in parallel by each RM (multiprogramming level).

Let p be a process incarnating RM rm and q a process not incarnating any RM. If q suspects p to have crashed and wants to takeover the role of rm, it proposes the change in the smallest consensus instance in rm's pool that q believes not to be decided yet. q repeats this step with subsequent instances until one of them decides on the INCARNATE request.

When q's proposal to incarnate rm is decided, q still needs to ensure that all consensus instances in which p had possibly voted have been decided. Otherwise, q might attribute one of such instances to a transaction different from the one p had originally done. There are at most $k-1$ such transactions. That is, if the incarnation change was decided in instance i, then q knows that p can only have voted on instances up to $i+k-1$, and q's incarnation effectively starts at instance $i+k$. To avoid blocking, q conservatively votes ABORT on all instances in the range $[i+1, i+k]$.

To recover the state of p, q must recover the updates sent along with votes for all instances smaller than i. These updates are learned along the the decisions of such instances and applied to the database.

5.6.2 Algorithm

Algorithm 17 defines the stubs implementing Algorithm 14 for an agent pid incarnating an RM rm. The agent keeps three variables:

V[*rm*] The set of votes that rm has received.

commitCounter[*rm*] A counter of committed transactions involving rm.

$rm2pid$ A flag indicating if the agent is incarnating rm and used to locally evaluate $R[rm]$. It is either pid, if pid is incarnating rm, or $\perp$, otherwise.

Transaction Termination

The *Vote* stub proposes rm's vote for transaction t on consensus instance $inst(rm, t)$. If the rm is voting on its own behalf, the current consensus instance is determined locally. With the vote, the resource manager proposes also the number of transactions already committed, $commitCounter$. This information is used later during recovery, as we explain below.

To vote for another resource manager rm', rm must know the instance that rm' would use to vote in t. This information may be transmitted by the transaction manager along with its commit request, since rm will only vote for rm' if it has

Algorithm 17 Uncoordinated implementation with stubs for Algorithm 14 (MPL k)

```
1:  Initialization
2:    V[rm] ← ∅                                          ◁ Votes I have seen.
3:    commitCounter ← 0                                  ◁ How many transactions I committed.
4:    rm2pid ← ⊥                                         ◁ Have not incarnated yet.
5:  R[rm]
6:    return rm2pid                                      ◁ Either my own pid or ⊥.
7:  OUTCOME(t)
8:    IF ∃⟨_, t, _, ABORT, _⟩ ∈ V[rm]                    ◁ Any ABORTs?
9:     THEN ABORT
10:    ELSE IF ∃⟨_, t, tset, _, _⟩ ∈ V[rm] : ∀s ∈ tset :   ◁ All COMMITs?
                  ⟨s, t, tset, COMMIT, _⟩ ∈ V[rm]
11:    THEN COMMIT
12:    ELSE UNDEFINED                                   ◁ Neither one nor the other
13: VOTE(⟨rm, t, tset, vote, update[rm]⟩)
14:   propose(inst(rm, t), ⟨rm, t, tset, vote, upd, Len(T[rm])⟩)   ◁ Vote + number of committed.
15: when decide(j, d)                                    ◁ [Decided on instance j.]
16:   IF d = ⟨r, t, tset, vote, upd, cn⟩), rm ∈ tset
17:    THEN V[rm] ← V[rm] ∪ {⟨r, t, tset, vote, upd⟩}
18:    ELSE IF d = ⟨INCARNATE, _, r⟩, r ≠ rm             ◁ Someone else was substituted.
19:    THEN V[rm] ← V[rm] ∪ {⟨r, t, {r, rm}, ABORT, ∅⟩}, inst(rm, t) = j   ◁ Make d an ABORT vote.
20:    ELSE IF d = ⟨INCARNATE, pid′, rm⟩, pid ≠ pid′     ◁ rm has been reincarnated,
21:    THEN rm2pid ← ⊥                                   ◁ so give it up.
```

voted COMMIT for itself and, hence, received the commit request from the transaction manager. The transaction manager, in turn, collects this information from the resource managers during the execution of the transactions.

The decision of a consensus instance j is learned by rm if it is one of its instances or one associated to a transaction in which rm is participant. This decision can be of three types: (i) a vote, in which case it is added to $V[rm]$; (ii) a change of incarnation of another resource manager, in which case it is added to $V[rm]$ as an ABORT vote; and (iii) a change of rm's incarnation to process pid', in which case the process $pid \neq pid'$ currently incarnating learns that its incarnation lasts only until instance $j + k$. The **when** clause that processes the decisions is only activated once the resource manager has completed recovery.

Recovering from Failures

The *Incarnate* stub determines the updates performed by previous incarnations and in which instance the new incarnation starts. The procedure consists of three steps: in the first step, pid (i.e., the process incarnating rm) proposes ⟨"`Incarnate`", pid, rm⟩

Algorithm 17 Uncoordinated implementation with stubs for Algorithm 14 (MPL k) (Continued)

```
22: Incarnate(rm)
23:   tranSet ← ∅
24:   i ← 0                                   ◁ Assume that 0 is the first instance identifier.
25:   while TRUE
26:     propose(i, ⟨"Incarnate", pid, rm⟩)    ◁ Vote until incarnation changes,
27:     wait decide(i, d)
28:       if d = ⟨"Incarnate", pid, rm⟩
29:        then
30:         pidRM ← pid                       ◁ Reincarnated
31:         break                             ◁ Stop the first iteration.
32:        else
33:         tranSet ← tranSet ⋃ {d}
34:         i ← i + 1
35:   for j ← 1..k − 1                        ◁ and then, for k − 1 possibly open instances,
36:     propose(i + j, ⟨rm, t, {rm}, Abort, ∅, 0⟩)   ◁ vote to close them.
37:     wait decide(i + j, d)
38:     if d = ⟨"Incarnate", _, rm⟩           ◁ Someone else is recovering.
39:      then
40:       rm2pid ← ⊥                          ◁ Give up the resource manager
41:       return ⟨⟩                           ◁ and stop looking for updates.
42:     tranSet ← tranSet ⋃ {d}
43:   for all ⟨rm, t, l, v, u, cn⟩ ∈ tranSet   ◁ Close all of those transactions:
44:     if ¬∃⟨p′, t, l′, Abort, ∅, cn′⟩ ∈ V[rm]   ◁ If t was not aborted,
45:      then
46:       for all p′ ∈ l, ¬∃⟨p′, t, l, v′, u′, cn′⟩ ∈ V[rm]   ◁ make sure to terminate it
47:         propose(inst(p′, t), ⟨p′, t, l, Abort, ∅, 0⟩)   ◁ voting for others if needed.
48:         wait until decide(inst(p′, t), d)
49:         V[rm] ← V[rm] ⋃ {d}
50:         if d = ⟨p′, t, _, Abort, _, _⟩ then break   ◁ Stop on Abort.
51:   U[rm] ← ⟨u_1, ..., u_m⟩ :               ◁ Now order updates of known transactions
          e_k = ⟨rm, t_k, l_k, v_k, u_k, cn_k⟩ ∈ V[rm],
          ∀p ∈ l_k, ∃⟨p, t_k, l_k, Commit, u_p⟩ ∈ V[rm],   ◁ that committed
          ∀i < j, cn_i < cn_j                 ◁ in commit order.
52:   return U[rm]
```

in its first consensus instance and until such a value is decided in an instance i, at which point pid becomes the incarnation of rm for transactions associated with consensus instances $i + k$, until displaced by another process. The second step terminates the transactions associated with instances $i + 1$ to $i + k - 1$, belonging to the incarnations being replaced. The third step determines the outcome of all transactions associated with instances smaller than $i + k$.

The protocol terminates by gathering the updates of committed transactions in the sequence $U[rm]$. Elements in $U[rm]$ are ordered according to the commit counter in each vote, making it consistent with the commit order of non-concurrent transactions. Algorithm 17 does not keep the T data structure nor any abridged version of it; the order of committed transactions is forgotten by the resource managers once their updates are applied. Upon recovery, $U[rm]$ is created and kept just until the recovery is terminated.

To improve performance, implementations should use checkpoints and reduce the number of consensus instances in which resource managers must propose. Most of the remaining instances can run in parallel to reduce the recovery latency.

5.7 Evaluation

In this section we compare the coordinated implementation of the log service with other relevant commit protocols in the literature. In Section 5.7.1 we give a brief overview of such approaches, compare them in terms of communication steps, number of messages, and resilience. In Section 5.7.2, we experimentally compare the coordinated implementation with an uncoordinated one. The uncoordinated implementation is based on Paxos Commit [Gray and Lamport, 2006].

5.7.1 Analytical Evaluation

In the 2-Phase Commit protocol (2PC) [Gray, 1978], the TM asks RMs to vote, collects their votes, decides on the transaction's outcome, and informs the RMs. The protocol therefore requires three communication steps and sends up to 3 R messages, where R is the number of RMs. If the TM crashes during the second step, the protocol blocks until the process is recovered.

In the commit protocol of Guerraoui *et al.* [Guerraoui et al., 1996] (GLS), the RMs do not centralize the decision on the TM: on the second step, they exchange their votes and, on the third step, they communicate their decisions to each other, using a total of $N + 2N^2$ messages—to simplify the analysis, we count messages sent by processes to themselves. Hence, in good runs, all RMs see the decision after three communication steps but, in case of suspicions of failures, they must resort to a consensus instance to ensure correct termination, what adds at least one step to the execution. Assuming an unreliable failure detector to solve consensus, the protocol tolerates any minority of RM crashes.

Paxos Commit (PC) [Gray and Lamport, 2006] has the same communication complexity and, in some cases, the same communication pattern as GLS in good runs. In PC, however, the second and third steps are used to run the Paxos consensus protocol [Lamport, 1998] to agree on the vote of each RM. In the case of

suspicions, an RM proposes ABORT on behalf of the suspected RM using its Paxos instance. In PC, the role of deciding on the transaction's outcome is logically dissociated from the RMs; they are played by the acceptors. PC is non-blocking in the presence of crashes of any minority of acceptors. When there is a single acceptor, PC can be configured to have the same communication pattern and resilience as 2PC. If every RM acts also as an acceptor, then it equals GLS. We report this latter case in Table 5.1, which summarizes aspects of each approach.

The Uncoordinated Log Service (Uncoord) is an extension of PC to cope with the ordering of transactions, durability of updates, and replacement of failed resource managers. These aspects, however, to not enter in our analytical evaluation, where PC and Uncoord are regarded as the same.

While the previous approaches try improve the resilience of 2PC by reducing and distributing the role of the TM, other protocols tried to handle its failure by other means. In the 3-Phase Commit (3PC) [Gray and Lamport, 2006] protocols, the TM is replaced once it has crashed. Problems may arise, though, if the TM has not in fact crashed, possibly leading to inconsistent termination [Bernstein et al., 1987]. Besides, to be replaceable, the TM cannot be the only one to keep data essential for the termination, and disseminating this data introduces more communication steps to the termination (See Table 5.1).

In the Coordinated Log Service, the replacement of a non-crashed TM may lead to unnecessary aborts, but it does not break the consistency of the protocol. By using Paxos in the coordinated log service, the protocol takes the same number of communications steps to terminate as 3PC: five. Because the same coordinator is used on many transactions, the cost of terminating them is amortized by aggregating votes to be proposed, reducing the number of instances executed in parallel. Parallel instances, in practice, impact negatively on each other due to resource contention.

Jiménez-Peres *et al.* [Jiménez-Peris et al., 2001] proposed a commit service abstraction and a set of implementations. Different from our abstraction, their commit service is defined for single commit instances. Their implementations provide an optimistic outcome after three steps, and a confirmation after the fourth step, when the transaction can be committed. As we have done with RMs and acceptors in PC, we analyse their protocol co-locating the commit servers and the RMs.

Table 5.1, below, compares the discussed approaches in terms of number of communication steps required by each protocol, the number of messages sent on each one, and their resilience, that is, the number of resouce manager failures that does not prevent the protocols from terminating.

	Comm. Steps	Messages	Resilience
2-PC	3	$3R$	0
GLS [Guerraoui et al., 1996]	3	$2R^2 + R$	$< R/2$
Paxos Commit [Gray and Lamport, 2006]	3	$2R^2 + R$	$< R/2$
Commit Service [Jiménez-Peris et al., 2001]	(3)4	$3R^2 + 2R$	$< R/2$
3-PC [Bernstein et al., 1987]	5	$5R$	—
Coord. Log Service	5	$5R$	$< R/2$

Table 5.1: The cost of some commit protocols

5.7.2 Experimental Evaluation

By sending updates along with their votes, RMs can be recovered using only the state of acceptors; sending them before could be a waste of resources, and sending them after could cause the protocol to block during recovery. We have implemented a variant of Paxos Commit in which votes carry RMs' updates. Moreover, we also augmented it with a procedure to recover RMs based on the acceptors state. We used this Uncoordinated Log Service implementation in our experimental evaluation to show the cost of storing updates on the acceptors instead of locally at the RMs. Analytically, this approach has the same costs and resilience of Paxos Commit.

We have prototyped the Uncoordinated and the Coordinated log service in Java, as well as the Paxos consensus protocol that underlies both of them, and compared them using the Sprint infrastructure. More details on the prototype can be found in [Camargos et al., 2007a]. The evaluation consisted of two benchmarks, explained below, run in the Emulab testbed [White et al., 2002]. All nodes used were equipped with 64-bit Xeon 3GHz processors, interconnected through a Gigabit Ethernet switch.

Micro-benchmark

In the first experiment we used a micro-benchmark of non-conflicting transactions, each comprising one write operation executed by one RM; each operation writes 0, 1000, or 7000 bytes of data, allowing us to evaluate the impact of the size of updates carried by votes. To assess the impact of disk writes on the log service, we have run experiments in which consensus acceptors have disk writes enabled and disabled (i.e., consistency relies on at most a minority of acceptors crashing simultaneously).

To compare the different configurations, we first determined the workload needed on each of them to reach a 10ms latency per transaction execution. Each entry in Table 5.2 presents the throughput, in transactions per second, of the coordinated and the uncoordinated techniques, and their ratio (number between parenthesis).

Going from the first to the second line evidences the cost of writing on stable storage: both approaches benefit from disabling the disk, although the uncoordinated approach benefits more. The reason is that the coordinated approach already optimizes disk access by using the same consensus instance and a single disk write for multiple transactions.

Disk	0 bytes	1000 bytes	7000 bytes
On	1539/184 (8.36)	771/189 (4.08)	96/97 (0.99)
Off	2936/1249 (2.35)	1559/1181 (1.32)	370/357 (1.04)

Table 5.2: Coord/Uncoord throughput ratio for 10ms latency

Going from the left to the right columns in Table 5.2 we see the effects of update sizes in the log service. As the size grows, fewer votes can fit in the same proposal, increasing the number of Paxos instances needed until each vote requires one full instance, as in the uncoordinated version. At this point, the benefits of the coordinator are minimal. Likewise, the gains of not writing on disk are smaller for bigger updates, because more time is spent transferring the data.

TPC-C based benchmark

We also evaluated our implementations with a variant of the TPC-C [TPCC,] benchmark in which clients submit requests without think times. The transactions and their frequencies, however, are those specified by TPC-C. All tables but one (the read only Items table) were range partitioned among RMs according to their primary keys; the read-only table was replicated on all RMs. As a result, at most 15% transactions involved more than one RM (up to all RMs). Update transactions were 92% of the workload and produced data to be logged not exceeding 1500 bytes. In the experiments, we varied the load by increasing the number of clients, and measured the resulting throughput in transactions per second and their respective response times. Each dotted curve in Figure 5.1 gives the theoretical relation between throughput and response time for different numbers of clients, as defined by the Little's Law: number of clients = throughput $\times$ response time.

As Figure 5.1 shows, very small loads (dotted curve with 2 clients) do not differ significantly in performance between configurations. With higher loads, the coordinated version outperforms the uncoordinated one and scales better. Although in 85% of the cases a single transaction requires one consensus instance regardless of the termination protocol, the coordinated version can group at least five simultaneous requests, and even when a single RM could perform the batching on its own,

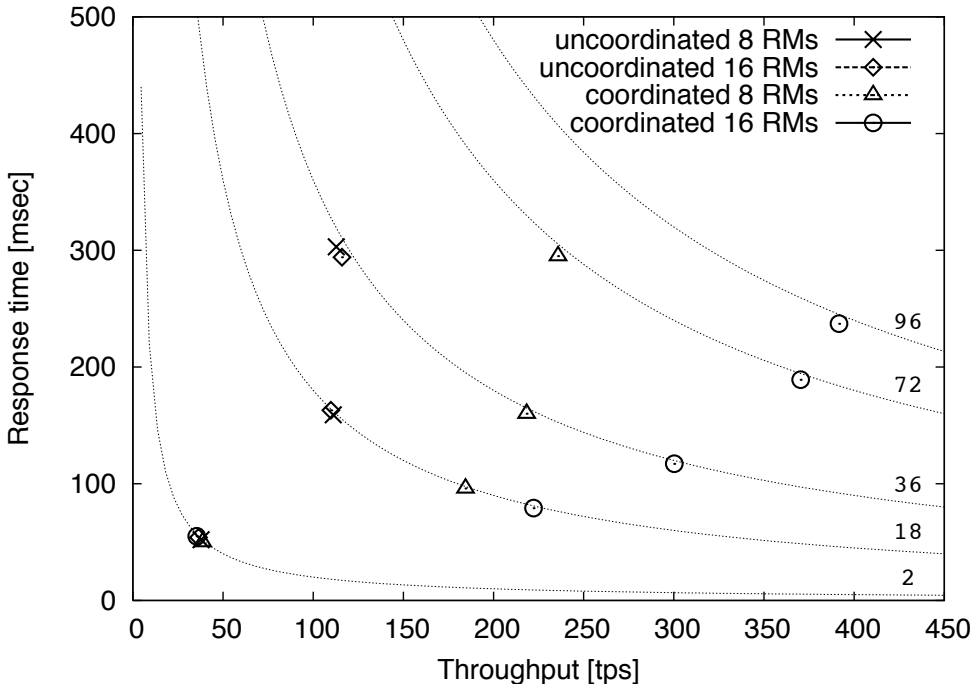

Figure 5.1: Maximum throughput versus response time of TPC-C transactions. The number of clients is shown next to the curves. Disk writes at acceptors were enabled.

the coordinator is still better off as it can combine data from different RMs.

5.8 Final Remarks and Related Work

In this chapter we have introduced the specification of a log service for transaction processing systems, which provides atomicity and durability to transactions through a non-blocking termination protocol. The service totally orders non-concurrent transactions. Should a resource manager fail, the service can be used to recover the resource manager's state prior to the crash and start a copy of it on a different and functional node. Moreover, it safely copes with multiple copies of a resource manager. Due to its general design and simple specification, we believe it can serve as basis for further work.

Some works in the literature have similarities with our service. In Stamus and Cristian's approach [Stamos and Cristian, 1993], for example, the log records of resource managers are aggregated and stored at the transaction manager, which is relied upon to implement the service. Their protocol, however, uses a byzantine agreement abstraction and makes stronger synchrony assumptions. Besides,

it does not consider the recovery of resource and transaction managers on different nodes in case of malfunctioning. Conversely, the log service of Daniels *et al.* [Daniels et al., 1987] does allow recovery on different machines but, instead, lacks the transaction termination feature. Also Mohan *et al.* [Mohan et al., 1985] used byzantine tolerant agreement abstractions. Their extended 2PC protocol can be seen as a byzantine uncoordinated log service implementation.

We also presented two highly available implementations of the log service, coordinated and uncoordinated, and provided a comparative experimental performance evaluation. As we have shown in the previous section, these implementations are representative of several well-known protocols in the literature. Although in theory the uncoordinated approach outperforms the coordinated one by two communication steps, in our experimental evaluation the coordinated approach has led to a much higher transaction throughput and smaller response times for small transactions. This result is explained by the higher number of messages sent in parallel in the uncoordinated version, negatively effecting on each other, and by the coordinator being able to terminate possibly many transactions using a single instance of consensus.

Resource managers and acceptors can be collocated to minimize the number of nodes in the system, as done in other commit protocols that are based on consensus [Gray and Lamport, 2006, Guerraoui et al., 1996]. However, we see a strong reason to decouple these two roles in practical scenarios: resource managers are generally more complex software artifacts and, therefore, more error prone than acceptors; the latter must be available for the sake of all transactions in the system and should not risk crashing because of a resource manager error. Besides, by decoupling these roles, the availability of the system becomes determined by the availability of quorum of acceptors, and more easily assessed.

Chapter 6

Conclusion

The difference between theory and practice is
bigger in practice than in theory.
Unknown author

6.1 Contributions

Agreement problems are recurrent in the development of fault-tolerant distributed systems and have been the subject of a large amount of research in the last decades. In spite of the large number of protocols proposed, we believe that concerning with practical aspects of implementing and deploying such protocols is missing, what limits their applications in real scenarios. In this thesis, we focus on the specification of practical protocols for agreement problems in distributed systems. By practical we mean adaptable crash-recovery protocols for asynchronous systems, which explore optimistic—yet realistic—assumptions to progress but do not require them to ensure correctness. Moreover, these protocols should harness building blocks readily available in todays infrastructures and semantic information to improve performance. In these areas, this thesis makes the following contributions.

Multicoordinated Consensus We have presented protocols that explore spontaneous ordering to solve consensus in the crash-recovery model. We generalized one of them, namely, B*-Consensus, as a multicoordinated mode of execution for agreement protocols. Agreement protocols traditionally run in rounds whose responsibility of progress is entrusted to a coordinator agent. In the multicoordinated mode, the coordinator of each round is replaced by a set of overlapping quorums of coordinators and, as long as at least one quorum of coordinators is alive, the round may still succeed in reaching agreement. Therefore, the multicoordinated mode provides better resilience than the single-coordinated rounds. Compared to the fast

mode [Lamport, 2006a], which completely avoids the coordinator, the multicoordinated mode provides better resilience by tolerating more acceptor failures at the expense of one communication step. We present a multicoordinated consensus protocol that extends Fast Paxos [Lamport, 2006a], with its fast and classic modes. The protocol can switch between classic, fast, and multicoordinated modes in runtime. This feature allows the protocol to be deployed in many different environments and to adapt to changes therein during the execution. For example, the protocol may adapt to changes in the network bandwidth, network latency, message loss rate, workload, and spontaneous message ordering. By using the right combination of round types and recovery technique, our protocol emulates most consensus protocols that we are aware of.

Multicoordinated Generalized Consensus While multicoordination reduces changes of rounds due to the failures of their coordinators, the use of semantic information about the values being agreed upon prevents round changes due to conflicting concurrent proposals. We have presented a protocol that combines both approaches in a synergism to improve the availability of agreement protocols. Our protocol, Multicoordinated Paxos, is an extension of Generalized Paxos [Lamport, 2004] to solve Generalized Consensus, and may be instantiated to solve several agreement protocols. We have presented one such instantiation that solves the Generic Broadcast problem [Pedone and Schiper, 2002].

Multicoordinated Agreement for Groups Corporative networks are generally organized hierarchically in groups of agents; agents inside a group communicate through fast links, but agents in different groups experience much larger delays. Hence, agreement protocols for these scenarios should avoid slow inter-group messages. On the one hand, fast protocols are too susceptible to collisions since the slow links could jeopardize spontaneous ordering in any minimally loaded system. On the other hand, using single-coordinated protocols would require messages to be sent from one group to another simply to reach the coordinator. Although using coordinator quorums instead of a single coordinator increases availability in this scenario, it does not avoid the extra communication. The solution we presented here is to split coordinator quorums in partially independent systems and ensure that no conflicting proposals are issued in the same quorum system. We also showed how this protocol can be improved by a recursive use of consensus and use of multicast technology. This way, in good periods of execution, all proposed values are learned by all agents in all groups in two inter-group message delays.

Log Service We presented a log service that abstracts the atomicity and durability properties in transaction termination. The main advantages of using this service to

implement distributed transactions are two. First, it encapsulates the underlaying system inside the service, simplifying application implementation since it may be oblivious to details. Second, it pushes atomicity and durability out of resource managers and, because the service may be implemented to be more or less available and fault tolerant, this improves the overall availability of the whole system.

6.2 Future Work

In extension to the work presented in this thesis, we believe that several points are worth further investigation.

Reconfiguration Policies The ability to change a protocol's execution mode is a very powerful tool in that it allows the protocol to react to environment changes. To the best of our knowledge, there has not been any work on self-adapting agreement protocols nor on the policies that should drive these adaptations. We would like to develop and experiment such policies and understand which ones are better suited for different distributed systems like clusters, computational grids, and corporative and overlay networks.

Byzantine Generalized Agreement Tolerating byzantine agents in consensus algorithms can be achieved by simply adjusting quorum intersection sizes to mask these agents' influence [Alvisi et al., 2000]. Consider for example a round with single coordinator in which the only coordinator is byzantine and submits multiple proposals. This byzantine behavior is, in fact, equivalent to multiple proposals being issued in a fast round. Since the fast-quorum requirement (Assumption 3) is enough to mask multiple proposals in fast-rounds, it is also enough to mask the coordinator's byzantine behavior. Similarly, the coordinator quorum requirement (Assumptions 4) may be strengthened to require the coord-quorums to share at least two coordinators and allow one coordinator to misbehave. Moreover, by employing a coord-quorum approach to start rounds, byzantine coordinators cannot force a denial-of-service by constantly creating new rounds. More important, while there has been no attempt to solve Generalized Consensus in byzantine settings, our approach would allow us to do so "out-of-the-box". This would allows us to generalize, for example, PBFT [Castro and Liskov, 1999], HQ [Cowling et al., 2006] and Zyzzyva [Kotla et al., 2007].

Hierarchical Agreement In the same way that coordinator quorums were used to reach agreement among groups, in Chapter 4, another layer of quorums that filters proposals before sending them to the coordinators could be added. In doing, we

can extend the multicoordinated collision fast protocols to several layers of a hierarchically organized network. The result would be an aggregation protocol in the lines of Astrolabe [Van Renesse et al., 2003], but with a final level of aggregation that provides agreement. Whether this protocol would be practical is an interesting question.

Partial Agreement In some cases, ensuring that a subset of learners eventually learn some decided value is enough to satisfy the application needs. While it is easy for acceptors to inform just some subset of the learners depending on the value accepted, it would be interesting to also limit the work done by the acceptors themselves. For example, consider the environment discussed in Chapter 4, where agents are divided in groups. If only the learners of some group G are interested in some value, then only the acceptors in G should get involved in choosing it. This partial agreement property seems a perfect fit for Generalized Consensus since, in this problem, acceptors are allowed to accept diverging c-structs, as long as they are compatible. If a single instance of Generalized Consensus among groups can be used to reach agreement at all levels, *i.e.*, inside groups, across a subset of them, or all together, is an interesting question that we would like to explore.

Multicoordinated Log Service In Chapter 5 we presented the log service abstraction for transaction termination. We also presented two implementations, one single-coordinated and another completely distributed. These are both extremes in a spectrum of which multicoordination is in the middle. We have not studied any multicoordinated implementation of the log service, but would like to do so. What is more, we would like to compare that with Collision-Fast Paxos and identify under which conditions each of these protocols would be the better option.

AMQP/Fix over the Log Service The Advanced Messaging Queue Protocol, AMQP, is a specification for transactional store and forward message queuing [Vinoski, 2006, A. M. Q. P. Working Group, 2006]. The Financial Exchange protocol, or FIX for short [The FIX Protocol Organization, 2008], is a specification for message exchange for financial applications. Both, AMQP and FIX, are widely used in trading applications and both resort to a transactional storage engine to provide the atomicity and durability that these applications require. Examples of storage engines used are MySQL and the Mnesia databases. We believe that the strong semantics and the simple interface of our log service are a perfect fit for these applications and plan on replacing the storage engine of one of the many open-source implementations of AMQP (Fix is proprietary). This would allows us to test the log service in different applications, identify its drawbacks, and compare it with other implementations.

Appendix A

Multicoordinated Paxos

A.1 Proof of Correctness

A.1.1 Preliminaries

Our proofs depend on a number of basic definitions and propositions, presented in this section. Apart from an extra proposition, everything in this section comes from [Lamport, 2004], since we want to follow the same notation and proof structure as presented in that paper.

Concerning round numbers (hereinafter called *ballot numbers*), we only assume that they are totally ordered by a relation $<$ and there is a smallest ballot number 0. We use *balnum* as an abbreviation for *ballot number*, and let *BalNum* be the set of all balnums. Balnums can be either fast or classic, but not both. Quorums of acceptors and coordinators depend upon the ballot number. We assume both the Fast Quorum Requirement (Assumption 3) and the Coord-quorum Requirement (Assumption 4). Lastly, we assume a c-struct set *CStruct* upon which Generalized Consensus is defined.

We start by defining a data structure called *ballot array*, used by our abstract algorithms. Ballot arrays keep the votes of each acceptor at each balnum and the ballot number at which each acceptor currently is (the highest-numbered balnum it has heard of). Vote entries are initialized with a value *none* that is not in *CStruct*. Due to that, we extend $\sqsubseteq$ to cope with *none* such that *none* $\sqsubseteq$ *none* but $\neg(v \sqsubseteq w)$ if either v or w (but not both) equals *none*.

Definition 3 (Ballot Array—Definition 1 of [Lamport, 2004]) *A* ballot array *bA is a mapping that assigns to each acceptor a a balnum $\widehat{bA}_a$ and to each acceptor a and balnum m a value $bA_a[m]$ that is a c-struct or equals none, such that for every acceptor a:*

- $bA_a[0] \neq$ *none,*

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

We say that a value v is *chosen at* balnum m if an m-quorum accepts v at m. We can also define chosen at with respect to a ballot array as follows.

Definition 4 (Chosen at—Definition 2 of [Lamport, 2004]) *A c-struct v is* chosen at *balnum m* in *ballot array bA iff there exists an m-quorum Q such that $v \sqsubseteq bA_a[m]$ for all acceptors a in Q. A c-struct v is* chosen in *ballot array bA iff it is chosen at m in bA for some balnum m.*

Considering that an acceptor a can only accept c-structs at balnums equal to or greater than its current one ($\widehat{bA}_a$ in ballot array *bA*), we define a c-struct to be *choosable at* balnum m iff it is or can still be chosen at m.

Definition 5 (Choosable at—Definition 3 of [Lamport, 2004]) *A c-struct v is* choosable at *balnum m* in *ballot array bA iff there exists an m-quorum Q such that $v \sqsubseteq bA_a[m]$ for every acceptor a in Q with $\widehat{bA}_a > m$.*

A c-struct is *safe at* balnum m iff it extends any c-struct choosable at a balnum j such that $j < m$, and we define a ballot array *bA* to be *safe* iff every entry $bA_a[m]$ different from *none* is safe at m, for every acceptor a. If acceptors accept only safe c-structs at balnums greater than or equal to their current ones, the algorithm guarantees that if v is ever chosen at a balnum i, then no acceptor will have accepted a c-struct w that does not extend v at a balnum j greater than i.

Definition 6 (Safe at—Definition 4 of [Lamport, 2004]) *A c-struct v is* safe at m in *bA iff $w \sqsubseteq v$ for every balnum $k < m$ and every c-struct w that is choosable at k. A ballot array bA is* safe *iff for every acceptor a and balnum k, if $bA_a[k]$ is a c-struct then it is safe at k in bA.*

The following proposition states that the chosen values of a safe ballot array are compatible. It implies that an algorithm can satisfy the consistency property of Generalized Consensus by having acceptors accept only safe values at balnums that are no lower than their current ones. Its detailed proof appears in [Lamport, 2004].

Proposition 1 (Proposition 1 of [Lamport, 2004]) *If a ballot array bA is safe, then the set of values that are chosen in bA is compatible.*

A.1.2 Abstract Multicoordinated Paxos

As in [Lamport, 2004], our proof of correctness starts with an abstract algorithm that can be more easily proved correct. The reason why we cannot use the same abstract algorithm as in [Lamport, 2004] is the difficulty of multicoordinated rounds to implement the variable *minTried* used there. As a result, we came up with an even more abstract algorithm, which can be implemented not only by Multicoordinated Paxos, but also by the abstract algorithm in [Lamport, 2004].

Our abstract algorithm is based upon a subset of the variables used by the abstract algorithm of [Lamport, 2004]:

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

propCmd The set of proposed commands. It initially equals the empty set.

bA A ballot array. It represents the current state of the voting. Initially, $\widehat{bA}_a = 0$, $bA_a[0] = \bot$ and $bA_a[m] = none$ for all $m > 0$. (Every acceptor casts a default vote for $\bot$ in ballot 0, so the algorithm begins with $\bot$ chosen.)

maxTried An array of c-structs, where $maxTried[m]$ is either a c-struct or equal to *none*, for every balnum m. Initially, $maxTried[0] = \bot$ and $maxTried[m] = none$ for all $m > 0$.

The Abstract Multicoordinated Paxos algorithm satisfies the following invariants, which, as we prove next, imply the properties Nontriviality and Consistency of Generalized Consensus.

maxTried **Invariant** For every balnum m, if $maxTried[m] \neq none$, then

1. $maxTried[m]$ is proposed.
2. $maxTried[m]$ is safe at m in *bA*.

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

1. $bA_a[m]$ is safe at m in *bA*.
2. If m is a classic balnum, then $bA_a[m] \sqsubseteq maxTried[m]$.
3. If m is a fast balnum, then $bA_a[m]$ is proposed.

learned **Invariant** For every learner l:

1. $learned[l]$ is proposed.
2. $learned[l]$ is the lub of a finite set of c-structs chosen in *bA*.

Proposition 2 *The* learned *invariant implies the Nontriviality property of Generalized Consensus.*

PROOF: By part 1 of the *learned* invariant. □

Proposition 3 *Invariants* bA *and* learned *imply the Consistency property of Generalized 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. *bA* is safe.
   PROOF: This follows from part 1 of the *bA* invariant and the definition of a safe ballot array (Definition 6).

LET: $S = \{v : v \text{ is chosen in } bA\}$

2. S is compatible.
   PROOF: By step 1 and Proposition 1.
3. For every learner l, $learned[l] \sqsubseteq \sqcup S$.
   PROOF: This is true by part 2 of the *learned* invariant and axiom CS3, 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 Multicoordinated Paxos has seven atomic actions, described below. A complete specification of the algorithm in TLA$^+$ is given in Section A.2.2.

Propose(C) for any command C. It is enabled iff $C \notin propCmd$. It sets *propCmd* to $propCmd \cup \{C\}$.

JoinBallot(a, m) for acceptor a and balnum m. It is enabled iff $\widehat{bA}_a < m$. It sets $\widehat{bA}_a$ to m.

StartBallot(m, w) for balnum m and c-struct w. It is enabled iff

- $maxTried[m] = none$,
- w is safe at m in *bA*, and
- $w \in Str(propCmd)$.

It sets $maxTried[m]$ to w.

Suggest(m, σ) for balnum m and c-seq σ. It is enabled iff

- *maxTried*$[m] \neq$ *none* and
- $\sigma \in Seq(propCmd)$.

It sets *maxTried*$[m]$ to *maxTried*$[m] \bullet \sigma$

ClassicVote(a, m, v) for acceptor a, balnum m, and c-struct v. It is enabled iff

- $m \geq \widehat{bA_a}$,
- v is safe at m in *bA*,
- $v \sqsubseteq$ *maxTried*$[m]$, and
- $bA_a[m] =$ *none* or $bA_a[m] \sqsubseteq v$.

It sets $bA_a[m]$ to v and $\widehat{bA_a}$ to m.

FastVote(a, C) for acceptor a and command C. It is enabled iff

- $C \in$ *propCmd*,
- $\widehat{bA_a}$ is a fast balnum, and
- $bA_a[\widehat{bA_a}] \neq$ *none*.

It sets $bA_a[\widehat{bA_a}]$ to $bA_a[\widehat{bA_a}] \bullet C$.

AbstractLearn(l, v) for learner l and c-struct v. It is enabled iff v is chosen in *bA*. It sets *learned*$[l]$ to *learned*$[l] \sqcup v$.

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

Proposition 4 *Abstract Multicoordinated Paxos satisfies the Stability property of Generalized 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 remains to prove that the abstract algorithm satisfies the invariants *maxTried*, *bA*, and *learned*. For the sake of simplicity, however, we use some extra notation in the proof. First, 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. Second, to avoid ambiguity, we let *maxTriedInv*, *bAInv*, and *learnedInv* be expressions representing the statements of the three invariant properties.

Proposition 5 *Abstract Multicoordinated Paxos satisfies the invariants* maxTried*,* bA*, 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*(C) is executed, where $C \in Cmd$.

   PROOF SKETCH: Action *Propose*(C) only changes variable *propCmd*, 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 *propCmd* only increases and c-structs composed of proposed commands remain composed of proposed values.

   1.1. 1. $maxTried' = maxTried$
   2. $bA' = bA$
   3. $learned' = learned$
   4. $propCmd' = propCmd \cup \{C\} \wedge C \notin propCmd$

   PROOF: By the definition of action *Propose*(C).

   1.2. $maxTriedInv'$ is true.

   From its definition, it suffices to:

   ASSUME: $maxTried[r]' \neq none$, for any balnum r

   PROVE: 1. $maxTried[r]' \in Str(propCmd')$ and
   2. $maxTried[r]'$ is safe at r in bA'.

   1.2.1. $maxTried[r]' \in Str(propCmd')$

   PROOF: By applying the assumption of step 1.2 and step 1.1.1 to the invariant *maxTriedInv* we verify that $maxTried[r]' \in Str(propCmd)$, and step 1.1.4 tells us that $propCmd \subset propCmd'$.

   1.2.2. $maxTried[r]'$ is safe at r in bA'.

   PROOF: By *maxTriedInv* and steps 1.1.1 and 1.1.2.

   1.2.3. Q.E.D.

   1.3. $bAInv'$ is true.

   From its definition, it suffices to:

   ASSUME: $bA_e[r]' \neq none$, for any acceptor e and balnum r

   PROVE: 1. $bA_e[r]'$ is safe at r in bA',
   2. r is classic $\Rightarrow bA_e[r]' \sqsubseteq maxTried[r]'$, and
   3. r is fast $\Rightarrow bA_e[r]' \in Str(propCmd')$.

   1.3.1. $bA_e[r]'$ is safe at r in bA'.

   PROOF: By *bAInv* and step 1.1.2.

   1.3.2. r is classic $\Rightarrow bA_e[r]' \sqsubseteq maxTried[r]'$

   PROOF: By *bAInv* and steps 1.1.1 and 1.1.2.

   1.3.3. r is fast $\Rightarrow bA_e[r]' \in Str(propCmd')$

   PROOF: Step 1.1.2 and *bAInv* imply that, if r is a fast balnum, $bA_e[r]' \in propCmd$. Step 1.1.4 shows that $propCmd \subset propCmd'$.

1.3.4. Q.E.D.

1.4. *learnedInv*′ is true.

LET: h be any learner, without loss of generality.

1.4.1. $learned[h]' \in Str(propCmd')$

PROOF: Step 1.1.3 and *learnedInv* imply that $learned[h]' \in Str(propCmd)$, and step 1.1.4 implies that $propCmd \subset propCmd'$.

1.4.2. $learned[h]'$ is the lub of a finite set of c-structs chosen in bA'.

PROOF: By *learnedInv* and steps 1.1.2 and 1.1.3.

1.4.3. Q.E.D.

1.5. Q.E.D.

2. CASE: Action $JoinBallot(a, m)$ is executed, where a is an acceptor and m is a ballot number.

PROOF SKETCH: Action $JoinBallot(a, m)$ only changes $\widehat{bA}_a$, setting it to m, which is bigger than $\widehat{bA}_a$. Invariant conditions that do not refer to $\widehat{bA}_a$ are obviously preserved. It remains to check the conditions stating that certain values are safe or chosen in bA'. 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 value w that is choosable at round k in bA' is also choosable at k in *bA*. By the definition of *safe*, this implies that a value x that is safe at a balnum s in *bA* is also safe at s in bA'.

2.1. 1. $\widehat{bA}_a < m$

2. $\widehat{bA}'_a = m$

3. $\forall i \in Acceptor \setminus \{a\} : \widehat{bA}'_i = \widehat{bA}_i$

4. $\forall i \in Acceptor, j \in BalNum : bA_i[j]' = bA_i[j]$

5. $propCmd' = propCmd$

6. $maxTried' = maxTried$

7. $learned' = learned$

PROOF: By the definition of action $JoinBallot(a, m)$.

2.2. If x is safe at s in *bA*, for any c-struct x and balnum s, then x is safe at s in bA'.

The proof is by contradiction, as follows.

ASSUME: There exist c-struct x and balnum s, such that

1. x is safe at s in *bA*
2. x is not safe at s in bA'

PROVE: FALSE

2.2.1. Choose c-struct w and balnum k such that $k < s$, w is choosable at k in bA', and $w \not\sqsubseteq x$.

PROOF: w and k exist by assumption 2 of step 2.2 and the definition of *safe at*.

2.2.2. w is choosable at k in *bA*.

2.2.2.1. Choose k-quorum Q such that

$$\forall e \in Q : \widehat{bA}_e' > k \Rightarrow w \sqsubseteq bA_e[k]'$$

PROOF: Q exists by the definition of *choosable at*.

2.2.2.2. $\forall e \in Q : \widehat{bA}_e > k \Rightarrow w \sqsubseteq bA_e[k]$

PROOF: Steps 2.1.(1-3) imply that $\forall i \in Acceptor : \widehat{bA}_i > k \Rightarrow \widehat{bA}_i' > k$. Moreover, step 2.1.4 implies $\forall i \in Acceptor : bA_e[k] = bA_e[k]'$. If we combine this two formulas with the one of step 2.2.2.1, we can derive the expression of the current step.

2.2.2.3. Q.E.D.

PROOF: By step 2.2.2.2 and the definition of *choosable at*.

2.2.3. Q.E.D.

PROOF: If w is choosable at $k < s$ in *bA* (step 2.2.2), and $w \not\sqsubseteq x$ (step 2.2.1), then x is not safe at s in *bA*, contradicting assumption 1 of step 2.2.

2.3. *maxTriedInv*′ is true.

From its definition, it suffices to:

ASSUME: *maxTried*$[r]' \neq$ *none*, for any balnum r

PROVE: 1. *maxTried*$[r]' \in$ *Str*(*propCmd*′) and
2. *maxTried*$[r]'$ is safe at r in bA'.

2.3.1. *maxTried*$[r]' \in$ *Str*(*propCmd*′)

PROOF: By *maxTriedInv* and steps 2.1.5 and 2.1.6.

2.3.2. *maxTried*$[r]'$ is safe at r in bA'.

PROOF: By *maxTriedInv*, step 2.1.6, and step 2.2.

2.3.3. Q.E.D.

2.4. *bAInv*′ is true.

From its definition, it suffices to:

ASSUME: $bA_e[r]' \neq$ *none*, for any agent e and balnum r

PROVE: 1. $bA_e[r]'$ is safe at r in bA',
2. r is classic $\Rightarrow bA_e[r]' \sqsubseteq$ *maxTried*$[r]'$, and
3. r is fast $\Rightarrow bA_e[r]' \in$ *Str*(*propCmd*′).

2.4.1. $bA_e[r]'$ is safe at r in bA'.

PROOF: By *bAInv*, step 2.1.4, and step 2.2.

2.4.2. r is classic $\Rightarrow bA_e[r]' \sqsubseteq$ *maxTried*$[r]'$

PROOF: By *bAInv* and steps 2.1.4 and 2.1.6.

2.4.3. r is fast $\Rightarrow bA_e[r]' \in$ *Str*(*propCmd*′)

PROOF: By *bAInv* and steps 2.1.4 and 2.1.5.

2.4.4. Q.E.D.

2.5. *learnedInv*′ is true.

LET: h be any learner, without loss of generality.

2.5.1. *learned*$[h]' \in$ *Str*(*propCmd*′)

PROOF: By *learnedInv* and steps 2.1.5 and 2.1.7.

2.5.2. *learned*$[h]'$ is the lub of a finite set of c-structs chosen in *bA*$'$.
PROOF: *learnedInv* and step 2.1.7 imply that *learned*$[h]'$ is the lub of a finite set of c-structs chosen in *bA*. Step 2.1.4 and the definition of a chosen value imply that a value that is chosen in *bA* is also chosen in *bA*$'$.
2.5.3. Q.E.D.
2.6. Q.E.D.

3. CASE: Action *StartBallot*(m, w) is executed, where m is a balnum and $w \in$ *CStruct*.
PROOF SKETCH: Action *StartBallot*(m, w) changes *maxTried*$[m]$ from *none* to w, which is ensured to be both proposed and safe at m in *bA*. The action does not change the other variables. It preserves the *maxTried* invariant because w is proposed and safe at m in *bA*. It preserves the *bA* invariant because it does not change *bA* and *bA*$_e[m]$ is ensured to equal *none*, for any acceptor e, by the *bA* invariant itself. It preserves the *learned* invariant because it does not change *learned* or *bA*.
3.1. 1. *maxTried*$[m] =$ *none*
2. w is safe at m in *bA*
3. $w \in$ *Str*(*propCmd*)
4. *propCmd*$' =$ *propCmd*
5. *maxTried*$[m]' = w$
6. $\forall i \in BalNum \setminus \{m\} : maxTried[i]' = maxTried[i]$
7. *bA*$' =$ *bA*
8. *learned*$' =$ *learned*
PROOF: By the definition of action *StartBallot*(m, w).
3.2. *maxTriedInv*$'$ is true.
From its definition, it suffices to:
ASSUME: *maxTried*$[r]' \neq$ *none*, for any balnum r
PROVE: 1. *maxTried*$[r]' \in$ *Str*(*propCmd*$'$) and
2. *maxTried*$[r]'$ is safe at r in *bA*$'$.
3.2.1. *maxTried*$[r]' \in$ *Str*(*propCmd*$'$)
PROOF: If $r = m$, by steps 3.1.(3-5). If $r \neq m$, it is implied by *maxTriedInv* and steps 3.1.4 and 3.1.6.
3.2.2. *maxTried*$[r]'$ is safe at r in *bA*$'$.
PROOF: If $r = m$, by steps 3.1.2, 3.1.5, and 3.1.7. If $r \neq m$, it is implied by *maxTriedInv* and steps 3.1.(6-7).
3.2.3. Q.E.D.
3.3. *bAInv*$'$ is true.
From its definition, it suffices to:
ASSUME: *bA*$_e[r]' \neq$ *none*, for any agent e and balnum r
PROVE: 1. *bA*$_e[r]'$ is safe at r in *bA*$'$,
2. r is classic $\Rightarrow$ *bA*$_e[r]' \sqsubseteq$ *maxTried*$[r]'$, and
3. r is fast $\Rightarrow$ *bA*$_e[r]' \in$ *Str*(*propCmd*$'$).
3.3.1. *bA*$_e[r]'$ is safe at r in *bA*$'$.

PROOF: By *bAInv* and step 3.1.7.

3.3.2. r is classic $\Rightarrow$ $bA_e[r]' \sqsubseteq maxTried[r]'$

PROOF: Step 3.1.1 and *bAInv* imply that $bA_e[m] = none$. Now, if we apply step 3.1.7, we get that $bA_e[m]' = none$. This implies, by the assumption of step 3.3, that $r \neq m$. Thus, the proof follows from *bAInv* and steps 3.1.(6-7).

3.3.3. r is fast $\Rightarrow$ $bA_e[r]' \in Str(propCmd')$

PROOF: By *bAInv* and steps 3.1.4 and 3.1.7.

3.3.4. Q.E.D.

3.4. *learnedInv*′ is true.

LET: h be any learner, without loss of generality.

3.4.1. $learned[h]' \in Str(propCmd')$

PROOF: By *learnedInv* and steps 3.1.4 and 3.1.8.

3.4.2. $learned[h]'$ is the lub of a finite set of c-structs chosen in bA'.

PROOF: By *learnedInv* and steps 3.1.(7-8).

3.4.3. Q.E.D.

3.5. Q.E.D.

4. CASE: Action $Suggest(m, \sigma)$ is executed, where m is a ballot number and σ is a c-seq.

PROOF SKETCH: Action $Suggest(m, \sigma)$ changes $maxTried[m]$ to $maxTried[m] \bullet \sigma$, where σ is a sequence of proposed commands. The action does not change the other variables. It preserves the *maxTried* invariant because σ is proposed and, by the definition of *safe at*, any extension of a value that is safe at m in *bA* is also safe at m in *bA*. It preserves the *bA* invariant because it does not change *bA* and the *bA* invariant ensures that $bA_e[m]$ is either *none* or a value v such that $v \sqsubseteq maxTried[m]$, for any acceptor e. It preserves the *learned* invariant because it does not change *learned* or *bA*.

4.1. 1. $maxTried[m] \neq none$
2. $\sigma \in Seq(propCmd)$
3. $propCmd' = propCmd$
4. $maxTried[m]' = maxTried[m] \bullet \sigma$
5. $\forall i \in BalNum \setminus \{m\} : maxTried[i]' = maxTried[i]$
6. $bA' = bA$
7. $learned' = learned$

PROOF: By the definition of action $Suggest(m, \sigma)$.

4.2. *maxTriedInv*′ is true.

From its definition, it suffices to:

ASSUME: $maxTried[r]' \neq none$, for any balnum r

PROVE: 1. $maxTried[r]' \in Str(propCmd')$ and
2. $maxTried[r]'$ is safe at r in bA'.

4.2.1. $maxTried[r]' \in Str(propCmd')$

PROOF: If $r = m$, by *maxTriedInv* and steps 4.1.(2-4). If $r \neq m$, it is implied

by *maxTriedInv* and steps 4.1.3 and 4.1.5.

4.2.2. $maxTried[r]'$ is safe at r in bA'.

PROOF: *maxTriedInv* implies that $maxTried[r]$ is safe at r in bA, and step 4.1.6 states that $bA = bA'$. Steps 4.1.(4-5) complete the proof by implying that $maxTried[r] \sqsubseteq maxTried[r]'$. This is enough because the definition of *safe at* implies that if w is safe at k in β, then v is safe at k in β, for any v such that $w \sqsubseteq v$.

4.2.3. Q.E.D.

4.3. *bAInv*′ is true.

From its definition, it suffices to:

ASSUME: $bA_e[r]' \neq none$, for any agent e and balnum r

PROVE: 1. $bA_e[r]'$ is safe at r in bA',
2. r is classic $\Rightarrow bA_e[r]' \sqsubseteq maxTried[r]'$, and
3. r is fast $\Rightarrow bA_e[r]' \in Str(propCmd')$.

4.3.1. $bA_e[r]'$ is safe at r in bA'.

PROOF: By *bAInv* and step 4.1.6.

4.3.2. r is classic $\Rightarrow bA_e[r]' \sqsubseteq maxTried[r]'$

PROOF: Step 4.1.6 implies that $bA_e[r]' = bA_e[r]$, *bAInv* implies that $bA_e[r] \sqsubseteq maxTried[r]$, and steps 4.1.(4-5) imply that $maxTried[r] \sqsubseteq maxTried[r]'$, completing the proof.

4.3.3. r is fast $\Rightarrow bA_e[r]' \in Str(propCmd')$

PROOF: By *bAInv* and steps 4.1.3 and 4.1.6.

4.3.4. Q.E.D.

4.4. *learnedInv*′ is true.

LET: h be any learner, without loss of generality.

4.4.1. $learned[h]' \in Str(propCmd')$

PROOF: By *learnedInv* and steps 4.1.3 and 4.1.7.

4.4.2. $learned[h]'$ is the lub of a finite set of c-structs chosen in bA'.

PROOF: By *learnedInv* and steps 4.1.(6-7).

4.4.3. Q.E.D.

4.5. Q.E.D.

5. CASE: Action $ClassicVote(a, m, v)$ is executed, where a is an acceptor, m is a ballot number, and $v \in CStruct$.

PROOF SKETCH: Action *ClassicVote*(a,m,v) sets $bA_a[m]$ to c-struct v only if $m \geq \widehat{bA}_a$, v is ensured to be safe at m in *bA*, and $bA_a[m]$ equals *none* or $bA_a[m] \sqsubseteq v$. It is also a pre-condition to this action that $v \sqsubseteq maxTried[m]$, which implies that v is proposed, by the *maxTried* invariant. Since only entry $bA_a[m]$ ($m \geq \widehat{bA}_a$) is changed together with $\widehat{bA}_a$, which is set to m, and the definition of *choosable at* considers only entries $bA_e[j]$ where $j < \widehat{bA}_e$, no value can be made unsafe at any balnum after the execution of this action. It preserves the *maxTried* invariant because it does not change *maxTried* or *propCmd* and it does not make any entry unsafe. It preserves the *bA* invariant because no entry is made unsafe and the only entry it changes in *bA* is set to a safe and proposed value. It preserves the *learned* invariant because it does not change *learned* and any value that is chosen in *bA* remains chosen after the action is executed, by the definition of *chosen*.

5.1. 1. $m \geq \widehat{bA}_a$
2. v is safe at m in *bA*
3. $v \sqsubseteq maxTried[m]$
4. $bA_a[m] = none \vee bA_a[m] \sqsubseteq v$
5. $propCmd' = propCmd$
6. $maxTried' = maxTried$
7. $\forall i \in Acceptor \setminus \{a\} : \widehat{bA}_i' = \widehat{bA}_i$
8. $\widehat{bA}_a' = m$
9. $\forall i \in Acceptor, j \in BalNum : (i \neq a \vee j \neq m) \Rightarrow bA_i[j]' = bA_i[j]$
10. $bA_a[m]' = v$
11. $learned' = learned$

PROOF: By the definition of action *ClassicVote*(a,v).

5.2. If x is safe at s in *bA*, for any c-struct x and balnum s, then x is safe at s in *bA*$'$.

The proof is by contradiction, as follows.

ASSUME: There exist c-struct x and balnum s, such that
1. x is safe at s in *bA*
2. x is not safe at s in *bA*$'$

PROVE: FALSE

5.2.1. Choose c-struct w and balnum k such that $k < s$, w is choosable at k in *bA*$'$, and $w \not\sqsubseteq x$.

PROOF: w and k exist by assumption 5.2.2 and the definition of *safe at*.

5.2.2. w is choosable at k in *bA*.

5.2.2.1. Choose k-quorum Q such that

$$\forall e \in Q : \widehat{bA}_e' > k \Rightarrow w \sqsubseteq bA_e[k]'$$

PROOF: Q exists by the definition of *choosable at*.

5.2.2.2. $\forall e \in Q : \widehat{bA}_e > k \Rightarrow w \sqsubseteq bA_e[k]$

PROOF: Steps 5.1.(8-9) imply that $\forall i \in Acceptor : \widehat{bA}_i' > k \Rightarrow bA_i[k]' = bA_i[k]$. This equation applied to the one of step 5.2.2.1 leads to

$$\forall e \in Q : \widehat{bA}_e' > k \Rightarrow w \sqsubseteq bA_e[k].$$

Steps 5.1.(1,7-8) imply that $\forall i \in \mathit{Acceptor} : \widehat{bA_i}' \geq \widehat{bA_i}$. Thus, $\forall e \in Q : \widehat{bA_e} > k \Rightarrow \widehat{bA_e}' > k$, which together with the previous equation leads us to

$$\forall e \in Q : \widehat{bA_e} > k \Rightarrow w \sqsubseteq bA_e[k].$$

5.2.2.3. Q.E.D.

PROOF: By step 5.2.2.2 and the definition of *choosable at*.

5.2.3. Q.E.D.

PROOF: If w is choosable at $k < s$ in *bA* (step 5.2.2), and $w \not\sqsubseteq x$ (step 5.2.1), then x is not safe at s in *bA*, contradicting assumption 1 of step 5.2.

5.3. *maxTriedInv*′ is true.

From its definition, it suffices to:

ASSUME: $\mathit{maxTried}[r]' \neq \mathit{none}$, for any balnum r

PROVE: 1. $\mathit{maxTried}[r]' \in \mathit{Str}(\mathit{propCmd}')$ and
2. $\mathit{maxTried}[r]'$ is safe at r in bA'.

5.3.1. $\mathit{maxTried}[r]' \in \mathit{Str}(\mathit{propCmd}')$

PROOF: By steps 5.1.(5-6).

5.3.2. $\mathit{maxTried}[r]'$ is safe at r in bA'.

PROOF: By *maxTriedInv*, step 5.1.6 and step 5.2.

5.3.3. Q.E.D.

5.4. *bAInv*′ is true.

From its definition, it suffices to:

ASSUME: $bA_e[r]' \neq \mathit{none}$, for any agent e and balnum r

PROVE: 1. $bA_e[r]'$ is safe at r in bA',
2. r is classic $\Rightarrow bA_e[r]' \sqsubseteq \mathit{maxTried}[r]'$, and
3. r is fast $\Rightarrow bA_e[r]' \in \mathit{Str}(\mathit{propCmd}')$.

5.4.1. $bA_e[r]'$ is safe at r in bA'.

PROOF: If $e = a$ and $r = m$, it follows from steps 5.1.2 and 5.1.10. Otherwise, it follows from *bAInv*, step 5.1.9, and step 5.2.

5.4.2. r is classic $\Rightarrow bA_e[r]' \sqsubseteq \mathit{maxTried}[r]'$

PROOF: If $e = a$ and $r = m$, it follows from steps 5.1.3 and 5.1.10. Otherwise, it follows from *bAInv* and step 5.1.9.

5.4.3. r is fast $\Rightarrow bA_e[r]' \in \mathit{Str}(\mathit{propCmd}')$

PROOF: If $e = a$ and $r = m$, it follows from steps 5.1.3 and 5.1.10, and *maxTriedInv*. Otherwise, it follows from *bAInv* and step 5.1.9.

5.4.4. Q.E.D.

5.5. *learnedInv*′ is true.

LET: h be any learner, without loss of generality.

5.5.1. $\mathit{learned}[h]' \in \mathit{Str}(\mathit{propCmd}')$

PROOF: By *learnedInv* and steps 5.1.5 and 5.1.11.

5.5.2. $\mathit{learned}[h]'$ is the lub of a finite set of c-structs chosen in bA'.

PROOF: *learnedInv* and step 5.1.11 imply that $\mathit{learned}[h]'$ is the lub of a finite

set of c-structs chosen in *bA*. Steps 5.1.(4,9-10) state that the only entry of *bA* that is modified, is extended. The definition of a chosen value, therefore, implies that a value that is chosen in *bA* is also chosen in bA', completing the proof.

5.5.3. Q.E.D.

5.6. Q.E.D.

6. CASE: Action *FastVote*(a, C) is executed, where a is an acceptor and $C \in Cmd$.

PROOF SKETCH: Action *FastVote*(a, C) sets $bA_a[\widehat{bA}_a]$ to c-struct $bA_a[\widehat{bA}_a] \bullet C$ only if $bA_a[\widehat{bA}_a]$ does not equal *none* and C is proposed. Since only $bA_a[\widehat{bA}_a]$ is changed and the definition of *choosable at* considers only entries $bA_e[m]$ where $m < \widehat{bA}_e$, no value can be made unsafe at any balnum after the execution of this action. It preserves the *maxTried* invariant because it does not change *maxTried* or *propCmd* and it does not make any entry unsafe. It preserves the *bA* invariant because no entry is made unsafe and the only entry it changes in *bA* is extended with a proposed value, and the extension of a safe c-struct is also safe. It preserves the *learned* invariant because it does not change *learned* and any value that is chosen in *bA* remains chosen after the action is executed, by the definition of *chosen*.

6.1. 1. $C \in propCmd$

2. $\widehat{bA}_a$ is a fast balnum

3. $bA_a[\widehat{bA}_a] \neq none$

4. $propCmd' = propCmd$

5. $maxTried' = maxTried$

6. $bA_a[\widehat{bA}_a]' = bA_a[\widehat{bA}_a] \bullet C$

7. $\forall i \in Acceptor : \widehat{bA}_i' = \widehat{bA}_i$

8. $\forall i \in Acceptor, j \in BalNum : (i \neq a \vee j \neq \widehat{bA}_a) \Rightarrow bA_i[j]' = bA_i[j]$

9. $learned' = learned$

PROOF: By the definition of action *FastVote*(a, v).

6.2. If x is safe at s in *bA*, for any c-struct x and balnum s, then x is safe at s in bA'.

The proof is by contradiction, as follows.

ASSUME: There exist c-struct x and balnum s, such that

1. x is safe at s in *bA*
2. x is not safe at s in bA'

PROVE: FALSE

6.2.1. Choose c-struct w and balnum k such that $k < s$, w is choosable at k in bA', and $w \not\sqsubseteq x$.

PROOF: w and k exist by assumption 6.2.2 and the definition of *safe at*.

6.2.2. w is choosable at k in *bA*.

6.2.2.1. Choose k-quorum Q such that

$$\forall e \in Q : \widehat{bA}_e' > k \Rightarrow w \sqsubseteq bA_e[k]'$$

PROOF: Q exists by the definition of *choosable at*.

6.2.2.2. $\forall e \in Q : \widehat{bA}_e > k \Rightarrow w \sqsubseteq bA_e[k]$
PROOF: Step 6.1.7 states that $\widehat{bA}'_e = \widehat{bA}_e$, and step 6.1.8 implies that $bA_e[k]' = bA_e[k]$ if $\widehat{bA}'_e > k$.
6.2.2.3. Q.E.D.
PROOF: By step 6.2.2.2 and the definition of *choosable at*.
6.2.3. Q.E.D.
PROOF: If w is choosable at $k < s$ in *bA* (step 6.2.2), and $w \not\sqsubseteq x$ (step 6.2.1), then x is not safe at s in *bA*, contradicting assumption 1 of step 6.2.
6.3. *maxTriedInv*′ is true.
From its definition, it suffices to:
ASSUME: *maxTried*$[r]' \neq$ *none*, for any balnum r
PROVE: 1. *maxTried*$[r]' \in$ *Str*(*propCmd*′) and
2. *maxTried*$[r]'$ is safe at r in *bA*′.
6.3.1. *maxTried*$[r]' \in$ *Str*(*propCmd*′)
PROOF: By steps 6.1.(4-5).
6.3.2. *maxTried*$[r]'$ is safe at r in *bA*′.
PROOF: By *maxTriedInv*, step 6.1.5 and step 6.2.
6.3.3. Q.E.D.
6.4. *bAInv*′ is true.
From its definition, it suffices to:
ASSUME: $bA_e[r]' \neq$ *none*, for any agent e and balnum r
PROVE: 1. $bA_e[r]'$ is safe at r in *bA*′,
2. r is classic $\Rightarrow bA_e[r]' \sqsubseteq$ *maxTried*$[r]'$, and
3. r is fast $\Rightarrow bA_e[r]' \in$ *Str*(*propCmd*′).
6.4.1. $bA_e[r]'$ is safe at r in *bA*′.
PROOF: If $e = a$ and $r = \widehat{bA}_a$, it follows from steps 6.1.3 and 6.1.6, *bAInv*, and the definition of *safe at*, which implies that the extension of a safe value is also safe. Otherwise, it follows from *bAInv*, step 6.1.8, and step 6.2.
6.4.2. r is classic $\Rightarrow bA_e[r]' \sqsubseteq$ *maxTried*$[r]'$
PROOF: It follows from *bAInv* and step 6.1.8, since $r = \widehat{bA}_e$ and $e = a$ imply that r is not classic, by step 6.1.2.
6.4.3. r is fast $\Rightarrow bA_e[r]' \in$ *Str*(*propCmd*′)
PROOF: If $e = a$ and $r = \widehat{bA}_a$, it follows from *bAInv* and steps 6.1.1 and 6.1.6. Otherwise, it follows from *bAInv* and step 6.1.8.
6.4.4. Q.E.D.
6.5. *learnedInv*′ is true.
LET: h be any learner, without loss of generality.
6.5.1. *learned*$[h]' \in$ *Str*(*propCmd*′)
PROOF: By *learnedInv* and steps 6.1.4 and 6.1.9.
6.5.2. *learned*$[h]'$ is the lub of a finite set of c-structs chosen in *bA*′.

PROOF: *learnedInv* and step 6.1.9 imply that $learned[h]'$ is the lub of a finite set of c-structs chosen in *bA*. Steps 6.1.(3,6-8) state that the only entry of *bA* that is modified, is extended. The definition of a chosen value, therefore, implies that a value that is chosen in *bA* is also chosen in bA', completing the proof.

6.5.3. Q.E.D.

6.6. Q.E.D.

7. CASE: Action $AbstractLearn(l, v)$ is executed, where l is a learner and $v \in CStruct$.

PROOF SKETCH: Action $AbstractLearn(l, v)$ only changes variable *learned*, which is the array of learned c-structs, and does that by extending one entry to the lub of it with a chosen c-struct. Invariants *maxTried* and *bA* are obviously preserved. The first part of the *learned* invariant is preserved because this extension is proposed, by the definition of *chosen at*, the *bA* invariant and axiom CS3. The second part is obviously preserved by its definition and the one of the action.

7.1. 1. v is chosen in *bA*

2. $propCmd' = propCmd$

3. $maxTried' = maxTried$

4. $bA' = bA$

5. $learned[l]' = learned[l] \sqcup v$

6. $\forall i \in Learner \setminus \{l\} : learned[i]' = learned[i]$

PROOF: By the definition of action $AbstractLearn(l, v)$.

7.2. $maxTriedInv'$ is true.

From its definition, it suffices to:

ASSUME: $maxTried[r]' \neq none$, for any balnum r

PROVE: 1. $maxTried[r]' \in Str(propCmd')$ and

2. $maxTried[r]'$ is safe at r in bA'.

7.2.1. $maxTried[r]' \in Str(propCmd')$

PROOF: By *maxTriedInv* and steps 7.1.(2-3).

7.2.2. $maxTried[r]'$ is safe at r in bA'.

PROOF: By *maxTriedInv* and steps 7.1.(3-4).

7.2.3. Q.E.D.

7.3. $bAInv'$ is true.

From its definition, it suffices to:

ASSUME: $bA_e[r]' \neq none$, for any acceptor e and balnum r

PROVE: 1. $bA_e[r]'$ is safe at r in bA',

2. r is classic $\Rightarrow bA_e[r]' \sqsubseteq maxTried[r]'$, and

3. r is fast $\Rightarrow bA_e[r]' \in Str(propCmd')$.

7.3.1. $bA_e[r]'$ is safe at r in bA'.

PROOF: By *bAInv* and step 7.1.4.

7.3.2. r is classic $\Rightarrow bA_e[r]' \sqsubseteq maxTried[r]'$

PROOF: By *bAInv* and steps 7.1.(3-4).

7.3.3. r is fast $\Rightarrow$ $bA_e[r]' \in Str(propCmd')$
PROOF: By *bAInv* and steps 7.1.2 and 7.1.4.
7.3.4. Q.E.D.
7.4. *learnedInv*′ is true.
LET: h be any learner, without loss of generality.
7.4.1. $learned[h]' \in Str(propCmd')$
PROOF: If $h \neq l$, it follows from *learnedInv* and steps 7.1.6 and 7.1.2. Otherwise, the definition of a chosen value, *bAInv*, and step 7.1.1 imply that $v \in Str(propCmd)$. *learnedInv* implies that $learned[l] \in Str(propCmd)$. Proposition 1 and *learnedInv* imply that v and $learned[l]$ are compatible, and axiom CS3 states that its lub exists and must be in $Str(propCmd)$. The proof is completed by steps 7.1.2 and 7.1.5.
7.4.2. $learned[h]'$ is the lub of a finite set of c-structs chosen in bA'.
PROOF: If $h \neq l$, it follows from *learnedInv* and step 7.1.6. Otherwise, it follows from *learnedInv* and steps 7.1.1 and 7.1.5.
7.4.3. Q.E.D.
7.5. Q.E.D.

□

A.1.3 Distributed Abstract Multicoordinated Paxos

As an intermediate step in our proof, we introduce a distributed version of the abstract algorithm in the previous section. This algorithm has the variables *propCmd*, *learned*, and *bA* with the same role as in the non-distributed abstract algorithm. It introduces the variables *dMaxTried*, a distributed version of *maxTried*, and *msgs*, used to simulate a message passing system by holding the messages sent between coordinators, acceptors, and learners.

propCmd The set of proposed commands. It initially equals the empty set.

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

bA A ballot array. It represents the current state of the voting. Initially, $\widehat{bA}_a = 0$, $bA_a[0] = \bot$ and $bA_a[m] = none$ for all acceptor a and ballot number $m > 0$. (Every acceptor casts a default vote for $\bot$ in ballot 0, so the algorithm begins with $\bot$ chosen.)

dMaxTried An array of arrays of c-structs, where $dMaxTried[c][m]$ is either a c-struct or equal to *none*, for every coordinator c and balnum m. Initially, $dMaxTried[c][0] = \bot$ and $dMaxTried[c][m] = none$ for every coordinator c and all balnum $m > 0$.

msgs The set of messages sent by coordinators and acceptors. (This set is used to simulate the message passing among processes.)

The distributed abstract algorithm is described in terms of the following actions. Its formal specification in TLA$^+$ is given in the appendix section A.2.3.

Propose(C) executed by the proposer of command C. The action is always enabled. It sets *propCmd* to *propCmd*$\cup\{C\}$, from where coordinators and acceptors can read C.

Phase1a(c,m) executed by coordinator c, for balnum m. The action is enabled iff $dMaxTried[c][m] = none$. It sends the message $\langle$"1a", $m\rangle$ to acceptors (adds it to *msgs*).

Phase1b(a,m) executed by acceptor a, for balnum m. The action is enabled iff

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

It sets $\widehat{bA}_a$ to m and sends the message $\langle$"1b", $m, bA_a\rangle$ to the coordinators.

Phase2Start(c,m,v) executed by coordinator c, for balnum m, and c-struct v. The action is enabled iff:

- $dMaxTried[c][m] = none$,
- There exists an m-quorum Q such that for all $a \in Q$, there is a message $\langle$"1b", $m, bA_a\rangle \in msgs$ coming from a, and
- $v = w \bullet \sigma$, where $\sigma \in Seq(propCmd)$, $w \in ProvedSafe(Q,m,\beta)$, and β is any ballot array such that, for every acceptor a in Q, $\widehat{\beta}_a = m$ and c has received a message $\langle$"1b", $m,\rho\rangle$ from a with $\rho = \beta_a$.

This action sets $dMaxTried[c][m]$ to v and sends the message $\langle$"2a", $m, v\rangle$ to acceptors.

Phase2aClassic(c,m,C) executed by coordinator c, for balnum m and command C. The action is enabled iff

- $C \in propCmd$ and
- $dMaxTried[c][m] \neq none$

This action sends the message $\langle$"2a", $m, c, dMaxTried[c][m] \bullet C\rangle$ to the acceptors and sets $dMaxTried[c][m]$ to $dMaxTried[c][m] \bullet C$.

$Phase2bClassic(a, m, v)$ executed by acceptor a, for balnum m and c-struct v. The action is enabled iff

- $m \geq \widehat{bA}_a$,
- there is an m-coordquorum L and a c-struct u such that, for every $c \in L$, acceptor a has received a phase "2a" message for balnum m with value w satisfying $u \sqsubseteq w$, i.e., u is a lower bound for the w values, and
- Either $bA_a[m]$ equals *none* and v equals u, or $bA_a[m]$ and u are compatible and v equals $bA_a[m] \sqcup u$.

It sets $bA_a[m]$ to v, $\widehat{bA}_a$ to m, and sends the message $\langle$"2b", $m, v\rangle$ to the learners.

$Phase2bFast(a, C)$ executed by acceptor a, for balnum m and command C. The action is enabled iff

- $\widehat{bA}_a$ is a fast balnum,
- $bA_a[\widehat{bA}_a] \neq none$, and
- $C \in propCmd$.

It sets $bA_a[\widehat{bA}_a]$ to $bA_a[\widehat{bA}_a] \bullet C$ and sends a message $\langle$"2b", $\widehat{bA}_a, bA_a[\widehat{bA}_a] \bullet C\rangle$ to the learners.

$Learn(l, v)$ executed by learner l, for c-struct v. It is enabled iff a has received phase "2b" messages for some round i from an i-quorum Q and v is a prefix of the values on those messages. 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. This implementation is stated by the following proposition.

Proposition 6 *Distributed Abstract Multicoordinated Paxos implements the Abstract Multicoordinated Paxos specification.*

To prove this proposition we provide a refinement mapping from the distributed version's states to the non-distributed version's [Abadi and Lamport, 1991].

In the following we replace the variables in the non-distributed algorithms by overlined versions. That is, we let expression $\overline{A}$ refer to expression E, in the non-distributed algorithm specification, where all occurrences of variables *propCmd*, *maxTried*, *bA*, and *learned* are replaced by $\overline{propCmd}$, $\overline{maxTried}$, $\overline{bA}$, and $\overline{learned}$, respectively. Non-overlined expressions refer to those in the distributed algorithm.

We give the refinement mapping by defining overlined variables based on the distributed algorithm's variables in a way that satisfies the non-distributed specification.

Let the overlined variables be defined as follows.

$\overline{propCmd} \triangleq propCmd$

$\overline{learned} \triangleq learned$

$\overline{maxTried} \triangleq$
$\quad$ LET $Tried(Q, m) \triangleq$ IF $\exists c \in Q : dMaxTried[c][m] = none$
$\qquad\qquad\qquad\qquad$ THEN $none$
$\qquad\qquad\qquad\qquad$ ELSE $\sqcap \{dMaxTried[c][m] : c \in Q\}$
$\qquad AllTried(m) \triangleq \{v \in \{Tried(Q, m) : Q$ is an m-coordquorum$\} :$
$\qquad\qquad\qquad\qquad v \neq none\}$
$\quad$ IN $[m \in BalNum \mapsto$ IF $AllTried(m) = \{\}$ THEN $none$
$\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad$ ELSE $\sqcup AllTried(m)]$

To prove that this is a valid refinement mapping and witnesses the implementation of the Abstract Multicoordinated Paxos by Distributed Abstract Multicoordinated Paxos we must show that the distributed version's intial states imply the non-distributed version's initial states, and that each step of the distributed version implies a step in the non-distributed version, be it one that changes the overlined variables or does not change anything (a stuttering step). To simplify these proofs we first prove some properties of c-structs and of our refinement mapping. (The first proposition actually regards c-structs in general)

Proposition 7 *If, for some ballot number m and ballot array bA, all elements of S are safe at m in bA, then $\sqcup S$ is also safe at m in bA.*

By the definition of "safe at", for all c-structs v choseable at round m in *bA* and for all $w \in S$, $v \sqsubseteq w$. Moreover, by the definition of "lower bound", v is a lower bound of S and, by the definition of $\sqcup$, $v \sqsubseteq \sqcup S$. Therefore, all elements of S extend v and all c-structs choosable at m in *bA*, being safe at m in *bA*. □

Proposition 8 *$AllTried(m)$ is compatible for $m \geq 0$.*

PROOF: By the definition of glb, the empty set is compatible and $\sqcup\{\} = \bot$. If $AllTried(m)$ is not empty, then it contains the $Tried(Q, m)$ for all Q such that $Tried(Q, m) \neq none$. By the definition of $Tried$ and $\sqcap$, if $Tried(Q, m) \neq none$, then it is a prefix of $dMaxTried[c][m]$ for all $c \in Q$.
Let $Tried(Q, m), Tried(R, m) \in AllTried(m)$. By the coord-quorum-assumption, there exists a coordinator $c \in Q \cap R$, and $AllTried(Q, m) \sqsubseteq dMaxTried[c][m]$ and

$AllTried(R,m) \sqsubseteq dMaxTried[c][m]$. Therefore, $dMaxTried[c][m]$ is an upper bound to $\{Tried(Q,m), Tried(R,m)\}$, and they are compatible. Because its elements are pairwise compatible and due to **CS3**, $AllTried(m)$ is compatible. □

Proposition 9 *dMaxTried* = *dmaxTried*′ ⇒ $\overline{maxTried} = \overline{maxTried}'$.

PROOF: $\overline{maxTried}$ is defined only over *dMaxTried* values. If *dMaxTried* does not change, then *maxTried* cannot change. □

Hereinafter we refer to to Distributed Abstract Multicoordinated Paxos as DAP and Abstract Multicoordinated Paxos as AP. As we mentioned before, the proof of Proposition 6, i.e., that DAP implements the AP, is divided in two steps: proving the implication among the initial states and among the steps. The first step is captured by Proposition 10 and the second step by Proposition 11, below.

Proposition 10 *DAP's initial state implies AP's initial state.*

PROOF SKETCH: First we prove that DAP's initial state implies that $\overline{maxTried}$ is initialized as specified in AP's. Because the correct initialization of the other variables are trivially implied (they have the same initialization), we conclude the proof.

1. $\overline{maxTried}[0] = \bot$
   PROOF: By the specification of DAP, $dMaxTried[c][0] = \bot$ for each coordinator c. By the definition of $\overline{maxTried}$, *Tried*, and glb, $Tried(Q,0) = \bot$ for any Q. Therefore, by definition of *AllTried*, $AllTried(0) = \bot$, and by the definition of lub, $\overline{maxTried}[0] = \bot$.
2. $\forall m > 0, \overline{maxTried}[m] = none$
   PROOF: By the specification of DAP, $dMaxTried[c][m] = none$ for each coordinator c and round $m > 0$. By the definition of $\overline{maxTried}$ and *Tried*, $Tried(Q,m) = none$ for any Q. Hence, by the definition of *AllTried*, $AllTried(m) = \{\}$. By the definition of $\overline{maxTried}$, $\overline{maxTried}[m] = none$ for any $m > 0$.
3. Q.E.D.

□

Proposition 11 *A DAP step implements an AP step (a possibly stuttering one).*

PROOF SKETCH: We consider each action of DAP and show that each step either implies a step of AP or that AP's variables—propCmd, learned, *bA*, and $\overline{maxTried}$—are left unchanged. We use TLA$^+$ UNCHANGED v notation to indicate that variable (or sequence of variables) v did not change in some step.

1. ASSUME: $\wedge\ C \in Cmd$
   $\wedge\ Propose(C)$
   PROVE: $\overline{Propose(C)}$

1.1. $C \notin propCmd$
PROOF: By the definition of *Propose*.
1.2. $propCmd' = propCmd \cup \{C\}$
PROOF: By the definition of *Propose*.
1.3. UNCHANGED $\langle learned, bA, \overline{maxTried}\rangle$
PROOF: By the definition of *Propose* and Proposition 9.
1.4. Q.E.D.

2. ASSUME: $\land\ c \in Coord$
$\land\ m \in BalNum$
$\land\ Phase1a(c, m)$
PROVE: UNCHANGED $\langle propCmd, learned, bA, \overline{maxTried}\rangle$
PROOF: By the definition of *Phase1a* and Proposition 9.

3. ASSUME: $\land\ c \in Coord$
$\land\ m \in BalNum$
$\land\ v \in CStruct$
$\land\ Phase2Start(c, m, v)$
PROVE: $\land \lor \land \overline{maxTried} = none$
$\land \lor \overline{maxTried}' = none$
$\lor\ StartBallot(m, \overline{maxTried}'[m])$
$\lor \land \overline{maxTried} \neq none$
$\land \lor \exists \sigma \in Seq(propCmd) :$
$\land\ \overline{maxTried}'[m] = \overline{maxTried}[m] \bullet \sigma$
$\land\ Suggest(m, \sigma)$
$\lor$ UNCHANGED $\overline{maxTried}$
$\land$ UNCHANGED $\langle propCmd, bA, learned\rangle$
3.1. ASSUME: $\overline{maxTried}[m] = none$
PROVE: $\lor\ StartBallot(m, \overline{maxTried}'[m])$
$\lor$ UNCHANGED $\overline{maxTried}$
3.1.1. ASSUME: $\overline{maxTried'}[m] \neq none$
PROVE: $StartBallot(m, \overline{maxTried}'[m])$
3.1.1.1. $\overline{maxTried}'[m] \sqsubseteq v$
PROOF: By the assumption, for all m-coordquorums Q such that $Tried(Q, m)' \in AllTried(m)'$, $c \in Q$, or $Tried(Q, m)'$ would equal $none$ and it would not belong to $AllTried(m)'$. By the definition of glb, $Tried(Q, m)' \sqsubseteq dMaxTried[c][m] = v$. Therefore, all elements of $AllTried(m)'$ are compatible prefixes of v, and v is an upper bound of $AllTried(m)'$. But, by the definition of $\overline{maxTried}$, $\overline{maxTried}'[m] =$

$\sqcup AllTried(m)'$ and by the definition of $\sqcup$ and Assumption 3, $\overline{maxTried}'[m]$ must be a prefix of v.

3.1.1.2. $\overline{maxTried}[m] = none$
PROOF: By assumption.

3.1.1.3. $\overline{maxTried}'[m]$ is safe at m in *bA*
PROOF: By the definitions of actions *ProvedSafe*, $Phase2Start$, and $Phase2aClassic$, for all $d \in Coord$, $dMaxTried[d][m]$ is first set to a safe value and then simply extended, therefore remaining safe. By the definition of $AllTried$ and Proposition 7, all elements of $AllTried(m)'$ are safe. Because $\overline{maxTried}[m]$ is an extension of such safe c-structs, it is also safe.

3.1.1.4. $\overline{maxTried}'[m] \in Str(propCmd)$
PROOF: By the definition of action $Phase2Start$, $v \in Str(propCmd)$. But, by step 3.1.1.1, $\overline{maxTried}'[m] \sqsubseteq v$ and $\overline{maxTried}'[m]$ is constructible from a subset of the commands in v. Therefore, by the definition of Str, $\overline{maxTried}'[m] \in Str(propCmd)$.

3.1.1.5. Q.E.D.

3.1.2. ASSUME: $\overline{maxTried}'[m] = none$
PROVE: UNCHANGED $\overline{maxTried}$
PROOF: By assumption.

3.1.3. Q.E.D.

3.2. ASSUME: $\overline{maxTried}[m] \neq none$
PROVE: $\vee \exists \sigma \in Seq(propCmd) :$
$\wedge\ \overline{maxTried}'[m] = \overline{maxTried}[m] \bullet \sigma$
$\wedge\ Suggest(m, \sigma)$
$\vee$ UNCHANGED $\overline{maxTried}$

3.2.1. ASSUME: $\overline{maxTried}'[m] \neq \overline{maxTried}[m]$
PROVE: $\exists \sigma \in Seq(propCmd) :$
$\wedge\ \overline{maxTried}'[m] = \overline{maxTried}[m] \bullet \sigma$
$\wedge\ Suggest(m, \sigma)$

3.2.1.1. $\exists \sigma \in Seq(propCmd) : \overline{maxTried}'[m] = \overline{maxTried}[m] \bullet \sigma$
PROOF: Let QD be the set of m-coordquorums Q such that $Tried(Q, m) \neq Tried(Q, m)'$; clearly by the assumption, QD is not empty and for all $Q \in QD$, $c \in Q$. Moreover, by the definition of $Tried$, $Tried(Q, m)' \sqsubseteq v$.
Let $SomeTried(SD, m) = \sqcup\{AllTried(S, m) : S \in SD\}$. Because for all $Q \in QD$, $Tried(Q, m) = none$, $SomeTried(\{S : S$ is an m-coordquorum and $S \notin QD\}, m) = AllTried(m)$.
Let Q and R be two m-coordquorums such that $Q \in QD$, $R \notin QD$, and $Q \cap R \neq \emptyset$. Moreover, let $d \in Q \cap R$. Therefore, $Tried(R, m) = Tried(R, m)'$,

$Tried(R, m)' \sqsubseteq dMaxTried[d][m]$ and $Tried(Q, m)' \sqsubseteq dMaxTried[d][m]$, and either $Tried(Q, m)' \sqsubseteq Tried(R, m)$ or $Tried(R, m) \sqsubseteq Tried(Q, m)'$. In the first case, $SomeTried(\{S : S \text{ is an } m\text{-coordquorum and } S \notin QD\}, m) = SomeTried(\{S : S \text{ is an } m\text{-coordquorum and } S \notin QD\} \cup \{Q\}, m)$, and is of no interest. In the second case, the equality may not hold, what would imply that $Tried(Q, m)'$ has some command that not in $AllTried(m)$. This second case must hold for some pair Q, R, since by assumption and the definition of $\overline{maxTried}$ $AllTried(m) \neq AllTried(m)'$. Without loss of generality, let R be such that $Tried(R, m)$ is maximal; by the definition of $\sqsubseteq$, $Tried(Q, m)' = Tried(R, m) \bullet \sigma$, for some sequence σ. As $Tried(Q, m)' \sqsubseteq v \in Seq(propCmd)$, $\sigma \in Seq(propCmd)$. Finally, by the definition of $\sqcup$, $AllTried(m)' = AllTried(m)$.

3.2.1.2. ASSUME: $\exists \sigma \in Seq(propCmd) :$
$\overline{maxTried}'[m] = \overline{maxTried}[m] \bullet \sigma$
PROVE: $Suggest(m, \sigma)$

PROOF: All pre and post-conditions of $Suggest$ are assumed:

- $maxTried[m] \neq none$,
- $\sigma \in Seq(propCmd)$, and
- $\overline{maxTried}'[m] = \overline{maxTried}[m] \bullet \sigma$.

3.2.1.3. Q.E.D.

3.2.2. ASSUME: $\overline{maxTried}'[m] = \overline{maxTried}[m]$
PROVE: UNCHANGED $\overline{maxTried}$

PROOF: By the assumption.

3.2.3. Q.E.D.

3.3. UNCHANGED $\langle propCmd, bA, learned \rangle$

PROOF: This is trivially true, because variables *propCmd*, *bA*, and *learned* are kept unchanged in *Phase2Start*.

3.4. Q.E.D.

4. ASSUME: $\land\ c \in Coord$
$\land\ m \in BalNum$
$\land\ C \in propCmd$
$\land\ Phase2aClassic(c, m, C)$
PROVE: $\land \lor Suggest(m, \langle C \rangle)$
$\lor$ UNCHANGED $\overline{maxTried}$
$\land$ UNCHANGED $\langle propCmd, learned, bA \rangle$

4.1. $\lor Suggest(m, \langle C \rangle)$
$\lor$ UNCHANGED $\overline{maxTried}$

PROOF SKETCH: A *Phase2aClassic* step either increases $\overline{maxTried}[m]$ by C or leaves it as it is. In the first case, it implements a *Suggest* step; in the second it implements a stutering step. We first show conditions that are necessary and sufficient for $\overline{maxTried}$ to stay unchanged on a *Phase2aClassic* step. We then show that if it changes, then a *Suggest* step follows.

4.1.1. ASSUME: $\overline{maxTried}'[m] \neq none$
PROVE: $\overline{maxTried}[m] \neq none$
PROOF: The proof is by contradiction. Suppose that $\overline{maxTried}[m] = none$. Then, by the definitions of *AllTried* and *Tried*, for all coordinator quorum m-coordquorum Q there is a coordinator $d \in Q$ such that $dMaxTried[d][m] = none$, and $Tried(Q, m) = none$. By the assumption, for some m-coordquorum Q, $Tried(Q, m)' \neq none$. Because $Tried(Q, m) = none$ and $Tried(Q, m)' \neq none$ and only $dMaxTried[c][m]$ was changed, c must be in in Q and $dMaxTried[c][m] = none$, but this contradicts a pre-condition of steps of type $Phase2aClassic$.

4.1.2. ASSUME: $\overline{maxTried}'[m] \neq \overline{maxTried}[m]$
PROVE: $\overline{maxTried}'[m] = \overline{maxTried}[m] \bullet C$
PROOF: By the assumption, $\overline{maxTried}'[m] \neq none$, and by the step 4.1.1, $\overline{maxTried}[m] \neq none$.
Let QD be the set of m-coordquorums Q such that $Tried(Q, m) \neq Tried(Q, m)'$; clearly, $\forall Q \in QD, c \in Q$.
By the assumption, QD is not empty and for all $Q \in QD$, $Tried(Q, m) \sqsubseteq dMaxTried[c][m]$ and $Tried(Q, m)' \sqsubseteq dMaxTried[c][m] \bullet C$. Because only $dMaxTried[c][m]$ was changed and $Tried(Q, m)' \neq Tried(Q, m)$, and by the definition of $\sqcap$, $Tried(Q, m)' = Tried(Q, m) \bullet C$.
Therefore, $\overline{maxTried}' = \sqcup(AllTried(m) \cup \{Tried(Q, m) \bullet C : Q \in QD\})$, and the set $AllTried(m) \cup \{Tried(Q, m) \bullet C : Q \in QD\}$ is compatible. Because the first set contains all commands of the second but C, the least upper bound of the first, $\overline{maxTried}$ differs from the least upper bound of the union, $\overline{maxTried}'$ only by the adition of C to the first, and, because they are compatible, $\overline{maxTried}' = \overline{maxTried} \bullet C$

4.1.3. ASSUME: $\overline{maxTried}' = \overline{maxTried} \bullet C$
PROVE: *Suggest*$(m, \langle C \rangle)$
4.1.3.1. $\langle C \rangle \in Seq(propCmd)$
PROOF: Because $C \in propCmd$ and by the definition of Seq.
4.1.3.2. $\overline{maxTried}[m] \neq none$
PROOF: By step 4.1.1.
4.1.3.3. $\overline{maxTried}'[m] = \overline{maxTried}[m] \bullet \langle C \rangle]$
PROOF: By step 4.1.2.
4.1.3.4. Q.E.D.

4.1.4. Q.E.D.

4.2. UNCHANGED $\langle propCmd, bA, learned \rangle$

PROOF: By the definition of *Phase2aClassic*, variables *propCmd*, *bA*, and *learned* are kept unchanged.

4.3. Q.E.D.

5. ASSUME: $\land\ a \in Acceptor$
$\land\ m \in BalNum$
$\land\ Phase1b(a, m)$
PROVE: $JoinBallot(a, m)$

PROOF: Any *Phase1b* step clearly implements a *JoinBallot* step, as all the latter's pre and post conditions are also required by the first.

6. ASSUME: $\land\ a \in Acceptor$
$\land\ m \in BalNum$
$\land\ v \in CStruct$
$\land\ Phase2bClassic(a, m, v)$
PROVE: $ClassicVote(a, m, v)$

6.1. $m \geq \widehat{bA_a}$

PROOF: By the definition of *Phase2bClassic*.

6.2. v is safe at round m in *bA*

PROOF: *Phase2bClassic* has as pre-condition that a has received a "2a" message from all coordinators in some coord-quorum L for round $\widehat{bA_a}$, and that there is a c-struct u such that for all such messages, u is a prefix of the value w in each one. Messages "2a" are sent in actions *Phase2Start* and *Phase2aClassic*, and in both cases the value sent is set to $dMaxTried[c][m]$, where c is the sender coordinator and m is the round in which the message was sent. Because $dMaxTried[c][m] = none$ is a pre-condition to action *Phase2Start* and it changes $dMaxTried[c][m]$ to something different from *none* and no other step changes it back to *none*, a *Phase2Start* step happens only once for a given c and m. A *Phase2aClassic* action is only enabled after this *Phase2Start* step, and it just appends some c-seq to the previous value of $dMaxTried[c][m]$. Therefore, the value set to $dMaxTried[c][m]$ by the *Phase2Start* step is always a prefix of $dMaxTried[c][m]$ in future states. Let $firstTried[c][m]$ be such initial value. By the definition of *Phase2Start* and Proposition 7, $firstTried[c][m]$ is safe at m in *bA*. Because $firstTried[c][m]$ is a prefix of any value sent by c on "2a" messages in round m, it is also a prefix of u, and u must be safe. Since v is equal to or an extension of u, it is also safe.

6.3. $v \sqsubseteq maxTried[m]$

PROOF: Let L be the m-coordquorum from which a has received the "2a" messages in the *Phase2BClassic* step and u be the common lower bound to all values received in such messages according to the action pre-condition. By the definition of *Tried* in $\overline{maxTried}$, by the definition of glb, and because $dMaxTried[c][m]$ is only extended for any coordinator c and balnum m, $u \sqsubseteq Tried(L, m)$. By the definition of $\overline{AllTried}$ in $\overline{maxTried}$ and lub, $Tried(l, m) \sqsubseteq \overline{maxTried}[m]$, and therefore $u \sqsubseteq \overline{maxTried}[m]$. Since $v \sqsubseteq u$, it follows that $v \sqsubseteq \overline{maxTried}[m]$

6.4. $\vee\ bA_a[m] = none$
$\vee\ bA_a[m] \sqsubseteq v$.
PROOF: By the definition of *Phase2bClassic*.

6.5. $bA_a[m]' = v$
PROOF: By the definition of *Phase2bClassic*.

6.6. UNCHANGED $\langle propCmd, \overline{maxTried}, learned\rangle$
PROOF: By Proposition 9 and the definition of *Phase2bClassic*.

6.7. Q.E.D.

7. ASSUME: $\wedge\ a \in Acceptor$
$\wedge\ C \in Cmd$
$\wedge\ Phase2bFast(a, C)$
PROVE: $FastVote(a, C)$
PROOF: Due to Proposition 9, and the definition of *Phase2bFast*, any *Phase2bFast* step is also a *FastVote* step, as all the latter's pre and post conditions are also satisfied by the first.

8. ASSUME: $\wedge\ l \in Learner$
$\wedge\ v \in CStruct$
$\wedge\ Learn(l, v)$
PROVE: $\overline{Learn}(l, v)$

8.1. v is chosen in *bA*
PROOF: The first pre-condition of *Learn* implies that all acceptors in some m-quorum Q executed action *Phase2bFast* or *Phase2bClassic*. Because commands and c-seqs can only be appended to $bA_a[m]$ in these actions, for all acceptors $a \in Q$, $v \sqsubseteq bA_a[m]$. Therefore, by the definition of **chosen at**, v is chosen at m in *bA*, and so is v is chosen at *bA*.

8.2. $\wedge\ learned'[l] = \overline{learned[l] \sqcup v}$
$\wedge$ UNCHANGED $\langle propCmd, \overline{maxTried}, bA\rangle$
PROOF: By the definition of *Learn*.

8.3. Q.E.D.
PROOF: By the definition of *AbstractLearn* and steps 8.1 and 8.2.

9. Q.E.D.

□

A.1.4 Multicoordinated Paxos

To prove correctness of the algorithm presented in Section 3.3, we first add the following history variables to the algorithm presented in the previous section.

crnd An array of balnums, where *crnd*[c] represents the current round of coordinator c. Initially 0.

cval An array of c-structs, where *cval*[c] represents the latest c-struct coordinator c has sent in a phase "2a" message for round *crnd*[c]. Initially $\bot$.

rnd An array of balnums, 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 balnums, where *vrnd*[a] is the round at which acceptor a has accepted the latest value. Initially 0.

vval An array of c-structs, where *vval*[a] is the c-struct acceptor a has accepted at *vrnd*[a]. Initially $\bot$.

msgs2 A set of messages sent by coordinators and acceptors. This variable is different from the original *msgs* variable.

We now make some simple changes to the algorithm's actions in order to update these history variables accordingly. Notice that the following algorithm is not exactly the same as in the previous section. The pre-conditions of actions *Phase1a*, *Phase2Start*, and *Phase2aClassic* are slightly more restrictive. However, it is an obvious implementation of the previous version.

Propose(C) executed by the proposer of command C. The action is always enabled. It sets *propCmd* to *propCmd*$\cup\{C\}$, from where coordinators and acceptors can read C.

Phase1a(c, m) executed by coordinator c, for balnum m. The action is enabled iff $\forall j \geq m$: *dMaxTried*[c][j] $= none$. It adds message $\langle$"1a", $m\rangle$ to *msgs* and *msgs2*.

Phase1b(a, m) executed by acceptor a, for balnum m. The action is enabled iff

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

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

Phase2Start(c, m, v) executed by coordinator c, for balnum m, and c-struct v. The action is enabled iff

- $\forall j \geq m$: *dMaxTried*[c][j] $= none$
- There exists an m-quorum Q such that for all $a \in Q$, there is a message $\langle$"1b", m, $bA_a\rangle \in msgs$ coming from a.
- $v = w \bullet \sigma$, where $\sigma \in Seq(propCmd)$, $w \in ProvedSafe(Q, m, \beta)$, and β is any ballot array such that, for every acceptor a in Q, $\widehat{\beta}_a = m$ and c has received a message $\langle$"1b", $m, \rho\rangle$ from a with $\rho = \beta_a$.

This action sets *dMaxTried*[c][m] and *cval*[c] to v, *crnd*[c] to m, and adds message $\langle$"2a", $m, v\rangle$ to *msgs* and *msgs2*.

Phase2aClassic(c, m, C) executed by coordinator c, for command C. The action is enabled iff

- $C \in propCmd$.
- $dMaxTried[c][m] \neq none$
- $\forall j > m : maxTried[c][j] = none$

This action adds message $\langle$"2a", m, c, $dMaxTried[c][m] \bullet C\rangle$ to *msgs* and *msgs2*, and sets $dMaxTried[c][m]$ and *cval*[c] to $dMaxTried[c][m] \bullet C$.

$Phase2bClassic(a, m, v)$ executed by acceptor a, for balnum m and c-struct v. The action is enabled iff

- $m \geq \widehat{bA_a}$,
- there is an m-coordquorum L and a c-struct u such that, for every $c \in L$, acceptor a has received a phase "2a" message for balnum m with value w satisfying $u \sqsubseteq w$, i.e., u is a lower bound for the w values, and
- Either $bA_a[m]$ equals *none* and v equals u, or $bA_a[m]$ and u are compatible and v equals $bA_a[m] \sqcup u$.

It sets $bA_a[m]$ and *vval*[a] to v, $\widehat{bA_a}$, *rnd*[a], and *vrnd*[a] to m, and adds message $\langle$"2b", $m, v\rangle$ to *msgs* and *msgs2*.

$Phase2bFast(a, C)$ executed by acceptor a, for balnum m and command C. The action is enabled iff

- $\widehat{bA_a}$ is a fast balnum,
- $bA_a[\widehat{bA_a}] \neq none$, and
- $C \in propCmd$.

It sets $bA_a[\widehat{bA_a}]$ and $vval[a]$ to $bA_a[\widehat{bA_a}] \bullet C$ and adds message $\langle$"2b", $\widehat{bA_a}, bA_a[\widehat{bA_a}] \bullet C\rangle$ to *msgs* and *msgs2*.

$Learn(l, v)$ executed by learner l, for c-struct v. Executed by learner l. It is enabled iff a has received phase "2b" messages for some round i from an i-quorum Q and v is a prefix of the values on those messages. It sets $learned[l]$ to $learned[l] \sqcup v$.

Variables *crnd*, *cval*, *rnd*, *vrnd*, *vval*, and *msgs2* appear in no pre-condition and are clearly history variables (they satisfy Abadi and Lamport's conditions H1-5 of [Abadi and Lamport, 1991]). 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: $crnd[c] = k \iff \begin{array}{ll} \wedge & dMaxTried[c][k] \neq none \\ \wedge & \forall j > k : dMaxTried[c][j] = none \end{array}$

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

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

InvDA7: $\langle$"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 $\rho[vrnd]$.

InvDA8: $\langle$"2a", $m, v\rangle \in msgs \iff \langle$"2a", $m, v\rangle \in msgs2$

InvDA9: $\langle$"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(C) executed by the proposer of command C. The action is always enabled. It sets *propCmd* to *propCmd* $\cup \{C\}$, from where coordinators and acceptors can read C.

This action remains the same.

Phase1a(c, m) executed by coordinator c, for balnum m. The action is enabled iff *crnd*$[c] < m$. It adds message $\langle$"1a", $m\rangle$ to *msgs* and *msgs2*.

By invariant InvDA1.

Phase1b(a, m) executed by acceptor a, for balnum m. The action is enabled iff

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

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

By invariants InvDA3 and InvDA6.

Phase2Start(c, m, v) executed by coordinator c, for balnum m, and c-struct v. The action is enabled iff

- $crnd[c] < m$
- There exists an m-quorum Q such that for all $a \in Q$, there is a message $\langle$"1b", $m, vval, vrnd\rangle \in msgs2$ coming from a.
- $v = w \bullet \sigma$, where $\sigma \in Seq(propCmd)$, $w \in ProvedSafe(Q, 1bMsg)$ (see *ProvedSafe* as defined in Section 3.3) , and *1Msg* is a mapping from every acceptor a in Q to the phase "1b" message of the previous condition coming from a.

This action sets *dMaxTried*$[c][m]$ and *cval*$[c]$ to v, *crnd*$[c]$ to m, and adds message $\langle$"2a", $m, v\rangle$ to *msgs* and *msgs2*.

By invariants InvDA1 and InvDA6. The equivalence between *ProvedSafe*(Q, m, β) (of Section A.1.3) and *ProvedSafe*$(Q, 1bMsg)$ (of Section 3.3) is given by InvDA6.

Phase2aClassic(c, m, C) executed by coordinator c, for command C. The action is enabled iff

- $C \in propCmd$.
- $crnd[c] = m$

This action adds message $\langle$"2a", $m, c, cval[c] \bullet C\rangle$ to *msgs* and *msgs2*, and sets $dMaxTried[c][m]$ and *cval*$[c]$ to $cval[c] \bullet C$.

By invariants InvDA1 and InvDA2.

$Phase2bClassic(a, m, v)$ executed by acceptor a, for balnum m and c-struct v. The action is enabled iff

- $m \geq rnd[a]$,
- there is an m-coordquorum L and a c-struct u such that, for every $c \in L$, acceptor a has received a phase "2a" message (through *msgs2*) for balnum m with value w satisfying $u \sqsubseteq w$, i.e., u is a lower bound for the w values, and
- Either $vrnd[a] < m$ and v equals u, or $vrnd[a] = m$, $vval[a]$ and u are compatible, and v equals $vval[a] \sqcup u$.

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

By invariants InvDA3, InvDA4, InvDA8.

$Phase2bFast(a, C)$ executed by acceptor a, for balnum m and command C. The action is enabled iff

- $rnd[a]$ is a fast balnum,
- $rnd[a] = vrnd[a]$, and
- $C \in propCmd$.

It sets $bA_a[\widehat{bA}_a]$ and *vval*$[a]$ to $vval[a] \bullet C$ and adds message $\langle$"2b", $m, vval[a] \bullet C\rangle$ to *msgs* and *msgs2*.

By invariants InvDA3, InvDA4, InvDA5, and the fact that $rnd[a] \geq vrnd[a]$, which can be inferred by the definition of a ballot array and invariants InvDA3 and InvDA4.

$Learn(l, v)$ executed by learner l, for c-struct v. Executed by learner l. It is enabled iff a has received (through *msgs2*) phase "2b" messages for some round i from an i-quorum Q and v is a prefix of the values on those messages. It sets $learned[l]$ to $learned[l] \sqcup v$.

By invariant InvDA9.

The resulting algorithm now has variables *bA*, *dMaxTried* 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(C) executed by the proposer of command C. The action is always enabled. It sets *propCmd* to *propCmd*$\cup\{C\}$, from where coordinators and acceptors can read C.

Phase1a(c, m) executed by coordinator c, for balnum m. The action is enabled iff *crnd*$[c] < m$. It adds message $\langle$"1a", $m\rangle$ to *msgs2*.

Phase1b(a, m) executed by acceptor a, for balnum m. The action is enabled iff

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

It sets *rnd*$[a]$ to m, and adds message $\langle$"1b", m, *vval*$[a]$, *vrnd*$[a]\rangle$ to *msgs2*.

Phase2Start(c, m, v) executed by coordinator c, for balnum m, and c-struct v. The action is enabled iff

- $crnd[c] < m$
- There exists an m-quorum Q such that for all $a \in Q$, there is a message $\langle$"1b", $m, vval, vrnd\rangle \in msgs2$ coming from a.
- $v = w \bullet \sigma$, where $\sigma \in Seq(propCmd)$, $w \in ProvedSafe(Q, 1bMsg)$, and *1Msg* is a mapping from every acceptor a in Q to the phase "1b" message of the previous condition coming from a.

This action sets *cval*$[c]$ to v, *crnd*$[c]$ to m, and adds message $\langle$"2a", $m, v\rangle$ to *msgs2*.

Phase2aClassic(c, m, C) executed by coordinator c, for command C. The action is enabled iff

- $C \in propCmd$.
- $crnd[c] = m$

This action adds message $\langle$"2a", m, c, $cval[c] \bullet C\rangle$ to *msgs2*, and sets *cval*$[c]$ to $cval[c] \bullet C$.

$Phase2bClassic(a, m, v)$ executed by acceptor a, for balnum m and c-struct v. The action is enabled iff

- $m \geq rnd[a]$,
- there is an m-coordquorum L and a c-struct u such that, for every $c \in L$, acceptor a has received a phase "2a" message (through *msgs2*) for balnum m with value w satisfying $u \sqsubseteq w$, i.e., u is a lower bound for the w values, and
- Either $vrnd[a] < m$ and v equals u, or $vrnd[a] = m$, $vval[a]$ and u are compatible, and v equals $vval[a] \sqcup u$.

It sets *vval*$[a]$ to v, *rnd*$[a]$ and *vrnd*$[a]$ to m, and adds message $\langle$"2b", $m, v\rangle$ to *msgs2*.

$Phase2bFast(a, C)$ executed by acceptor a, for balnum m and command C. The action is enabled iff

- $rnd[a]$ is a fast balnum,
- $rnd[a] = vrnd[a]$, and
- $C \in propCmd$.

It sets *vval*$[a]$ to $vval[a] \bullet C$ and adds message $\langle$"2b", m, $vval[a] \bullet C\rangle$ to *msgs2*.

$Learn(l, v)$ executed by learner l, for c-struct v. Executed by learner l. It is enabled iff a has received (through *msgs2*) phase "2b" messages for some round i from an i-quorum Q and v is a prefix of the values on those messages. It sets $learned[l]$ to $learned[l] \sqcup v$.

The algorithm presented in Section 3.3 is a stricter implementation of the algorithm above, which can be easily verified by simply comparing their actions. This concludes the proof that Multicoordinated Paxos satisfies the safety requirements of Generalized Consensus. Section A.2.4 presents the unambiguous TLA$^+$ specification of the basic algorithm presented in Section 3.3 and Section A.2.5 presents its complete TLA$^+$ specification including actions for collision recovery and a simplified version of the mechanism for reducing the number of disk writes presented in Section 3.4.5.

A.1.5 Collision Recovery

The mechanisms for collision recovery described in Section 3.4.4 simply simulate the execution of basic actions of the algorithm in a way compatible with the actual behavior. The interpretation of a collision as a phase "1a" message for the following balnum, for example, simulates a *Phase1a* action for that balnum. Since action *Phase1a* simply sends a "1a" message with no guarantee of delivery, the process detecting the collision could be the only one receiving such a message. Given that these basic actions are already proved correct, such mechanisms cannot violate the safety properties of our specification.

A.1.6 Liveness

That the basic algorithm provides means for ensuring liveness is easy to see, since new rounds of any type can always be started and a classic round has the same liveness requirements as Classic Paxos. The discussion in Section 3.4.7 extends Multicoordinated Paxos in this sense and sketches its liveness conditions. We now prove that the extended Multicoordinated Paxos algorithm presented in Section 3.4.7 satisfies

the Liveness property of Generalized Consensus, given that its liveness condition is eventually satisfied. We refer to the algorithm as EMCP.

Proposition 12 *If there is a proposer p, a coordinator c, a learner l, a quorum of acceptors Q, and a non-empty set of coordinators C such that the liveness condition MCLiv(p,l,c,Q,C) holds at some time t_0 and then forever, then l eventually learns a c-struct containing the command v proposed by p.*

PROOF: The proof is divided into the following steps:

1. No coordinator other than c executes action *Phase1a* after t_0
   By the definition of *MCLiv*(p,l,c,Q,C), only c believes to be leader after instant t_0. Since this is a pre-condition for executing action *Phase1a* in the extended algorithm, only c does so.
2. There is a time $t_1 \geq t_0$ after which $crnd[c]$ does not change
   PROOF SKETCH: The proof is divided in two steps. First we prove that only a finite number rounds may be started by coordinated recovery and then that the same holds true for rounds started in action *Phase1a*.
   2.1. There is a time $t_1' \geq t_0$ after which no coordinated recovery is performed
   PROOF: Let $i-1$ be the highest-numbered round in which some message was sent before t_0. If the coordinators of $i-1$ perceive a conflict in $i-1$ then they may perform coordinated recovery to start round i by sending "`2a`" messages with different c-structs and, hence, another conflict might happen in round i. Due to *MCLiv*(p,l,c,Q,C), after t_0 all functional coordinators receive all messages in the same order in i and, by the determinism of the protocol, they perceive the same conflicts in i and choose the same c-struct to send in the "`2a`" message of round $i+1$, if performing another coordinated recovery. Therefore, no conflict happens in round $i+1$ or in any bigger round and no more coordinated recoveries are executed. Hence, there is a time $t_1' \geq t_0$ after round round $i+1$ started after which no coordinated recoveries are performed.
   2.2. There is a time $t_1'' \geq t_1'$ after which action *Phase1a* is not executed anymore
   PROOF: By step 2.1, coordinated recovery will not be executed after time t_1' and by *MCLiv*(p,l,c,Q,C), uncoordinated recovery can only be executed a limited amount of times. Hence, there is a time $t_1''' \geq t_1'$ after which no recovery is performed.
   Due to step 1 there is a finite number of rounds that have been started before t_0 by coordinators different from c and by the previous paragraph there is a time after which no coordinated or uncoordinated recovery is performed and, therefore, there is a time $t_1'''' \geq t_1'''$ after which c has will receive no more "`skip`" messages informing about rounds bigger than its current one. Hence, c will not start any bigger round unless it suspects that there are no coord-quorums for round $crnd[c]$ whose all coordinators are alive. By *MCLiv*(p,l,c,Q,C), after

t_0, c will only start rounds whose coordinators do not fail. Hence, there is a time $t_1'''' \geq t_1''''$ after which c does not start any round due to suspecting that coord-quorums are not available for its current round.

If t_1'' is bigger than or equal to t_1'''', then *Phase1a* is not executed after t_1''.

2.3. ASSUME: 1. There is a time $t_1' \geq t_0$ after which no coordinated recovery is performed.

2. There is a time $t_1'' \geq t_1'$ after which action *Phase1a* is not executed anymore

PROVE: There is a time $t_1 \geq t_1''$ after which action $Phase2Start(c, i)$ is executed, where i the highest-numbered round for which c executed action *Phase1a*

PROOF: By the definition of the extended algorithm, c keeps retransmitting the "1a" message for round i to all acceptors. By assumption, acceptors in Q do not crash after t_0 and, therefore, receive such "1a" messages. After they execute action $Phase1b$ for round i, they keep re-sending their $1b$ messages and c will eventually execute $Phase2Start$.

2.4. Q.E.D.

PROOF: $crnd[c]$ can only be changed by executing action $Phase2Start$ or by performing coordinated or uncoordinated recovery, and *Phase2Start* can only be executed for a round after *Phase1a* has been executed for the same round. By steps 2.1 and 2.2 there is a time t_0'' after which no recovery or *Phase1a* actions is executed. By step 2.3, c eventually executes action *Phase2Start* for the highest-numbered round for which it has executed action *Phase1a* at some instant $t_1 \geq t_1''$.

3. There is a time $t_2 \geq t_1$ after which action *Phase2Start*(d,*crnd*[c]) will have been executed by every coordinator d of round $crnd[c]$.

PROOF: By the same reasoning of step 2.

4. There is a time $t_3 \geq t_2$ after which the command $v \in \sqcap \{cval[d] : d$ is a coordinator of round $crnd[c]\}$.

PROOF: By steps 2 and 3, there is a time t_3' after which all coordinators of round $crnd[c]$ can execute action *Phase2aClassic*. By assumption, all conflicting proposals received in $crnd[c]$ are received in the same order and, therefore, added to $cval[d]$ in the same order by every acceptor d. Hence, these c-structs are compatible.

By the protocol specification, proposer p retransmits v and all coordinators of round $crnd[c]$ eventually receive it. Therefore, there is a time $t_3 \geq t_3'$ after which v is part of $cval[d]$ of any coordinator d of round $crnd[c]$. Since they are all compatible, by **CS4**, v is contained in their greatest lower bound.

5. There is a time $t_4 \geq t_3$ after which $v \in \sqcap \{vval[a] : a \in Q\}$ and $vrnd[a] = crnd[c], a \in Q$.

PROOF: By step 2, no new rounds are created after some time $t_1 \geq t_0$. Hence, no acceptor can accept any value in a round bigger than $crnd[c]$ after t_1. By step 4, all c-structs sent to acceptors in "2a" messages are compatible and because coordinators keep retransmitting their "2a" messages, all acceptors in Q eventually receive one containing v from every acceptor in some coord-quorum for round $crnd[c]$. Hence, by the definition of the algorithm, every acceptor a eventually accepts some c-struct containing v that is compatible with the c-structs accepted by the other acceptors. Hence, there is a time $t_4 \geq t_3$ after which $v \in \sqcap\{vval[a] : a \in Q\}$ and $vrnd[a] = crnd[c], a \in Q$.

6. Eventually l learns a c-struct containing v
   PROOF: By steps 5 and 2 and the specification of EMCF, learners eventually received compatible c-structs from all acceptors in Q in round $crnd[c]$ containing v. Hence, l eventually learns a c-struct containing v.
7. Q.E.D.

A.2 TLA$^+$ Specifications

A.2.1 Helper Specifications

Order Relations

This module was defined in [Lamport, 2004].

MODULE *OrderRelations*

We make some definitions for an arbitrary ordering relation $\sqsubseteq$ on a set S. The module will be used by *instantiang* $\sqsubseteq$ and S with a particular operator and Set.

CONSTANTS $S, \_ \sqsubseteq \_$

We define *IsPartialOrder* to be the assertion that $\sqsubseteq$ is an (irreflexive) partial order on a set S, and *IsTotalOrder* to be the assertion that it is a total ordering of S.

$$\begin{aligned}
&IsPartialOrder \triangleq \\
&\quad \land \forall u, v, w \in S : (u \sqsubseteq v) \land (v \sqsubseteq w) \Rightarrow (u \sqsubseteq w) \\
&\quad \land \forall u, v \in S \quad : (u \sqsubseteq v) \land (v \sqsubseteq u) \Rightarrow (u = v)
\end{aligned}$$

$$\begin{aligned}
&IsTotalOrder \triangleq \\
&\quad \land IsPartialOrder \\
&\quad \land \forall u, v \in S : (u \sqsubseteq v) \lor (v \sqsubseteq u)
\end{aligned}$$

We now define the glb (greatest lower bound) and lub (least upper bound) operators. To define *GLB*, we first define $IsLB(lb, T)$ to be true iff lb is a lower bound of T, and $IsGLB(lb, T)$ to be true iff lb is a glb of T. the value of $GLB(T)$ is unspecified if T has no glb. The definition for upper bounds are analogous.

$$\begin{aligned}
IsLB(lb, T) &\triangleq \land lb \in S \\
&\quad\ \land \forall v \in T : lb \sqsubseteq v \\
IsGLB(lb, T) &\triangleq \land IsLB(lb, T) \\
&\quad\ \land \forall v \in S : IsLB(v, T) \Rightarrow (v \sqsubseteq lb) \\
GLB(T) &\triangleq \text{CHOOSE } lb \in S : IsGLB(lb, T) \\
v \sqcap w &\triangleq GLB(\{v, w\}) \\
IsUB(ub, T) &\triangleq \land ub \in S \\
&\quad\ \land \forall v \in T : v \sqsubseteq ub \\
IsLUB(ub, T) &\triangleq \land IsUB(ub, T) \\
&\quad\ \land \forall v \in S : IsUB(v, T) \Rightarrow (ub \sqsubseteq v) \\
LUB(T) &\triangleq \text{CHOOSE } ub \in S : IsLUB(ub, T) \\
v \sqcup w &\triangleq LUB(\{v, w\})
\end{aligned}$$

Command Structs

This module was defined in [Lamport, 2004].

MODULE *CStructs*

EXTENDS *Sequences* The *Sequences* module defines the operation *Seq*

We declare the assumed objects as parameters. We use *Bottom* instead of $\bot$.

CONSTANTS Cmd, $CStruct$, $\_ \bullet \_$, $Bottom$

We write $v ** \sigma$ as the overloaded version of $v \bullet \sigma$ for a command sequence σ. The recursive definition below defines the function $conc[w, t] = w ** t$.

$$
\begin{array}{l}
v ** s \triangleq \\
\quad \text{LET } conc[w \in CStruct, t \in Seq(Cmd)] \triangleq \\
\qquad \text{IF } t = \langle\rangle \text{ THEN } w \\
\qquad\qquad \text{ELSE } conc[w \bullet Head(t), Tail(t)] \\
\quad \text{IN } conc[v, s]
\end{array}
$$

$$Str(P) \triangleq \{Bottom ** s : s \in Seq(P)\}$$

Our algorithms use a value *none* that is not a c-struct and extend the relation $\sqsubseteq$ to the element *none* so that $none \sqsubseteq none$, *none* $\not\sqsubseteq v$, and $v \not\sqsubseteq$ *none* for any c-struct v.

$$
\begin{array}{l}
none \triangleq \text{CHOOSE } n : n \notin CStruct \\
v \sqsubseteq w \triangleq \ \lor \land v \in CStruct \\
\qquad\qquad\quad \land w \in CStruct \\
\qquad\qquad\quad \land \exists\, s \in Seq(Cmd) : w = v ** s \\
\qquad\qquad \lor \land v = none \\
\qquad\qquad\quad \land w = none \\
v \sqsubset w \triangleq (v \sqsubseteq w) \land (v \neq w)
\end{array}
$$

We now import the definitions of the *OrderRelations* module with *CStruct* substituted for S and $\sqsubseteq$ substituted for $\preceq$

INSTANCE *OrderRelations* WITH $S \leftarrow CStruct$

We now define compatibility of c-structs and of sets of c-structs, and of contains, giving them obvious operator nomes.

$$
\begin{array}{ll}
AreCompatible(v, w) & \triangleq \exists\, ub \in CStruct : IsUB(ub, \{v, w\}) \\
IsCompatible(S) & \triangleq \forall\, v, w \in S : AreCompatible(v, w) \\
Contains(v, C) & \triangleq \exists\, s, t \in Seq(Cmd) : \\
 & \qquad v = ((Bottom ** s) \bullet C) ** t
\end{array}
$$

Here are the formal statements of assumptions $CS0$-$CS4$.

$$CS0 \triangleq \forall\, v \in CStruct, C \in Cmd : v \bullet C \in CStruct$$

$$CS1 \triangleq CStruct = Str(Cmd)$$

$$CS2 \triangleq IsPartialOrder$$

$$
\begin{array}{l}
CS3 \triangleq \forall\, P \in \text{SUBSET } Cmd \setminus \{\{\}\} : \\
\qquad \land \forall\, v, w \in Str(P) : \\
\qquad\quad \land v \sqcap w \in Str(P) \\
\qquad\quad \land IsGLB(v \sqcap w, \{v, w\}) \\
\qquad\quad \land AreCompatible(v, w) \Rightarrow \land v \sqcup w \in Str(P)
\end{array}
$$

$$\wedge\ IsLUB(v \sqcup w,\ \{v,\ w\})$$

$CS4 \triangleq \forall\, v,\ w \in CStruct,\ C \in Cmd :$
$\quad AreCompatible(v,\ w) \wedge Contains(v,\ C) \wedge Contains(w,\ C) \Rightarrow$
$\quad\quad Contains(v \sqcap w,\ C)$

ASSUME $CS0 \wedge CS1 \wedge CS2 \wedge CS3 \wedge CS4$

Paxos Constants

This module was defined in [Lamport, 2004], but was extended as needed to define multicoordinated algorithms. For example, it was extended with the definition of $CoordQuorum(m)$.

MODULE *PaxosConstants*

This module defines the data structures for the abstract algorithm. It is basically the same module *PaxosConstants* found in the Generalized Paxos paper, except for the introduction of constants *Coord* and $CoordQuorum(\_)$, and assumption *CoordQuorumAssumption*.

EXTENDS *CStructs*, *FiniteSets*

Module *FiniteSets* defines $IsFiniteSet(S)$ to be true iff S is a finite set

We introduce the parameter *IsFast*, where $IsFast(m)$ is true iff m is a fast ballot number. The ordering relation $\preceq$ on ballot numbers is also a parameter.

CONSTANTS *BalNum*, *Zero*, $\_ \preceq \_$, $IsFast(\_)$

We assume that *Zero* is the first balnum, and that $\preceq$ is a total ordering of the set *BalNum* of balnums.

ASSUME $\wedge\ Zero \in BalNum$
$\quad \wedge$ LET $PO \triangleq$ INSTANCE *OrderRelations*
$\quad\quad\quad\quad$ WITH $S \leftarrow BalNum,\ \sqsubseteq \leftarrow \preceq$
$\quad\quad$ IN $PO\,!\,IsTotalOrder$

We define $i \prec j$ to be true iff $i \preceq j$ for two different balnums i and j

$i \prec j \triangleq (i \preceq j) \wedge (i \neq j)$

If B is a set of ballot numbers that contains a maximum element, then $Max(B)$ is defined to equal that maximum. Otherwise, its value is unspecified.

$Max(B) \triangleq$ CHOOSE $i \in B : \forall\, j \in B : j \preceq i$

This section of the module is the only part that differs the original *PaxosConstants* module presented in the Generalized *Paxos* paper. We have added the constants *Coord* and *CoordQuorum*, and the assumptions that coordinator quorums are sets of coordinators which intersect if they refer to the same ballot number.

CONSTANTS *Learner*, *Acceptor*, $Quorum(\_)$, *Coord*, $CoordQuorum(\_)$

$QuorumAssumption \triangleq$
$\quad \forall\, i \in BalNum :$
$\quad\quad \wedge\ Quorum(i) \subseteq$ SUBSET *Acceptor*

$\land \forall\, j \in BalNum :$
$\quad \land \forall\, Q \in Quorum(i),\, R \in Quorum(j) : Q \cap R \neq \{\}$
$\quad \land\, IsFast(j) \Rightarrow$
$\quad\quad \forall\, Q \in Quorum(i),\, R1,\, R2 \in Quorum(j) :$
$\quad\quad\quad Q \cap R1 \cap R2 \neq \{\}$

ASSUME $QuorumAssumption$

$CoordQuorumAssumption \triangleq$
$\quad \forall\, i \in BalNum :$
$\quad\quad \land\, CoordQuorum(i) \subseteq \text{SUBSET } Coord$
$\quad\quad \land \forall\, Q,\, R \in CoordQuorum(i) : Q \cap R \neq \{\}$

ASSUME $CoordQuorumAssumption$

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 $\beta.vote[m]$ and $\hat{\beta}_a$ as $\beta.mbal[a]$.

$BallotArray \triangleq$
$\quad \{beta \in [vote : [Acceptor \to [BalNum \to CStruct \cup \{none\}]],$
$\quad\quad\quad mbal : [Acceptor \to BalNum]] :$
$\quad\quad \forall\, a \in Acceptor :$
$\quad\quad\quad \land\, beta.vote[a][Zero] \neq none$
$\quad\quad\quad \land\, IsFiniteSet(\{m \in BalNum : beta.vote[a][m] \neq none\})$
$\quad\quad\quad \land \forall\, m \in BalNum :$
$\quad\quad\quad\quad (beta.mbal[a] \prec m) \Rightarrow (beta.vote[a][m] = none)\}$

We now formalize the definitions of *chosen at*, *safe at*, etc. We translate the *English* terms into obvious operator names. For example, $IsChosenAt(v,\, m,\, \beta\,)$ is defined to be true iff v is chosen at m in β , assuming that v is a c-struct, m is a balnum, and β a ballot array. (We don't care what $IsChosenAt(v,\, m,\, \beta\,)$ means for other values of v, m, and β.) We also assert the three propositions as theorems.

$IsChosenAt(v,\, m,\, beta) \triangleq$
$\quad \exists\, Q \in Quorum(m) : \forall\, a \in Q : (v \sqsubseteq beta.vote[a][m])$

$IsChosenIn(v,\, beta) \triangleq$
$\quad \exists\, m \in BalNum : IsChosenAt(v,\, m,\, beta)$

$IsChoosableAt(v,\, m,\, beta) \triangleq$
$\quad \exists\, Q \in Quorum(m) :$
$\quad\quad \forall\, a \in Q : (m \prec beta.mbal[a]) \Rightarrow (v \sqsubseteq beta.vote[a][m])$

$IsSafeAt(v,\, m,\, beta) \triangleq$
$\quad \forall\, k \in BalNum :$
$\quad\quad (k \prec m) \Rightarrow \forall\, w \in CStruct :$
$\quad\quad\quad IsChoosableAt(w,\, k,\, beta) \Rightarrow (w \sqsubseteq v)$

$IsSafe(beta) \triangleq$
$\quad \forall\, a \in Acceptor,\, k \in BalNum :$
$\quad\quad (beta.vote[a][k] \neq none) \Rightarrow IsSafeAt(beta.vote[a][k],\, k,\, beta)$

THEOREM Proposition 1

$\forall\, beta \in BallotArray :$
$\quad IsSafe(beta) \Rightarrow IsCompatible(\{v \in CStruct : IsChosenIn(v, beta)\})$

$ProvedSafe(Q, m, beta) \triangleq$
$\quad \text{LET } k \triangleq Max(\{i \in BalNum :$
$\qquad\qquad (i \prec m) \wedge (\exists\, a \in Q : beta.vote[a][i] \neq none)\})$
$\qquad RS \triangleq \{R \in Quorum(k) : \forall\, a \in Q \cap R : beta.vote[a][k] \neq none\}$
$\qquad g(R) \triangleq GLB(\{beta.vote[a][k] : a \in Q \cap R\})$
$\qquad G \triangleq \{g(R) : R \in RS\}$
$\quad \text{IN IF } RS = \{\} \text{ THEN } \{beta.vote[a][k] :$
$\qquad\qquad a \in \{b \in Q : beta.vote[b][k] \neq none\}\}$
$\qquad\qquad \text{ELSE IF } IsCompatible(G) \text{ THEN } \{LUB(G)\}$
$\qquad\qquad\qquad \text{ELSE } \{\}$

THEOREM Proposition 2
$\quad \forall\, m \in BalNum \setminus \{Zero\},\ beta \in BallotArray :$
$\qquad \forall\, Q \in Quorum(m) :$
$\qquad\quad \wedge\, IsSafe(beta)$
$\qquad\quad \wedge\, \forall\, a \in Q : m \preceq beta.mbal[a]$
$\qquad\quad \Rightarrow \forall\, v \in ProvedSafe(Q, m, beta) : IsSafeAt(v, m, beta)$

$IsConservative(beta) \triangleq$
$\quad \forall\, m \in BalNum,\ a, b \in Acceptor :$
$\qquad \wedge\, \neg IsFast(m)$
$\qquad \wedge\, beta.vote[a][m] \neq none$
$\qquad \wedge\, beta.vote[b][m] \neq none$
$\qquad \Rightarrow AreCompatible(beta.vote[a][m], beta.vote[b][m])$

THEOREM Proposition 3
$\quad \forall\, beta \in BallotArray :$
$\qquad IsConservative(beta) \Rightarrow$
$\qquad\quad \forall\, m \in BalNum \setminus \{Zero\} :$
$\qquad\qquad \forall\, Q \in Quorum(m) : ProvedSafe(Q, m, beta) \neq \{\}$

A.2.2 Abstract Multicoordinated Paxos

This module specifies an abstract version of the Multicoordinated Paxos algorithm. It is used as the first step in the proof of correctness of Multicoordinated Paxos.

MODULE *AbstractMCPaxos*

This module specifies the Abstract MultiCoordinated Paxos algorithm. It resembles the specification of the Abstract Generalized Paxos algorithm presented in the Generalized Paxos paper, but some changes are required since we do not have a *minTried* variable.

EXTENDS *PaxosConstants*

VARIABLES *propCmd*, *maxTried*, *bA*, *learned*

Invariants

Type invariant.

$$\begin{aligned}
TypeInv \triangleq\ & \land propCmd \subseteq Cmd \\
& \land learned \in [Learner \to CStruct] \\
& \land bA \in BallotArray \\
& \land maxTried \in [BalNum \to CStruct \cup \{none\}]
\end{aligned}$$

Other invariants satisfied by the algorithm.

$$\begin{aligned}
&maxTriedInvariant \triangleq \\
&\quad \forall\, m \in BalNum : \\
&\quad\ (maxTried[m] \neq none) \Rightarrow \\
&\qquad \land maxTried[m] \in Str(propCmd) \\
&\qquad \land IsSafeAt(maxTried[m],\, m,\, bA)
\end{aligned}$$

$$\begin{aligned}
&bAInvariant \triangleq \\
&\quad \forall\, a \in Acceptor,\, m \in BalNum : \\
&\quad\ (bA.vote[a][m] \neq none) \Rightarrow \\
&\qquad \land IsSafeAt(bA.vote[a][m],\, m,\, bA) \\
&\qquad \land \neg IsFast(m) \Rightarrow (bA.vote[a][m] \sqsubseteq maxTried[m]) \\
&\qquad \land IsFast(m) \Rightarrow (bA.vote[a][m] \in Str(propCmd))
\end{aligned}$$

$$\begin{aligned}
&learnedInvariant \triangleq \\
&\quad \forall\, l \in Learner : \land learned[l] \in Str(propCmd) \\
&\qquad\qquad\qquad \land \exists\, S \in \text{SUBSET } CStruct : \\
&\qquad\qquad\qquad\quad \land IsFiniteSet(S) \\
&\qquad\qquad\qquad\quad \land \forall\, v \in S : IsChosenIn(v,\, bA) \\
&\qquad\qquad\qquad\quad \land learned[l] = LUB(S)
\end{aligned}$$

Actions

$Propose(C)$ specifies the action of proposing command C

$$\begin{aligned}
&Propose(C) \triangleq \\
&\quad \land\ \ C \notin propCmd \\
&\quad \land\ \ propCmd' = propCmd \cup \{C\} \\
&\quad \land\ \ \text{UNCHANGED } \langle maxTried,\, bA,\, learned\rangle
\end{aligned}$$

Action $JoinBallot(a, m)$ increases the current ballot number of agent a, setting it to m.

$$\begin{aligned}
&JoinBallot(a,\, m) \triangleq \\
&\quad \land bA.mbal[a] \prec m \\
&\quad \land bA' = [bA \text{ EXCEPT } !.mbal[a] = m] \\
&\quad \land \text{UNCHANGED } \langle propCmd,\, maxTried,\, learned\rangle
\end{aligned}$$

Action *StartBallot*(m, w) changes $maxTried[m]$ from *none* to w, where w is proposed and safe at m in bA.

$$\begin{aligned}
&StartBallot(m,\, w) \triangleq \\
&\quad \land maxTried[m] = none \\
&\quad \land IsSafeAt(w,\, m,\, bA) \\
&\quad \land w \in Str(propCmd) \\
&\quad \land maxTried' = [maxTried \text{ EXCEPT } ![m] = w] \\
&\quad \land \text{UNCHANGED } \langle propCmd,\, bA,\, learned\rangle
\end{aligned}$$

Action *Suggest*(m, σ) extends $maxTried[m]$ with σ if $maxTried[m] \neq$ *none*.
The expression $[maxTried$ EXCEPT $![m] =$ @ $**s]$ represents a vector (in fact, it is a function) which is almost the same as $maxTried$ except for entry m ($![m]$ in the expression), which is set to the previous value of that entry (@ in the expression) extended with command sequence s.

$Suggest(m, s) \triangleq$
$\quad \wedge \ s \in Seq(propCmd)$
$\quad \wedge \ maxTried[m] \neq none$
$\quad \wedge \ maxTried' = [maxTried$ EXCEPT $![m] = @ ** s]$
$\quad \wedge$ UNCHANGED $\langle propCmd, bA, learned \rangle$

Action $ClassicVote(a, v)$ sets $\widehat{bA}_a$ to m and $bA_a[m]$ to v if
(i) $\widehat{bA}_a \preceq m$
(ii) v is safe at m in bA,
(iii) $v \sqsubseteq$ *maxTried*$[m]$, and
(iv) either $bA_a[m]$ equals *none* or v is an extension of it.

$ClassicVote(a, m, v) \triangleq$
$\quad \wedge \ bA.mbal[a] \preceq m$
$\quad \wedge \ IsSafeAt(v, m, bA)$
$\quad \wedge \ v \sqsubseteq maxTried[m]$
$\quad \wedge \ \vee \ bA.vote[a][m] = none$
$\quad\quad \vee \ bA.vote[a][m] \sqsubseteq v$
$\quad \wedge \ bA' = [bA$ EXCEPT $!.mbal[a] = m, \ !.vote[a][m] = v]$
$\quad \wedge$ UNCHANGED $\langle propCmd, maxTried, learned \rangle$

Action $FastVote(a, C)$ extends $bA_a[\widehat{bA}_a]$ with proposed command C if $\widehat{bA}_a$ is a fast balnum and $bA_a[\widehat{bA}_a] \neq$ *none* . Expression $[bA$ EXCEPT $!.vote[a][bA.mbal[a]] =$ @ $\bullet C]$ follows the same principle explained in action $Suggest(m, C)$.

$FastVote(a, C) \triangleq$
$\quad \wedge \ C \in propCmd$
$\quad \wedge \ IsFast(bA.mbal[a])$
$\quad \wedge \ bA.vote[a][bA.mbal[a]] \neq none$
$\quad \wedge \ bA' = [bA$ EXCEPT $!.vote[a][bA.mbal[a]] = @ \bullet C]$
$\quad \wedge$ UNCHANGED $\langle propCmd, maxTried, learned \rangle$

Action $AbstractLearn(l, v)$ extends $learned[l]$ to the least upper bound of $learned[l]$ and v, if v is chosen.

$AbstractLearn(l, v) \triangleq$
$\quad \wedge \ IsChosenIn(v, bA)$
$\quad \wedge \ learned' = [learned$ EXCEPT $![l] = @ \sqcup v]$
$\quad \wedge$ UNCHANGED $\langle propCmd, maxTried, bA \rangle$

Complete Specification

Initial predicate

$Init \triangleq \ \wedge \ propCmd = \{\}$
$\quad\quad \wedge \ learned \quad = [l \in Learner \mapsto Bottom]$
$\quad\quad \wedge \ bA = [vote \mapsto$
$\quad\quad\quad\quad [a \in Acceptor \mapsto$

$$[m \in BalNum \mapsto \text{IF } m = Zero \text{ THEN } Bottom \text{ ELSE } none]],$$
$$mbal \mapsto [a \in Acceptor \mapsto Zero]]$$
$$\land\ maxTried = [m \in BalNum \mapsto \text{IF } m = Zero \text{ THEN } Bottom \text{ ELSE } none]$$

Actions combined into the next-state relation.

$$\begin{aligned} Next \triangleq\ & \lor \exists\, C \in Cmd : Propose(C) \\ & \lor \exists\, a \in Acceptor,\, m \in BalNum : JoinBallot(a,\, m) \\ & \lor \exists\, m \in BalNum,\, w \in CStruct : StartBallot(m,\, w) \\ & \lor \exists\, m \in BalNum,\, s \in Seq(Cmd) : Suggest(m,\, s) \\ & \lor \exists\, a \in Acceptor,\, C \in Cmd : FastVote(a,\, C) \\ & \lor \exists\, a \in Acceptor,\, m \in BalNum,\, v \in CStruct : \\ & \qquad ClassicVote(a,\, m,\, v) \\ & \lor \exists\, l \in Learner,\, v \in CStruct : AbstractLearn(l,\, v) \end{aligned}$$

We define *Spec* to be the complete specification.

$$Spec \triangleq Init \land \Box[Next]_{\langle propCmd,\, learned,\, bA,\, maxTried \rangle}$$

The following theorem asserts the invariance of our invariants

THEOREM
$$Spec \Rightarrow \Box(TypeInv \land maxTriedInvariant \land bAInvariant \land learnedInvariant)$$

The following asserts that our specification *Spec* implies/implements the specification Spec from module *GeneralConsensus*.

$$GC \triangleq \text{INSTANCE } GeneralConsensus$$
THEOREM $Spec \Rightarrow GC\,!\,Spec$

A.2.3 Distributed Abstract Multicoordinated Paxos

This module specifies a distributed version of the abstract algorithm in the previous section. It adds the exchange of messages and shows how the distributed data structures can be mapped back to the non-distributed algorithm. It is used as the second step in the proof of correctness of Multicoordinated Paxos.

MODULE *DistAbsMCPaxos*

EXTENDS *PaxosConstants*

The following variables are similar to those in *AbstractMCPaxos* module. They are changed in this module as they are in *AbstractMCPaxos*.

VARIABLES *propCmd*, *bA*, *learned*

We describe the state of the message-passing system by the value of the variable *msgs*. *I.e.*, processes send messages to each othe by simply putting them in the *msgs* variable.

As we do not specify liveness for this protocol, we do not explicitly model message loss. Because no action is required to happen, messages can simply be ignored, modeling the loss of a message by never executing the action that would read the message.

Message duplication is modeled by not removing the message from $msgs$ once it is read.

VARIABLE $msgs$

We define Msg to be the set of all possible messages. For the sake of clarity and avoiding errors, we let messages be records instead of tuples. For example, the message $\langle$"2a", m, $v\rangle$ in the text becomes a record with type field "2a", bal field m, and val field v.

$$\begin{array}{lll} Msg \triangleq & & [type : \{\text{“1a”}\},\ bal : BalNum] \\ & \cup & [type : \{\text{“1b”}\},\ bal : BalNum,\ acc : Acceptor, \\ & & \quad vote : [BalNum \to CStruct \cup \{none\}]] \\ & \cup & [type : \{\text{“2a”}\},\ bal : BalNum,\ val : CStruct, \\ & & \ coord : Coord] \\ & \cup & [type : \{\text{“2b”}\},\ bal : BalNum,\ acc : Acceptor, \\ & & \ val : CStruct] \end{array}$$

$dMaxTried$ is a distributed version of $maxTried$. Each coordinator stores in $dMaxTried$ the longest c-struct it has send in a round, and $MaxTried$, below, will map it to $maxTried$.

VARIABLE $dMaxTried$

Refinement Mapping

$dMaxTried$ is mapped to $maxTried$ by the $MaxTried$ operator. It maps the c-structs tried by all coordinators in some $coord$-quorum Q in some round m to a single c-struct $Tried(Q, m)$. The set of all $Tried(Q, m)$, $AllTried(m)$, is then mapped to $maxTried$.

$$\begin{array}{l} MaxTried \triangleq \\ \quad \text{LET } Tried(Q, m) \triangleq \text{IF } \exists\, c \in Q : dMaxTried[c][m] = none \\ \qquad\qquad\qquad\qquad \text{THEN } none \\ \qquad\qquad\qquad\qquad \text{ELSE } GLB(\{dMaxTried[c][m] : c \in Q\}) \\ \\ \qquad AllTried(m) \triangleq \{v \in \{Tried(Q, m) : Q \in CoordQuorum(m)\} \\ \qquad\qquad\qquad\qquad : v \neq none\} \\ \\ \quad \text{IN } [m \in BalNum \mapsto \text{IF } AllTried(m) = \{\} \\ \qquad\qquad\qquad\qquad \text{THEN } none \\ \qquad\qquad\qquad\qquad \text{ELSE } LUB(AllTried(m))] \end{array}$$

Invariants

Type invariant. implies Abstract! $TypeInv$.

$$\begin{array}{ll} TypeInv \triangleq & \wedge\ propCmd \subseteq Cmd \\ & \wedge\ learned \in [Learner \to CStruct] \\ & \wedge\ bA \in BallotArray \\ & \wedge\ dMaxTried \in [Coord \to [BalNum \to CStruct \cup \{none\}]] \\ & \wedge\ msgs \subseteq Msg \end{array}$$

Actions

Action $Propose(C)$ adds command C to $propCmd$.

It implements $AbstractMCPaxos\,!\,Propose(C)$ directly.

$$
\begin{array}{l}
Propose(C) \triangleq \\
\quad \land \quad C \notin propCmd \\
\quad \land \quad propCmd' = propCmd \cup \{C\} \\
\quad \land \quad \text{UNCHANGED } \langle bA,\, learned,\, msgs,\, dMaxTried \rangle
\end{array}
$$

Action $Phase1a(c,\, m)$ executes phase $1a$ for round m at coordinator c, sending a "1a" message to all acceptors.

It has no counterpart on $AbstractMCPaxos$.

$$
\begin{array}{l}
Phase1a(c,\, m) \triangleq \\
\quad \land\, dMaxTried[c][m] = none \\
\quad \land\, msgs' = msgs \cup \{[type \mapsto \text{"1a"},\, bal \mapsto m]\} \\
\quad \land \text{ UNCHANGED } \langle propCmd,\, bA,\, learned,\, dMaxTried \rangle
\end{array}
$$

Action $Phase1b(a,\, m)$ executes phase $1b$ for round m at acceptor a.

It implements $JoinBallot(a,\, m)$, and sends a "1b" message to coordinators.

$$
\begin{array}{l}
Phase1b(a,\, m) \triangleq \\
\quad \land\, bA.mbal[a] \prec m \\
\quad \land\, [type \mapsto \text{"1a"},\, bal \mapsto m] \in msgs \\
\quad \land\, bA' = [bA \text{ EXCEPT } !.mbal[a] = m] \\
\quad \land\, msgs' = msgs \cup \\
\qquad\qquad \{[type \mapsto \text{"1b"},\, bal \mapsto m,\, acc \mapsto a,\, vote \mapsto bA.vote[a]]\} \\
\quad \land \text{ UNCHANGED } \langle propCmd,\, learned,\, dMaxTried \rangle
\end{array}
$$

Action $Phase2Start(c,\, m,\, v)$ starts phase $2a$ for round m at coordinator c, proposing the c-struct v.

It implements $AbstractMCPaxos\,!\,StartBallot(m,\, MaxTried'[m])$. Because not all $Phase2Start$ change $MaxTried[m]$, some steps are stuttering regarding $AbstractMCPaxos$.

$$
\begin{array}{l}
Phase2Start(c,\, m,\, v) \triangleq \\
\quad \land\, dMaxTried[c][m] = none \\
\quad \land\, \exists\, Q \in Quorum(m) : \\
\qquad \land\, \forall\, a \in Q : \exists\, msg \in msgs : \land\, msg.type = \text{"1b"} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \land\, msg.bal \;\; = m \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \land\, msg.acc \;\; = a \\
\qquad \land \text{ LET } 1bMsg \triangleq [a \in Q \mapsto \text{CHOOSE } msg \in msgs : \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \land\, msg.type = \text{"1b"} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \land\, msg.bal \;\; = m \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \land\, msg.acc \;\; = a] \\
\qquad\qquad beta \triangleq [vote \mapsto [a \in Q \mapsto 1bMsg[a].vote], \\
\qquad\qquad\qquad\qquad\quad mbal \mapsto [a \in Q \mapsto m]] \\
\qquad \text{IN } \;\; \exists\, w \in ProvedSafe(Q,\, m,\, beta),\, s \in Seq(propCmd) : \\
\qquad\qquad \land\, v = w \mathbin{**} s
\end{array}
$$

$\land\ dMaxTried' = [dMaxTried \text{ EXCEPT } ![c][m] = v]$
$\land\ msgs' = msgs \cup$
$\quad \{[type \mapsto \text{“2a”},\ bal \mapsto m,\ val \mapsto v,\ coord \mapsto c]\}$
$\land$ UNCHANGED $\langle propCmd,\ bA,\ learned\rangle$

Action $Phase2aClassic(c,\ m,\ C)$ is executed by coordinator c of ballot m, for command C. It adds the command C to c's previously tried value and forward the new value to the acceptors.

It implements the $AbstractMCPaxos\,!\,Suggest(m,\ \langle C\rangle)$ action.

$Phase2aClassic(c,\ m,\ C) \triangleq$
$\land\ C \in propCmd$
$\land\ dMaxTried[c][m] \neq none$
$\land\ dMaxTried' =$
$\quad [dMaxTried \text{ EXCEPT } ![c][m] = dMaxTried[c][m] **C]$
$\land\ msgs' = msgs \cup$
$\quad \{[type \mapsto \text{“2a”},\ bal \mapsto m,\ val \mapsto dMaxTried'[c][m],\ coord \mapsto c]\}$
$\land$ UNCHANGED $\langle propCmd,\ bA,\ learned\rangle$

Action $Phase2bClassic(a,\ m,\ v)$ is executed by acceptor a in round m to accept v.

It implements action $AbstractMCPaxos\,!\,ClassicVote(a,\ m,\ v)$.

$Phase2bClassic(a,\ m,\ v) \triangleq$
$\land\ bA.mbal[a] \preceq m$
$\land\ \exists\, L \in CoordQuorum(m),\ u \in CStruct :$
$\quad \land\ \forall\, c \in L : \exists\, msg \in msgs :\ \land\ msg.type = \text{“2a”}$
$\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\ \land\ msg.bal = m$
$\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\ \land\ msg.coord = c$
$\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\ \land\ u \sqsubseteq msg.val$
$\quad \land\ \lor\ \land\ bA.vote[a][m] = none$
$\qquad\qquad\ \land\ v = u$
$\qquad\ \lor\ \land\ AreCompatible(bA.vote[a][m],\ u)$
$\qquad\qquad\ \land\ v = bA.vote[a][m] \sqcup u$
$\land\ bA' = [bA \text{ EXCEPT } !.vote[a][m] = v,\ !.mbal[a] = m]$
$\land\ msgs' = msgs \cup \{[type \mapsto \text{“2b”},\ bal \mapsto m,\ acc \mapsto a,\ val \mapsto v]\}$
$\land$ UNCHANGED $\langle propCmd,\ learned,\ dMaxTried\rangle$

Action $Phase2bFast(a,\ C)$ is executed by acceptor a to accept command C, comming directly from the proposers (fast accept).

It implements the $AbstractMCPaxos\,!\,FastVote(a,\ C)$ action.

$Phase2bFast(a,\ C) \triangleq$
$\land\ C \in propCmd$
$\land\ IsFast(bA.mbal[a])$
$\land\ bA.vote[a][bA.mbal[a]] \neq none$
$\land\ bA' = [bA \text{ EXCEPT } !.vote[a][bA.mbal[a]] = @ \bullet C]$
$\land\ msgs' = msgs \cup \{[type \mapsto \text{“2b”},\ bal \mapsto bA.mbal[a],\ acc \mapsto a,$
$\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\ val \mapsto bA'.vote[a][bA.mbal[a]]]\}$
$\land$ UNCHANGED $\langle propCmd,\ learned,\ dMaxTried\rangle$

Action $Learn(l, v)$ executed by learner l to learn c-struct v.

It implements the $AbstractMCPaxos\,!\,AbstractLearn(a, v)$ action.

$$
\begin{array}{l}
Learn(l, v) \triangleq \\
\quad \land \exists\, m \in BalNum : \\
\qquad \exists\, Q \in Quorum(m) : \\
\qquad\quad \forall\, a \in Q : \exists\, msg \in msgs : \land\, msg.type = \text{“2b”} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \land\, msg.bal = m \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \land\, msg.acc = a \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \land\, v \sqsubseteq msg.val \\
\quad \land\, learned' = [learned \text{ EXCEPT } ![l] = @ \sqcup v] \\
\quad \land \text{UNCHANGED } \langle propCmd, bA, dMaxTried, msgs \rangle
\end{array}
$$

Full Specification

Init defines the initial state.

$$
\begin{array}{l}
Init \triangleq \land\, propCmd = \{\} \\
\qquad\quad \land\, learned \;\; = [l \in Learner \mapsto Bottom] \\
\qquad\quad \land\, bA = [vote \mapsto \\
\qquad\qquad\qquad [a \in Acceptor \mapsto \\
\qquad\qquad\qquad\quad [m \in BalNum \mapsto \text{IF } m = Zero \text{ THEN } Bottom \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{ELSE } none]], \\
\qquad\qquad\quad mbal \mapsto [a \in Acceptor \mapsto Zero]] \\
\qquad\quad \land\, dMaxTried = [c \in Coord \mapsto [m \in BalNum \mapsto \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \text{IF } m = Zero \text{ THEN } Bottom \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \text{ELSE } none]] \\
\qquad\quad \land\, msgs = \{\}
\end{array}
$$

Next defines how action are combined to generate the next states.

$$
\begin{array}{l}
Next \triangleq \lor\, \exists\, C \in Cmd : Propose(C) \\
\qquad\quad \lor\, \exists\, m \in BalNum,\, c \in Coord : Phase1a(c, m) \\
\qquad\quad \lor\, \exists\, m \in BalNum,\, v \in CStruct,\, c \in Coord : \\
\qquad\qquad\quad Phase2Start(c, m, v) \\
\qquad\quad \lor\, \exists\, m \in BalNum,\, s \in Seq(Cmd),\, c \in Coord : \\
\qquad\qquad\quad Phase2aClassic(c, m, s) \\
\qquad\quad \lor\, \exists\, a \in Acceptor,\, m \in BalNum : Phase1b(a, m) \\
\qquad\quad \lor\, \exists\, m \in BalNum,\, a \in Acceptor,\, v \in CStruct : \\
\qquad\qquad\quad Phase2bClassic(a, m, v) \\
\qquad\quad \lor\, \exists\, a \in Acceptor,\, C \in Cmd : Phase2bFast(a, C) \\
\qquad\quad \lor\, \exists\, l \in Learner,\, v \in CStruct : Learn(l, v)
\end{array}
$$

Spec is defined as the complete specification.

$$Spec \triangleq Init \land \Box[Next]_{\langle propCmd,\, bA,\, learned,\, dMaxTried,\, msgs \rangle}$$

The following theorem asserts the invariance of *TypeInv*

THEOREM $Spec \Rightarrow \Box\, TypeInv$

The following theorem implies that there is a refinement mapping from *DistAbsMCPaxos* to *AbstractMCPaxos*. Therefore, *DistAbsMCPaxos* implements *AbstractMCPaxos*. Because *GeneralConsensus* is implemented by *AbstractMCPaxos*, *DistAbsMCPaxos* also implements *GeneralConsensus*

$AB \triangleq$ INSTANCE *AbstractMCPaxos* WITH $maxTried \leftarrow MaxTried$
$GC \triangleq$ INSTANCE *GeneralConsensus*

THEOREM $Spec \Rightarrow AB\,!\,Spec$
THEOREM $Spec \Rightarrow GC\,!\,Spec$

A.2.4 Basic Multicoordinated Paxos

This module specifies the Multicoordinated Paxos algorithm as presented in Section 3.3.

MODULE *DistMCPaxos*

EXTENDS *PaxosConstants*

VARIABLES $crnd$, $cval$, rnd, $vrnd$, $vval$, $learned$, $msgs$
CONSTANT *Proposer*

$cVars \triangleq \langle crnd, cval\rangle$
$aVars \triangleq \langle rnd, vrnd, vval\rangle$

$$\begin{aligned} Msg \triangleq\ & [type : \{\text{“propose”}\},\ cmd : Cmd] \cup \\ & [type : \{\text{“1a”}\},\ bal : BalNum] \cup \\ & [type : \{\text{“1b”}\},\ bal : BalNum,\ vval\ : CStruct, \\ & \qquad\qquad vrnd : BalNum,\ acc : Acceptor] \cup \\ & [type : \{\text{“2a”}\},\ bal : BalNum,\ val : CStruct,\ coord : Coord] \cup \\ & [type : \{\text{“2b”}\},\ bal : BalNum,\ val : CStruct,\ acc : Acceptor] \end{aligned}$$

$$\begin{aligned} TypeInv \triangleq\ & \land\ crnd \in [Coord \to BalNum] \\ & \land\ cval\ \in [Coord \to CStruct] \\ & \land\ rnd\ \in [Acceptor \to BalNum] \\ & \land\ vrnd \in [Acceptor \to BalNum] \\ & \land\ vval\ \in [Acceptor \to CStruct] \\ & \land\ learned \in [Learner \to CStruct] \\ & \land\ msgs \subseteq Msg \end{aligned}$$

Actions

Action $Send(msg)$ implements the sending of message msg.
$Send(msg) \triangleq msgs' = msgs \cup \{msg\}$

$DistProvedSafe(Q, 1bMsg)$ below is the TLA$^+$ version of the $ProvedSafe(Q, 1bMsg)$ function presented in Section 3.3.
$DistProvedSafe(Q, 1bMsg) \triangleq$
LET

$vals(S) \triangleq \{1bMsg[a].vval : a \in S\}$
$vrnds \triangleq \{1bMsg[a].vrnd : a \in Q\}$
$k \triangleq Max(vrnds)$
$kacceptors \triangleq \{a \in Q : 1bMsg[a].vrnd = k\}$
$QinterR \triangleq \{Q \cap R : R \in Quorum(k)\}$
$QinterRAtk \triangleq \{S \in QinterR : S \subseteq kacceptors\}$
$Gamma \triangleq \{GLB(vals(inter)) : inter \in QinterRAtk\}$
IN
IF $QinterRAtk = \{\}$ THEN $vals(kacceptors)$
ELSE $\{LUB(Gamma)\}$

The following actions are the same as those presented in Section 3.3, just translated into TLA$^+$.

Propose(p, C)

$Propose(p, C) \triangleq$
$\land$ $Send([type \mapsto$ "propose", $cmd \mapsto C])$
$\land$ UNCHANGED $\langle cVars, aVars, learned\rangle$

*Phase*1*a*(c, m)

$Phase1a(c, m) \triangleq$
$\land\ c \in$ UNION $CoordQuorum(m)$
$\land\ crnd[c] \prec m$
$\land\ Send([type \mapsto$ "1a", $bal \mapsto m])$
$\land$ UNCHANGED $\langle cVars, aVars, learned\rangle$

*Phase*1*b*(a, m)

$Phase1b(a, m) \triangleq$
$\land\ rnd[a] \prec m$
$\land\ [type \mapsto$ "1a", $bal \mapsto m] \in msgs$
$\land\ rnd' = [rnd$ EXCEPT $![a] = m]$
$\land\ Send([type \mapsto$ "1b", $bal \mapsto m$, $vval \mapsto vval[a]$,
$vrnd \mapsto vrnd[a]$, $acc \mapsto a])$
$\land$ UNCHANGED $\langle cVars, vrnd, vval, learned\rangle$

*Phase*2*Start*(c, m)

$Phase2Start(c, m) \triangleq$
$\land\ crnd[c] \prec m$
$\land\ \exists\, Q \in Quorum(m) :$
$\land\ \forall\, a \in Q : \exists\, msg \in msgs :$ $\land\ msg.type =$ "1b"
$\land\ msg.bal = m$
$\land\ msg.acc = a$
$\land$ LET $1bMsg \triangleq [a \in Q \mapsto$ CHOOSE $msg \in msgs :$
$\land\ msg.type =$ "1b"
$\land\ msg.bal = m$
$\land\ msg.acc = a]$
IN $\exists\, w \in DistProvedSafe(Q, 1bMsg) :$
$\land\ crnd' = [crnd$ EXCEPT $![c] = m]$

```
                ∧ cval′ = [cval EXCEPT ![c] = w]
                ∧ Send([type ↦ "2a", bal ↦ m, val ↦ w, coord ↦ c])
                ∧ UNCHANGED ⟨aVars, learned⟩
```

Phase2aClassic(*c*)

```
Phase2aClassic(c) ≜
  ∃ msg ∈ msgs :
    ∧ msg.type = "propose"
    ∧ cval′ = [cval EXCEPT ![c] = @ • msg.cmd]
    ∧ Send([type ↦ "2a", bal ↦ crnd[c], val ↦ cval′[c], coord ↦ c])
    ∧ UNCHANGED ⟨crnd, aVars, learned⟩
```

Phase2bClassic(*a*, *m*)

```
Phase2bClassic(a, m) ≜
  ∧ rnd[a] ⪯ m
  ∧ ∃ L ∈ CoordQuorum(m) :
      ∧ ∀ c ∈ L : ∃ msg ∈ msgs : ∧ msg.type = "2a"
                                 ∧ msg.bal = m
                                 ∧ msg.coord = c
      ∧ LET
          A2aMsg(c) ≜ CHOOSE msg ∈ msgs : ∧ msg.type = "2a"
                                          ∧ msg.bal = m
                                          ∧ msg.coord = c
          L2aMsgs ≜ {A2aMsg(c) : c ∈ L}
          L2aVals ≜ {msg.val : msg ∈ L2aMsgs}
        IN
          ∧ ∨ ∧ vrnd[a] ≺ m
              ∧ vval′ = [vval EXCEPT ![a] = GLB(L2aVals)]
            ∨ ∧ vrnd[a] = m
              ∧ AreCompatible(vval[a], GLB(L2aVals))
              ∧ vval′ = [vval EXCEPT ![a] = @ ⊔ GLB(L2aVals)]
          ∧ rnd′ = [rnd EXCEPT ![a] = m]
          ∧ vrnd′ = [vrnd EXCEPT ![a] = m]
          ∧ Send([type ↦ "2b", bal ↦ m, val ↦ vval′[a], acc ↦ a])
          ∧ UNCHANGED ⟨cVars, learned⟩
```

Phase2bFast(*a*)

```
Phase2bFast(a) ≜
  ∧ IsFast(rnd[a])
  ∧ rnd[a] = vrnd[a]
  ∧ ∃ msg ∈ msgs :
      ∧ msg.type = "propose"
      ∧ vval′ = [vval EXCEPT ![a] = @ • msg.cmd]
      ∧ Send([type ↦ "2b", bal ↦ rnd[a], val ↦ vval′[a], acc ↦ a])
      ∧ UNCHANGED ⟨cVars, rnd, vrnd, learned⟩
```

Learn(*l*)

$$
\begin{array}{l}
Learn(l) \triangleq \\
\quad \exists\, m \in BalNum : \\
\quad\quad \exists\, Q \in Quorum(m) : \\
\quad\quad\quad \land \forall\, a \in Q : \exists\, msg \in msgs : \land\, msg.type = \text{“2b”} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \land\, msg.bal\ \ = m \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \land\, msg.acc\ = a \\
\quad\quad\quad \land \text{LET} \\
\qquad\qquad A2bMsg(a) \triangleq \text{CHOOSE}\ msg \in msgs : \land\, msg.type = \text{“2b”} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \land\, msg.bal = m \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \land\, msg.acc \in Q \\
\qquad\qquad Q2bMsgs \triangleq \{A2bMsg(a) : a \in Q\} \\
\qquad\qquad Q2bVals\ \triangleq \{msg.val : msg\ \in Q2bMsgs\} \\
\qquad\quad \text{IN} \\
\qquad\qquad \land\, learned' = [learned\ \text{EXCEPT}\ ![l] = @ \sqcup GLB(Q2bVals)] \\
\qquad\qquad \land\, \text{UNCHANGED}\ \langle cVars,\, aVars,\, msgs\rangle
\end{array}
$$

Full Specification

Init defines the initial state.

$$
\begin{array}{l}
Init \triangleq\ \land\, learned = [l \in Learner \mapsto Bottom] \\
\qquad\quad \land\, crnd = [c \in Coord \mapsto Zero] \\
\qquad\quad \land\, cval\ = [c \in Coord \mapsto Bottom] \\
\qquad\quad \land\, rnd\ \ = [a \in Acceptor \mapsto Zero] \\
\qquad\quad \land\, vrnd = [a \in Acceptor \mapsto Zero] \\
\qquad\quad \land\, vval\ = [a \in Acceptor \mapsto Bottom] \\
\qquad\quad \land\, learned = [l \in Learner \mapsto Bottom] \\
\qquad\quad \land\, msgs = \{\}
\end{array}
$$

Next defines how action are combined to generate the next states.

$$
\begin{array}{l}
Next \triangleq\ \lor \exists\, p \in Proposer,\, C \in Cmd\ : Propose(p,\, C) \\
\qquad\quad \lor \exists\, c \in Coord,\, m \in BalNum : Phase1a(c,\, m) \\
\qquad\quad \lor \exists\, c \in Coord,\, m \in BalNum : Phase2Start(c,\, m) \\
\qquad\quad \lor \exists\, c \in Coord : Phase2aClassic(c) \\
\qquad\quad \lor \exists\, a \in Acceptor,\, m \in BalNum : Phase1b(a,\, m) \\
\qquad\quad \lor \exists\, a \in Acceptor,\, m \in BalNum : Phase2bClassic(a,\, m) \\
\qquad\quad \lor \exists\, a \in Acceptor : Phase2bFast(a) \\
\qquad\quad \lor \exists\, l\ \in Learner : Learn(l)
\end{array}
$$

Spec is defined as the complete specification.

$$Spec \triangleq\ Init \land \Box[Next]_{\langle cVars,\, aVars,\, learned,\, msgs\rangle}$$

The following theorem asserts the invariance of *TypeInv*.

THEOREM $Spec \Rightarrow \Box\, TypeInv$

The following theorem asserts the *DistMCPaxos* implements *GeneralConsensus*

$PropCmd \triangleq \{m.cmd : m \in \{mm \in msgs : mm.type = \text{“propose”}\}\}$

$GC \triangleq$ INSTANCE $GeneralConsensus$ WITH $propCmd \leftarrow PropCmd$

THEOREM $Spec \Rightarrow GC\,!\,Spec$

A.2.5 Complete Multicoordinated Paxos

This module specifies the Multicoordinated Paxos algorithm without dependencies, also specifying the collision detection mechanisms and a simplified version of the mechanism presented in Section 3.4.5 to reduce the number of disk writes.

MODULE $MultiCoordPaxos$

The module imports two standard modules. Module $Naturals$ defines the set Nat of naturals and the ordinary arithmetic operators; module $FiniteSets$ defines $IsFiniteSet(s)$ to be true iff S is a finite set and defines $Cardinality(S)$ to be the number of elements in S, if s is finite.

EXTENDS $Naturals$, $FiniteSets$, $CStructs$

Constants

The next statement declares the specification's constant parameters, which have the following meanings:

Acceptor	the set of acceptors.
Learner	the set of learners.
FastNum	the set of fast round numbers.
Quorum(i)	the set of i-quorums.
Coord	the set of coordinators.
CoordQuorum(i)	the set of coordinator quorums of round i.

CONSTANTS $Acceptor$, $Learner$, $FastNum$, $Quorum(\_)$, $Coord$, $CoordQuorum(\_)$

$PosNat$ is defined to be the set of positive integers.

$PosNat \triangleq Nat \setminus \{0\}$

$RNum$ is the set of round numbers. A round number is composed of two parts: the incarnation number and the sequence number.

$RNum \triangleq Nat \times PosNat$

A round number with sequence number 0 is not a valid round number, but it is used to represent some agents' states. $RType$ is defined to represent such a set.

$RType \triangleq Nat \times Nat$

We must define a precedence relation between round numbers. We define $\prec$ to represent the relation that round i precedes round j iff i has a lower incarnation number than j or they have the same incarnation number but i has a lower sequence number than j. We also define $\preceq$, $\succ$, and $\succeq$ accordingly.

$i \prec j \triangleq$ IF $i[1] < j[1]$ THEN TRUE ELSE $i[2] < j[2]$

$i \preceq j \triangleq i \prec j \lor i = j$
$i \succ j \triangleq j \prec i$
$i \succeq j \triangleq j \preceq i$

$Max(S)$ is defined to be the maximum of a finite set S of Round Numbers.

$Max(S) \triangleq$ CHOOSE $i \in S : \forall\, j \in S : i \succeq j$

the following statement asserts the assumption that $FastNum$ is a set of round numbers.

ASSUME $FastNum \subseteq RNum$

$ClassicNum$ is defined to be the set of classic round numbers.

$ClassicNum \triangleq RNum \setminus FastNum$

$FairNum(c)$ is defined to be the set of classic round numbers for which coordinator c is, itself, a coordinator quorum.

$FairNum(c) \triangleq \{i \in ClassicNum : \{c\} \in CoordQuorum(i)\}$

The following assumption asserts that the set of acceptors is finite. It is needed to ensure progress.

ASSUME $IsFiniteSet(Acceptor)$

The following asserts the assumptions that $Quorum(i)$ is a set of sets of acceptors, for every round number i, and that the $Quorum$ Requirement holds.

```
ASSUME ∀ i ∈ RNum :
          ∧ Quorum(i) ⊆ SUBSET Acceptor
          ∧ ∀ j ∈ RNum :
              ∧ ∀ Q ∈ Quorum(i), R ∈ Quorum(j) : Q ∩ R ≠ {}
              ∧ (j ∈ FastNum) ⇒
                  ∀ Q ∈ Quorum(i) : ∀ R1, R2 ∈ Quorum(j) :
                    Q ∩ R1 ∩ R2 ≠ {}
```

The following asserts the assumptions that $CoordQuorum(i)$ is a set of sets of coordinators, for every round number i, and that every coordinator is, itself, a coordinator quorum of infinitely many classic rounds for every possible incarnation.

```
ASSUME ∧ ∀ i ∈ RNum : CoordQuorum(i) ⊆ SUBSET Coord
       ∧ ∀ c ∈ Coord, i ∈ RType :
           ∃ j ∈ PosNat : ∧ j > i[2]
                          ∧ ⟨i[1], j⟩ ∈ FairNum(c)
```

The following asserts the assumption that coordinator quorums of the same round should always intersect.

ASSUME $\forall\, i \in RNum : \forall\, Q, R \in CoordQuorum(i) : Q \cap R \neq \{\}$

Message is defined to be the set of all possible messages. A message is a record having a *type* field indicating what message it is, and a *rnd* field indicating the round number. What other fields, if any, a message has depends on its type.

```
Message ≜  [type : {"phase1a"}, rnd : RNum]
      ∪    [type : {"phase1b"}, rnd : RNum, vval : CStruct,
                                vrnd : RType, acc : Acceptor]
```

$\cup$ $[type : \{\text{“phase2a”}\}, rnd : RNum, val : CStruct, coord : Coord]$
$\cup$ $[type : \{\text{“phase2b”}\}, rnd : RNum, val : CStruct, acc : Acceptor]$

Variables and State Predicates

The following statement declares the specification's variables.

VARIABLES rnd, $vrnd$, $vval$, $crnd$, $cval$, $amLeader$, $sentMsg$, $proposed$, $learned$, $goodSet$

Defining the following tuples of variables makes it more convenient to state which variables are left unchanged by the actions.

$aVars \triangleq \langle rnd, vrnd, vval \rangle$ Acceptor variables
$cVars \triangleq \langle crnd, cval \rangle$ Coordinator variables
$oVars \triangleq \langle amLeader, proposed, learned, goodSet \rangle$ Most other variables
$vars \triangleq \langle aVars, cVars, oVars, sentMsg \rangle$ All variables

$TypeOK$ is the type-correctness invariant, asserting that the value of each variable is an element of the proper set (its “type”). Type correctness of the specification means that $TypeOK$ is an invariant—— that is, it is true in every state of every behavior allowed by the specification.

$$
\begin{aligned}
&TypeOK \triangleq \\
&\quad \land \ rnd \in [Acceptor \to RType] \\
&\quad \land \ vrnd \in [Acceptor \to RType] \\
&\quad \land \ vval \in [Acceptor \to CStruct] \\
&\quad \land \ crnd \in [Coord \to RType] \\
&\quad \land \ cval \in [Coord \to CStruct \cup \{none\}] \\
&\quad \land \ amLeader \in [Coord \to \text{BOOLEAN}] \\
&\quad \land \ sentMsg \in \text{SUBSET } Message \\
&\quad \land \ proposed \in \text{SUBSET } Cmd \\
&\quad \land \ learned \in [Learner \to CStruct] \\
&\quad \land \ goodSet \subseteq Acceptor \cup Coord
\end{aligned}
$$

$Init$ is the initial predicate that describes the initial values of all variables.

$$
\begin{aligned}
&Init \triangleq \\
&\quad \land rnd = [a \in Acceptor \mapsto \langle 0, 0 \rangle] \\
&\quad \land vrnd = [a \in Acceptor \mapsto \langle 0, 0 \rangle] \\
&\quad \land vval = [a \in Acceptor \mapsto Bottom] \\
&\quad \land crnd = [c \in Coord \mapsto \langle 0, 0 \rangle] \\
&\quad \land cval = [c \in Coord \mapsto none] \\
&\quad \land amLeader \in [Coord \to \text{BOOLEAN}] \\
&\quad \land sentMsg = \{\} \\
&\quad \land proposed = \{\} \\
&\quad \land learned = [l \in Learner \mapsto Bottom] \\
&\quad \land goodSet \in \text{SUBSET } (Acceptor \cup Coord)
\end{aligned}
$$

Action Definitions

$Send(m)$ describes the state change that represents the sending of a message m. It is used as a conjunt in defining the algorithm actions.

$Send(msg) \triangleq sentMsg' = sentMsg \cup \{msg\}$

Coordinator Actions

Action $Phase1a(c, i)$ specifies the execution of phase $1a$ of round i by coordinator c. Different from the previous specifications, this action changes $crnd$ and $cval$. This is done for liveness, to prevent a coordinator from continuously starting new rounds. It could be done by adding a new variable but we just thought that this way was easier and compliant with other $Paxos$ specifications.

$Phase1a(i, c) \triangleq$
$\land\ amLeader[c]$
$\land\ c \in$ UNION $CoordQuorum(i)$
$\land\ crnd[c] \prec i$
$\land\ \lor\ crnd[c] = \langle i[1], 0\rangle$
$\quad \lor \exists\, m \in sentMsg : \land\ crnd[c] \prec m.rnd$
$\qquad\qquad\qquad\qquad\ \land\ m.rnd[1] = i[1]$
$\qquad\qquad\qquad\qquad\ \land\ m.rnd \prec i$
$\quad \lor \land\ crnd[c] \notin FairNum(c)$
$\qquad\ \land\ i[1] = crnd[c][1]$
$\land\ crnd' = [crnd$ EXCEPT $![c] = i]$
$\land\ cval' = [cval$ EXCEPT $![c] = none]$
$\land\ Send([type \mapsto$ "phase1a", $rnd \mapsto i])$
$\land$ UNCHANGED $\langle aVars, oVars\rangle$

Reasons for executing $Phase1a$:
1 - Did not do anything in this incarnation
2 - Some round interfered with round $crnd[c]$
3 - Round $crnd[c]$ might have collisions and cannot ensure liveness in the presence of failures

$MsgsFrom(Q, i, phase)$ is defined to be the set of messages in $sentMsg$ of type *phase* (which may equal "phase1b" or "phase2b") sent in round i by the acceptors in the set Q.

$MsgsFrom(Q, i, phase) \triangleq$
$\quad \{m \in sentMsg : (m.type = phase) \land (m.acc \in Q) \land (m.rnd = i)\}$

If M is the set of round i phase $1b$ messages sent by the acceptors in a quorum Q, then $IsPickableVal(Q, i, M, v)$ is true according to the following rule, easily derived from the definition of $ProvedSafe$ in the paper. It allows the coordinator to send the value v in a phase $2a$ message for round i.

$IsPickableVal(Q, i, M, v) \triangleq$
$\quad$ LET $vr(a) \triangleq$ (CHOOSE $m \in M : m.acc = a).vrnd$
$\qquad vv(a) \triangleq$ (CHOOSE $m \in M : m.acc = a).vval$
$\qquad k \triangleq Max(\{vr(a) : a \in Q\})$
$\qquad RS \triangleq \{R \in Quorum(k) : \forall\, a \in Q \cap R : vr(a) = k\}$
$\qquad g(R) \triangleq GLB(\{vv(a) : a \in R \cap Q\})$
$\qquad G \triangleq \{g(R) : R \in RS\}$
$\qquad PrSafe \triangleq$
$\qquad\quad$ IF $RS = \{\}$ THEN $\{vv(a) : a \in \{b \in Q : vr(b) = k\}\}$
$\qquad\qquad\qquad$ ELSE $\{LUB(G)\}$

$\quad$ IN $\exists\, w \in PrSafe, s \in Seq(proposed) : v = w \mathbin{**} s$

$Phase2Start(i, c, v)$ specifies the first execution of $phase2a$ in round i by coordinator c.

```
Phase2Start(i, c, v) ≜                          has executed phase1a, but not
    ∧ ∨ ∧ crnd[c] = i                           phase2a, or another coordinator has
          ∧ cval[c] = none                      executed phase1a.
      ∨ crnd[c] ≺ i

    ∧ ∃ Q ∈ Quorum(i) :
        ∧ ∀ a ∈ Q : ∃ m ∈ MsgsFrom(Q, i, "phase1b") : m.acc = a
        ∧ IsPickableVal(Q, i, MsgsFrom(Q, i, "phase1b"), v)
    ∧ cval' = [cval EXCEPT ![c] = v]
    ∧ crnd' = [crnd EXCEPT ![c] = i]
    ∧ Send([type ↦ "phase2a", rnd ↦ i, val ↦ v, coord ↦ c])
    ∧ UNCHANGED ⟨aVars, oVars⟩
```

*Phase*2*a*(*i*, *c*, *v*) specifies other executions of *phase*2*a* in round *i* by coordinator *c*.

```
Phase2a(i, c, v) ≜                              has executed Phase2Start.
    ∧   crnd[c] = i
    ∧   cval[c] ≠ none
    ∧   ∃ C ∈ proposed : v = cval[c] ∗∗ C
    ∧ cval' = [cval EXCEPT ![c] = v]
    ∧ Send([type ↦ "phase2a", rnd ↦ i, val ↦ v, coord ↦ c])
    ∧ UNCHANGED ⟨crnd, aVars, oVars⟩
```

NextRound(*i*) is the round number following *i* in the same incarnation.

```
NextRound(i) ≜ [i EXCEPT ![2] = @ + 1]
```

*NextRoundP*1*b*(*Q*, *i*) is the set of phase 1*b* messages for round *NextRound*(*i*) sent by acceptors in *Q*.

```
NextRoundP1b(Q, i) ≜ MsgsFrom(Q, NextRound(i), "phase1b")
```

Action *CoordinatedRecovery*(*i*, *c*, *v*) specifies our variation of coordinated recovery. With this action, coordinator *c* attempts to recover from a collision in round *i* by sending round *NextRound*(*i*) phase 2*a* messages for the value *v*. To ensure liveness, *NextRound*(*i*) should be a fair round, but this is not a requirement for correctness.

```
CoordinatedRecovery(i, c, v) ≜
    LET j ≜ NextRound(i)
    IN  ∧ crnd[c] ≺ j
        ∧ ∃ Q ∈ Quorum(j) :
            ∧ ∀ a ∈ Q : ∃ m ∈ NextRoundP1b(Q, i) : m.acc = a
            ∧ IsPickableVal(Q, j, NextRoundP1b(Q, i), v)
        ∧ cval' = [cval EXCEPT ![c] = v]
        ∧ crnd' = [crnd EXCEPT ![c] = j]
        ∧ Send([type ↦ "phase2a", rnd ↦ j, val ↦ v, coord ↦ c])
        ∧ UNCHANGED ⟨aVars, oVars⟩
```

coordLastMsg(*c*) is defined to be the last message that coordinator *c* sent, if $crnd[c] \succ \langle 0, 0\rangle$.

```
coordLastMsg(c) ≜
    IF cval[c] = none
        THEN [type ↦ "phase1a", rnd ↦ crnd[c]]
        ELSE [type ↦ "phase2a", rnd ↦ crnd[c],
```

$val \mapsto cval[c],\ coord \mapsto c]$

In action $CoordRetransmit(c)$, coordinator c retransmits the last message it sent. This action is a stuttering action (meaning it does not change the value of any variable, so it is a no-op) if that message is still in $sentMsg$. However, this action is needed because c might have failed after first sending the message and subsequently have been repaired after the message was removed from $sentMsg$.

$$
\begin{array}{l}
CoordRetransmit(c) \triangleq \\
\quad \land\ crnd[c] \in RNum \\
\quad \land\ Send(coordLastMsg(c)) \\
\quad \land\ \text{UNCHANGED}\ \langle aVars,\ cVars,\ amLeader,\ proposed, \\
\qquad\qquad\qquad learned,\ goodSet \rangle
\end{array}
$$

$CoordNext(c)$ is the next-state action of coordinator c—— that is, the disjunct of the algorithm's complete next-state action that represents actions of that coordinator.

$$
\begin{array}{l}
CoordNext(c) \triangleq \\
\quad \lor \exists\, i \in RNum : \lor Phase1a(i,\ c) \\
\qquad\qquad\qquad\qquad \lor \exists\, v \in CStruct : \lor Phase2Start(i,\ c,\ v) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \lor Phase2a(i,\ c,\ v) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \lor CoordinatedRecovery(i,\ c,\ v) \\
\quad \lor CoordRetransmit(c)
\end{array}
$$

Acceptor Actions

Action $Phase1b(i,\ a)$ specifies the execution of phase $1b$ for round i by acceptor a.

$$
\begin{array}{l}
Phase1b(i,\ a) \triangleq \\
\quad \land\ rnd[a] \prec i \\
\quad \land\ [type \mapsto \text{“phase1a”},\ rnd \mapsto i] \in sentMsg \\
\quad \land\ rnd' = [rnd\ \text{EXCEPT}\ ![a] = i] \\
\quad \land\ Send([type \mapsto \text{“phase1b”},\ rnd \mapsto i,\ vrnd \mapsto vrnd[a], \\
\qquad vval \mapsto vval[a],\ acc \mapsto a]) \\
\quad \land\ \text{UNCHANGED}\ \langle cVars,\ oVars,\ vrnd,\ vval \rangle
\end{array}
$$

$MsgsFromCoordQuorum(Q,\ i,\ phase)$ is defined to be the set of messages in $sentMsg$ of type *phase* (which may equal “phase1a” or “phase2a”) sent in round i by the coordinators in the set Q.

$$
\begin{array}{l}
MsgsFromCoordQuorum(Q,\ r,\ phase) \triangleq \\
\quad \{m \in sentMsg : \land\ (m.type = phase) \\
\qquad\qquad\qquad\qquad \land\ (m.coord \in Q) \\
\qquad\qquad\qquad\qquad \land\ (m.rnd = r)\}
\end{array}
$$

Action $Phase2b(i,\ a,\ v)$ specifies the execution of phase $2b$ for round i by acceptor a, upon receipt of either a phase $2a$ message or a proposal (for a fast round) with value v. This action actually implements actions $Phase2bClassic$ and $Phase2bFast$ of the previous specifications

$$
\begin{array}{l}
Phase2b(i,\ a,\ v) \triangleq \\
\quad \land\quad rnd[a] \preceq i \\
\quad \land\quad \lor vrnd[a] \prec i \\
\qquad\quad \lor vval[a] \sqsubset v \\
\quad \land\quad \lor \exists\, Q \in CoordQuorum(i),\ u \in CStruct : \\
\qquad\quad \land \forall\, c \in Q : \exists\, m \in MsgsFromCoordQuorum(Q,\ i, \text{“phase2a”}) :
\end{array}
$$

```
                                    ∧ c = m.coord
                                    ∧ u ⊑ m.val
                ∧ ∨ ∧ vrnd[a] ≺ i
                    ∧ v = u
                  ∨ ∧ vrnd[a] = i
                    ∧ AreCompatible(vval[a], u)
                    ∧ v = vval[a] ⊔ u
              ∨ ∧ i ∈ FastNum
                ∧ vrnd[a] = i
                ∧ ∃ C ∈ proposed : v = vval[a] ∗∗ C   Sent in classic 2a.
    ∧   rnd'  = [rnd  EXCEPT ![a] = i]
    ∧   vrnd' = [vrnd EXCEPT ![a] = i]
    ∧   vval' = [vval EXCEPT ![a] = v]
    ∧   Send([type ↦ "phase2b", rnd ↦ i, val ↦ v, acc ↦ a])
    ∧   UNCHANGED ⟨cVars, oVars⟩
```

Action $CollisionDetection(i, a)$ specifies the action acceptor a must take when it detects a collision on the values proposed by a coordinator quorum or on the values accepted by an acceptor quorum for round i. In such a case, acceptor a sends a $phase1b$ message for round $i + 1$.

```
CollisionDetection(i, a) ≜
    ∧ rnd[a] ≼ i
    ∧ ∨ ∃ Q ∈ CoordQuorum(i) :          collision in a multicoordinated round
          ∃ m1, m2 ∈ MsgsFromCoordQuorum(Q, i, "phase2a") :
                ∧ m1.coord ≠ m2.coord
                ∧ ¬AreCompatible(m1.val, m2.val)
      ∨ ∃ Q ∈ Quorum(i) :               collision in a fast round
          ∃ m1, m2 ∈ MsgsFrom(Q, i, "phase2b") :
                ∧ m1.acc ≠ m2.acc
                ∧ ¬AreCompatible(m1.val, m2.val)
    ∧ rnd' = [rnd EXCEPT ![a] = i]
    ∧ Send([type ↦ "phase1b", rnd ↦ NextRound(i), vrnd ↦ vrnd[a],
            vval ↦ vval[a], acc ↦ a])
    ∧ UNCHANGED ⟨cVars, oVars, vrnd, vval⟩
```

Action $UncoordinatedRecovery(i, a, v)$ specifies our variation of uncoordinated recovery. With this action, acceptor a attempts to recover from a collision in round i by sending a phase $2b$ message for round $NextRound(i)$ with value v.

```
UncoordinatedRecovery(i, a, v) ≜
  LET j ≜ NextRound(i)
  IN  ∧ j ∈ FastNum
      ∧ rnd[a] ≼ i
      ∧ ∃ Q ∈ Quorum(j) :
          ∧ ∀ b ∈ Q : ∃ m ∈ NextRoundP1b(Q, i) : m.acc = b
          ∧ IsPickableVal(Q, j, NextRoundP1b(Q, i), v)
      ∧ rnd' = [rnd EXCEPT ![a] = j]
      ∧ vrnd' = [vrnd EXCEPT ![a] = j]
      ∧ vval' = [vval EXCEPT ![a] = v]
      ∧ Send([type ↦ "phase2b", rnd ↦ j, val ↦ v, acc ↦ a])
      ∧ UNCHANGED ⟨cVars, oVars⟩
```

$accLastMsg(a)$ is defined to be the last message sent by acceptor a, if $rnd[a] \succ \langle 0, 0\rangle$

```
accLastMsg(a) ≜
  IF vrnd[a] ≺ rnd[a]
     THEN [type ↦ "phase1b", rnd ↦ rnd[a], vrnd ↦ vrnd[a],
           vval ↦ vval[a], acc ↦ a]
     ELSE [type ↦ "phase2b", rnd ↦ rnd[a], val ↦ vval[a],
           acc  ↦ a]
```

In action $AcceptorRetransmit(a)$ acceptor a retransmits the last message it sent.

```
AcceptorRetransmit(a) ≜
  ∧ rnd[a] ∈ RNum
  ∧ Send(accLastMsg(a))
  ∧ UNCHANGED ⟨aVars, cVars, amLeader, proposed,
               learned, goodSet⟩
```

$AcceptorNext(a)$ is the next-state action of acceptor a—— that is, the disjunct of the next-state action that represents actions of that acceptor.

```
AcceptorNext(a) ≜
  ∨ ∃ i ∈ RNum : ∨ Phase1b(i, a)
                 ∨ ∃ v ∈ CStruct : ∨ Phase2b(i, a, v)
                                   ∨ UncoordinatedRecovery(i, a, v)
                 ∨ CollisionDetection(i, a)
  ∨ AcceptorRetransmit(a)
```

Other Actions

Action $Propose(v)$ represents the proposal of a value v by some proposer.

```
Propose(v) ≜
    ∧  proposed′ = proposed ∪ {v}
    ∧  UNCHANGED ⟨aVars, cVars, amLeader, sentMsg,
                  learned, goodSet⟩
```

Action $Learn(l, v)$ represents the learning of a value v by learner l.

```
Learn(l, v) ≜
    ∧ ∃ i ∈ RNum :
       ∃ Q ∈ Quorum(i) :
         ∀ a ∈ Q :
           ∃ m ∈ sentMsg : ∧ m.type = "phase2b"
                           ∧ m.rnd = i
                           ∧ m.acc = a
                           ∧ v ⊑ m.val
    ∧ learned′ = [learned EXCEPT ![l] = @ ⊔ {v}]
    ∧ UNCHANGED ⟨aVars, cVars, amLeader, sentMsg, proposed, goodSet⟩
```

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 $amLeader[c]$.)

$$
\begin{array}{l}
LeaderSelection \triangleq \\
\quad \land\ amLeader' \in [\,Coord \to \text{BOOLEAN}\,] \\
\quad \land\ \text{UNCHANGED}\ \langle aVars,\ cVars,\ sentMsg,\ proposed, \\
\qquad\qquad\qquad learned,\ goodSet\rangle
\end{array}
$$

Action $Fail(a)$ specifies the failure of agent a.

$$
\begin{array}{l}
Fail(a) \triangleq \\
\quad \land\ a \in goodSet \\
\quad \land\ goodSet' = goodSet \setminus \{a\} \\
\quad \land\ \text{UNCHANGED}\ \langle aVars,\ cVars,\ amLeader,\ sentMsg, \\
\qquad\qquad\qquad proposed,\ learned\rangle
\end{array}
$$

$Repair(a)$ specifies the recovery of agent a. For simplicity, we model the loss of state in $crnd[a]$, $cval[a]$, or $rnd[a]$ during recovery.

$$
\begin{array}{l}
Repair(a) \triangleq \\
\quad \land\ a \notin goodSet \\
\quad \land\ goodSet' = goodSet \cup \{a\} \\
\quad \land\ \text{IF}\ a \in Coord \\
\qquad \text{THEN}\ \land\ crnd' = [crnd\ \text{EXCEPT}\ ![a][1] = @ + 1,\ ![a][2] = 0] \\
\qquad\qquad\ \ \land\ cval' = [cval\ \text{EXCEPT}\ ![a] = none] \\
\qquad \text{ELSE}\ \text{UNCHANGED}\ cVars \\
\quad \land\ \text{IF}\ a \in Acceptor \\
\qquad \text{THEN}\ \land\ rnd' = [rnd\ \text{EXCEPT}\ ![a][1] = @ + 1,\ ![a][2] = 1] \\
\qquad \text{ELSE}\ \text{UNCHANGED}\ rnd \\
\quad \land\ \text{UNCHANGED}\ \langle vrnd,\ vval,\ amLeader,\ sentMsg,\ proposed,\ learned\rangle
\end{array}
$$

Action $FailOrRepair$ allows the failure or recovery of an agent a. Since this action may be performed at any time, the specification makes no assumption about which agents are good. (However, progress is guaranteed only under an assumption about the value of $goodSet$.)

$$
\begin{array}{l}
FailOrRepair \triangleq\ \exists\, a \in (Coord \cup Acceptor) : \\
\qquad\qquad\qquad\qquad \lor\ Fail(a) \\
\qquad\qquad\qquad\qquad \lor\ Repair(a)
\end{array}
$$

Action $LoseMsg(m)$ removes message m from $sentMsg$. It is always enabled unless m is the last message sent by an acceptor or coordinator in $goodSet$. Hence, the only assumption the specification makes about message loss is that the last message sent by an agent in $goodSet$ is not lost. Because $sentMsg$ includes messages in an agent's output buffer, this effectively means that a non-failed process always has the last message it sent in its output buffer, ready to be retransmitted.

$$
\begin{array}{l}
LoseMsg(m) \triangleq \\
\quad \land \neg \lor \land\ m.type = \text{“phase1a”} \\
\qquad\quad \land \exists\, c \in \text{UNION}\ CoordQuorum(m.rnd) : \\
\qquad\qquad\quad \land\ m = coordLastMsg(c) \\
\qquad\qquad\quad \land\ c\ \ \in goodSet \\
\qquad \lor \land\ m.type = \text{“phase2a”} \\
\qquad\quad \land\ m = coordLastMsg(m.coord)
\end{array}
$$

$\quad\quad\quad\quad \land\ m.coord \in goodSet$
$\quad\quad\quad \lor\ \land\ m.type \in \{\text{“phase1b”}, \text{“phase2b”}\}$
$\quad\quad\quad\quad\ \land\ m = accLastMsg(m.acc)$
$\quad\quad\quad\quad\ \land\ m.acc \in goodSet$
$\quad \land\ sentMsg' = sentMsg \setminus \{m\}$
$\quad \land\ \text{UNCHANGED}\ \langle aVars,\ cVars,\ amLeader,\ proposed,$
$\quad\quad\quad\quad\quad\quad\quad learned,\ goodSet\rangle$

Action *OtherAction* is the disjunction of all actions other than ones peformed by acceptors or coordinators, plus the *LeaderSelection* action (which represents leader-selection actions performed by the coordinators).

$OtherAction \triangleq$
$\quad \lor \exists\, v \in Cmd : Propose(v)$
$\quad \lor \exists\, v \in CStruct,\ l \in Learner : Learn(l,\ v)$
$\quad \lor LeaderSelection$
$\quad \lor FailOrRepair$
$\quad \lor \exists\, m \in sentMsg : LoseMsg(m)$

Next is the algorithm's complete next-state action.

$Next \triangleq$
$\quad \lor \exists\, c \in Coord : CoordNext(c)$
$\quad \lor \exists\, a \in Acceptor : AcceptorNext(a)$
$\quad \lor\ OtherAction$

Formula *Spec* is the complete specification of the Multi-coordinated Paxos algorithm without fairness.

$Spec \triangleq\ Init \land \Box[Next]_{vars}$

The following are the safety properties of Generalized Consensus.

$Nontriviality \triangleq\ \forall\, l \in Learner :$
$\quad\quad\quad\quad \Box(learned[l] \in Str(proposed))$

$Stability \quad\ \triangleq\ \forall\, l \in Learner,\ v \in CStruct :$
$\quad\quad\quad\quad \Box((learned[l]\ = v) \Rightarrow \Box(v \sqsubseteq learned[l]))$

$Consistency \triangleq\ \forall\, l1,\ l2 \in Learner :$
$\quad\quad\quad\quad\quad \Box AreCompatible(learned[l1],\ learned[l2])$

The following theorem asserts the correctness of the algorithm.

$\text{THEOREM}\ Spec \Rightarrow \Box(\,TypeOK) \land Nontriviality$
$\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \land\ Stability$
$\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \land\ Consistency$

Appendix B

Log Service

The specifications presented in Sections 5.3 and 5.5, are simplified versions of our complete specifications. For the unabridged versions, we used Lamport's Temporal Logic of Actions (TLA$^+$) specification language [Lamport, 2002a]. The language borrows most of its formalism from basic mathematics, and reading it should be straightforward except maybe for a few constructs. The TLA$^+$cheat-sheet should be enough to clarify any other doubts[Lamport, 2002b].

B.1 Abstract Specification

B.1.1 Constants

This module specifies constants used by all the other specifications.

MODULE *LogServiceConstants*

This module specifies the *LogService*'s constants.

The service's constant parameters:

RM	the set of resource managers.
TID	the set of transaction ids.
Update	the type update.
ApplyUpdate	interface to database.
GetUpdate	update at r, by t.

CONSTANTS RM, TID, $Update$,
$ApplyUpdate(\_)$,
$GetUpdate(\_, \_)$

$NoRM \triangleq$ CHOOSE $r : r \notin RM$

$NoTID \triangleq$ CHOOSE $t : t \notin TID$

The environments's constant parameters:

PID	the set of all processes.

CONSTANTS PID

$NoPID \triangleq$ CHOOSE $p : p \notin PID$

B.1.2 Specification

MODULE *LogService*

This module specifies the *LogService*. That is, it specifies a centralized log service would look like, and how resource managers should use it.

EXTENDS *Naturals*, *FiniteSets*, *Sequences*,
LogServiceConstants

The environment's variables:
badProc crashed processes
suspect suspicions.

VARIABLES $badProc$, Environments's
$suspect$ Who suspects whom?

The Service's variables:
vHist the history of votes.
tHist the history of committed transactions
LastConcSet LastConcSet concurrent to the last committed.
terminatingAt transactions terminating at rm.
terminatedAt transactions terminated at rm.

VARIABLES $vHist$, $tHist$, $rm2pid$, Log service's
$LastConcSet$

The Resource Managers' variables:
terminatingAt transactions terminating at rm.
terminatedAt transactions terminated at rm.

VARIABLES $terminatingAt$, Resource Manager's
$terminatedAt$,
$pid2rm$

The Transaction Managers's variables:
termReq transactions requested to terminate.
part participants on each transaction.

VARIABLES $termReq$, Transaction Managers'
$part$

Defining some aliases.

$svars \triangleq \langle vHist, tHist, LastConcSet, rm2pid \rangle$
$rvars \triangleq \langle terminatingAt, terminatedAt, pid2rm \rangle$
$tvars \triangleq \langle part, termReq \rangle$

$evars \triangleq \langle badProc, suspect \rangle$
$avars \triangleq \langle svars, rvars, tvars, evars \rangle$

Types

$UNKNOWN \triangleq$ CHOOSE $v : v \notin \{\text{TRUE, FALSE}\}$

$Votes \triangleq [rm : RM,\ tr : TID,\ tset : \text{SUBSET } RM,$
$vt : \{\text{“Commit”, “Abort”}\},\ upd : Update]$

Invariants

Type invariant.

$TypeInvariant \triangleq$

$\wedge\ vHist \in \text{SUBSET } Votes$
$\wedge\ tHist \in Seq(\text{SUBSET } TID)$
$\wedge\ LastConcSet \in \text{SUBSET } TID$
$\wedge\ rm2pid \in [RM \to PID \cup \{NoPID\}]$
$\wedge\ terminatingAt \in [PID \to \text{SUBSET } TID]$
$\wedge\ terminatedAt \in [PID \to \text{SUBSET } TID]$
$\wedge\ pid2rm \in [PID \to RM \cup \{NoRM\}]$
$\wedge\ part \in [TID \to \text{UNION } \{[S \to PID] : S \in \text{SUBSET } RM\}]$
$\wedge\ termReq \in \text{SUBSET } TID$
$\wedge\ badProc \in \text{SUBSET } PID$
$\wedge\ suspect \in [PID \to [PID \to \text{BOOLEAN }]]$

Initial State

$Init \triangleq$

$\wedge\ vHist = \{\}$
$\wedge\ tHist = \langle\rangle$
$\wedge\ LastConcSet = \{\}$
$\wedge\ rm2pid = [r \in RM \mapsto NoPID\ ]$
$\wedge\ terminatingAt = [p \in PID \mapsto \{\}]$
$\wedge\ terminatedAt = [p \in PID \mapsto \{\}]$
$\wedge\ pid2rm = [r \in PID \mapsto NoRM]$
$\wedge\ part = [t \in TID \mapsto [e \in \{\} \mapsto \{\}]]$
$\wedge\ termReq = \{\}$
$\wedge\ badProc = \{\}$
$\wedge\ suspect = [p \in PID \mapsto [q \in PID \mapsto \text{FALSE}]]$

No vote received. No transaction committed,

No RM is incarnated. No transactions terminating at any RM. No transactions terminating at any RM. No RM is incarnated.

No participant in any transaction.

No termination request issued. No bad process.

Operators

Operator $IsInvolved(t, rm)$ checks whether rm is involved in transaction t.

$IsInvolved(t, rm) \triangleq$ IF $\exists\, v \in vHist : v.tr = t \wedge rm \in v.rm$
THEN TRUE

```
                ELSE IF ∃ v ∈ vHist : v.tr = t ∧ v.vt = "Commit"
                     THEN FALSE
                     ELSE UNKNOWN
```

Operator $Updates(rm)$ gets the updates performed by rm in committed transactions.

```
Updates(rm) ≜
  LET UpdateOn(t) ≜ (CHOOSE v ∈ vHist : v.tr = t ∧ v.rm = rm).upd

      upd[i ∈ 0 .. Len(tHist)] ≜
        IF i = 0
        THEN ⟨⟩
        ELSE Append(upd[i − 1], {UpdateOn(t) :
                                   t ∈ {e ∈ tHist[i] :
                                          IsInvolved(rm, e) = TRUE}})

  IN upd[Len(tHist)]
```

Operator $Outcome(h, t)$ gives the termination status of transaction t, considering the history of votes h.

```
Outcome(h, t) ≜ IF ∃ v ∈ h : v.tr = t ∧ v.vt = "Abort"
                THEN "Abort"
                ELSE IF ∃ v ∈ h :
                          ∧ v.tr = t
                          ∧ ∀ p ∈ v.tset :
                              ∃ vv ∈ h : ∧ vv.rm = p
                                         ∧ vv.tr = t
                                         ∧ vv.vt = "Commit"
                     THEN "Commit"
                     ELSE "Undefined"
```

Actions

Environment

This action crashes a good process.

```
Crash ≜ ∧ ∃ p ∈ (PID \ badProc) : badProc′ = badProc ∪ {p}
        ∧ UNCHANGED ⟨suspect⟩
```

This action changes the suspicion status.

```
ChangeSuspicion ≜ ∧ ∃ p ∈ (PID \ badProc), q ∈ PID :
                    ∧ p ≠ q
                    ∧ suspect′ = [suspect EXCEPT ![p][q] = ¬@]
                  ∧ UNCHANGED ⟨badProc⟩
```

EnvActions:
- process crash.

• change suspicions.

$$EnvActions \triangleq \land \lor Crash \\ \lor ChangeSuspicion \\ \land \text{UNCHANGED } \langle svars, rvars, tvars \rangle$$

Transaction Manager

Adds a resource manager to a non-terminated transaction.

The transaction manager will only succeed in the adding a resource manager r to a transaction t if r was not crashed by the time it was contacted, and replied to operation request.

$$AddRM \triangleq \land \exists\, t \in (TID \setminus termReq), \\ r \in \{r \in RM : \land rm2pid[r] \neq NoPID \\ \land rm2pid[r] \notin badProc\} : \\ \land r \notin \text{DOMAIN } part[t] \\ \land part' = [part \text{ EXCEPT } ![t] = \\ [e \in \text{DOMAIN } @ \cup \{r\} \mapsto \text{IF } e \in \text{DOMAIN } @ \\ \text{THEN } @[e] \\ \text{ELSE } rm2pid[r]]] \\ \land \text{UNCHANGED } \langle termReq, rm2pid \rangle$$

t has not tried to terminate yet. r has been incarnated, replied to an operation, and was not a participant yet

Request the termination of a transaction.

The transaction manager can, at any time, try to terminate a transaction it has not tried to terminate before, and that executed some operation.

$$RequestTerm \triangleq \land \exists\, t \in (TID \setminus termReq) : \\ \land part[t] \neq \langle\rangle \\ \land termReq' = termReq \cup \{t\} \\ \land \text{UNCHANGED } \langle part, rm2pid \rangle$$

t has not tried to terminate yet.

The disjunction of Transaction Manager's actions.
• Add a resource manager as a participant.
• Try to terminate a transaction.

$$TMActions \triangleq \land \lor AddRM \\ \lor RequestTerm \\ \land \text{UNCHANGED } \langle svars, rvars, evars \rangle$$

Log Service

Action $Incarnate(pid, rm)$ is executed by process pid to incarnate resource manager rm.

$$Incarnate \triangleq \\ \text{LET } ApplyUpdates(u) \triangleq \\ \text{LET } apply[i \in 1 \,..\, Len(u)] \triangleq \\ \text{IF } i = Len(u) \text{ THEN } ApplyUpdate(u[i]) \\ \text{ELSE } ApplyUpdate(u[i]) \land apply[i+1] \\ \text{IN } \text{IF } u = \langle\rangle \text{ THEN TRUE} \\ \text{ELSE } apply[1]$$

```
IN  ∃ p ∈ PID \ badProc, r ∈ RM :
      ∧ pid2rm[p] = NoRM
      ∧ ∨ rm2pid[r] = NoPID
        ∨ rm2pid[r] ≠ NoPID ∧ suspect[p][rm2pid[r]]
      ∧ rm2pid' = [rm2pid EXCEPT ![r] = p]
      ∧ pid2rm' = [pid2rm EXCEPT ![p] = r]
      ∧ ApplyUpdates(Updates(r))
      ∧ UNCHANGED ⟨terminatingAt, terminatedAt, vHist, tHist, LastConcSet⟩
```

Action $Vote(v)$ adds v to $vHist$, if not there yet.

```
Vote(v) ≜                                                  Voted transactions not
  LET ConcSet(vh) ≜ {t ∈ {v.tr : ∧ v ∈ vh} :               terminated yet.
                     Outcome(vh, t) = "Undefined"}
      Commits ≜ ∧ Outcome(vHist', v.tr) = "Commit"
                ∧ ∨ ∧ v.tr ∈ LastConcSet
                    ∧ tHist' = [tHist EXCEPT ![Len(tHist)] = @ ∪ {v.tr}]
                    ∧ LastConcSet' = LastConcSet \ {v.tr}
                  ∨ ∧ v.tr ∉ LastConcSet
                    ∧ tHist' = Append(tHist, {v.tr})
                    ∧ LastConcSet' = ConcSet(vHist') \ {v.tr}
                ∧ UNCHANGED rm2pid

      Aborts ≜ ∧ Outcome(vHist', v.tr) = "Abort"
               ∧ UNCHANGED ⟨tHist, LastConcSet, rm2pid⟩

  IN  ∧ ¬∃ ov ∈ vHist : ∧ ov.rm = v.rm                     Add v to vHist
                        ∧ ov.tr = v.tr
      ∧ vHist' = vHist ∪ {v}
      ∧ StayUndef ∨ Commits ∨ Aborts
```

Action $Terminate(rm, t)$ is executed by rm to step towards transaction t's termination.

```
VoteForMyself(r, t) ≜
    ∧ t ∈ (termReq
        \ (terminatingAt[rm2pid[r]] ∪ terminatedAt[rm2pid[r]]))
    ∧ terminatingAt' = [terminatingAt EXCEPT ![rm2pid[r]] = @ ∪ {t}]
    ∧ ∨ Vote([rm ↦ r, tr ↦ t, tset ↦ DOMAIN part[t],          Vote
             vt ↦ "Commit", upd ↦ GetUpdate(r, t)])          commit.
      ∨ Vote([rm ↦ r, tr ↦ t, tset ↦ {r}, vt ↦ "Abort", upd ↦ {}])   Vote
    ∧ UNCHANGED ⟨terminatedAt⟩                                abort.

VoteForOthers(r, t) ≜                                      rm tried to termi-
    ∧ t ∈ terminatingAt[rm2pid[r]]                         nate t t is stuck.
    ∧ Outcome(vHist, t) = "Undefined"
    ∧ ∃ s ∈ DOMAIN part[t] :
      ∧ suspect[rm2pid[r]][rm2pid[s]]
      ∧ Vote([rm ↦ s, tr ↦ t, tset ↦ DOMAIN part[t],
```

$vt \mapsto$ "Abort", $upd \mapsto \{\}]$)
$\land$ UNCHANGED $\langle terminatingAt, terminatedAt \rangle$

$Learn(r, t) \triangleq$
$\land\ Outcome(vHist, t) \neq$ "Undefined" Since t has terminated, rm learns it.
$\land\ terminatingAt' = [terminatingAt$ EXCEPT $![rm2pid[r]] = @ \setminus \{t\}]$
$\land\ terminatedAt' = [terminatedAt$ EXCEPT $![rm2pid[r]] = @ \cup \{t\}]$
$\land$ UNCHANGED $\langle svars \rangle$

$Terminate \triangleq$
$\exists\, r \in RM, t \in TID :$ r is a participant of t. the process is alive, and is still the same
$\land\ r \in$ DOMAIN $part[t]$
$\land\ part[t][r] \notin badProc$
$\land\ part[t][r] = rm2pid[r]$
$\land\ \lor\ VoteForMyself(r, t)$
$\lor\ VoteForOthers(r, t)$
$\lor\ Learn(r, t)$
$\land$ UNCHANGED $\langle pid2rm \rangle$

$RMActions \triangleq$
$\land\ \lor\ Incarnate$
$\lor\ Terminate$
$\land$ UNCHANGED $\langle tvars, evars \rangle$

Specification

$Next \triangleq \lor\ RMActions$
$\lor\ TMActions$
$\lor\ EnvActions$

$Spec \triangleq Init \land \Box[Next]_{\langle svars, tvars, rvars, evars \rangle}$

Theorems

The specification is type safe.

THEOREM $Spec \Rightarrow \Box\, TypeInvariant$

AC-Validity If an RM decides to commit a transaction, then all RMs voted to commit the transaction.

$AC\_Validity \triangleq \forall\, t \in TID : Outcome(vHist, t) =$ "Commit"
$\Rightarrow \forall\, r \in$ DOMAIN $part[t] :$
$\exists\, v \in vHist :\ \land\ v.rm = r$
$\land\ v.tr\ = t$
$\land\ v.vote =$ "Commit"

AC-Agreement It is impossible for one RM to commit a transaction and another one to abort the transaction.

$AC\_Agreement \triangleq$ TRUE From the definition of $Outcome$.

AC-Non-Triviality If all RMs vote to commit the transaction and no RM is suspected throughout the execution of the protocol, then the decision is commit.

$$
\begin{array}{l}
AC\_Non\_Triviality \triangleq \forall\, t \in TID : \\
\quad \land \forall\, r,\, s \in part[t] : \Box(\neg suspect[rm2pid[r]][rm2pid2[s]]) \\
\quad \land \forall\, r \in part[t] : \exists\, v \in vHist : \land\, v.rm = r \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \land\, v.t = t \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \land\, v.vt = \text{“Commit”} \\
\quad \Rightarrow \\
\quad \Diamond Outcome(vHist) = \text{“Commit”}
\end{array}
$$

AC-Termination Non-faulty RMs eventually decide.

$$
\begin{array}{l}
AC\_Termination \triangleq \\
\quad \land \forall\, t \in TID : \Diamond\Box(Outcome(vHist,\, t) \in \{\text{“Commit”}, \text{“Abort”}\}) \\
\quad \land \forall\, t \in TID : \forall\, r \in part[t] : \Diamond\Box(t \in terminatedAt[r])
\end{array}
$$

THEOREM $Spec \Rightarrow \land AC\_Validity \land AC\_Non\_Triviality \land AC\_Termination$

B.1.3 Correctness

We want to prove that the log service's specification (LSS) we gave at Section 5.3 satisfies the atomic commitment and the R-Consistency properties, recalled below. We first prove an invariant of the algorithm and then proceed to prove each property individually.

- **AC-Validity** If an RM decides to commit a transaction, then all RMs voted to commit the transaction.
- **AC-Agreement** It is impossible for one RM to commit a transaction and another one to abort the transaction.
- **AC-Non-Triviality** If all RMs vote to commit the transaction and no RM is suspected throughout the execution of the protocol, then the decision is commit.
- **AC-Termination** Non-faulty RMs eventually decide.
- **Durability** Committed data is saved by the system such that, even in the event of a failure and system restart, the data is available in its correct state.

Invariant 1 *At any point in time there is at most one process incarnating any given resource manager.*

PROOF: At the initial state, defined by $Init$, no resource manager is incarnated by any process. Resource managers are incarnated only by executing action $Incarnate$, that replaces the previous resource manager by the new one. Therefore, at most

one process can be incarnating a resource manager at any point in time, and this relation is kept in $rm2pid$. □

Invariant 2 *Given a transaction t, if $Outcome(vHist, t)$ ≠"Undefined" at some point in time, then it will equal $Outcome(vHist, t)$ at any later time.*

PROOF: The variable $vHist$ is only changed in action $Vote$, where a vote v is added to $vHist$ only if there was no other vote for the same transaction $v.tr$ and resource manager $v.rm$ in$vHist$. Therefore, the only change allowed to $vHist$ is the addition of new votes.
At the initial state $vHist = \{\}$ and, by the definition of action $Outcome$, $Outcome(\{\}, t)$ equals "Undefined". From this state, "Abort" and "Commit" votes for t can be added to $vHist$.
By the definition of $Outcome$, if an "Abort" is ever added, then $Outcome(vHist, t)$ will equal "Abort" at any future evaluation. If a "Commit" vote is added for each resource manager involved in t, then $Outcome(vHist, t)$ will equal "Commit" and, because no "Abort" vote for t could be added later, it will equal "Commit" at any future evaluation. □

Proposition 13 (AC-Valitidy) *LSS satisfies the AC-Validity property.*

PROOF: By the specification, the only action that issues "Commit" votes is action $VoteForMyself$. As its name suggests, the action is executed for a resource manager r only to cast its own vote and, therefore, a vote v such that $v.vt$ ="Commit" and $v.rm = r$ can only be casted by r itself.
By specification, $Outcome(vHist, t)$ will equal "Commit" only if a "Commit" vote has been issued for t from all resource managers involved. By the previous paragraph, we know that each resource manager involved in t must have issued its own "Commit" vote for t, and the **AC-Validity** property is true. □

Proposition 14 (AC-Agreement) *LSS satisfies the AC-Validity property.*

PROOF: Suppose that $Outcome(vHist, t)$ evaluated to "Commit" at some point in time for some transaction t. By the initial state it was initially evaluated to "Undefined" and turned to "Commit" at some later state. By the Invariant A.2, it must have turned directly from "Undefined" to "Commit", and will be "Commit" on any future evalution. Therefore, any resource manager either sees "Commit" or "Undefined"; because a resource manager will give t for terminated only if $Outcome(vHist, t)$ ≠"Undefined", all resource managers will see the same termination outcome. Changing "Commit" for "Abort" renders the equivalent result, and therefore the **AC-Agreement** property is true. □

Proposition 15 (AC-Non-Triviality) *LSS satisfies the AC-Non-Triviality property.*

PROOF: By the specification, for any transaction t, a resource manager will only vote "Abort" abort on behalf of another resource manager if it suspects that it is crashed. If there are no suspicions, then all resource manager will vote for themselves. If no resource manager votes "Abort" for itself, then only "Commit" votes will be issued and added to $vHist$. Because, by assumption, all crashes are eventually suspected, the lack of suspicions implies that no resource manager crashed. Therefore, eventually all resource managers involved in t will have their votes added to $vHist$ and, at this point in time, $Outcome(vHist, t)$ will turn from "Undefined" to "Commit", satisfying the **AC-Non-Triviality** property. □

To prove the next property we must assume some kind of fairness on the system; we assume the following: actions that become enabled and remain in such state until executed, are eventually executed. This property is equivalent to the Weak Fairness described in [Lamport, 2002a].

Proposition 16 (AC-Termination) *LSS satisfies the AC-Termination property.*

Assuming that any transaction t that is started will eventually be requested to terminate, resource managers will eventually crash or vote for themselves. As, by assumption, non-faulty resource managers eventually suspect any crashed ones and vote on their behalf if they have issued their own votes, eventually some "Abort" vote or all "Commit" votes for t will be gathered by the log service, t will be terminated, and resource managers that did not crash will learn the transaction's outcome, hence, satisfying the **AC-Termination** property. □

Proposition 17 (Durability) *LSS satisfies the Durability property.*

Let s be the state of some given database. We denote by $s.u$ the state obtained by applying an update u to the database at state s. To prove the the **Durability** property we use the following assumptions regarding this change of states. We call two transactions "non-concurrent" if their termination procedure was executed within non-overlapping periods of time.

Assumption 6 *Applying an update is a deterministic operation. That is, given two databases at states s_1 and s_2 and and update u, $(s_1 = s_2) \Rightarrow (s_1.u = s_2.u)$.*

Assumption 7 *Updates to different data items are commutable. That is, given two updates u_1 and u_2 and a database in some state s, if u_1 and u_2 do not write to the same data item, then $s.u_1.u_2 = s.u_2.u_1$.*

Assumption 8 *Concurrent transactions do not access the same data items. I.e., if two transactions execute their commit procedure in parallel, then they do not read items written by each other.*

PROOF SKETCH: We divide the proof in several steps. First we show that any non-concurrent transactions that committed, who possibly accessed the same data items, are totally ordered in $tHist$, according to their termination order. Then we show that the reincarnation procedure apply the updates of committed transactions in the same order they were committed (the order in $tHist$) and, finally, show that this ensures that a recovered resource manager has the same committed stated as before crashing or being replaced.

1. ASSUME: $(C, \leq_C)$ is the poset where C is the set of committed transactions of some run of the system, $\leq_C$ their commit order, and
   $vHist = (H, \leq_H)$, in that run.
   PROVE: $\forall t_1, t_2 \in C : t_1 \leq_C t_2 \Rightarrow t_1 \leq_H t_2$
   PROOF: Let t_1 and t_2 be two non-concurrent transactions that committed and, without loss of generality, let t_1 be the one that committed first. By the specification, all resource managers involved in t_1 executed the action $Vote$ with "Commit" vote for t_1 before any has voted for t_2.
   By the definition of $Vote$, when the first vote for t_1 was issued, t_1 was added to $ConcSet$. After the last vote, t_1 was removed from $ConcSet$ and added to $vHist$, and $LastConcSet$ was changed to a set not containing t_2. When t_2 receives its first vote to commit, it is added to $ConcSet$, and upon the last vote, it is removed from $ConcSet$ and added to $vHist$. Because t_2 cannot belong to the $LastConcSet$ defined when t_1 was committed, t_2 will belong to a set in $tHist$ different from the one t_1 belongs, by the specification.
   Recalling the meaning of the data-structure $tHist$, if a transaction t_1 belongs to the set $tHist[i]$ for some natural number i, then $t_1 \leq_H t_2$, for any transaction t_2 that belongs to $tHist[j]$ for any $j > i$.
2. ASSUME: r is a resource manager,
   t is a transaction that committed, and
   r was involved in t.
   PROVE: If $Updates(r)$ is evaluated after t committed, then
   $\exists i \in 1..Len(Updates(r)) : Updates(r)[i] = u$,
   where u are the updates executed by r on transaction t.
   PROOF: Because t committed, by the definition of $Vote$, there must be a set in $tHist$ to whom t belongs. By de definition of $Updates$ and $IsInvolved$, r will be identified as participant of t and, by the definition of $UpdateOn$, inside the definition of $Updates$, r's updates on t are in $Updates(r)$.
3. ASSUME: r is a resource manager,
   t_1 and t_2 are non-concurrent transactions,
   t_1 and t_2 committed,
   t_1 committed before t_2,

r was involved in t_1 and t_2,
r's updates on t_1 and t_2 are u_1 and u_2, respectively.
PROVE: $(Updates(r)[i] = u_1) \wedge (Updates(r)[j] = u_2) \Rightarrow i < j$
PROOF: The definition of $Updates$ constructs $Updates(r)$ accessing $tHist$ backwards, but orderly. Because each update is added to $tHist$ before the previous one in the sequence, the final result is that updates are in the same order as their respective transaction. By the step 2, both u_1 and u_2 are in $Updates(r)$ and, by step 1, t_1 is ordered before t_2 in $tHist$, and u_1 appear before u_2 in $Updates(r)$.

4. Q.E.D.
PROOF: By the definition of $ApplyUpdates$, updates are applied sequentially. Consequently, by steps 1,2, and 3, all updates of non-concurrent transactions are applied in the recovering resource manager in the same order they were originally executed. By Assumption 8, concurrent transactions access only different items and are, by Assumption 7, commutable. Finally, by Assumption 8, the recovering resource manager must have the same committed state as in the previous incarnation. □

B.2 Coordinated Implementation

B.2.1 Specification

MODULE *CoordLogService*

This module is specifies the log service's Coordinated Implementation.

EXTENDS *Naturals*, *FiniteSets*, *Sequences*,
LogServiceConstants,
Consensus

CONSTANTS *Coord*

The environment's variables:

badProc	processes that crashed
suspect	process X process suspicions.
msgs	messages sent.

VARIABLES *badProc*,
suspect,
msgs

The Coordinators's variables:

vHistAt	votes received per coordinator.
tHistAt	transactions that committed.
recSet	set of awaiting recovery transactions.
bSet	sets of votes to be proposed.
instances	instance to be used by the coordinator(s).

VARIABLES *vHist*,
tHist,
recSet,
bSet,
instances

The Resource Managers' variables:

terminatingAt	transactions terminating at rm.
terminatedAt	transactions terminated at rm.
pid2rm	PID $\mapsto$ RM.
incarns	the incarnation of each rm.
outcome	local view of Outcome.

VARIABLES *terminatingAt*,
terminatedAt,
*pid*2*rm*,
incarns,
outcome

The Transaction Managers's variables:

termReq	transactions requested to terminate.
part	participants on each transaction, and incarnating process.

VARIABLES $termReq$,
$part$

Defining some aliases.

$svars \triangleq \langle vHist, tHist, recSet, bSet, instances \rangle$
$rvars \triangleq \langle terminatingAt, terminatedAt, pid2rm, incarns, outcome \rangle$
$tvars \triangleq \langle part, termReq \rangle$
$evars \triangleq \langle badProc, suspect \rangle$
$ovars \triangleq \langle decision \rangle$
$avars \triangleq \langle svars, rvars, tvars, evars, ovars, msgs \rangle$

Types

$Incarnating \triangleq$ CHOOSE $p : p \notin (PID \cup \{NoRM\})$

Vote extended with PID

$EVotes \triangleq [rm : RM, tr : TID, tset : \text{SUBSET } RM,$
$vt : \{\text{“Commit”}, \text{“Abort”}\}, upd : Update, pid : PID]$

Votes to incarnate a resource manager.

$IncarnT \triangleq [vt : \{\text{“Incarnate”}\}, pid : PID, rm : RM]$

Types of messages exchanged.

$Msgs \triangleq [type : \{\text{“Recover”}\}, rm : RM, pid : PID] \cup$
$[type : \{\text{“Recovered”}\}, rm : RM, pid : PID,$
$upd : Seq(Update), inc : Nat] \cup$
$[type : \{\text{“Incarnated”}\}, rm : RM, inc : Nat] \cup$
$[type : \{\text{“Vote”}\}, rm : RM, pid : PID, tr : TID,$
$tset : \text{SUBSET } RM, vt : \{\text{“Commit”}, \text{“Abort”}\}, upd : Update] \cup$
$[type : \{\text{“Terminated”}\}, tr : TID, out : \{\text{“Commit”}, \text{“Abort”}\}]$

Invariants

Type invariant.

$TypeInvariant \triangleq$
$\wedge\ vHist \in \text{SUBSET } EVotes$
$\wedge\ tHist \in Seq(TID \cup IncarnT)$
$\wedge\ bSet \in [Coord \to \text{SUBSET } (EVotes \cup IncarnT)]$
$\wedge\ recSet \in \text{SUBSET } IncarnT$
$\wedge\ instances \in Nat$
$\wedge\ terminatingAt \in [PID \to \text{SUBSET } TID]$
$\wedge\ terminatedAt \in [PID \to \text{SUBSET } TID]$
$\wedge\ pid2rm \in [PID \to RM \cup \{Incarnating, NoRM\}]$
$\wedge\ outcome \in [PID \to [TID \to \{\text{“Undefined”}, \text{“Abort”}, \text{“Commit”}\}]]$
$\wedge\ part \in [TID \to \text{UNION } \{[S \to PID] : S \in \text{SUBSET } RM\}]$
$\wedge\ termReq \in \text{SUBSET } TID$
$\wedge\ badProc \in \text{SUBSET } PID$

$\land$ *suspect* $\in [PID \to [PID \to \text{BOOLEAN}\ ]]$
$\land$ *msgs* $\in \text{SUBSET}\ Msgs$
$\land$ *ConsensusTypeInv*

Initial State

Init $\triangleq$

$\land$ *vHist*	$= \{\}$	No vote received, no transaction committed, nor being committed.
$\land$ *tHist*	$= \langle\rangle$	
$\land$ *bSet*	$= [c \in Coord \mapsto \{\}]$	
$\land$ *recSet*	$= [c \in Coord \mapsto \{\}]$	
$\land$ *instances*	$= 0$	No transactions terminating. No transactions terminating. No RM is incarnated.
$\land$ *terminatingAt*	$= [p \in PID \mapsto \{\}]$	
$\land$ *terminatedAt*	$= [p \in PID \mapsto \{\}]$	
$\land$ *pid2rm*	$= [r \in PID \mapsto NoRM]$	
$\land$ *outcome*	$= [p \in PID \mapsto [t \in TID \mapsto \text{“Undefined”}]]$	
$\land$ *incarns*	$= [p \in PID \mapsto 0]$	No transaction started. No termination request issued. No bad process.
$\land$ *part*	$= [t \in TID \mapsto [e \in \{\} \mapsto \{\}]]$	
$\land$ *termReq*	$= \{\}$	
$\land$ *badProc*	$= \{\}$	
$\land$ *suspect*	$= [p \in PID \mapsto [q \in PID \mapsto \text{FALSE}]]$	
$\land$ *msgs*	$= \{\}$	No message sent
$\land$ *ConsensusInit*		

Operators

The *pid* of the process currently incarnating r.

$rm2pid(r) \triangleq$
IF $\exists\, p \in PID,\ i \in 1\,..\,Len(tHist)$:
$\land\ tHist[i] = [vt \mapsto \text{“Incarnate”},\ pid \mapsto p,\ rm \mapsto r]$
$\land \neg\exists\, j \in i+1\,..\,Len(tHist),\ op \in PID$:
$tHist[j] = [vt \mapsto \text{“Incarnate”},\ pid \mapsto op,\ rm \mapsto r]$
THEN CHOOSE $p \in PID$:
$\exists\, i \in 1\,..\,Len(tHist)$:
$\land\ tHist[i] = [vt \mapsto \text{“Incarnate”},\ pid \mapsto p,\ rm \mapsto r]$
$\land \neg\exists\, j \in i+1\,..\,Len(tHist),\ op \in PID$:
$tHist[j] = [vt \mapsto \text{“Incarnate”},\ pid \mapsto op,\ rm \mapsto r]$
ELSE *NoPID*

Operator $Outcome(h,\ t)$ gives the termination status of transaction t, considering the history of votes h.

$Outcome(h,\ t) \triangleq$
IF $\exists\, v \in h : v.tr = t \land v.vt = \text{“Abort”}$
THEN “Abort”
ELSE IF $\exists\, v \in h$:
$\land\ v.tr = t$
$\land \forall\, p\ \in v.tset$:

$\exists\, vv \in h : \wedge\, vv.rm = p$
$\wedge\, vv.tr = t$
$\wedge\, vv.vt =$ "Commit"
THEN "Commit"
ELSE "Undefined"

Actions

Environment

This action crashes a good process.

$Crash \triangleq$
$\wedge\, \exists\, p \in (PID \setminus badProc) : badProc' = badProc \cup \{p\}$
$\wedge$ UNCHANGED $\langle suspect \rangle$

This action changes the suspicion status.

$ChangeSuspicion \triangleq$
$\wedge\, \exists\, p \in (PID \setminus badProc),\ q \in PID :$
$\wedge\, p \neq q$
$\wedge\, suspect' = [suspect$ EXCEPT $![p][q] = \neg @]$
$\wedge$ UNCHANGED $\langle badProc \rangle$

EnvActions:
- process crash.
- change suspicions.

$EnvActions \triangleq \wedge\, Crash \vee ChangeSuspicion$
$\wedge$ UNCHANGED $\langle msgs, svars, tvars, rvars, ovars \rangle$

Transaction Manager

Adds a resource manager to a non-terminated transaction.

The transaction manager will only succeed in the adding a resource manager r to a transaction t if r was not crashed by the time it was contacted, and replied to operation request.

$AddRM \triangleq$
$\wedge\, \exists\, t \in (TID \setminus termReq),$
$r \in \{r \in RM : \wedge\, rm2pid(r) \neq NoPID$
$\wedge\, rm2pid(r) \notin badProc\} :$
$\wedge\, r \notin$ DOMAIN $part[t]$
$\wedge\, part' = [part$ EXCEPT $![t] =$
$[e \in$ DOMAIN $@ \cup \{r\} \mapsto$ IF $e \in$ DOMAIN $@$
THEN $@[e]$
ELSE $rm2pid(r)]]$
$\wedge$ UNCHANGED $\langle termReq \rangle$

t has not tried to terminate yet. r has been incarnated, replied to an operation, and was not a participant yet.

Request the termination of a transaction.

The transaction manager can, at any time, try to terminate a transaction it has not tried to terminate before, and that executed some operation.

$RequestTerm \triangleq \land \exists\, t \in (TID \setminus termReq) :$ t has not tried to terminate yet.
$\quad \land part[t] \neq \langle\rangle$
$\quad \land termReq' = termReq \cup \{t\}$
$\land$ UNCHANGED $\langle part \rangle$

The disjunction of Transaction Manager's actions.
- Add a resource manager as a participant.
- Try to terminate a transaction.

$TMActions \triangleq \land \lor AddRM$
$\quad \lor RequestTerm$
$\land$ UNCHANGED $\langle rvars, evars, svars, msgs, ovars \rangle$

Log Service

Starts the incarnate procedure.

$IncarnateStart(p, r) \triangleq$ p is neither incarnating nor trying and r is not incarnated or its process is supected.
$\quad \land pid2rm[p] = NoRM$
$\quad \land \lor rm2pid(r) = NoPID$
$\quad\quad \lor rm2pid(r) \neq NoPID \land suspect[p][rm2pid(r)]$
$\quad \land pid2rm' = [pid2rm$ EXCEPT $![p] = Incarnating]$
$\quad \land msgs' = msgs \cup \{[type \mapsto$ "Recover", $pid \mapsto p,\ rm \mapsto r]\}$
$\quad \land$ UNCHANGED $\langle incarns \rangle$

Ends the incarnate procedure.

$IncarnateEnd(p, r) \triangleq$
$\quad$ LET $ApplyUpdates(u) \triangleq$
$\quad\quad$ LET $apply[i \in 1 \,..\, Len(u)] \triangleq$
$\quad\quad\quad$ IF $i = Len(u)$ THEN $ApplyUpdate(u[i])$
$\quad\quad\quad\quad\quad$ ELSE $ApplyUpdate(u[i]) \land apply[i+1]$

$\quad\quad$ IN IF $u = \langle\rangle$ THEN TRUE
$\quad\quad\quad\quad$ ELSE $apply[1]$

$\quad$ IN $\land pid2rm[p] = Incarnating$
$\quad\quad \land \exists\, m \in msgs :$
$\quad\quad\quad \land m.type =$ "Recovered"
$\quad\quad\quad \land m.pid = p$
$\quad\quad\quad \land m.rm = r$
$\quad\quad\quad \land pid2rm' = [pid2rm$ EXCEPT $![p] = r]$
$\quad\quad\quad \land incarns' = [incarns$ EXCEPT $![p] = m.inc]$
$\quad\quad\quad \land ApplyUpdates(m.upd)$
$\quad\quad \land$ UNCHANGED $\langle msgs \rangle$

Executed by process p to give up incarnating r, when another process incarnates it.

$Desincarnate(p, r) \triangleq$
$\land\ incarns[p] > 0$
$\land\ \exists\, m \in msgs :$
$\quad\land\ m.type = \text{“Incarnated”}$
$\quad\land\ m.rm = r$
$\quad\land\ m.inc > incarns[p]$
$\land\ incarns' = [incarns \text{ EXCEPT } ![p] = 0]$
$\land\ \text{UNCHANGED } \langle msgs, pid2rm\rangle$

IncarnateStub is a “stub” to the abstract log service *Incarnate* action.

$IncarnateStub \triangleq$
$\land\ \exists\, p \in PID \setminus badProc, r \in RM :$ p is good
$\quad\lor\ IncarnateStart(p, r)$
$\quad\lor\ IncarnateEnd(p, r)$
$\quad\lor\ Desincarnate(p, r)$
$\land\ \text{UNCHANGED } \langle terminatingAt, terminatedAt, outcome\rangle$

VoteForMyself is executed to vote on some transaction.

$VoteForMyself(r, t) \triangleq$
rm has not tried to terminate t.
$\land\ t \in (termReq \setminus (terminatingAt[part[t][r]] \cup terminatedAt[part[t][r]]))$
$\land\ terminatingAt' = [terminatingAt \text{ EXCEPT } ![part[t][r]] = @ \cup \{t\}]$
$\land\ \lor\ msgs' = msgs \cup \{[type \mapsto \text{“Vote”}, pid \mapsto part[t][r], rm \mapsto r,$
$\qquad\qquad tr \mapsto t, tset \mapsto \text{DOMAIN } part[t],$
$\qquad\qquad vt \mapsto \text{“Commit”}, upd \mapsto GetUpdate(r, t)]\}$ Vote commit.
$\quad\lor\ msgs' = msgs \cup \{[type \mapsto \text{“Vote”}, pid \mapsto part[t][r], rm \mapsto r,$
$\qquad\qquad tr \mapsto t, tset \mapsto \{r\}, vt \mapsto \text{“Abort”}, upd \mapsto \{\}]\}$
$\land\ \text{UNCHANGED } \langle terminatedAt, outcome\rangle$ Vote Abort

VoteForOthers is executed to vote for some slow participant.

$VoteForOthers(r, t) \triangleq$ rm tried to terminate t but did not succeed yet.
$\land\ t \in terminatingAt[part[t][r]]$
$\land\ outcome[part[t][r]][t] = \text{“Undefined”}$
$\land\ \neg\exists\, m \in msgs :$
$\quad\land\ m.type = \text{“Terminated”}$
$\quad\land\ m.tr = t$
$\land\ \exists\, s \in \text{DOMAIN } part[t] :$
$\quad\land\ suspect[part[t][r]][part[t][s]]$
$\quad\land\ msgs' = msgs \cup$
$\qquad\{[type \mapsto \text{“Vote”}, pid \mapsto part[t][s], rm \mapsto s, tr \mapsto t,$ Send an
$\qquad\ tset \mapsto \text{DOMAIN } part[t], vt \mapsto \text{“Abort”}, upd \mapsto \{\}]\}$ abort vote.
$\land\ \text{UNCHANGED } \langle terminatingAt, terminatedAt, outcome\rangle$

Learn action is performed when a new transaction has terminated.

$Learn(r, t) \triangleq$ $outcome[t]$ is undefined but t terminated.
$\land\ outcome[part[t][r]][t] = \text{“Undefined”}$
$\land\ \exists\, m \in msgs :$

$\land\ m.type = \text{“Terminated”}$
$\land\ m.tr = t$
$\land\ outcome' = [outcome \text{ EXCEPT } ![part[t][r]][t] = m.out]$
$\land\ terminatingAt' = [terminatingAt \text{ EXCEPT } ![part[t][r]] = @ \setminus \{t\}]$
$\land\ terminatedAt' = [terminatedAt \text{ EXCEPT } ![part[t][r]] = @ \cup \{t\}]$
$\land\ \text{UNCHANGED } \langle msgs \rangle$

Action *TerminateStub* is executed by rm to step towards transaction t's termination.

It is a "stub" to the abstract log service's Terminate action.

$TerminateStub \triangleq$
$\land \exists\, r \in RM, t \in TID :$
$\land\ r \in \text{DOMAIN } part[t]$
$\land\ part[t][r] \notin badProc$
$\land\ pid2rm[part[t][r]] = r$
$\land\ \lor\ VoteForMyself(r, t)$
$\lor\ VoteForOthers(r, t)$
$\lor\ Learn(r, t)$
$\land\ \text{UNCHANGED } \langle incarns, pid2rm \rangle$

r is a participant of t. the process is still alive, and sees itself as the rm. first attempt to terminate t. other attempts Learn that it was decided.

The disjunction of Resource Manager's actions.
- Execute the incarnation procedure.
- Try to terminate a transaction.

$RMActions \triangleq$
$\land\ \lor\ IncarnateStub$
$\lor\ TerminateStub$
$\land\ \text{UNCHANGED } \langle tvars, evars, ovars, svars \rangle$

These two actions implement the handling of incarnation requests.

$IncarnateRequest \triangleq$
$\land \exists\, p \in PID, r \in RM, c \in Coord, m \in msgs :$
$\land\ m.type = \text{“Recover”} \land m.pid = p \land m.rm = r$
$\land \neg\exists\, i \in 1 \,..\, Len(tHist) :$
$tHist[i] = [vt \mapsto \text{“Incarnate”}, pid \mapsto p, rm \mapsto r]$ p is not incarnating.
$\land\ recSet' = [recSet \text{ EXCEPT } ![c] = @ \cup$
$\{[vt \mapsto \text{“Incarnate”}, pid \mapsto p, rm \mapsto r]\}]$
$\land\ bSet' = [bSet \text{ EXCEPT } ![c] = @ \cup$
$\{[vt \mapsto \text{“Incarnate”}, pid \mapsto p, rm \mapsto r]\}]$
$\land\ \text{UNCHANGED } \langle vHist, tHist, instances, ovars, msgs \rangle$

$IncarnateReply \triangleq$
LET $Urm(tinc) \triangleq$
LET $try[i \in 0 \,..\, tinc] \triangleq$
IF $i = 0$

```
                THEN ⟨⟩
                ELSE try[i − 1] ∘ IF ∧ tHist[i] ∈ TID
                                     ∧ ∃ v ∈ vHist :
                                         ∧ v.tr  = tHist[i]
                                         ∧ v.rm = tHist[tinc].rm
                                     THEN ⟨(CHOOSE v ∈ vHist :
                                                ∧ v.tr  = tHist[i]
                                                ∧ v.rm = tHist[tinc].rm).upd⟩
                                     ELSE ⟨⟩
        IN  try[tinc]

    Inc(tinc) ≜ Cardinality({i ∈ 1 .. tinc :
                                ∧ tHist[i] ∉ TID
                                ∧ tHist[i].rm = tHist[tinc].rm})

IN  ∃ tinc ∈ 1 .. Len(tHist), c ∈ Coord :
      ∧ tHist[tinc] ∈ recSet[c]
      ∧ recSet′ = [recSet EXCEPT ![c] = @ \ {tHist[tinc]}]
      ∧ msgs′ = msgs ∪
            {[type ↦ "Recovered", rm ↦ tHist[tinc].rm, upd ↦ Urm(tinc),
              inc ↦ Inc(tinc), pid ↦ tHist[tinc].pid],
             [type ↦ "Incarnated", rm ↦ tHist[tinc].rm, inc ↦ Inc(tinc)]}
      ∧ UNCHANGED ⟨vHist, tHist, bSet, instances, ovars⟩
```

Not reincarn transaction,

is committed before *tinc*, *r* took part in it.

This action handles votes issued by participants.

```
VoteRequest ≜
    ∃ c ∈ Coord, m ∈ msgs :
    ∧ m.type = "Vote"
    ∧ ¬∃ ev  ∈ (bSet[c] ∩ EVotes) ∪ vHist :
       ∧ ev.rm = m.rm
       ∧ ev.tr  = m.tr
       ∧ bSet′  = [bSet EXCEPT ![c] = @ ∪
                    {[pid ↦ m.pid, rm ↦ m.rm, tr ↦ m.tr,
                      tset ↦ m.tset, vt ↦ m.vt, upd ↦ m.upd]}]
    ∧ UNCHANGED ⟨vHist, tHist, recSet, instances, msgs, ovars⟩
```

The next two actions are used in proposing and deciding consensus instances. It is used for handlling votes as well as incarnation changes.

```
CoordPropose ≜
    ∧ ∃ c ∈ Coord :
       ∧ bSet[c] ≠ {}
       ∧ Propose(instances, bSet[c])
    ∧ UNCHANGED ⟨svars, msgs⟩

CoordDecide ≜
    LET D ≜ Decide(instances)
```

```
EVotesInD ≜ {v ∈ D : ∧ v.vt ∈ {"Commit", "Abort"}
                     ∧ ¬∃ ov ∈ vHist : ∧ ov.rm = v.rm
                                       ∧ ov.tr  = v.tr}

V2V(v)    ≜ [rm ↦ v.rm, tr ↦ v.tr, tset ↦ v.tset, pid ↦ v.pid,
             vt  ↦ IF rm2pid(v.rm) = v.pid THEN v.vt ELSE "Abort",
             upd ↦ IF rm2pid(v.rm) = v.pid THEN v.upd ELSE {}]

VotesInD  ≜ {V2V(v) : v ∈ EVotesInD}

Set2Seq(S) ≜
  LET set2seq[SS ∈ SUBSET S] ≜
        IF SS = {} THEN ⟨⟩
                   ELSE LET ss ≜ CHOOSE ss ∈ SS : TRUE
                            IN  Append(set2seq[SS \ {ss}], ss)
  IN  set2seq[S]

newCommitted ≜ {t ∈ TID : ∧ Outcome(vHist, t)  = "Undefined"
                          ∧ Outcome(vHist', t) = "Commit"}

newTermMsgs ≜ {[type ↦ "Terminated", tr ↦ t, out ↦ "Commit"] :
                          t ∈ newCommitted}
              ∪
              {[type ↦ "Terminated", tr ↦ v.tr, out ↦ "Abort"] :
                          v ∈ {vv ∈ VotesInD : vv.vt = "Abort"}}

IncarnationReqInD ≜ {i ∈ D : i.vt = "Incarnate"}

IN  ∧ D ≠ NoProposal
    ∧ vHist' = vHist ∪ VotesInD
    ∧ msgs'  = msgs ∪ newTermMsgs
    ∧ tHist' = tHist ∘ (IF newCommitted ≠ {}
                        THEN Set2Seq(newCommitted)
                        ELSE ⟨⟩)
                     ∘ Set2Seq(IncarnationReqInD)
    ∧ instances' = instances + 1
    ∧ bSet'      = [c ∈ Coord ↦ bSet[c] \ D]
    ∧ UNCHANGED ⟨recSet, ovars⟩
```

The disjunction of Coordinator's actions.
- Process requests to incarnate a resource manager.
- Process votes from resource managers.
- Handle consensus instances.

```
CoordActions ≜
  ∧ ∨ IncarnateRequest ∨ IncarnateReply
    ∨ VoteRequest
    ∨ CoordPropose ∨ CoordDecide
  ∧ UNCHANGED ⟨tvars, evars, rvars⟩
```

Specification

The next-state action, as a disjunction of all possible action.

$Next \triangleq$
$\quad \lor RMActions \lor CoordActions$ Implement $RMActions$
$\quad \lor TMActions$ Implement $TMActions$
$\quad \lor EnvActions$ Implement $EnvActions$

The specification.

$Spec \triangleq Init \land \Box[Next]_{\langle avars \rangle}$

Refinement Mapping

$rm\_pid2rm \triangleq [r \in RM \mapsto \text{IF } pid2rm[p] = \text{“Incarnating”} \text{ THEN } NoRM$
$\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{ELSE } pid2rm[p]]$

$rm\_rm2pid \triangleq [r \in RM \mapsto rm2pid(r)]$

$rm\_LastConcSet \triangleq \{\}$

$rm\_vHist \triangleq \{[f \in \text{DOMAIN } v \setminus \{\text{“pid”}\} \mapsto v.f] : v \in vHist\}$

$rm\_tHist \triangleq \text{LET } Test(e) \triangleq e \notin IncarnT$
$\qquad\qquad\qquad\quad \text{IN } \quad SelectSeq(vHist, Test)$

B.2.2 Implementation Proof

To prove that the Coordinated Log Service (CLS) is, indeed, an implementation of the Log Service's specification (LS) we give a refinement mapping of the CLS's variables to the LS's, and show that the execution of CLS's actions implies the execution of one of LS's actions, or in a stuttering step. For a thorough explanation of refinement mappings the reader is referred to the following works: [Lamport, 1996] and [Abadi and Lamport, 1991].

We substitute every expression of the specification for an overlined expression with the same name, meaning that any variable defined in its scope is replaced by an overlined one, with the same name; these overlined variables witness the implementation of the specification. We prove that these witnesses exist by defining them from the variables in the implementation, i.e., by giving a refinement mapping.

The actual refinement is defined at the end of the specification. Below we simply rename each definition to conform the overlined notation. Variables that are not redefined are the same as in the implementation.

$\overline{pid2rm} \triangleq rm\_pid2rm$

$\overline{rm2pid} \triangleq rm\_rm2pid$

$\overline{LastConcSet} \triangleq rm\_LastConcSet$

$\overline{vHist} \triangleq rm\_vHist$

$\overline{tHist} \triangleq rm\_tHist$

Proposition 18 $Spec \Rightarrow \overline{Spec}$

1. ASSUME: $Init$
   PROVE: $\overline{Init}$
   PROOF: Except for $LastConcSet$ and $rm2pid$, all the variables are initialized in $Init$ exactly as their overlined counterparts in $\overline{Init}$. By the refinement mapping, $\overline{LastConcSet}$ is always the empty set, therefore conforming the initialization in $\overline{Init}$. Finally, by the definition of operator $rm2pid$, $tHist = \langle\rangle$ implies that $rm2pid$ maps from all resource managers to $NoPid$.

2. ASSUME: $Next$
   PROVE: $\overline{Next} \vee \text{UNCHANGED } \langle\overline{svars, tvars, rvars, evars}\rangle$
   2.1. ASSUME: $RMActions$
   PROVE: $\overline{RMActions}$
   2.1.1. ASSUME: $IncarnateStub \wedge \text{UNCHANGED } \langle tvars, evars\rangle$
   PROVE: $\Diamond \wedge \overline{Incarnate} \vee \text{UNCHANGED } \langle svars, rvars\rangle$
   $\wedge \text{UNCHANGED } \langle\overline{tvars, evars}\rangle$

   PROOF SKETCH: We want to show that the execution of $IncarnateStart(p, r)$ for some process p and resource manager r leads to the execution $IncarnateEnd(p, r)$, if p does not crash and its messages are lost. Because $IncarnateEnd(p, r)$ can only be executed if $IncarnateStart(p, r)$ was previously executed and because the pre-conditions of $\overline{Incarnate}$ shared with $IncarnateStart$ do not change until $IncarnateEnd$ is executed, and the conditions of $\overline{Incarnate}$ shared with $IncarnateEnd$ complement the set of $\overline{Incarnate}$ pre and post-conditions already true, the execution of $IncarnateEnd$ implies an $\overline{Incarnate}$ step. If just the start action is performed, then it implies a stuttering step of $\overline{Spec}$.

   PROOF: The $IncarnateStub$ action is a disjunction of actions
   - $IncarnateStart$,
   - $IncarnateEnd$, and
   - $Desincarnate$

   It is clear by the specification that action $IncarnateEnd(p, r)$ cannot execute for a resource manager r and process p before an $IncarnateStart(p, r)$ is executed: $IncarnateEnd(p, r)$ only executes after receiving a message $m = [type \mapsto \langle Recovered\rangle, rm \mapsto r, pid \mapsto p]$, and such message will not be sent by action $IncarnateReply$ before a vote $v = [vt \mapsto$ "`Incarnate`"$, pid \mapsto p, rm \mapsto r]$ is added to $tHist$. Hence, v will only be added to $tHist$ in action $CoordDecide$, after being proposed in action $CoordPropose$. $CoordPropose$

can only propose such value if it belongs to $bSet[c]$, for some coordinator c, what can only happen if a request message for p to incarnate r is received in action $IncarnateRequest$, and such message is only sent by the execution of action $IncarnateStart(p, r)$.
When $IncarnateStart(p, r)$ is executed, it adds a "`Recover`" message with fields $pid = p$ and $rm = r$ to $msgs$; this is the first pre-condition for action $IncarnateRequest(p, r)$ to execute. The second pre-condition is satisfied until action $CoordDecide$ adds the "`Incarnate`" vote for p and r to $tHist$, what can only happen if $IncarnateRequest$ has been executed first, since $IncarnateRequest$ is the only action that creates "`Incarnate`" votes.
When the action $IncarnateRequest$ is executed, it adds an "`Incarnate`" vote for p and r to $bSet[c]$, for some coordinator c, making it not empty. This is the only pre-condition for the $CoordPropose$ action execute for coordinator c, and the action is eventually executed. Since coordinators insist on proposing its $bSet[c]$ until it is empty, and only removes votes from it if they are decided in some instance, c will keep proposing the "`Incarnate`" vote until it is decided or c crashes. Coordinators are deterministic state machines, and can be replicated at will (their state is only based on the outcomes of consensus instances) and, therefore, as long as coordinators can recover after crashes or infinitely many of them are available, some coordinator eventually completes the execution of $IncarnateRequest$ and $IncarnateReply$.
By the consensus problem definition, **C-Progress** ensures that a decision will eventually be reached on each instance (given that the minimum number of acceptors eventually stay up long enough for the instances to finish). If p crashes, then the request for incarnation is simply discarded or is decided but will be followed by another request for the same r.
When an instance containing the vote for p to incarnate r is decided, the vote is added to $tHist$, and action $IncarnateReply$ will be enabled for all coordinators that proposed it. If c crashes before this action is performed, p will be blocked and never execute another action, as if it had crashed. Because the change made by $IncarnateStart(r, p)$ to variable $pid2rm[p]$ does not affect $\overline{pid2rm}$, this would imply that $\langle \overline{svars}, \overline{rvars} \rangle$ did not change.
Once $IncarnateReply$ is performed, p will eventually receive the "`Recovered`" message, unless c crashes, enabling action $IncarnateEnd$. By the definition of Urm, p will receive all the updates performed by previous incarnations of r. By the definition of $ApplyUpdates$, the p will apply all the updates and recover the committed state r had on its previous incarnation. The pre condition of $IncarnateStub$, $IncarnatedStart$, and $IncarnatedEnd$, and the post-conditions of $IncarnateEnd$ imply the pre and post-conditions of $\overline{Incarnate}$.
By the assumption, variables in $\langle \overline{tvars}, \overline{evars} \rangle$ do not change.

2.1.2. ASSUME: $TerminateStub \wedge$ UNCHANGED $\langle tvars, evars, ovars, svars\rangle$
PROVE: $\Diamond \wedge \overline{Terminate} \vee$ UNCHANGED $\langle \overline{svars, rvars}\rangle$
$\wedge$ UNCHANGED $\langle \overline{tvars, evars}\rangle$

PROOF SKETCH: The pre-conditions of action $TerminateStub$ are the same as those of $\overline{Terminate}$. Therefore, it is enough to show that each of $TerminateStub$'s sub-actions, $VoteForMyself$, $VoteForOthers$, $Learn$, and the actions they lead to, imply a step of their equivalent overlined actions.

2.1.2.1. ASSUME: $\overline{VoteForMyself(r, t)}$
PROVE: $\Diamond \wedge \overline{VoteForMyself(r, t)}$
$\vee$ UNCHANGED $\langle \overline{terminatingAt, vHist, tHist}\rangle$
$\wedge$ UNCHANGED $\langle \overline{terminatedAt}\rangle$

Because the pre-condition and the first post-condition of both actions are the same, it is enough to prove that the second post-condition of action $\overline{VoteForMyself(r, t)}$, the addition of a message m to $msgs$, may the execution of $\overline{Vote(v)}$, where m is "`Vote`" message and v is a vote, and the fields vt, upd, tr, $tset$, and rm of m and v are equal, or has no effect on variables $\langle \overline{terminatingAt, vHist, tHist}\rangle$.

If message m is received by some coordinator c, a vote with its contents, therefore equal to v, is added to $bSet[c]$, enabling the action $CoordPropose$. Action $CoordPropose$ will be executed with a proposal containing this vote until it is decided and added to $vHist$ by action $CoordDecide$, where all coordinators can see it ($vHist$ is changed deterministically based on the consensus outcomes, and would be the same for all coordinators if represented independently at each one.), or until c crashes. If no coordinator succeeds in getting the vote decided, then either another vote, resulting from the execution of $VoteForOthers(r, t)$ will be decided, or all resource managers involved in transaction t will have crashed before their "`Vote`" messages are seen by non-faulty coordinators. It is up to the transaction manager, to then vote to abort the transaction; in the case the transaction manager also crashes and no vote for t is ever decided, t is simply forgotten, implying that $\langle \overline{svars, rvars}\rangle$ is kept unchanged.

$CoordDecide$ also appends newly committed transactions to $tHist$: each transaction is added in a different set, as if they were not concurrent, and $\overline{LastConcSet}$ is always empty, ensuring its type invariance and constructing $tHist$ in a way compatible with the specification of $\overline{tHist}$. Because this is the only action to change $vHist$ and $tHist$, the mapping to $\overline{vHist}$ and $\overline{tHist}$ is correct.

$\langle \overline{tvars, evars}\rangle$ are kept since none of actions changes them.

2.1.2.2. ASSUME: $\overline{VoteForOthers(r, t)}$
PROVE: $\Diamond \wedge \overline{VoteForOthers(r, t)} \vee$ UNCHANGED $\langle \overline{vHist, tHist}\rangle$
$\wedge$ UNCHANGED $\langle \overline{terminatedAt, terminatingAt}\rangle$

PROOF: It is true by the same arguments of step 2.1.2.1.

2.1.2.3. ASSUME: $Learn(r, t)$
PROVE: $\wedge \overline{Learn(r, t)} \vee$ UNCHANGED $\langle \overline{terminatedAt, terminatingAt} \rangle$
$\wedge$ UNCHANGED $\langle \overline{svars} \rangle$

PROOF: As action $Learn(r, t)$ has the same post-conditions of $\overline{Learn(r, t)}$, it is enough to show that the pre-conditions of the first imply the pre-conditions of the latter. Since the reception of "`Terminated`" message for transaction t implies that it was sent and since it is only sent by action $CoordDecide$ if the transaction has terminated, the reception of such message implies that the $\overline{Outcome(vHist, t)} \neq$ "`Undefined`".

2.1.2.4. Q.E.D.

2.1.3. Q.E.D.

2.2. $TMActions \Rightarrow \overline{TMActions}$

2.2.1. $AddRM \Rightarrow \overline{AddRM}$

PROOF: Trivially true, since the their definitions are equal.

2.2.2. $RequestTerm \Rightarrow \overline{RequestTerm}$

PROOF: Trivially true, since the their definitions are equal.

2.2.3. UNCHANGED $\langle rvars, evars, svars, msgs, ovars \rangle$
$\Rightarrow$ UNCHANGED $\langle \overline{svars, rvars, evars} \rangle$

PROOF:Clearly true since either the left-hand side of the expression contains all variables in the spec.

2.2.4. Q.E.D.

2.3. $EnvActions \Rightarrow \overline{EnvActions}$

PROOF: Trivially true, since their definitions are equal.

2.4. Q.E.D.

3. Q.E.D.

□

B.3 Uncoordinated Implementation

B.3.1 Specification

MODULE *CoordLogService*

This module is specifies the log service's Coordinated Implementation.

EXTENDS *Naturals*, *FiniteSets*, *Sequences*,
LogServiceConstants,
Consensus

CONSTANTS *Coord*

The environment's variables:

badProc	processes that crashed
suspect	process X process suspicions.
msgs	messages sent.

VARIABLES $badProc$,
$suspect$,
$msgs$

The Coordinators's variables:

vHistAt	votes received per coordinator.
tHistAt	transactions that committed.
recSet	set of awaiting recovery transactions.
bSet	sets of votes to be proposed.
instances	instance to be used by the coordinator(s).

VARIABLES $vHist$,
$tHist$,
$recSet$,
$bSet$,
$instances$

The Resource Managers' variables:

terminatingAt	transactions terminating at rm.
terminatedAt	transactions terminated at rm.
pid2rm	PID $\mapsto$ RM.
incarns	the incarnation of each rm.
outcome	local view of Outcome.

VARIABLES $terminatingAt$,
$terminatedAt$,
$pid2rm$,
$incarns$,
$outcome$

The Transaction Managers's variables:

termReq	transactions requested to terminate.
part	participants on each transaction, and incarnating process.

VARIABLES $termReq$,
$part$

Defining some aliases.

$svars \triangleq \langle vHist, tHist, recSet, bSet, instances \rangle$
$rvars \triangleq \langle terminatingAt, terminatedAt, pid2rm, incarns, outcome \rangle$
$tvars \triangleq \langle part, termReq \rangle$
$evars \triangleq \langle badProc, suspect \rangle$
$ovars \triangleq \langle decision \rangle$
$avars \triangleq \langle svars, rvars, tvars, evars, ovars, msgs \rangle$

Types

$Incarnating \triangleq \text{CHOOSE } p : p \notin (PID \cup \{NoRM\})$

Vote extended with PID

$EVotes \triangleq [rm : RM,\ tr : TID,\ tset : \text{SUBSET } RM,$
$\quad vt : \{\text{“Commit”}, \text{“Abort”}\},\ upd : Update,\ pid : PID]$

Votes to incarnate a resource manager.

$IncarnT \triangleq [vt : \{\text{“Incarnate”}\},\ pid : PID,\ rm : RM]$

Types of messages exchanged.

$Msgs \triangleq [type : \{\text{“Recover”}\},\ rm : RM,\ pid : PID] \cup$
$\quad [type : \{\text{“Recovered”}\},\ rm : RM,\ pid : PID,$
$\quad upd : Seq(Update),\ inc : Nat] \cup$
$\quad [type : \{\text{“Incarnated”}\},\ rm : RM,\ inc : Nat] \cup$
$\quad [type : \{\text{“Vote”}\},\ rm : RM,\ pid : PID,\ tr : TID,$
$\quad tset : \text{SUBSET } RM,\ vt : \{\text{“Commit”}, \text{“Abort”}\},\ upd : Update] \cup$
$\quad [type : \{\text{“Terminated”}\},\ tr : TID,\ out : \{\text{“Commit”}, \text{“Abort”}\}]$

Invariants

Type invariant.

$TypeInvariant \triangleq$
$\quad \wedge\ vHist \in \text{SUBSET } EVotes$
$\quad \wedge\ tHist \in Seq(TID \cup IncarnT)$
$\quad \wedge\ bSet \in [Coord \to \text{SUBSET } (EVotes \cup IncarnT)]$
$\quad \wedge\ recSet \in \text{SUBSET } IncarnT$
$\quad \wedge\ instances \in Nat$
$\quad \wedge\ terminatingAt \in [PID \to \text{SUBSET } TID]$
$\quad \wedge\ terminatedAt \in [PID \to \text{SUBSET } TID]$
$\quad \wedge\ pid2rm \in [PID \to RM \cup \{Incarnating, NoRM\}]$
$\quad \wedge\ outcome \in [PID \to [TID \to \{\text{“Undefined”}, \text{“Abort”}, \text{“Commit”}\}]]$
$\quad \wedge\ part \in [TID \to \text{UNION } \{[S \to PID] : S \in \text{SUBSET } RM\}]$
$\quad \wedge\ termReq \in \text{SUBSET } TID$
$\quad \wedge\ badProc \in \text{SUBSET } PID$
$\quad \wedge\ suspect \in [PID \to [PID \to \text{BOOLEAN }]]$
$\quad \wedge\ msgs \in \text{SUBSET } Msgs$
$\quad \wedge\ ConsensusTypeInv$

Initial State

$Init \triangleq$
$\quad \wedge\ vHist = \{\}$
$\quad \wedge\ tHist = \langle\rangle$
$\quad \wedge\ bSet = [c \in Coord \mapsto \{\}]$
$\quad \wedge\ recSet = [c \in Coord \mapsto \{\}]$

No vote received, no transaction committed, nor being committed.

$\land\ instances = 0$
$\land\ terminatingAt = [p \in PID \mapsto \{\}]$
$\land\ terminatedAt = [p \in PID \mapsto \{\}]$
$\land\ pid2rm = [r \in PID \mapsto NoRM]$
$\land\ outcome = [p \in PID \mapsto [t \in TID \mapsto \text{“Undefined”}]]$
$\land\ incarns = [p \in PID \mapsto 0]$
$\land\ part = [t \in TID \mapsto [e \in \{\} \mapsto \{\}]]$
$\land\ termReq = \{\}$
$\land\ badProc = \{\}$
$\land\ suspect = [p \in PID \mapsto [q \in PID \mapsto \text{FALSE}]]$
$\land\ msgs = \{\}$
$\land\ ConsensusInit$

No transactions terminating. No transactions terminating. No RM is incarnated.

No transaction started. No termination request issued. No bad process.

No message sent

Operators

The pid of the process currently incarnating r.

$$
\begin{array}{l}
rm2pid(r) \triangleq \\
\quad \text{IF } \exists\, p \in PID,\, i \in 1\,..\,Len(tHist) : \\
\qquad \land\ tHist[i] = [vt \mapsto \text{“Incarnate”},\ pid \mapsto p,\ rm \mapsto r] \\
\qquad \land\ \neg\exists\, j \in i+1\,..\,Len(tHist),\, op \in PID : \\
\qquad\quad tHist[j] = [vt \mapsto \text{“Incarnate”},\ pid \mapsto op,\ rm \mapsto r] \\
\quad \text{THEN CHOOSE } p \in PID : \\
\qquad\quad \exists\, i \in 1\,..\,Len(tHist) : \\
\qquad\qquad \land\ tHist[i] = [vt \mapsto \text{“Incarnate”},\ pid \mapsto p,\ rm \mapsto r] \\
\qquad\qquad \land\ \neg\exists\, j \in i+1\,..\,Len(tHist),\, op \in PID : \\
\qquad\qquad\quad tHist[j] = [vt \mapsto \text{“Incarnate”},\ pid \mapsto op,\ rm \mapsto r] \\
\quad \text{ELSE } NoPID
\end{array}
$$

Operator $Outcome(h,\, t)$ gives the termination status of transaction t, considering the history of votes h.

$$
\begin{array}{l}
Outcome(h,\, t) \triangleq \\
\quad \text{IF } \exists\, v \in h : v.tr = t \land v.vt = \text{“Abort”} \\
\quad \text{THEN “Abort”} \\
\quad \text{ELSE IF } \exists\, v \in h : \\
\qquad\quad \land\ v.tr = t \\
\qquad\quad \land\ \forall\, p \in v.tset : \\
\qquad\qquad \exists\, vv \in h : \land\ vv.rm = p \\
\qquad\qquad\qquad\qquad\ \land\ vv.tr = t \\
\qquad\qquad\qquad\qquad\ \land\ vv.vt = \text{“Commit”} \\
\qquad\quad \text{THEN “Commit”} \\
\qquad\quad \text{ELSE “Undefined”}
\end{array}
$$

Actions

Environment

This action crashes a good process.

$Crash \triangleq$
$\quad \land \exists\, p \in (PID \setminus badProc) : badProc' = badProc \cup \{p\}$
$\quad \land$ UNCHANGED $\langle suspect \rangle$

This action changes the suspicion status.

$ChangeSuspicion \triangleq$
$\quad \land \exists\, p \in (PID \setminus badProc),\ q \in PID :$
$\quad\quad \land\ p \neq q$
$\quad\quad \land\ suspect' = [suspect$ EXCEPT $![p][q] = \neg @]$
$\quad \land$ UNCHANGED $\langle badProc \rangle$

EnvActions:
- process crash.
- change suspicions.

$EnvActions \triangleq \land Crash \lor ChangeSuspicion$
$\quad\quad\quad\quad \land$ UNCHANGED $\langle msgs, svars, tvars, rvars, ovars \rangle$

Transaction Manager

Adds a resource manager to a non-terminated transaction.

The transaction manager will only succeed in the adding a resource manager r to a transaction t if r was not crashed by the time it was contacted, and replied to operation request.

$AddRM \triangleq$
$\quad \land \exists\, t \in (TID \setminus termReq),$
$\quad\quad r \in \{r \in RM : \land\ rm2pid(r) \neq NoPID$
$\quad\quad\quad\quad\quad\quad\ \land\ rm2pid(r) \notin badProc\} :$
$\quad\quad \land\ r \notin$ DOMAIN $part[t]$
$\quad\quad \land\ part' = [part$ EXCEPT $![t] =$
$\quad\quad\quad\quad [e \in$ DOMAIN $@ \cup \{r\} \mapsto$ IF $e \in$ DOMAIN $@$
$\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad$ THEN $@[e]$
$\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad$ ELSE $rm2pid(r)]]$
$\quad \land$ UNCHANGED $\langle termReq \rangle$

t has not tried to terminate yet.
r has been incarnated, replied to an operation, and was not a participant yet.

Request the termination of a transaction.

The transaction manager can, at any time, try to terminate a transaction it has not tried to terminate before, and that executed some operation.

$RequestTerm \triangleq \land \exists\, t \in (TID \setminus termReq) :$
$\quad\quad\quad\quad\quad\quad\quad\quad\ \land\ part[t] \neq \langle\rangle$
$\quad\quad\quad\quad\quad\quad\quad\quad\ \land\ termReq' = termReq \cup \{t\}$
$\quad\quad\quad\quad\quad\quad\quad\ \land$ UNCHANGED $\langle part \rangle$

t has not tried to terminate yet.

The disjunction of Transaction Manager's actions.
- Add a resource manager as a participant.
- Try to terminate a transaction.

$TMActions \triangleq \land \lor AddRM$

```
        ∨ RequestTerm
    ∧ UNCHANGED ⟨rvars, evars, svars, msgs, ovars⟩
```

Log Service

Starts the incarnate procedure.

```
IncarnateStart(p, r) ≜
    ∧ pid2rm[p] = NoRM
    ∧ ∨ rm2pid(r) = NoPID
      ∨ rm2pid(r) ≠ NoPID ∧ suspect[p][rm2pid(r)]
    ∧ pid2rm' = [pid2rm EXCEPT ![p] = Incarnating]
    ∧ msgs' = msgs ∪ {[type ↦ "Recover", pid ↦ p, rm ↦ r]}
    ∧ UNCHANGED ⟨incarns⟩
```

p is neither incarnating nor trying and r is not incarnated or its process is supected.

Ends the incarnate procedure.

```
IncarnateEnd(p, r) ≜
  LET ApplyUpdates(u) ≜
      LET apply[i ∈ 1 .. Len(u)] ≜
          IF i  = Len(u) THEN ApplyUpdate(u[i])
                         ELSE ApplyUpdate(u[i]) ∧ apply[i + 1]

      IN  IF u = ⟨⟩ THEN TRUE
                    ELSE apply[1]

  IN  ∧ pid2rm[p] = Incarnating
      ∧ ∃ m ∈ msgs :
          ∧ m.type = "Recovered"
          ∧ m.pid = p
          ∧ m.rm = r
          ∧ pid2rm' = [pid2rm EXCEPT ![p] = r]
          ∧ incarns' = [incarns EXCEPT ![p] = m.inc]
          ∧ ApplyUpdates(m.upd)
      ∧ UNCHANGED ⟨msgs⟩
```

Executed by process p to give up incarnating r, when another process incarnates it.

```
Desincarnate(p, r) ≜
    ∧ incarns[p] > 0
    ∧ ∃ m ∈ msgs :
        ∧ m.type = "Incarnated"
        ∧ m.rm = r
        ∧ m.inc > incarns[p]
    ∧ incarns' = [incarns EXCEPT ![p] = 0]
    ∧ UNCHANGED ⟨msgs, pid2rm⟩
```

IncarnateStub is a "stub" to the abstract log service *Incarnate* action.

```
IncarnateStub ≜
    ∧ ∃ p  ∈ PID \ badProc, r ∈ RM :        p is good
        ∨ IncarnateStart(p, r)
```

$\lor\ IncarnateEnd(p,\ r)$
$\lor\ Desincarnate(p,\ r)$
$\land$ UNCHANGED $\langle terminatingAt,\ terminatedAt,\ outcome\rangle$

VoteForMyself is executed to vote on some transaction.

$VoteForMyself(r,\ t) \triangleq$
rm has not tried to terminate t.
$\land\ t \in (termReq \setminus (terminatingAt[part[t][r]] \cup terminatedAt[part[t][r]]))$
$\land\ terminatingAt' = [terminatingAt$ EXCEPT $![part[t][r]] = @ \cup \{t\}]$
$\land\ \lor\ msgs' = msgs \cup \{[type \mapsto$ "Vote", $pid \mapsto part[t][r],\ rm \mapsto r,$
$tr \mapsto t,\ tset \mapsto$ DOMAIN $part[t],$
$vt \mapsto$ "Commit", $upd \mapsto GetUpdate(r,\ t)]\}$ Vote commit.
$\lor\ msgs' = msgs \cup \{[type \mapsto$ "Vote", $pid \mapsto part[t][r],\ rm \mapsto r,$
$tr \mapsto t,\ tset \mapsto \{r\},\ vt \mapsto$ "Abort", $upd \mapsto \{\}]\}$ Vote *Abort*.
$\land$ UNCHANGED $\langle terminatedAt,\ outcome\rangle$

VoteForOthers is executed to vote for some slow participant.

$VoteForOthers(r,\ t) \triangleq$ rm tried to terminate t but did not succeed yet.
$\land\ t \in terminatingAt[part[t][r]]$
$\land\ outcome[part[t][r]][t] =$ "Undefined"
$\land\ \neg\exists\ m \in msgs :$
$\land\ m.type =$ "Terminated"
$\land\ m.tr = t$
$\land\ \exists\ s \in$ DOMAIN $part[t] :$
$\land\ suspect[part[t][r]][part[t][s]]$
$\land\ msgs' = msgs \cup$
$\{[type \mapsto$ "Vote", $pid \mapsto part[t][s],\ rm \mapsto s,\ tr \mapsto t,$ Send an
$tset \mapsto$ DOMAIN $part[t],\ vt \mapsto$ "Abort", $upd \mapsto \{\}]\}$ abort vote.
$\land$ UNCHANGED $\langle terminatingAt,\ terminatedAt,\ outcome\rangle$

Learn action is performed when a new transaction has terminated.

$Learn(r,\ t) \triangleq$ $outcome[t]$ is undefined but t terminated.
$\land\ outcome[part[t][r]][t] =$ "Undefined"
$\land\ \exists\ m \in msgs :$
$\land\ m.type =$ "Terminated"
$\land\ m.tr = t$
$\land\ outcome' = [outcome$ EXCEPT $![part[t][r]][t] = m.out]$
$\land\ terminatingAt' = [terminatingAt$ EXCEPT $![part[t][r]] = @ \setminus \{t\}]$
$\land\ terminatedAt' = [terminatedAt$ EXCEPT $![part[t][r]] = @ \cup \{t\}]$
$\land$ UNCHANGED $\langle msgs\rangle$

Action *TerminateStub* is executed by rm to step towards transaction t's termination.
It is a "stub" to the abstract log service's Terminate action.

$TerminateStub \triangleq$

```
∧ ∃ r ∈ RM, t ∈ TID :
    ∧ r ∈ DOMAIN part[t]
    ∧ part[t][r] ∉ badProc
    ∧ pid2rm[part[t][r]] = r
    ∧ ∨ VoteForMyself(r, t)
      ∨ VoteForOthers(r, t)
      ∨ Learn(r, t)
∧ UNCHANGED ⟨incarns, pid2rm⟩
```

r is a participant of t. the process is still alive, and sees itself as the rm. first attempt to terminate t. other attempts Learn that it was decided.

The disjunction of Resource Manager's actions.
- Execute the incarnation procedure.
- Try to terminate a transaction.

```
RMActions ≜
    ∧ ∨ IncarnateStub
      ∨ TerminateStub
    ∧ UNCHANGED ⟨tvars, evars, ovars, svars⟩
```

These two actions implement the handling of incarnation requests.

```
IncarnateRequest ≜
    ∧ ∃ p ∈ PID, r ∈ RM, c ∈ Coord, m ∈ msgs :
        ∧ m.type = "Recover" ∧ m.pid = p ∧ m.rm = r
        ∧ ¬∃ i ∈ 1 .. Len(tHist) :
            tHist[i] = [vt ↦ "Incarnate", pid ↦ p, rm ↦ r]
        ∧ recSet' = [recSet EXCEPT ![c] = @ ∪
                      {[vt ↦ "Incarnate", pid ↦ p, rm ↦ r]}]
        ∧ bSet'   = [bSet   EXCEPT ![c] = @ ∪
                      {[vt ↦ "Incarnate", pid ↦ p, rm ↦ r]}]
    ∧ UNCHANGED ⟨vHist, tHist, instances, ovars, msgs⟩
```

p is not incarnating.

```
IncarnateReply ≜
    LET Urm(tinc) ≜
        LET try[i ∈ 0 .. tinc] ≜
            IF i = 0
            THEN ⟨⟩
            ELSE try[i − 1] ∘ IF ∧ tHist[i] ∈ TID
                                 ∧ ∃ v ∈ vHist :
                                     ∧ v.tr  = tHist[i]
                                     ∧ v.rm = tHist[tinc].rm
                              THEN ⟨(CHOOSE v ∈ vHist :
                                       ∧ v.tr  = tHist[i]
                                       ∧ v.rm = tHist[tinc].rm).upd⟩
                              ELSE ⟨⟩
        IN  try[tinc]

      Inc(tinc) ≜ Cardinality({i ∈ 1 .. tinc :
                                 ∧ tHist[i] ∉ TID
                                 ∧ tHist[i].rm = tHist[tinc].rm})
```

Not reincarn transaction, is committed before *tinc*, *r* took part in it.

```
    IN  ∃ tinc ∈ 1 .. Len(tHist), c ∈ Coord :
          ∧ tHist[tinc] ∈ recSet[c]
          ∧ recSet' = [recSet EXCEPT ![c] = @ \ {tHist[tinc]}]
          ∧ msgs' = msgs ∪
                {[type ↦ "Recovered", rm ↦ tHist[tinc].rm, upd ↦ Urm(tinc),
                  inc ↦ Inc(tinc), pid ↦ tHist[tinc].pid],
                 [type ↦ "Incarnated", rm ↦ tHist[tinc].rm, inc ↦ Inc(tinc)]}
          ∧ UNCHANGED ⟨vHist, tHist, bSet, instances, ovars⟩
```

This action handles votes issued by participants.

```
VoteRequest ≜
    ∃ c ∈ Coord, m ∈ msgs :
    ∧ m.type = "Vote"
    ∧ ¬∃ ev  ∈ (bSet[c] ∩ EVotes) ∪ vHist :
       ∧ ev.rm = m.rm
       ∧ ev.tr  = m.tr
       ∧ bSet'  = [bSet EXCEPT ![c] = @ ∪
                    {[pid ↦ m.pid, rm ↦ m.rm, tr ↦ m.tr,
                      tset ↦ m.tset, vt ↦ m.vt, upd ↦ m.upd]}]
    ∧ UNCHANGED ⟨vHist, tHist, recSet, instances, msgs, ovars⟩
```

The next two actions are used in proposing and deciding consensus instances. It is used for handlling votes as well as incarnation changes.

```
CoordPropose ≜
    ∧ ∃ c ∈ Coord :
       ∧ bSet[c] ≠ {}
       ∧ Propose(instances, bSet[c])
    ∧ UNCHANGED ⟨svars, msgs⟩


CoordDecide ≜
    LET D ≜ Decide(instances)

        EVotesInD ≜ {v ∈ D : ∧ v.vt ∈ {"Commit", "Abort"}
                             ∧ ¬∃ ov ∈ vHist : ∧ ov.rm = v.rm
                                               ∧ ov.tr  = v.tr}

        V2V(v)    ≜ [rm ↦ v.rm, tr ↦ v.tr, tset ↦ v.tset, pid ↦ v.pid,
                     vt  ↦ IF rm2pid(v.rm) = v.pid THEN v.vt ELSE "Abort",
                     upd ↦ IF rm2pid(v.rm) = v.pid THEN v.upd ELSE {}]

        VotesInD  ≜ {V2V(v) : v ∈ EVotesInD}

        Set2Seq(S) ≜
            LET set2seq[SS ∈ SUBSET S] ≜
                IF SS = {} THEN ⟨⟩
                           ELSE LET ss ≜ CHOOSE ss ∈ SS : TRUE
                                IN  Append(set2seq[SS \ {ss}], ss)
            IN  set2seq[S]
```

$$newCommitted \triangleq \{t \in TID : \land Outcome(vHist, t) = \text{“Undefined”} \\ \land Outcome(vHist', t) = \text{“Commit”}\}$$

$$newTermMsgs \triangleq \{[type \mapsto \text{“Terminated”}, tr \mapsto t, out \mapsto \text{“Commit”}] : \\ t \in newCommitted\} \\ \cup \\ \{[type \mapsto \text{“Terminated”}, tr \mapsto v.tr, out \mapsto \text{“Abort”}] : \\ v \in \{vv \in VotesInD : vv.vt = \text{“Abort”}\}\}$$

$$IncarnationReqInD \triangleq \{i \in D : i.vt = \text{“Incarnate”}\}$$

$$\begin{array}{ll} \text{IN} & \land D \neq NoProposal \\ & \land vHist' = vHist \cup VotesInD \\ & \land msgs' = msgs \cup newTermMsgs \\ & \land tHist' = tHist \circ (\text{IF } newCommitted \neq \{\} \\ & \qquad\qquad \text{THEN } Set2Seq(newCommitted) \\ & \qquad\qquad \text{ELSE } \langle\rangle) \\ & \qquad \circ Set2Seq(IncarnationReqInD) \\ & \land instances' = instances + 1 \\ & \land bSet' = [c \in Coord \mapsto bSet[c] \setminus D] \\ & \land \text{UNCHANGED } \langle recSet, ovars\rangle \end{array}$$

The disjunction of Coordinator's actions.
- Process requests to incarnate a resource manager.
- Process votes from resource managers.
- Handle consensus instances.

$$CoordActions \triangleq \\ \begin{array}{ll} & \land \lor IncarnateRequest \lor IncarnateReply \\ & \quad \lor VoteRequest \\ & \quad \lor CoordPropose \lor CoordDecide \\ & \land \text{UNCHANGED } \langle tvars, evars, rvars\rangle \end{array}$$

Specification

The next-state action, as a disjunction of all possible action.

$$Next \triangleq \\ \begin{array}{ll} \lor RMActions \lor CoordActions & \text{Implement } RMActions \\ \lor TMActions & \text{Implement } TMActions \\ \lor EnvActions & \text{Implement } EnvActions \end{array}$$

The specification.

$$Spec \triangleq Init \land \Box[Next]_{\langle avars\rangle}$$

Refinement Mapping

$$rm\_pid2rm \triangleq [r \in RM \mapsto \text{IF } pid2rm[p] = \text{“Incarnating”} \text{ THEN } NoRM$$

$$\text{ELSE } pid2rm[p]]$$

$$rm\_rm2pid \triangleq [r \in RM \mapsto rm2pid(r)]$$

$$rm\_LastConcSet \triangleq \{\}$$

$$rm\_vHist \triangleq \{[f \in \text{DOMAIN } v \setminus \{\text{“pid”}\} \mapsto v.f] : v \in vHist\}$$

$$rm\_tHist \triangleq \text{LET } Test(e) \triangleq e \notin IncarnT \text{ IN } SelectSeq(vHist, Test)$$

B.3.2 Implementation Proof

To prove that the Coordinated Log Service (CLS) is, indeed, an implementation of the Log Service's specification (LS) we give a refinement mapping of the CLS's variables to the LS's, and show that the execution of CLS's actions implies the execution of one of LS's actions, or in a stuttering step. For a thorough explanation of refinement mappings the reader is referred to the following works: [Lamport, 1996] and [Abadi and Lamport, 1991].

We substitute every expression of the specification for an overlined expression with the same name, meaning that any variable defined in its scope is replaced by an overlined one, with the same name; these overlined variables witness the implementation of the specification. We prove that these witnesses exist by defining them from the variables in the implementation, i.e., by giving a refinement mapping.

The actual refinement is defined at the end of the specification. Below we simply rename each definition to conform the overlined notation. Variables that are not redefined are the same as in the implementation.

$$\overline{pid2rm} \triangleq rm\_pid2rm$$

$$\overline{rm2pid} \triangleq rm\_rm2pid$$

$$\overline{LastConcSet} \triangleq rm\_LastConcSet$$

$$\overline{vHist} \triangleq rm\_vHist$$

$$\overline{tHist} \triangleq rm\_tHist$$

Proposition 19 $Spec \Rightarrow \overline{Spec}$

1. ASSUME: $Init$
   PROVE: $\overline{Init}$

PROOF: Except for $LastConcSet$ and $rm2pid$, all the variables are initialized in $Init$ exactly as their overlined counterparts in $\overline{Init}$. By the refinement mapping, $\overline{LastConcSet}$ is always the empty set, therefore conforming the initialization in $\overline{Init}$. Finally, by the definition of operator $rm2pid$, $tHist = \langle\rangle$ implies that $rm2pid$ maps from all resource managers to $NoPid$.

2. ASSUME: $Next$
   PROVE: $\overline{Next} \vee$ UNCHANGED $\langle\overline{svars, tvars, rvars, evars}\rangle$
   2.1. ASSUME: $RMActions$
   PROVE: $\overline{RMActions}$
   2.1.1. ASSUME: $IncarnateStub \wedge$ UNCHANGED $\langle tvars, evars\rangle$
   PROVE: $\Diamond \wedge \overline{Incarnate} \vee$ UNCHANGED $\langle svars, rvars\rangle$
   $\wedge$ UNCHANGED $\langle\overline{tvars, evars}\rangle$

PROOF SKETCH: We want to show that the execution of $IncarnateStart(p, r)$ for some process p and resource manager r leads to the execution $IncarnateEnd(p, r)$, if p does not crash and its messages are lost. Because $IncarnateEnd(p, r)$ can only be executed if $IncarnateStart(p, r)$ was previously executed and because the pre-conditions of $\overline{Incarnate}$ shared with $IncarnateStart$ do not change until $IncarnateEnd$ is executed, and the conditions of $\overline{Incarnate}$ shared with $IncarnateEnd$ complement the set of $\overline{Incarnate}$ pre and post-conditions already true, the execution of $IncarnateEnd$ implies an $\overline{Incarnate}$ step. If just the start action is performed, then it implies a stuttering step of $\overline{Spec}$.

PROOF: The $IncarnateStub$ action is a disjunction of actions

- $IncarnateStart$,
- $IncarnateEnd$, and
- $Desincarnate$

It is clear by the specification that action $IncarnateEnd(p, r)$ cannot execute for a resource manager r and process p before an $IncarnateStart(p, r)$ is executed: $IncarnateEnd(p, r)$ only executes after receiving a message $m = [type \mapsto \langle Recovered\rangle, rm \mapsto r, pid \mapsto p]$, and such message will not be sent by action $IncarnateReply$ before a vote $v = [vt \mapsto$ "`Incarnate`"$, pid \mapsto p, rm \mapsto r]$ is added to $tHist$. Hence, v will only be added to $tHist$ in action $CoordDecide$, after being proposed in action $CoordPropose$. $CoordPropose$ can only propose such value if it belongs to $bSet[c]$, for some coordinator c, what can only happen if a request message for p to incarnate r is received in action $IncarnateRequest$, and such message is only sent by the execution of action $IncarnateStart(p, r)$.

When $IncarnateStart(p, r)$ is executed, it adds a "`Recover`" message with fields $pid = p$ and $rm = r$ to $msgs$; this is the first pre-condition for action $IncarnateRequest(p, r)$ to execute. The second pre-condition is satisfied until action $CoordDecide$ adds the "`Incarnate`" vote for p and r to $tHist$, what can only happen if $IncarnateRequest$ has been executed first, since $IncarnateRequest$ is the only action that creates "`Incarnate`" votes.

When the action $IncarnateRequest$ is executed, it adds an "`Incarnate`" vote for p and r to $bSet[c]$, for some coordinator c, making it not empty. This is the only pre-condition for the $CoordPropose$ action execute for coordinator c, and the action is eventually executed. Since coordinators insist on proposing its $bSet[c]$ until it is empty, and only removes votes from it if they are decided in some instance, c will keep proposing the "`Incarnate`" vote until it is decided or c crashes. Coordinators are deterministic state machines, and can be replicated at will (their state is only based on the outcomes of consensus instances) and, therefore, as long as coordinators can recover after crashes or infinitely many of them are available, some coordinator eventually completes the execution of $IncarnateRequest$ and $IncarnateReply$.

By the consensus problem definition, **C-Progress** ensures that a decision will eventually be reached on each instance (given that the minimum number of acceptors eventually stay up long enough for the instances to finish). If p crashes, then the request for incarnation is simply discarded or is decided but will be followed by another request for the same r.

When an instance containing the vote for p to incarnate r is decided, the vote is added to $tHist$, and action $IncarnateReply$ will be enabled for all coordinators that proposed it. If c crashes before this action is performed, p will be blocked and never execute another action, as if it had crashed. Because the change made by $IncarnateStart(r, p)$ to variable $pid2rm[p]$ does not affect $\overline{pid2rm}$, this would imply that $\langle \overline{svars, rvars} \rangle$ did not change.

Once $IncarnateReply$ is performed, p will eventually receive the "`Recovered`" message, unless c crashes, enabling action $IncarnateEnd$. By the definition of Urm, p will receive all the updates performed by previous incarnations of r. By the definition of $ApplyUpdates$, the p will apply all the updates and recover the committed state r had on its previous incarnation. The pre condition of $IncarnateStub$, $IncarnatedStart$, and $IncarnatedEnd$, and the post-conditions of $IncarnateEnd$ imply the pre and post-conditions of $\overline{Incarnate}$.

By the assumption, variables in $\langle \overline{tvars, evars} \rangle$ do not change.

2.1.2. ASSUME: $TerminateStub \land \text{UNCHANGED } \langle tvars, evars, ovars, svars \rangle$
PROVE: $\Diamond \land \overline{Terminate} \lor \text{UNCHANGED } \langle \overline{svars, rvars} \rangle$
$\land \text{UNCHANGED } \langle \overline{tvars, evars} \rangle$

PROOF SKETCH: The pre-conditions of action $TerminateStub$ are the same as those of $\overline{Terminate}$. Therefore, it is enough to show that each of $TerminateStub$'s sub-actions, $VoteForMyself$, $VoteForOthers$, $Learn$, and the actions they lead to, imply a step of their equivalent overlined actions.

2.1.2.1. ASSUME: $VoteForMyself(r, t)$
PROVE: $\Diamond \land \overline{VoteForMyself(r, t)}$
$\lor \text{UNCHANGED } \langle \overline{terminatingAt, vHist, tHist} \rangle$
$\land \text{UNCHANGED } \langle \overline{terminatedAt} \rangle$

Because the pre-condition and the first post-condition of both actions are the same, it is enough to prove that the second post-condition of action $VoteForMyself(r,t)$, the addition of a message m to $msgs$, may the execution of $\overline{Vote(v)}$, where m is "`Vote`" message and v is a vote, and the fields vt, upd, tr, $tset$, and rm of m and v are equal, or has no effect on variables $\langle \overline{terminatingAt, vHist, tHist} \rangle$.

If message m is received by some coordinator c, a vote with its contents, therefore equal to v, is added to $bSet[c]$, enabling the action $CoordPropose$. Action $CoordPropose$ will be executed with a proposal containing this vote until it is decided and added to $vHist$ by action $CoordDecide$, where all coordinators can see it ($vHist$ is changed deterministically based on the consensus outcomes, and would be the same for all coordinators if represented independently at each one.), or until c crashes. If no coordinator succeeds in getting the vote decided, then either another vote, resulting from the execution of $VoteForOthers(r,t)$ will be decided, or all resource managers involved in transaction t will have crashed before their "`Vote`" messages are seen by non-faulty coordinators. It is up to the transaction manager, to then vote to abort the transaction; in the case the transaction manager also crashes and no vote for t is ever decided, t is simply forgotten, implying that $\langle \overline{svars, rvars} \rangle$ is kept unchanged.

$CoordDecide$ also appends newly committed transactions to $tHist$: each transaction is added in a different set, as if they were not concurrent, and $\overline{LastConcSet}$ is always empty, ensuring its type invariance and constructing $tHist$ in a way compatible with the specification of $\overline{tHist}$. Because this is the only action to change $vHist$ and $tHist$, the mapping to $\overline{vHist}$ and $\overline{tHist}$ is correct.

$\langle \overline{tvars, evars} \rangle$ are kept since none of actions changes them.

2.1.2.2. ASSUME: $VoteForOthers(r,t)$
PROVE: $\Diamond \wedge \overline{VoteForOthers(r,t)} \vee \text{UNCHANGED } \langle \overline{vHist, tHist} \rangle$
$\wedge \text{ UNCHANGED } \langle \overline{terminatedAt, terminatingAt} \rangle$

PROOF: It is true by the same arguments of step 2.1.2.1.

2.1.2.3. ASSUME: $Learn(r,t)$
PROVE: $\wedge \overline{Learn(r,t)} \vee \text{UNCHANGED } \langle \overline{terminatedAt, terminatingAt} \rangle$
$\wedge \text{ UNCHANGED } \langle \overline{svars} \rangle$

PROOF: As action $Learn(r,t)$ has the same post-conditions of $\overline{Learn(r,t)}$, it is enough to show that the pre-conditions of the first imply the pre-conditions of the latter. Since the reception of "`Terminated`" message for transaction t implies that it was sent and since it is only sent by action $CoordDecide$ if the transaction has terminated, the reception of such message implies that the $\overline{Outcome(vHist,t)} \neq$ "`Undefined`".

2.1.2.4. Q.E.D.
2.1.3. Q.E.D.
2.2. $TMActions \Rightarrow \overline{TMActions}$
2.2.1. $AddRM \Rightarrow \overline{AddRM}$
PROOF: Trivially true, since the their definitions are equal.
2.2.2. $RequestTerm \Rightarrow \overline{RequestTerm}$
PROOF: Trivially true, since the their definitions are equal.
2.2.3. UNCHANGED $\langle rvars, evars, svars, msgs, ovars \rangle$
$\Rightarrow$ UNCHANGED $\langle \overline{svars, rvars, evars} \rangle$
PROOF:Clearly true since either the left-hand side of the expression contains all variables in the spec.
2.2.4. Q.E.D.
2.3. $EnvActions \Rightarrow \overline{EnvActions}$
PROOF: Trivially true, since their definitions are equal.
2.4. Q.E.D.
3. Q.E.D.

□

Bibliography

[A. M. Q. P. Working Group, 2006] A. M. Q. P. Working Group (2006). AMQP: Advanced message queuing, version 0.8. 120

[Abadi and Lamport, 1991] Abadi, M. and Lamport, L. (1991). The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284. 139, 150, 206, 220

[Aguilera et al., 1998] Aguilera, M. K., Chen, W., and Toueg, S. (1998). Failure detection and consensus in the crash-recovery model. In *Proc. of the 12th International Symposium on Distributed Computing*. 12, 17, 39, 40

[Aguilera et al., 2000] Aguilera, M. K., Delporte-Gallet, C., Fauconnier, H., and Toueg, S. (2000). Thrifty generic broadcast. In *Proceedings of the 14th International Symposium on Distributed Computing (DISC)*, pages 268–282, Toledo, Spain. Springer-Verlag. 42, 70

[Aguilera et al., 2001] Aguilera, M. K., Gallet, C. D., Fauconnier, H., and Toueg, S. (2001). Stable leader election. In *DISC '01: Proceedings of the 15th International Conference on Distributed Computing*, pages 108–122, London, UK. Springer-Verlag. 32

[Aguilera and Toueg, 1998] Aguilera, M. K. and Toueg, S. (1998). Failure detection and randomization: A hybrid approach to solve consensus. *SIAM J. Comput.*, 28(3):890–903. 40

[Alvisi et al., 2000] Alvisi, Malkhi, Pierce, Reiter, and Wright (2000). Dynamic byzantine quorum systems. In *International Conference on Dependable Systems and Networks (IEEE Computer Society) (replacing both IEEE FTCS (after 29th) and IFIP DCCA (after 7th))*, volume 1. 119

[Ben-Or, 1983] Ben-Or, M. (1983). Another advantage of free choice (extended abstract): Completely asynchronous agreement protocols. In *PODC '83: Proceedings of the second annual ACM symposium on Principles of distributed computing*, pages 27–30, New York, NY, USA. ACM Press. 2, 10, 71

[Bernstein et al., 1987] Bernstein, P., Hadzilacos, V., and Goodman, N. (1987). *Concurrency Control and Recovery in Database Systems*. Addison-Wesley. 94, 112, 113

[Bracha and Toueg, 1983] Bracha, G. and Toueg, S. (1983). Resilient consensus protocols. In *PODC '83: Proceedings of the second annual ACM symposium on Principles of distributed computing*, pages 12–26, New York, NY, USA. ACM. 10

[Burrows, 2006] Burrows, M. (2006). The chubby lock service for loosely-coupled distributed systems. In *USENIX'06: Proceedings of the 7th conference on USENIX Symposium on Operating Systems Design and Implementation*, pages 24–24, Berkeley, CA, USA. USENIX Association. 32

[Camargos et al., 2006a] Camargos, L., Madeira, E. R. M., and Pedone, F. (2006a). Optimal and practical wab-based consensus algorithms. In *Euro-Par 2006 Parallel Processing*, volume 4128 of *Lecture Notes in Computer Science*, pages 549–558, Berlin / Heidelberg. Springer. vii, viii, 2, 11, 12, 13

[Camargos et al., 2006b] Camargos, L., Pedone, F., and Schmidt, R. (2006b). A primary-backup protocol for in-memory database replication. In *NCA '06: Proceedings of the Fifth IEEE International Symposium on Network Computing and Applications*, pages 204–211, Washington, DC, USA. IEEE Computer Society. vii

[Camargos et al., 2007a] Camargos, L., Pedone, F., and Wieloch, M. (2007a). Sprint: a middleware for high-performance transaction processing. In *EuroSys '07: Proceedings of the 2007 conference on EuroSys*, pages 385–398, New York, NY, USA. ACM Press. vii, 113

[Camargos et al., 2008a] Camargos, L., Schmidt, R., and Pedone, F. (2008a). Multicoordinated agreement protocols for higher availabilty. In *NCA '08: Proceedings of the Seventh IEEE International Symposium on Network Computing and Applications*, Washington, DC, USA. IEEE Computer Society. viii, 13

[Camargos et al., 2008b] Camargos, L., Wieloch, M., Pedone, F., and Madeira, E. (2008b). A highly available log service for transaction termination. In *Proceedings of the seventh International Symposium on Parallel and Distributed Computing (ISPDC 2008)*. vii, viii

[Camargos et al., 2007b] Camargos, L. J., Schmidt, R. M., and Pedone, F. (2007b). Multicoordinated paxos: Brief announcement. In *PODC '07: Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, pages 316–317, New York, NY, USA. ACM Press. viii, 13

[Castro and Liskov, 1999] Castro, M. and Liskov, B. (1999). Practical byzantine fault tolerance. In *OSDI '99: Proceedings of the third symposium on Operating*

systems design and implementation, pages 173–186, Berkeley, CA, USA. USENIX Association. 12, 119

[Chandra et al., 1996] Chandra, T. D., Hadzilacos, V., and Toueg, S. (1996). The weakest failure detector for solving consensus. *Journal of the ACM*, 43(4):685–722. 2, 11, 62, 102

[Chandra and Toueg, 1996] Chandra, T. D. and Toueg, S. (1996). Unreliable failure detectors for reliable distributed systems. *Communications of the ACM*, 43(2):225–267. 11, 12, 96

[Cowling et al., 2006] Cowling, J., Myers, D., Liskov, B., Rodrigues, R., and Shrira, L. (2006). Hq replication: A hybrid quorum protocol for byzantine fault tolerance. In *Proceedings of the Seventh Symposium on Operating Systems Design and Implementations (OSDI)*, Seattle, Washington. 119

[Cristian and Fetzer, 1999] Cristian, F. and Fetzer, C. (1999). The timed asynchronous distributed system model. *Parallel and Distributed Systems, IEEE Transactions on*, 10(6):642–657. 11

[Daniels et al., 1987] Daniels, D. S., Spector, A. Z., and Thompson, D. S. (1987). Distributed logging for transaction processing. In Dayal, U. and Traiger, I., editors, *Proceedings of the ACM SIGMOD Annual Conference*, pages 82–96, San Francisco, CA. ACM, ACM Press. 116

[Dolev et al., 1987] Dolev, D., Dwork, C., and Stockmeyer, L. (1987). On the minimal synchronism needed for distributed consensus. *Journal of the ACM (JACM)*, 34(1):77–97. 11, 39

[Dolev et al., 1997] Dolev, D., Friedman, R., Keidar, I., and Malkhi, D. (1997). Failure detectors in omission failure environments. In *Symposium on Principles of Distributed Computing*, page 286. 39

[Dutta and Guerraoui, 2002] Dutta, P. and Guerraoui, R. (2002). Fast indulgent consensus with zero degradation. *Lecture Notes in Computer Science*, 2485. 12

[Dwork et al., 1988] Dwork, C., Lynch, N., and Stockmeyer, L. (1988). Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323. 11

[Fischer et al., 1985] Fischer, M. J., Lynch, N. A., and Paterson, M. S. (1985). Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382. 2, 10

[Gray, 1978] Gray, J. (1978). Notes on database op. systems. In *Advanced Course: Operating Systems*, pages 393–481. 111

[Gray and Lamport, 2006] Gray, J. and Lamport, L. (2006). Consensus on transaction commit. *ACM TODS*, 31(1):133–160. 4, 94, 111, 112, 113, 116

[Guerraoui et al., 1996] Guerraoui, R., Larrea, M., and Schiper, A. (1996). Reducing the cost for non-blocking in atomic commitment. In *ICDCS '96: Proceedings of the 16th International Conference on Distributed Computing Systems (ICDCS '96)*, page 692, Washington, DC, USA. IEEE Computer Society. 95, 111, 113, 116

[Hadzilacos and Toueg, 1993] Hadzilacos, V. and Toueg, S. (1993). *Fault-tolerant broadcasts and related problems*, pages 97–145. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2 edition. 41

[Hurfin et al., 1998] Hurfin, M., Mostefaoui, A., and Raynal, M. (1998). Consensus in asynchronous systems where processes can crash and recover. In *Proceedings Seventeenth IEEE Symposium on Reliable Distributed Systems, IEEE Comput.*, pages 280–286, Soc, Los Alamitos, CA. 12, 39, 40

[Hurfin et al., 2002] Hurfin, M., Mostéfaoui, A., and Raynal, M. (2002). A versatile family of consensus protocols based on chandra-toueg's unreliable failure detectors. *IEEE Trans. Comput.*, 51(4):395–408. 12

[Hurfin and Raynal, 1999] Hurfin, M. and Raynal, M. (1999). A simple and fast asynchronous consensus protocol based on a weak failure detector. *Distrib. Comput.*, 12(4):209–223. 12

[Jiménez-Peris et al., 2001] Jiménez-Peris, R., Patino-Martinez, M., Alonso, G., and Arevalo, S. (2001). A low-latency non-blocking commit service. *Distributed Computing Conference (DISC'01)*, pages 93–107. 95, 112, 113

[Kooh and Haddad, 1999] Kooh, N. F. and Haddad, S. (1999). Reaching agreement in hierarchical groups. In *Proceedings of the 12th International Conference on Parallel and Distributed Computing Systems*, Fort Lauderdale, USA. IASTED Press. 91

[Kotla et al., 2007] Kotla, R., Alvisi, L., Dahlin, M., Clement, A., and Wong, E. (2007). Zyzzyva: speculative byzantine fault tolerance. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 45–58, New York, NY, USA. ACM. 119

[Lamport, 1978] Lamport, L. (1978). Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565. 3

[Lamport, 1996] Lamport, L. (1996). Refinement in state-based formalisms. 206, 220

[Lamport, 1998] Lamport, L. (1998). The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169. 2, 9, 12, 23, 27, 39, 71, 94, 111

[Lamport, 2001] Lamport, L. (2001). Paxos made simple. *ACM SIGACT News (Distributed Computing Column)*, 32(4):18–25. 12, 23

[Lamport, 2002a] Lamport, L. (2002a). *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Professional. 6, 185, 194

[Lamport, 2002b] Lamport, L. (2002b). A summary of tla+. Internet. 185

[Lamport, 2004] Lamport, L. (2004). Generalized consensus and paxos. Technical Report MSR-TR-2005-33, Microsoft Research. 3, 5, 42, 43, 54, 118, 121, 122, 123, 158, 160

[Lamport, 2006a] Lamport, L. (2006a). Fast paxos. *Distributed Computing*, 19(2):79–103. 12, 23, 28, 31, 32, 57, 58, 62, 118

[Lamport, 2006b] Lamport, L. (2006b). Lower bounds for asynchronous consensus. *Distributed Computing*, 19(2):104–125. 9, 13, 31, 40

[Lamport and Massa, 2004] Lamport, L. and Massa, M. (2004). Cheap paxos. In *DSN '04: Proceedings of the 2004 International Conference on Dependable Systems and Networks (DSN'04)*, Washington, DC, USA. IEEE Computer Society. 39, 59

[Lampson, 2001] Lampson, B. (2001). The abcd's of paxos. In *PODC '01: Proceedings of the twentieth annual ACM symposium on Principles of distributed computing*, New York, NY, USA. ACM Press. 3, 12

[Lorch et al., 2006] Lorch, J. R., Adya, A., Bolosky, W. J., Chaiken, R., Douceur, J. R., and Howell, J. (2006). The smart way to migrate replicated stateful services. In *EuroSys '06: Proceedings of the 2006 EuroSys conference*, pages 103–115, New York, NY, USA. ACM Press. 59

[Martin and Alvisi, 2006] Martin, J. P. and Alvisi, L. (2006). Fast byzantine consensus. *Dependable and Secure Computing, IEEE Transactions on*, 3(3):202–215. 12

[Mohan et al., 1985] Mohan, C., Strong, R., and Finkelstein, S. (1985). Method for distributed transaction commit and recovery using byzantine agreement within clusters of processors. *SIGOPS Oper. Syst. Rev.*, 19(3):29–43. 116

[Oki and Liskov, 1988] Oki, B. M. and Liskov, B. H. (1988). Viewstamped replication: A new primary copy method to support highlyavailable distributed systems. In *PODC '88: Proceedings of the seventh annual ACM Symposium on Principles of distributed computing*, pages 8–17, New York, NY, USA. ACM Press. 39

[Oliveira et al., 1997] Oliveira, R., Guerraoui, R., and Schiper, A. (1997). Consensus in the crash-recover model. Technical Report TR-97/239, EPFL – Départment d'Informatique, Lausanne, Switzerland. 39

[Pedone et al., 2003] Pedone, F., Guerraoui, R., and Schiper, A. (2003). The database state machine approach. *Distrib. Parallel Databases*, 14(1):71–98. 73

[Pedone and Schiper, 1999] Pedone, F. and Schiper, A. (1999). Generic broadcast. In *Proceedings of the 13th International Symposium on Distributed Computing (DISC'99, formerly WDAG)*. 41, 42, 70, 92

[Pedone and Schiper, 2002] Pedone, F. and Schiper, A. (2002). Handling message semantics with generic broadcast protocols. *Distributed Computing*, 15(2):97–107. 3, 5, 33, 41, 42, 64, 70, 118

[Pedone et al., 2002a] Pedone, F., Schiper, A., Urbán, P., and Cavin, D. (2002a). Solving agreement problems with weak ordering oracles. In *EDCC-4: Proceedings of the 4th European Dependable Computing Conference on Dependable Computing*, pages 44–61, London, UK. Springer-Verlag. 2, 10, 11, 39

[Pedone et al., 2002b] Pedone, F., Schiper, A., Urbán, P., and Cavin, D. (2002b). Weak ordering oracles for failure detection-free systems. In *Proc. Int'l Conf. on Dependable Systems and Networks (DSN), supplemental volume*. 2, 10, 14, 15, 39

[Rabin, 1983] Rabin, M. O. (1983). Randomized byzantine generals. In *Proc. of the 24th Annu. IEEE Symp. on Foundations of Computer Science*, pages 403–409. 2, 10, 71

[Schiper, 1997] Schiper, A. (1997). Early consensus in an asynchronous system with a weak failure detector. *Distrib. Comput.*, 10(3):149–157. 12

[Schmidt et al., 2007] Schmidt, R., Camargos, L., and Pedone, F. (2007). On collision-fast atomic broadcast. Technical report, EPFL. viii, 3, 75, 76, 77, 78, 79, 88, 91

[Silberschatz et al., 2001] Silberschatz, A., Korth, H. F., and Sudarshan, S. (2001). *Database Systems Concepts*. McGraw-Hill Science/Engineering/Math, fourth edition. 96

[Song et al., 2008] Song, Y. J., van Renesse, R., Schneider, F. B., and Dolev, D. (2008). The building blocks of consensus. In *ICDCN*, pages 54–72. 71

[Sousa et al., 2002] Sousa, A., Pereira, J., Moura, F., and Oliveira, R. (2002). Optimistic total order in wide area networks. In *Proceedings of the 21st IEEE Symposium on Reliable Distributed Systems*, pages 190–199. IEEE CS. 91

[Stamos and Cristian, 1993] Stamos, J. W. and Cristian, F. (1993). Coordinator log transaction execution protocol. *Distributed and Parallel Databases*, 1(4):383–408. 115

[The FIX Protocol Organization, 2008] The FIX Protocol Organization (2008). Fix protocol. Internet site. 120

[TPCC,] TPCC, T. P. P. C. http://www.tpc.org/. 114

[Van Renesse et al., 2003] Van Renesse, R., Birman, K. P., and Vogels, W. (2003). Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. *ACM Trans. Comput. Syst.*, 21(2):164–206. 120

[Vicente and Rodrigues, 2002] Vicente, P. and Rodrigues, L. (2002). An indulgent uniform total order algorithm with optimistic delivery. In *Proceedings of the 21st Symposium on Reliable Distributed Systems*, pages 92–101, Osaka University, Suita, Japan. IEEE. 91

[Vinoski, 2006] Vinoski, S. (2006). Advanced message queuing protocol. *IEEE Internet Computing*, 10(6):87–89. 120

[White et al., 2002] White, B., Lepreau, J., Stoller, L., Ricci, R., Guruprasad, S., Newbold, M., Hibler, M., Barb, C., and Joglekar, A. (2002). An integrated experimental environment for distributed systems and networks. In *Proc. of the Fifth Symposium on Operating Systems Design and Implementation, OSDI02*, pages 255–270, Boston, MA. USENIX Association. 113

[Zielinski, 2004] Zielinski, P. (2004). Paxos at war. Technical Report UCAM-CL-TR-593, University of Cambridge, Computer Laboratory. 12

[Zielinski, 2005] Zielinski, P. (2005). Optimistic generic broadcast. In *Proceedings of the 19th International Symposium on Distributed Computing*, pages 369–383, Kraków, Poland. 42, 70