J Braz Comput Soc (2010) 16: 49–68
DOI 10.1007/s13173-010-0001-7

ORIGINAL PAPER

Multicoordinated agreement for groups of agents

Lasaro Camargos · Rodrigo Schmidt ·
Edmundo Madeira · Fernando Pedone

Received: 30 July 2009 / Accepted: 1 March 2010 / Published online: 23 April 2010
© The Brazilian Computer Society 2010

Abstract Agents in agreement protocols play well-distinct roles. *Proposers* propose values to the *acceptors*, which will accept proposals and inform the *learners* so they detect that an agreement has been reached. A fourth role is that of the *coordinator*, who filters the proposals from proposers to acceptors. While proposers, learners, and coordinators are easily replaced, substituting an acceptor is prohibitive. Protocols that do not employ a coordinator are less resilient to acceptor failures. Protocols that use one coordinator are more resilient to acceptor failures, at the expense of one extra communication step even in the absence of failures. Moreover, they require replacing the coordinator as soon as it fails, a reconfiguration that, although relatively inexpensive, diminishes the protocol availability. Hence, either option, i.e., one or zero coordinator, has its drawbacks. In previous works, we have presented an alternative: multicoordinated agreement protocols. Such protocols are as resilient as single-coordinated protocols but require less reconfiguration to cope with coordinator failures. In fact, most reconfiguration can be done in parallel to the execution of the protocol's normal steps. Multicoordination can be applied to several problems. In this paper we exemplify its use in solving consensus and then introduce a fast multicoordinated agreement protocol for agents organized in groups, an abstraction for fast local area networks interconnected by slower links.

Keywords Multicoordinated · Agreement · Consensus · Broadcast · Groups

L. Camargos (✉) · E. Madeira
IC, Institute of Computing UNICAMP, University of Campinas, PO Box 6176, 13083-970 Campinas, SP, Brazil
e-mail: lasaro@ic.unicamp.br

E. Madeira
e-mail: edmundo@ic.unicamp.br

R. Schmidt
Facebook, 1601 S California Avenue, Palo Alto, CA 94304, USA
e-mail: rschmidt@facebook.com

F. Pedone
Faculty of Informatics, University of Lugano (USI), Via Giuseppe Buffi 6, 6904 Lugano, Switzerland
e-mail: fernando.pedone@unisi.ch

1 Introduction

A distributed application is a composite of agents that perform local actions and exchange information to cooperatively perform some 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 effects of their actions or rollback to a previous state; a certain agent must be excluded from any future interactions. Synchronizing actions, here, means solving an agreement problem.

Many 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, in turn, precedes the learning of the decision. Fault-tolerant algorithms may retry this cycle over and over to circumvent failures and, in each of these *rounds*, it must be ensured that any previously achieved decision is honored in the next ones.

There are three types of rounds. In *fast* rounds, proposals are sent from *proposer* agents to the *acceptor* agents. Acceptors store and forward the proposals to the *learners*, the agents that take an action based on agreement reached through the protocol. Roughly speaking, fast rounds require

that more than two thirds of the acceptors be available to allow progress under the *possibility* of failures. In *classic* rounds, proposals go first through a *leader* (or *coordinator*) that filters all but one of them before forwarding to the acceptors. This extra step allows reducing acceptor quorums to a majority of processes.

In a previous work we have introduced *multicoordinated* rounds, which replace the leader of classic rounds for multiple *coordinators* [7]. This approach drives the availability of individual rounds up by not centralizing their processing on a single coordinator and without requiring larger acceptor quorums. With improved availability, the protocol does not have to start a new round at the first sign of failures and can choose the next set of coordinators while the current round is still executing.

While such gains are minimal in the execution of a single instance of consensus, they are considerable when long sequences of rounds are necessary. This is the case, for example, when implementing a replicated state machine [22, 28]. 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 is used to decide on each command of the sequence. To make the implementation efficient, all instances run in parallel, share the same elected coordinator, and overlap parts of their rounds. Hence, availability gains reflect not only on one but on all ongoing decisions.

A better way of implementing a replicated state machine is to use protocols that let the application specify which commands must be ordered and which ones need not be [25, 32]. By not ordering commands that do not have to be, protocols can be more efficient. Although this kind of protocols is out of the scope of this paper, it is important to know that the techniques presented here also apply to such protocols. In fact, we have presented one such protocol elsewhere [7].

The contribution of this paper is twofold. First, this paper presents multicoordination in much more detail than presented in [8], a brief announcement, and without the complexity of Generalized Consensus as in [9]. Multicoordination can and should be incorporated in the design of agreement protocols and, by detailing the approach, we take another step in such a direction. Second, we present novel algorithms for reaching agreement in hierarchically organized distributed systems. More specifically, we consider a topology that is commonplace in the data centers of large corporations: groups of nodes, with large-bandwidth low-latency links connecting the nodes within the same group, and slow and limited links connecting nodes across groups. In such environments, latency is clearly a major concern and reconfiguration procedures that render the agreement protocol temporarily 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 of concurrent proposals. Besides the use of a multicoordinated approach, we employed multicast primitives and consensus to restrict some reconfigurations within groups, where they are less expensive. As a side effect contribution we also describe CFPaxos, a collision-fast protocol previously published only as a technical report [37].

We start our presentation by defining the system model we consider (Sect. 2.1). Next, we review the consensus problem and approaches to solve it (Sects. 2.2–2.5), including multicoordination. After we further detail the use of multicoordination to solve consensus (Sect. 3), we formalize agreement in networks of groups and the use of standard approaches to solve it (Sect. 4), followed by our multicoordinated protocols (Sect. 5), its correctness proofs and related work. Last, we summarize the contributions in this paper and our further steps (Sect. 6).

2 Distributed agreement problems

2.1 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 a multiple of them. Reasoning in terms of agents allows us to specify problems and algorithms more concisely and in terms of heterogeneous agents. For example, a client/server application may be described in terms of two kinds of agents, *client* and *server* agents, while an e-mailing system may be described in terms of *senders* and *receivers*.

Distributed systems can be classified in different axes according to the way agents exchange information, the way they fail and recover, and the relative speeds at which they perform the computation. In this paper 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 for an agent to execute any action or for a message to be transmitted. If such bounds exist and the number of failures can be limited in time, however, then the protocols that we present in this paper ensure some liveness properties.

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, that is, 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.

2.2 The consensus problem

The consensus problem is an invaluable tool in the study of agreement problems because many of them can actually be reduced to this problem. Consensus is specified in terms of three kinds of agents: *proposer*, *learner*, and *acceptor*. To the best of our knowledge, the specification based on roles was introduced by Lamport [23].

In the consensus problem, agents must agree on a single value out of a given set of proposals. Proposers issue proposals, one out of which will become the decision. Once a decision is reached, learners must become aware of it. 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, there are three safety requirements of consensus [27]:

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.

In many distributed systems, clients experience a high churn and cannot be expected to stay up or connected for long periods. While it is OK to state the safety requirements in terms of proposers and learners, the clients of a consensus service, the same does not apply for the liveness requirements. Hence, we state the liveness in terms of the acceptors, which are part of the application infrastructure and, therefore, more reliable. 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.

The well-known FLP result states that no fault-tolerant consensus algorithm can ensure termination in the presence of failures [17] under the assumed asynchronous crash–recovery model. Phrased in terms of acceptors, this result implies that quorums must equal the set of all acceptors. Hence, in order to be fault-tolerant, algorithms must make extra assumptions about the system to ensure liveness.

2.3 Circumventing FLP

One way of circumventing the FLP result is through randomization. Bracha and Toueg's algorithm, for example, relies on the assumption 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, a property they labeled *fair scheduling* [4]. Somewhat the same principle lies in Rabin's [35] and Ben-Or's [3] protocols, in which agents rely on a random bit generator to eventually choose the same bit as proposal. If queried in rounds by the agents, then for every round there is a non-zero probability that all agents chose the same random bit and agreement is reached with probability 1.

Pedone et al.'s protocol replaces the random bit generator by the assumption that, for every round, there is a non-zero probability that messages will be received in the same order by all agents [33]. This property is abstracted by *weak ordering oracles* [34]. Pedone et al.'s protocols [33], however, were devised for the crash–stop model and do not tolerate lossy communication channels. These shortcomings were addressed in enhanced versions of the protocols [5], namely R*-Consensus and B*-Consensus. B*-Consensus was actually the basis for the multicoordinated rounds that we describe later in this paper.

Several other works have circumvented the FLP result by assuming a partially synchronous model [12–14, 16]. The concept of unreliable failure detectors (UFD) [12] is probably the most notorious result in this model. The UFD encapsulate the minimal synchrony assumptions required to solve consensus as abstract properties ensuring that, eventually, some state will be reached in which progress can be made. The weakest UFD that can be used to solve consensus, $\Diamond\mathcal{W}$ [11], 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.

The Ω leader election oracle [11] is an abstraction equivalent to $\Diamond\mathcal{W}$. Briefly, Ω ensures that nonfaulty agents eventually agree on the identity of some nonfaulty agent to be the *leader*.

In spite of their strength, UFD and randomization are still fallible in the sense that they give no guarantees of when they will provide correct information. To make sure that no

irreversible decision is taken based on these tools, one extra assumption is needed [12, 23]:

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

2.4 Rounds and coordinators

Paxos [23] is a notorious example of Ω-based consensus protocol. In Paxos, 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 possible previously decided values and use such values, if existent, in the new rounds. Moreover, in starting a new round, the coordinator must prevent previous ones from deciding if they have not done so yet. Hence, if rounds are indiscriminately started, it may happen that no round ever decides on any value. To avoid such a scenario, a *leader* coordinator is selected to start new rounds through a leader election oracle like Ω. Because the Ω oracle is allowed to make mistakes and elect multiple agents at the same time, rounds may be started even in the absence of failures, preventing progress toward a decision until mistakes cease to happen.

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. The new leader must synchronize with acceptors to block previously started rounds and, while this synchronization is done, no decision can be reached. That is, the protocol is unavailable.

Fast Paxos [26] 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. Hereafter we refer to these round types respectively as classic and fast rounds.

Without the leader to filter out proposals, different acceptors may accept different values, which we call a *collision*. Collisions may prevent any value from ever being decided and to recover from them, quorums of acceptors must be bigger in fast than in classic rounds. Therefore, fast rounds are potentially less available than classic ones, as stated by the following requirement.

Assumption 2 (Simple fast quorum requirement) If Q, R, and S are quorums, then $Q \cap R \cap S \neq \emptyset$.

This requirement is actually stronger than necessary, but serves as a good generalization. The actual Fast-Quorum requirement is presented later in the paper.

2.5 Multicoordinated rounds

As the name suggests, multicoordinated rounds have multiple coordinators to which the proposers send their proposals in parallel. Similarly to what happens in single coordinated rounds, coordinators in a multicoordinated round forward the proposals from proposers to the acceptors. Acceptors, however, only take into consideration proposals forwarded by a quorum of the round coordinators. This way, values may be accepted even if some coordinators crash, as long as a full quorum is alive. By requiring these coordinator quorums to intersect, the protocol ensures that no two different values will be considered on the same round and, thus, no collision happens on the acceptors. Hence, acceptor quorums are only required to satisfy Assumption 1, being more resilient than in Fast Paxos. 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 [7].

Many algorithms in the literature may be seen as instances of Fast Paxos, in spite of several differences in their actual specifications. As an example, consider differences in the assumed failure model: some algorithms were specified for the crash–stop model [12, 15, 19, 20, 36], some for the crash–recovery [1, 18, 23, 24], and some for the Byzantine [10, 28, 29, 40]. Therefore, by presenting multicoordination as an extension to Fast Paxos we make it simpler to derive multicoordinated modes to the other protocols. We do so in the next section by reviewing a multicoordinated consensus protocol [8].

3 Multicoordinated consensus

Our multicoordinated consensus protocol, MCC for short, is an extended version of Fast Paxos [25] that, in turn, extends the classic Paxos protocol [23]. As a result, MCC has the same requirements of the Paxos protocols as well as the same structure. While here we present MCC directly, we refer the reader to [7] to a progressive explanation starting with Paxos, extending it to Fast Paxos, and finally presenting MCC.

The rounds in MCC are uniquely identified by round numbers totally ordered by a relation $<$. Nonetheless, their execution does not have to follow this order. In fact, actions

referring to different rounds may even interleave. In this paper, we assume that round numbers correspond to the set of natural numbers. For a discussion of other types of round numbers and their interesting properties, the reader is referred to [7].

Acceptors and coordinators are divided in quorums per round. The quorums of acceptors and of coordinators of round i are called, respectively, i-quorums and i-coordquorums. We say that a value is *chosen* in a round i if all acceptors in an i-quorum have accepted the value while in i. The goal of each round is to choose a value while ensuring the following property: If a value v is chosen in round i, then no acceptor will ever accept a value different from v in any round j such that $j > i$. This is done in two phases: phase 1 serves to identify previously decided values and phase 2 tries to decide on some value while maintaining consistency with previous decisions, identified in the phase 1.

We initially require that Assumptions 1 and 2 be satisfied. In the next section we show that a combination of the first and a weakened version of the second are sufficient. Moreover, MCC also requires Assumption 3 be satisfied.

Assumption 3 (Coordquorum requirement) For any two quorums of coordinators P and Q for the same classic round, $P \cap Q \neq \emptyset$.

3.1 The algorithm

The actions of Algorithm 1 are defined in terms of the variables manipulated by the agents. The function of each variable will become clear from the explanation of the actions. 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*.

3.1.1 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 and, to allow the execution of fast rounds, to acceptors alike. To minimize communication overhead, proposals may be initially sent only to coordinators and sufficient acceptors to form the quorums needed to decide; if needed, the proposals are then re-submitted in broadcast. Observe that this is only possible if multiple rounds share the same quorums, otherwise proposers would not know where to send proposals, because they are oblivious to rounds.

3.1.2 Phase one

In Paxos and Fast Paxos, each round i has a 1-to-1 association to a coordinator agent c, and only such a coordinator can execute action *Phase1a*(c, i). In MCC, the association is 1-to-many, and any of such coordinators can execute the action, no matter how they are subdivided in coordquorums. 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.

In reply, an acceptor a executes *Phase1b*(a, i) if it has not heard of any round bigger than i, where a has heard of j if it already executed actions *Phase1b*(a, j) or *Phase2b*(a, j). In this case, a joins round i by setting *rnd*[a] to i, and sends a message $\langle$"`1b`"$, a, i, vrnd[a], vval[a]\rangle$ 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 yet accepted by a, then *vrnd*[a] equals its initial state and *vval*[a] equals an invalid proposal, *none*.

By the precondition that $rnd[a] < i$ 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.

3.1.3 Phase two

The second phase starts once a coordinator c receives $\langle$"`1b`"$, a, i, vrnd, vval\rangle$ messages for the same round i from all acceptors a in an i-quorum Q and executes action *Phase2a*(c, i). Observe that a coordinator c executes *Phase2a*(c, i) at most once for each round i, but any coordinator of round i can start the second phase even if it did not start phase one, 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.

In executing *Phase2a*(c, i), 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. c picks *val* based on the "`1b`" messages received from the acceptors in Q in round i, as explained next. This procedure is formally defined as function *PickValue*(c, Q, i) in Algorithm 2.

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

Algorithm 1 Multicoordinated Consensus

```
1:  Proposer Actions:
2:  Propose(a, v) ≜
3:    preconditions:
4:      a ∈ proposers
5:    actions:
6:      send ⟨"propose", v⟩ to coordinators ∪ acceptors
7:  Phase One:
8:  Phase1a(c, i) ≜
9:    pre-conditions:
10:     c ∈ ⋃ i-coordquorum
11:     crnd[c] < i
12:   actions:
13:     send ⟨"1a", c, 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
22: Phase Two:
23: Phase2a(c, i) ≜
24:   preconditions:
25:     c ∈ ⋃ i-coordquorum
26:     crnd[c] ≤ i
27:     ∃Q : Q is an i-quorum and ∀a ∈ Q, c received ⟨"1b", a, i, rnd, val⟩
28:   actions:
29:     crnd[c] ← i
30:     cval[c] ← PickValue(c, Q, i)
31:     send ⟨"2a", c, crnd[c], PickValue(c, Q, i)⟩ to acceptors
32: PickValue(c, Q, i) is defined in Algorithm 2.
33: Phase2b(a, i) ≜
34:   preconditions:
35:     a ∈ acceptors
36:     rnd[a] ≤ i
37:     vrnd[a] < i
38:     ∃C, val : C is an i-coordquorum and ∀c ∈ C a received ⟨"2a", c, i, val⟩ from c
39:   actions:
40:     rnd[a] ← i
41:     vrnd[a] ← i
42:     IF val ≠ Any
43:      THEN vval[a] ← val
44:      ELSE vval[a] ← CHOOSE v: a received ⟨"propose", v⟩ from some proposer
45:     send ⟨"2b", a, i, val⟩ to learners
46: Learn(l) ≜
47:   preconditions:
48:     l ∈ learners
49:     ∃Q : Q is an i-quorum and ∀a ∈ Q, l received ⟨"2b", a, i, val⟩
50:   actions:
51:     learned[l] ← val
```

Algorithm 2 *PickValue*(Q) in Multicoordinated Consensus

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

by Assumption 1, no such a quorum has or will succeed in choosing any value in j. As a result, c can pick any proposed value.

Otherwise, if some valid value is received by c, let k be the greatest round number *vrnd* received among 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.

Whether the round was fast or not is irrelevant to the agents until the moment in which the coordinator must execute function *PickValue*(c, Q, i) and pick a value. At this point, if i is a fast round and if either the first or the third case happens, that is those in which any value may be picked, then c can tell the acceptors to accept proposals directly from proposers. It does so by picking a special value *Any* in *PickValue*. Irrespectively of the value picked, *Any* or otherwise, c sends the message $\langle$"2a", $c, i, val\rangle$ to acceptors.

Once an acceptor a accepts some value *val* in round i, by executing action *phase2b*, it sends a message $\langle$"2b", $a, i, val\rangle$ to warn the learners. The exact way in which action *phase2b* works, however, depends on whether the round is fast or not, if the coordinator has sent the *Any* special value, and on the concept of coordinator quorums. We focus on these points in the following sections. In any case, 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*$[l]$.

3.2 Accepting values in fast rounds

As we have stated above, each round is associated with many coordinators, each of which is allowed to execute action *phase2a* and send a proposal to the acceptors and, therefore, it is possible that acceptors receive multiple proposals in the same round i.

If the acceptors were to accept the first proposal they see, then the round may not succeed in the sense that no single value is accepted by a quorum of acceptors. To avoid this situation, in non-fast rounds an acceptor accepts a value iff it was received from all coordinators in an i-coordquorum. Since we required coordinator quorums to satisfy Assumption 3, no two different values can ever be accepted in the multicoordinated (non-fast) round.

In a fast round, however, if an acceptor a receives the value *Any* from a coordinator, then a will accept proposals directly from the proposers. Since no restriction is put on the number of proposers in the system, we cannot organize them in quorums to avoid accepting multiple values, and it may happen that different acceptors accept different values in the same fast round. If we define quorums as any majority of the acceptors (satisfying Assumption 1 only), having multiple values accepted in the same round might lead to the following run of MCC:

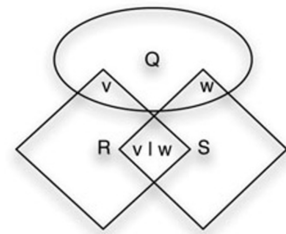

Fig. 1 Q is an i-quorum, R and S are k-quorums, and v and w are the values accepted by acceptors in $Q \cap R$ and $Q \cap S$, respectively

1. In the fast round i, acceptor a_v accepts value v and acceptor a_w accepts value $w \neq v$.
2. In round $i+1$, every coordinator c of $i+1$ receives "1b" messages from a quorum Q containing a_v and a_w.

At this point, there are two different values, v and w, and two i-quorums, R and S, such that $Q \cap R = a_v$ and $Q \cap S = a_w$. Moreover, 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 acceptors in $(S \cup R) \setminus Q$ accept. By Assumption 1, R and S have a non-empty intersection, which prevents both values from being chosen, but since this intersection does not intersect Q, c cannot decide which value to pick. Even worse, if all acceptors in $R \setminus S$ accepted v and all acceptors in $S \setminus R$ accepted w, then if the acceptors $R \cap S$ crash definitively, the algorithm will never be able to tell whether v or w were already decided and cannot progress any further. This scenario is depicted in Fig. 1.

The way to avoid such a scenario 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 above, also intersects Q. If this is ensured, the situation discussed above 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. In MCC and Fast Paxos, the following requirement is sufficient:

Assumption 4 (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 $Q \cap R \cap S \neq \emptyset$.

If every set of $n-E$ acceptors is a *fast quorum* and every set of $n-F$ acceptors is a *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 $\lfloor\frac{2n}{3}\rfloor+1$ 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 $\lfloor\frac{3n}{4}\rfloor+1$ 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 [27] (Fast Learning Theorem).

Even though we can avoid the situation in which the algorithm cannot safely execute more rounds, the acceptance of multiple values in fast rounds can still prevent the round from succeeding, if no complete quorum accepts the same value. There are different ways of recovering from such a *collision* [26], and all reduce to executing a higher-numbered round. However, depending on how this new round is chosen, latency can be reduced considerably. We refer the reader to [26] and [7] for in-depth discussions of collision recovery.

3.3 Correctness and liveness

The MCC algorithm satisfies the safety properties of consensus mainly because it ensures that rounds are consistent with previous decisions. As described in the definition of *Phase1b*, executing action *Phase1b* for a given round prevents the same acceptor from executing action *Phase2b* for any smaller round, and action *Phase2a* selects the previously chosen value, if any. 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.

In the simplified case where rounds are single-coordinated, 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 (by using an Ω oracle) and the proposer proposes some value. This is equivalent to the rounds of the original Paxos protocol [23].

A similar condition may seem harder to achieve for multicoordinated rounds, 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, at most one value is accepted in such rounds. Coordinators of rounds executed afterward contact only the acceptors and are not influenced by the multiple coordinators of lower rounds. Roughly speaking, a multicoordinated round i will finish if no other round is started, some proposal is made, and at least one of its coordquorums is composed of non-crashed coordinators and all of its coordinators receive the same proposal. Moreover, because our protocol is an extension of Fast Paxos, it can switch to single-coordinated rounds at any moment and ensure liveness under the same conditions of classic Paxos. We formalized this argument in [6].

4 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. The busyness model of online retailers, such as Amazon.com, and social networking sites, such as Facebook and Orkut, mandate that the user data, such as profiles and shopping carts, be available to their millions of users at nearby data centers for a better user experience. Data must also be available at other locations to support mobility. In some cases, although clients may access their data on any data center, updates are performed in a single location and only later propagated to the other locations. While simpler to implement, this strategy incurs in higher latency between an update and the availability of the written data at the closest data center; allowing updates everywhere, instead, is preferable. Implementing such an update-everywhere database replication may be done atop of agreement protocols (e.g., Pedone et al. [30]) but, to be effective, such agreement protocols must be efficient and resilient in this networking scenario which we refer to as networks of groups.

Networks of groups 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 exchange messages through low-latency communication channels, those in different groups may experience latencies that are orders of magnitude higher. More specifically, we consider networks abstracted by Fig. 2, 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, 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, Fig. 2 depicts the acceptors in the system as another set, A. To ensure fault-tolerance, A should be composed of 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.

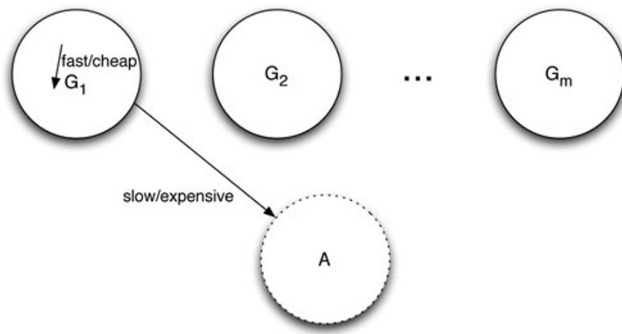

Fig. 2 Agents distributed in groups in a wide area network. Agents in a group $G_i, 0 \leq i \leq m$, are physically close to each other. Agents in A are spread geographically

4.1 A round type for network of groups

4.1.1 *Multicoordinated fast rounds*

Consider the price of solving standard consensus in a network of groups using a leader-based protocol such as 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 sections 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 messages tend to be ordered spontaneously and collisions are more unlikely. 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 of proposals and, by consequence, larger acceptor quorums and, therefore, lesser resiliency and more messages crossing the boundaries of groups. To minimize the number of messages exchanged, one agent may be selected from each group to aggregate the groups' proposals and forward them to the acceptors instead of letting every agent propose in every fast round. Such a fast round with aggregation is, in fact, a multicoordinated round in which each agent aggregating proposals is a coordquorum. Observe however that this multicoordinated fast round does not satisfy the coordquorum requirement (Assumption 3).

When comparing single-coordinated and fast rounds, we see that we either need to ensure that a single value is proposed or we allow conflicting values be proposed but require Assumption 4 be true and more acceptors be available. The same holds true for multicoordinated classic and multicoordinated fast rounds, as the one described in the previous paragraphs of this section. That is, we either enforce that only a single proposal per round can possibly be accepted by satisfying the coordquorum requirement or we relax this requirement and satisfy the fast-quorum requirement. Both requirements can only be relaxed at the same time if absence of conflicts is guaranteed in some other way. This is what is accomplished by *collision-fast* rounds.

4.1.2 *Fast rounds in spite of collisions*

Guaranteeing that proposals do not conflict is exactly what the Collision-Fast Paxos protocol [37] 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 a 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.

By combining CFPaxos and the multicoordinated fast rounds presented in the previous section we get a round type that is perfectly suited for reaching agreement in networks of groups. In Sect. 5 we introduce two of such combined protocols, a basic and an optimized version. To simplify the presentation of our protocols, we first overview the CFPaxos protocol in the following section.

4.2 Collision-fast Paxos

In CFPaxos, learners must eventually learn a mapping from all proposers to the values that have been 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 [37], or v-maps. These mappings are also what name the problem solved by CFPaxos, that is, the Mapping Consensus problem [37], or M-Consensus for short.

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 $(v : D \to R) \in VMap$ 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 $v : \{a\} \to \{b\}$, 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." We also define the $\bullet$ operator, which extends a v-map with an s-map—where v is a v-map, s is an s-map, and $Dom(v)$ is the domain of v-map v—as follows:

$v \bullet s = w$, 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 $v \bullet \sigma = w$. Therefore, $\sqsubseteq$ is a partial order relation on v-map sets.

Given a v-map set V, we say that v-map v is a lower bound of V iff $v \sqsubseteq w$ for all w in V. A *greatest lower bound* (glb) of V is a lower bound v of V such that $w \sqsubseteq v$ for every lower bound w of V, and we represent it by $\sqcap V$. Similarly, we say that v is an upper bound of V iff $w \sqsubseteq v$ for all w in V. A *least upper bound* (lub) of V is an upper bound v of V such that $v \sqsubseteq w$ for every upper bound w of V, and we represent it by $\sqcup V$. For simplicity of notation, we use $v \sqcap w$ and $v \sqcup w$ to represent $\sqcap\{v, w\}$ and $\sqcup\{v, w\}$, respectively. A set V of v-maps is *compatible* iff for every pair of v-maps $v, w \in V$, for all elements $e \in Dom(v) \cap Dom(w)$, $v(e) = w(e)$. Note that, since $\sqsubseteq$ is a reflexive partial order on the set of v-maps, if a glb or lub of V exists, then it is unique, and that the existence of the lub $\sqcup V$ of a set of v-maps V depends on V being compatible.

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 $\{Nil\}$.

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 $\perp$ [37]:

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 similarity between the specification of M-Consensus and consensus is not coincidental. These problems are, in fact, equivalent: one can solve consensus by deterministically choosing a proposal from the M-Consensus solution and, conversely and, conversely, one solve M-Consensus by building v-map whose range equals the decision of 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 Algorithm

CFPaxos [37] 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 with a single coordinator agents that may start them. Each round is associated with a subset of proposers, which are the only ones allowed to send their proposals to the acceptors in that round. As a result, 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, if a collision-fast proposer q has not proposed yet, then it abstains from proposing by sending the s-map $\{q \mapsto Nil\}$ to the learners.

Acceptors accept s-maps to extend their accepted v-maps. Since they only receive s-maps with value other than *Nil* 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. Formally, 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;
- after the second communication step, learners receive the messages from the acceptors and collision-fast proposers and learn the v-map decided.

If the other collision-fast proposers have proposed before receiving p's proposal then, in the absence of failures, at most three communication steps before the protocol terminates and all proposed values are learned.

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-maps in the new round. The procedure is similar to the other Paxos algorithms and, in special, to Generalized [25] and Multicoordinated Paxos [9]: 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 reader is referred to the original work [37], 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.

5 Multicoordinated agreement for groups

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 such as the one in Fig. 2. By assigning one collision-fast proposer to each group, to whom proposers in the same 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 to replace it with another proposer in the same group. If no agent is up to the task of collision-fast proposer, however, then the leader has no option but not select one from the given group, forcing regular proposers to the resort to other groups to have their proposals considered or stop proposing.

In this section we show how we can extend CFPaxos to tolerate the failure of collision-fast proposers in a group without changing rounds nor leaving the proposers "orphan" in situations in which CFPaxos would do so. Our extension consists in using multicoordination and recursively applying consensus inside each group. Because collision-fast proposers execute the tasks that we have associated with coordinators in the previous sections, notably, 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, 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 proposals).

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 in Assumption 5.

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

Assumption 5 is easily satisfied by having each quorum contain any majority of the coordinators of round i in a group G.

In the following section 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, ExtendedMCF, in Sect. 5.2.

5.1 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 $\bot$.

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 with a set of coordinator quorums. This scheme may be defined as 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 3 presents the actions of BasicMCF.

5.1.1 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.

5.1.2 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 $\langle\text{“1a”}, c, i\rangle$ message.

An acceptor a reacts to a $\langle\text{“1a”}, c, i\rangle$ 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 $\langle\text{“1b”}, a, i, vrnd[a], vval[a]\rangle$. Through this “1b” message, a promises to c that it will not accept any v-map in any round smaller than i.

5.1.3 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 $\langle\text{“2S”}, c, i, v\rangle$ 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 $\langle\text{“2S”}, d, i, v\rangle$ 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 $\langle\text{“propose”}, w\rangle$ message, in which case c sets $cval[c]$ to w and then sends a $\langle\text{“2a”}, c, i, \{group[c] \mapsto cval[c]\}\rangle$ message to the acceptors and all the other coordinators of round i. The second is after receipt of a $\langle\text{“2a”}, _, i, _, \{G \mapsto w\}\rangle$ message where $\{G \mapsto W\}$ is an s-map from group $G \neq group[p]$ to a value $w \neq Nil$. In this case, c sets $cval[c]$ to *Nil* and sends message $\langle\text{“2a”}, c, i, \{group[c] \mapsto cval[c]\}\rangle$ directly to the learners.

Algorithm 3 BasicMCF

```
1:  Proposer Actions:
2:  Propose(p, v) ≜
3:    preconditions:
4:      p ∈ proposers
5:    actions:
6:      send ⟨"propose", v⟩ to all coordinators of group group[p]
7:  Phase One:
8:  Phase1a(c, i) ≜
9:    preconditions:
10:     c is the coordinator of i
11:     crnd[c] < i
12:   actions:
13:     send ⟨"1a", c, i⟩ to acceptors
14: Phase1b(a, i) ≜
15:   preconditions:
16:     a ∈ acceptors
17:     rnd[a] < i
18:     received ⟨"1a", c, i⟩ from coordinator c
19:   actions:
20:     rnd[a] ← i
21:     send ⟨"1b", a, i, vrnd[a], vval[a]⟩ to c
22: Phase Two:
23: Phase2Start(c, i) ≜
24:   preconditions:
25:     c is the coordinator of i
26:     crnd[c] < i
27:     ∃Q : Q is a quorum and ∀a ∈ Q, c received ⟨"1b", a, i, rnd, val⟩
28:   actions:
29:     LET v ≜ PickValue(c, Q, i)
30:     IN
31:       crnd[c] ← i
32:       IF v = ⊥
33:        THEN
34:         send ⟨"2S", c, i, v⟩ to ⋃_{G∈Γ} G_i-coordquorum
35:         IF c ∈ ⋃_{G∈Γ} G_i-coordquorum
36:          THEN cval[c] ← none
37:        ELSE
38:         send ⟨"2S", c, i, v⟩ to ⋃_{G∈Γ} G_i-coordquorum ∪acceptors
39:         IF c ∈ ⋃_{G∈Γ} G_i-coordquorum
40:          THEN cval[c] ← v(group[c])
41: PickValue(c, Q, i) ≜
42:   LET
43:     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}
44:     S ≜ {v : c received ⟨"1b", a, i, k, v⟩ from a ∈ Q} \ {none}
45:   IN  IF S = {} THEN ⊥ ELSE ⊔S • {G ∈ Γ ↦ Nil}
46: Phase2Prepare(c, i) ≜
47:   preconditions:
48:     c ∈ ⋃_{G∈Γ} G_i-coordquorum
49:     crnd[c] < i
50:     c received ⟨"2S", c, i, v⟩
51:   actions:
52:     crnd[c] ← i
53:     IF v = ⊥ THEN cval[c] ← none ELSE cval[c] ← v(group[c])
54: Phase2a(c, i) ≜
55:   preconditions:
56:     c ∈ ⋃_{G∈Γ} G_i-coordquorum
57:     crnd[c] = i ∧ cval[c] = none
58:     ∃v : (c received ⟨"propose", v⟩) ∨ (received ⟨"2a", d, i, {G ↦ w}⟩: G ≠ group[c] ∧ w ≠ Nil = v)
59:   actions:
60:     cval[c] ← {G ↦ v}
61:     IF v = Nil
62:      THEN send ⟨"2a", c, crnd[c], cval[c]⟩ to learners
63:      ELSE send ⟨"2a", c, crnd[c], cval[c]⟩ to ⋃_{G∈Γ} G_i-coordinators ∪acceptors
64: Phase2b(a, i) ≜
65:   preconditions:
66:     a ∈ acceptors
67:     rnd[a] ≤ i
68:     (a received ⟨"2S", c, i, v⟩: v ≠ ⊥ ∧ (vnrd[a] < i ∨ vval[a] = none))∨
        (∃L, v ≠ Nil : L is an G_i-coordquorum ∧∀c ∈ L: a received ⟨"2a", c, i, {G ↦ v}⟩)
```

Algorithm 3 BasicMCF (Continued)

69: **actions:**
70: IF a received $\langle$"2S", $c, i, v\rangle$: $v \neq \bot \wedge (vnrd[a] < i \vee vval[a] = none)$
71: THEN $vval[a] \leftarrow v$
72: ELSE IF $\exists L, v \neq Nil$: (L is an G_i-coordquorum) $\wedge$
($\forall c \in L$: a received $\langle$"2a", $c, i, \{G \mapsto v\}\rangle$) $\wedge$
$(vnrd[a] < i \vee vval[a] = none)$
73: THEN $vval[a] \leftarrow \{G \mapsto v\} \bullet \{G \in \Gamma \mapsto Nil\}$
74: ELSE $vval[a] \leftarrow vval[a] \bullet \{G \mapsto v\}$
75: $rnd[a] \leftarrow vrnd[a] \leftarrow i$
76: send $\langle$"2b", $a, i, vval[a]\rangle$ to *learners*

$Learn(l) \triangleq$
78: **preconditions:**
79: $l \in learners$
80: $\exists Q : Q$ is a quorum **and** $\forall a \in Q$, l received $\langle$"2b", $a, i, val\rangle$
81: $\exists \gamma \in \Gamma : \forall G \in \gamma : \exists P : P$ is a G_i-coordquorum **and** $\forall c \in P$, l received $\langle$"2b", $c, i, \{G \mapsto Nil\}\rangle$
82: **actions:**
83: LET $Q2Vals \triangleq \{v : l$ received $m = \langle$"2b", $a, i, v\rangle, a \in Q\}$
84: IN $learned[l] \leftarrow learned[l] \sqcup (\sqcap Q2bVals \bullet \{G \in \gamma \mapsto Nil\})$

An acceptor a executes action $Phase2b(a, i)$ upon receipt of a $\langle$"2S", $c, i, v\rangle$ message 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, that is $vval[a] = none$. 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. Therefore, besides setting $rnd[a]$ and $vrnd[a]$ to i, a 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.

5.1.4 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 in 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 in a round lower than or equal to k. In this case, the function evaluates $\sqcup S$ extended with $\{G \mapsto Nil\}$ for every group G. Because the coordinator of round k must have picked a v-map that extended v-maps possibly chosen in 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 than 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 in round k. In addition, because acceptors only accept v-maps not mapping to *Nil*, $\sqcup S$ does not map to *Nil*.

5.1.5 Learning a value

Learning a v-map takes place 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*})).

5.1.6 Handling collisions

Because the communication latencies within a single G group are very low, there is a high probability that proposals will be spontaneously ordered at the coordinators. This, in turn, makes it likely that at least one G_i-coordquorum for each round i will forward the same proposal to the acceptors. In the cases when spontaneous ordering does not hold and conflicting proposals are forward to the acceptors, the acceptors simply will not accept any s-map for group G and it will be treated as if the whole group had failed. That is, a new round is started so that new acceptors in G have another chance of proposing.

5.1.7 Example round

Figure 3 shows a sample execution of a single round of the BasicMCF protocol. The scenario depicted has three groups of agents, G1, G2, and G3. Even though the acceptor agents belong to these three groups, they are depicted outside the groups to make explicit their message exchange; the location of the learners is irrelevant. No value has been proposed and no mapping has been accepted in this execution of the protocol. Some messages unnecessary in a run without failures as well as the communication inside groups G2 and G3 were omitted to keep the figure simple.

The coordinator of the round, one of the coordinators of group G1, starts the round by sending a 1A message to the acceptors and waiting for their 1B reply. Next, the coordinator sends a 2S message to the acceptors and all other coordinators of the round stating that nothing has yet been accepted. In an implementation of atomic broadcast, these steps are executed for multiple instances of the protocol within the same message exchange, as explained in [37].

When a proposer of group G1 sends its proposal to its group's coordinators, the later forward the proposal to the acceptors and coordinators of groups G2 and G3. Observe that in parallel, coordinators of G2 also send 2A messages to acceptors and coordinators, implying that a proposal was also made inside the group. The coordinators of G3, however, only send 2A after receiving the 2A from G1, and send their messages to the learners, implying that they have nothing to propose. Assuming that the proposal of G1 was $\{G1 \mapsto c\}$ and the proposal of G2 was $\{G2 \mapsto d\}$, at the end of the round, the learners will learn the mapping $(v : G1 \rightarrow c, G2 \rightarrow d, G3 \rightarrow Nil)$.

5.2 Adding intra-group reconfiguration

Although reconfiguration happens less often in BasicMCF than in the original CFPaxos, it may still happen if coordinators inside groups are unreliable. We now discuss how to extend BasicMCF in such a way that, in certain situations, failures are handled inside groups without changing

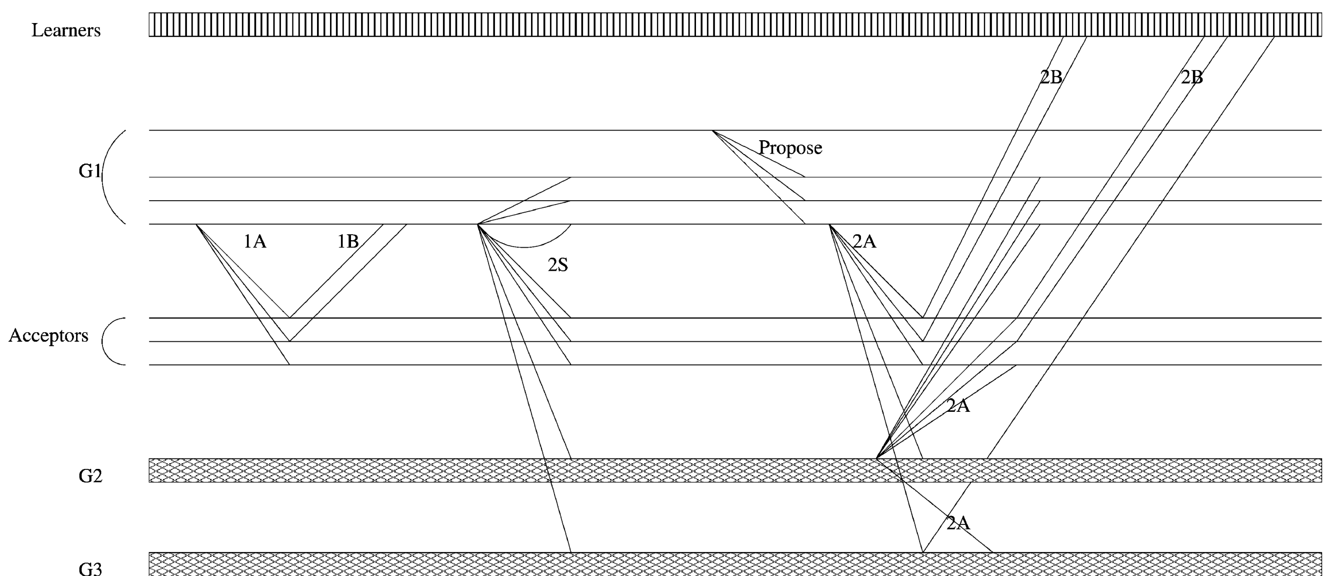

Fig. 3 A sample BasicMCF round; some messages have been omitted and acceptors are depicted outside their groups for clarity. After the round coordinator in G1 starts the round, two values from proposers in G1 and G2 are decided; as the number of proposals raises, the amortized cost of each decision decreases

rounds. This extended algorithm, which 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 intergroup message exchanges addresses all coordinators of a group instead of specific ones. Inside each group, agents agree on exactly which coordinators should actually form coordquorums 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; it cannot propose another. Acceptors and learners, which receive messages from the coordinators, can be informed on-the-fly about the composition of coordquorums. To allow them to distinguish different coordquorum 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 detail 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 until 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 adjustable time and then propose such values 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`" message 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.

5.3 Correctness and liveness

The correctness and liveness properties of CFPaxos were formally proven when the protocol was introduced [37]. 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 will 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. ExtendedMCF is discussed afterward.

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 preconditions. 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 latter c must be in the list of collision-fast proposers of round i. As we mentioned 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 $\overline{Phase2Prepare(c,i)}$ for every coordinator c in some G_i-coordquorum and due to the reception of the same "2S" message (and $\overline{Phase2Start(c,i)}$, 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 $\overline{Phase2a(c,i)}$.

Action $\overline{Phase2b(a,i)}$ 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, $\overline{Phase2b(a,i)}$ in fact implements action $Phase2a(a,i)$. With a similar argument we conclude that action $\overline{Learn(l)}$ implements $Learn(l)$.

It is straightforward to see that ExtendedMCF implements BasicMCF since each action in BasicMCF is executed in the same way in ExtendedMCF or under more preconditions. 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 to 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.

6 Related works

In [8] we have presented a multicoordinated consensus protocol (MCC) that extends Fast Paxos [26], 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. It is also an important tool in the study of agreement protocols, by using the right combination of round types and recovery technique, our protocol emulates most consensus protocols that we are aware of. In [9] we have shown the use of the multicoordinated mode in solving generic broadcast and discussed its relation to generalized consensus. We reviewed the multicoordinated execution mode in Sect. 2.5 and MCC in Sect. 3.

In this paper we tackled the problem of efficiently reaching agreement in a network organized as groups of agents. Our solution is an extension of the Collision-Fast Paxos protocol [37], which does not consider groups. For the reader's convenience we explained CFPaxos in Sect. 4.2. The same problem, agreement in groups of agents, was studied by Kooh and Haddad [21]. 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

Table 1 Comparison of message and time complexity in scenario with G groups with c coordinators, L learners, A acceptors, and 1 proposal per group

	Delay		Number of messages		Decisions
	Intra	Inter	Intra	Inter	
Paxos	0	3		$G + A * (L + 1)$	1
Fast Paxos	0	2		$G * A + A * L$	1
CFPaxos	1	2	G	$(G - 1) * G + G * A + A * L$	G
BasicMCF	1	2	$c * G$	$c^2 * (G - 1) * G + c * G * A + A * L$	G
ExtendedMCF	1	2	G	$c * (G - 1) * G + c * G * A + A * L$	G

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. The cost of such an approach depends on the choice of consensus algorithm used inside each group. 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 is the case, for example, when solving atomic broadcast; solutions based on consensus require each command proposed but not decided to be proposed again in a new consensus instance. Because our protocols solve M-Consensus instead, all proposals may be part of a single decision.

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 [38, 39]. 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 of 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 [31].

Steward [2] solves Byzantine atomic broadcast among agents organized in groups; to tolerate Byzantine failures Steward assumes the availability of a Byzantine agreement protocol inside each group to make the group behave as a single agent in a Paxos instance among the groups, and as such requires $3f + 1$ agents inside each group, where f is the maximum number of Byzantine failures tolerated inside that group. Our protocols do not tolerate Byzantine failures and, therefore, only require $2f + 1$ agents inside each group, if group coordquorums are defined as any majority of the coordinators of group for a given round.

Because each group behave as single agent in Steward, less intergroup messages are needed when compared to our protocols, which require each coordinator to send its own messages outbound its group. That is, our protocols trade a weaker failure model and extra intergroup messages for the need to reach agreement inside a group for every proposed message. Since the extra intergroup messages are sent in parallel, no extra delay is added. Moreover, because our protocols extend CFPaxos and not Paxos, they end up delivering more proposals per intergroup agreement than Seward's, finally resulting in a similar intergroup message complexity but with a much smaller delay for delivering each proposed command. (Remember that our protocol decides on a v-map of proposals, not on a single proposal.)

In Table 1 we compare the message and time complexity of our protocols and the Paxos-like related ones; we omitted Kooh and Haddad's protocol because its analysis depends on the consensus protocol employed, and also Steward because it solves a different problem, Byzantine Atomic Broadcast. Given the particularities of each protocol and the many execution scenarios, we restrict the analysis to a simple sample scenario, with the topology described below, in the absence of failures, and for phase 2 only, given that phase 1 may be executed a priori and has no impact in the absence of failures.

- There are G groups.
- There are C coordinators in the round, and each group has $c = C/G$ coordinators.
- There are A acceptors.
- There are L learners.
- 1 proposal is made in each group in the round.

The table shows the inter- and intra-group delays and the number of inter- and intra-group messages for each protocol. The table also shows the number of values, out of the G proposals, that are decided at the end of the round. As shown by the values of the last column, simply executing multiple instances of Paxos to achieve G decisions requires $(G-1)*A*L$ more messages and $(G-1)*3$ more communication delays than CFPaxos. If the needed Paxos instances are executed in parallel and messages aggregated, then numbers similar to CFPaxos could be achieved. However, failures in each instance would have to be handled individually

or at the expense of greater complexity. CFPaxos and the MCF both decide on G values, but the BasicMCF protocol requires an order of c^2 times more messages than CFPaxos, the price for the extra availability provided by multicoordination. With ExtendedMCF, the c^2 factor is reduced to c. It also reduces any extra delay due to the higher number of messages in the system.

7 Conclusion

Multicoordination is a mode of execution for agreement protocols. This mode has advantages of both fast and classic execution modes, namely, not dependency on a single leader and small acceptor quorums. While it does require more coordinator agents to participate in the protocol, these are simpler and easily replaceable, unlike acceptors.

In this paper we have reviewed previous works on multicoordination, aggregating their results in a single place. Moreover, we presented a novel protocol for reaching agreement between agents organized in groups. Such a scenario reflects the network topologies that are commonplace in many organizations, such as online retailers.

A general drawback of multicoordination with respect to the other execution modes is its increased message complexity: the number of messages from coordinators to acceptors is multiplied by the number of coordinators in each round. This is the reason why multicoordination should not be blindly applied. Instead, it should be employed by adaptable protocols capable of changing modes to adapt to changes of the environment and executing in multicoordinated mode only when appropriate.

As our next steps we plan to study, develop, and experiment with adaptation policies for multi-mode agreement protocols. This is an important step in the development of truly adaptable protocols. Moreover, we would like to explore the multicoordinated mode as a way to mask Byzantine failures.

Acknowledgements The authors would like to thank the CNPq, FAPESP, and the Hasler Foundation (project #2316), for their support, as well as the anonymous reviewers for their insightful feedback.

References

1. Aguilera M, Chen W, Toueg S (1998) Failure detection and consensus in the crash–recovery model. In: Proceedings of the 12th international symposium on distributed computing, September 1998
2. Amir Y, Danilov C, Dolev D, Kirsch J, Lane J, Nita-Rotaru C, Olsen J, Zage D. Steward: scaling Byzantine fault-tolerant replication to wide area networks. IEEE Trans Depend Secure Comput 99(2):5555
3. 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. ACM, New York, pp 27–30
4. Bracha G, Toueg S (1983) Resilient consensus protocols. In: PODC ’83: proceedings of the second annual ACM symposium on principles of distributed computing. ACM, New York, pp 12–26
5. Camargos L, Madeira E, Pedone F (2006) Optimal and practical WAB-based consensus algorithms. In: Euro-Par 2006 parallel processing. Lecture notes in computer science, vol 4128. Springer, Berlin, pp 549–558
6. Camargos L, Pedone F, Schmidt R (2006) A primary-backup protocol for in-memory database replication. In: NCA ’06: proceedings of the fifth IEEE international symposium on network computing and applications. IEEE Computer Society, Washington, pp 204–211
7. Camargos L, Schmidt R, Pedone F (2006) Multicoordinated Paxos. Technical Report 2006/2, EPFL and University of Lugano, 2006
8. Camargos L, Schmidt R, Pedone F (2007) Multicoordinated Paxos: brief announcement. In: PODC ’07: proceedings of the twenty-sixth annual ACM symposium on principles of distributed computing. ACM, New York, pp 316–317
9. Camargos L, Schmidt R, Pedone F (2008) Multicoordinated agreement protocols for higher availability. In: NCA ’08: proceedings of the seventh IEEE international symposium on network computing and applications. IEEE Computer Society, Washington
10. Castro M, Liskov B (1999) Practical Byzantine fault tolerance. In: OSDI ’99: proceedings of the third symposium on operating systems design and implementation. USENIX Association, Berkeley, pp 173–186
11. Chandra TD, Hadzilacos V, Toueg S (1996) The weakest failure detector for solving consensus. J ACM 43(4):685–722
12. Chandra TD, Toueg S (1996) Unreliable failure detectors for reliable distributed systems. Commun ACM 43(2):225–267
13. Cristian F, Fetzer C (1999) The timed asynchronous distributed system model. IEEE Trans Parallel Distrib Syst 10(6):642–657
14. Dolev D, Dwork C, Stockmeyer L (1987) On the minimal synchronism needed for distributed consensus. J ACM 34(1):77–97
15. Dutta P, Guerraoui R (2002) Fast indulgent consensus with zero degradation. In: Lecture notes in computer science, vol 2485. Springer, Berlin
16. Dwork C, Lynch N, Stockmeyer L (1988) Consensus in the presence of partial synchrony. J ACM 35(2):288–323
17. Fischer M, Lynch N, Paterson M (1985) Impossibility of distributed consensus with one faulty process. J ACM 32(2):374–382
18. Hurfin M, Mostefaoui A, Raynal M (1998) Consensus in asynchronous systems where processes can crash and recover. In: Proceedings seventeenth IEEE symposium on reliable distributed systems. IEEE Computer Society, Los Alamitos, pp 280–286
19. Hurfin M, Mostéfaoui A, Raynal M (2002) A versatile family of consensus protocols based on Chandra–Toueg’s unreliable failure detectors. IEEE Trans Comput 51(4):395–408
20. Hurfin M, Raynal M (1999) A simple and fast asynchronous consensus protocol based on a weak failure detector. Distrib Comput 12(4):209–223
21. Kooh N, Haddad S (1999) Reaching agreement in hierarchical groups. In: Proceedings of the 12th international conference on parallel and distributed computing systems. IASTED Press, Fort Lauderdale
22. Lamport L (1978) Time, clocks, and the ordering of events in a distributed system. Commun ACM 21(7):558–565
23. Lamport L (1998) The part-time parliament. ACM Trans Comput Syst 16(2):133–169
24. Lamport L (2001) Paxos made simple. ACM SIGACT News 32(4):18–25

25. Lamport L (2004) Generalized consensus and Paxos. Technical Report MSR-TR-2005-33, Microsoft Research
26. Lamport L (2006) Fast Paxos. Distrib Comput 19(2):79–103
27. Lamport L (2006) Lower bounds for asynchronous consensus. Distrib Comput 19(2):104–125
28. Lampson B (2001) The abcd's of Paxos. In: PODC '01: Proceedings of the twentieth annual ACM symposium on principles of distributed computing. ACM, New York
29. Martin JP, Alvisi L (2006) Fast Byzantine consensus. IEEE Trans Dependable Secure Comput 3(3):202–215
30. Pedone F, Guerraoui R, Schiper A (2003) The database state machine approach. Distrib Parallel Databases 14(1):71–98
31. Pedone F, Schiper A (1999) Generic broadcast. In: Proceedings of the 13th international symposium on distributed computing (DISC'99, formerly WDAG)
32. Pedone F, Schiper A (2002) Handling message semantics with generic broadcast protocols. Distrib Comput 15(2):97–107
33. Pedone F, Schiper A, Urbán P, Cavin D (2002) Solving agreement problems with weak ordering oracles. In: EDCC-4: proceedings of the 4th European dependable computing conference on dependable computing. Springer, London, pp 44–61
34. Pedone F, Schiper A, Urbán P, Cavin D (2002) Weak ordering oracles for failure detection-free systems. In: Proceedings of the international conference on dependable systems and networks (DSN), supplemental volume
35. Rabin MO (1983) Randomized Byzantine generals. In: Proceedings of the 24th annual IEEE symposium on foundations of computer science, pp 403–409
36. Schiper A (1997) Early consensus in an asynchronous system with a weak failure detector. Distrib Comput 10(3):149–157
37. Schmidt R, Camargos L, Pedone F (2007) On collision-fast atomic broadcast. Technical report, EPFL
38. Sousa A, Pereira J, Moura F, Oliveira R (2002) Optimistic total order in wide area networks. In: Proceedings of the 21st IEEE symposium on reliable distributed systems. IEEE Computer Society, New York, pp 190–199
39. Vicente P, Rodrigues L (2002) An indulgent uniform total order algorithm with optimistic delivery. In: Proceedings of the 21st symposium on reliable distributed systems, Osaka University, Suita, Japan. IEEE, New York, pp 92–101
40. Zielinski P (2004) Paxos at war. Technical Report UCAM-CL-TR-593, University of Cambridge, Computer Laboratory