Multicoordinated Agreement Protocols for Higher Availability

Lásaro Camargos [†⋆] lasaro@ic.unicamp.br

Rodrigo Schmidt [‡] rodrigo.schmidt@epfl.ch

Fernando Pedone [⋆] fernando.pedone@unisi.ch

[†] Universidade Estadual de Campinas (Unicamp), Brazil
[‡] École Polytechnique Fédérale de Lausanne (EPFL), Switzerland
[⋆] University of Lugano (USI), Switzerland

Abstract

Adaptability and graceful degradation are important features in distributed systems. Yet, consensus and other agreement protocols, basic building blocks of reliable distributed systems, lack these features and must perform expensive reconfiguration even in face of single failures. In this paper we describe multicoordinated mode of execution for agreement protocols that has improved availability and tolerates failures in a graceful manner. We exemplify our approach by presenting a Generic Broadcast algorithm. Our protocol can adapt to environment changes by switching to different execution modes. Finally, we show how our algorithm can solve the Generalized Consensus and its many instances (e.g., consensus, atomic broadcast, reliable broadcast).

1. Introduction

Distributed systems can provide higher availability than centralized ones by remaining functional despite failures of some of their components. Ensuring application-level consistency in a distributed system often requires agreement among the components.

Consensus is a common denominator of many agreement problems. Given a set of processes, some of which willing to make a proposal (e.g., what step to take next, what value set a variable to), solving consensus requires them to agree on one of the proposals. Despite the multitude of consensus algorithms, they all have similar execution patterns: They run successive rounds, each of which tries to reach the decision. When a round fails to lead to a decision, or so it seems to some processes, a new round is started to continue the procedure. If a decision is reached in a round, then successive rounds can only abide to such decision.

Some protocols execute their rounds in a coordinated way, resorting to a *leader* process to start and control each round [7, 12, 13]. Leaders must be monitored and replaced in case of failures to prevent blocking. On the one hand, aggressive failure detection allows skipping stuck rounds rapidly. On the other hand, it may incur in unnecessary rounds being started, an expensive approach since, for each new round, the leader must poll other processes to detect previous decisions.

To avoid handling leader failures, some protocols let processes skip the leader when proposing (e.g., [13, 4]). In the absence of failures and if a single proposal is issued, then such protocols can implement *fast* rounds, which save one communication step with respect to classic rounds. If multiple proposals are issued in parallel, however, real or apparent ties may prevent processes from deciding in a fast round even in the absence of failures. Moreover, the simple possibility of *conflicts*, as we call such concurrent proposals, imposes a lower resilience to failures if compared to classic rounds, because more processes are required to be functional to brake a tie [15].

Hence, both classic and fast protocols may pose problems to availability in that they are either too sensitive to the failure or suspicion of leaders or impose a lower overall resilience. In this paper we advocate a third approach which removes the dependency on the leader without decreasing the resilience of the protocol by using *coordinators* instead of a leader. In each round, multiple coordinators share the job of the leader, allowing a round to continue even if some coordinators fail. Hence, availability in these *multicoordinated* rounds does not depend on fast reaction to failures, and failure detection can use larger timeouts for higher precision [8] and minimize useless round changes. The price for the improved availability is extra communication: multicoordinated rounds have one communication step more than fast rounds, and each coordinator of a round may send as many messages as the leader in a classic round. No extra stable storage access is needed.

Although less costly than in fast rounds, collisions in multicoordinated rounds may also prevent the round from deciding even in stable periods of execution (i.e., no failures or suspicions). However, consensus is often used in a sequence of instances, not individual ones (e.g.[3, 16]). On implementing the State Machine Replication [11], for example, one can use consensus to agree on each command of the sequence to be executed by every replica [16, 12]. In many systems, however, commands may commute and there is no need for totally ordering them since the final state is the same independently of the order in which they are applied. Simple consensus, as applied to state machine replication (i.e., in a sequence), is too strong to capture this notion and a collision may happen even if the two concurrent proposals are commutable. In a different problem, namely Generic Broadcast [18], processes can agree on sequences of commands while taking into account the commutability of commands to mitigate the effects of collisions.

In this paper we present a multicoordinated generic broadcast protocol. In fact, because different round types may suit better different scenarios (e.g, if conflicts are rare and nodes reliable, than fast rounds may be better than classic and multicoordinated rounds), we have made generic broadcast protocol multimode, that is, it can switch between multicoordinated, fast, and classic modes at runtime, and use that to adapt to changes in the environment. To the best of our knowledge, Fast Paxos, which had classic and fast rounds [13], was the first multimode agreement protocol.

In the next section we discuss the Generic Broadcast problem in asynchronous systems. In Section 3 we present our algorithm and in Section 4 we discuss some of its practical aspects. Generic Broadcast is an instantiation of the even more general agreement problem named Generalized Consensus [13]. We discuss how our generic broadcast algorithm can be turned into a Generalized Consensus protocol in Section 5, and discuss other related works in Section 5. We conclude the paper in Section 6.[1]

2. Generic Broadcast

2.1. Overview

The Generic Broadcast problem can be described in terms of a set of *learner* processes progressively learning about commands broadcast by a set of *proposer* processes. In the context of a distributed application, *proposers* can be thought of as clients issuing commands and *learners* as the application servers executing the commands they learn. Clients might also be learners to know whether their issued commands were accepted by the system.

In this problem, processes agree on a partially ordered set (poset) of proposed commands. Commands are added to the poset concurrently while learners learn increasing prefixes of it. The partial order abides to a single rule: conflicting commands must be ordered, where conflicts are determined by the application. If all or no commands conflict, then generic broadcast degenerates to Atomic Broadcast or to Reliable Broadcast [10], respectively.

Consider a replicated application in which learner replicas execute commands according to the prefix they have learned. Notice that, because processes agree on a poset, the learned prefixes need not be prefixes of each other. If all non-commutable commands conflict, then replicas may get to different state, but they will always be conciliable. That is, consistency is ensured because non-commutable commands are executed in the same order by all replicas.

Briefly, the safety requirements of Generic Broadcast, which we formally restate later, are three: only proposed commands enter the poset (Nontriviality); no command is removed from the poset (Stability); and, the prefixes of the agreed poset seen by different learners at any point in time may differ, but can always be merged into a larger prefix (Consistency).

The first two properties are self-explanatory. The third one, Consistency, assert that however the states of two learners diverge by learning commands in different orders, they can be equaled by learning about the commands seen by the other. That is, no unsolvable conflict ever happens.

The liveness requirement is more complicated since it should not prevent progress if any subset of proposers or learners fail. Recall that such roles can be assigned to clients and we cannot require them not to fail. Therefore, we introduce another set of processes, about which we can make reliability assumptions; we call these processes the *acceptors*. We call a *quorum* any finite set of acceptors that is enough to allow liveness, and define the liveness requirement of Generic Broadcast as follows: for any proposer p and learner l, if p, l, and a quorum Q of acceptors are nonfaulty and p proposes a command C, then l eventually learns about C (Liveness).

2.2. Model

We assume an asynchronous crash-recovery model in which processes communicate by exchanging messages with no bounds on the time it takes for messages to be transmitted or actions to be executed. Messages can be lost or

[1]We stated part of the results presented here in a Brief Announcement [6].

duplicated but not corrupted; processes can fail by stopping only and never perform incorrect actions. Processes are assumed to have some sort of local stable storage to keep part of their state in between failures. Although we assume processes may recover, they are not obliged to do so once they have failed. For simplicity, a process is considered to be nonfaulty iff it never fails.

The well-known FLP result [9] states that under such circumstances no consensus algorithm can ensure a liveness property similar to ours if quorums are defined to tolerate the failure of any single acceptor. Notice that by defining all commands as conflicting in Generic Broadcast we force all learners to learn the exact same sequence of commands. If learners ignore all but the first command in such a sequence, then they will be solving the consensus problem. Hence, the FLP result has the same implications in Generic Broadcast. As a result, fault-tolerant algorithms must make extra assumptions about the system. We discuss the extra assumptions we make in Section 4.2.

To properly define Generic Broadcast in our formalism, we must first further describe the posets that are central to the problem and the operations we use to manipulate them. In Section 5 we show how these posets are just an instantiation of a more general data structure, C-Structs [13], and how our algorithm can be used solve a more general problem, namely, Generalized Consensus [13].

2.3. Command Histories and Formal Definition

Let Cmd be the set of commands proposable in Generic Broadcast and $\asymp$ a reflexive and symmetric conflict relation over the commands of Cmd. That is, $\forall C, D \in Cmd$, C conflicts with D and vice versa iff $C \asymp D$. We call the partially ordered set $(S, \prec)$ a command history, or c-hists for short, iff (i)$S \subseteq Cmd$ and (ii)$\forall C, D \in S$, $C \asymp D \Rightarrow (C \prec D$ or $D \prec C)$. We define $\bullet$ as an operator that appends a command from Cmd to a command history respecting the conflict relation $\asymp$. The operator $\bullet$ is defined for sequences of commands as the ordered application of $\bullet$ to each command in the sequence. More formally, if $(S, \prec)$ is a c-hist and $\langle C_1, \ldots, C_m \rangle$ is a sequence of commands in Cmd, then:

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

and

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

We represent the empty c-hist by $\perp$ and say that c-hist g extends c-hist h (or that h is a prefix of g), noted $h \sqsubseteq g$, if there exists a sequence σ of commands such that $g = h \bullet \sigma$. Hence, $\perp \sqsubseteq h$ for any c-hist h. Moreover, given a set T of c-hists, we say that c-hist h is a lower bound of T iff $h \sqsubseteq g$ for all g in T. A greatest lower bound (glb) of T is a lower bound h of T such that $g \sqsubseteq h$ for every lower bound g of T, and we represent it by $\sqcap T$. In other words, $\sqcap T$ is the largest prefix of the c-hists in T. Similarly, we say that h is an upper bound of T iff $g \sqsubseteq h$ for all g in T. A least upper bound (lub) of T is an upper bound h of T such that $h \sqsubseteq g$ for every upper bound g of T, and we represent it by $\sqcup T$. That is, h is a c-hist of which all c-hists in T are prefixes. If $\sqsubseteq$ is a reflexive partial order on the set of c-hists and a glb or lub of T exists, then it is unique. For simplicity of notation, we use $g \sqcap h$ and $g \sqcup h$ to represent $\sqcap\{g, h\}$ and $\sqcup\{g, h\}$, respectively. Two command histores g and h are defined to be *compatible* iff they have a common upper bound, and a set S of c-hist is compatible iff its elements are pairwise compatible. (One could think of $\sqcap$ and $\sqcup$ *roughly* as the intersection and the union of c-hists.) As an example of the meaning of such definitions, consider the following set of c-hists, where $a \leftarrow b$ means $a \prec b$.

$$T = \left\{ \begin{tikzcd}
a & c \arrow[l] & b \arrow[ll, bend right] & d \arrow[l] & \\
a & & b \arrow[ll] & d \arrow[l] & \\
a & c \arrow[l] & b \arrow[ll, bend right] & & e \arrow[lll, bend right]
\end{tikzcd} \right\}$$

The lower bounds of T are $\{\perp, a, a \leftarrow b\}$, being $\sqcap T = a \leftarrow b$ (the glb of T) and $\sqcup T =$ $\begin{tikzcd} a & c \arrow[l] & b \arrow[ll, bend right] & d \arrow[l] & e \arrow[lll, bend right] \end{tikzcd}$ (the lub of T).

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

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

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

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

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

3. Multicoordinated Generic Broadcast

3.1. About Rounds and Quorums

We present our Multicoordinated Generic Broadcast algorithm (MGB) in the next section. It is given as a set of atomic actions, organized and executed in rounds; MGB assumes an unbounded number of rounds, totally ordered by a relation $<$. For simplicity, it can be assumed that round numbers correspond to the set of natural numbers. We show how to benefit from different definitions of round numbers in Section 4.3. Although there is a total order among rounds, their execution does not have to follow it, and actions referring to different rounds can be interleaved.

For each round, an acceptor can "accept" one c-hist and then extend it with other commands. The purpose of a round is to get c-hists with the same prefix accepted by a quorum of acceptors, a situation in which we say the prefix has been *chosen*. The algorithm guarantees that if a prefix is chosen at some round, then any c-hists accepted and chosen in any later rounds are extensions of such a prefix. Therefore, a learner can safely learn a c-hist h as soon as it knows that h has been chosen. To prevent two incompatible prefixes from being chosen, we require quorums of acceptors to have non-empty intersections.

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

Intuitively, this assumption forbids two unconnected partitions of our acceptors to concurrently choose different c-hists. In fact, this requirement is very general and any general algorithm for asynchronous consensus must satisfy a similar one, as shown in [15]. A simple way to ensure this is to define quorums as any majority of the acceptors.

Each round is divided into two phases. The first phase serves to identify previously chosen command histories and the second phase to extend these histories with new commands. To orchestrate rounds, MGB assumes a set of *coordinator* processes, besides proposers, acceptors, and learners. We classify rounds in *classic*, *fast*, and *multicoordinated* according to the role played by coordinators in the round.

Classic rounds have a single coordinator, responsible for starting the round by triggering the first phase, and by forwarding the commands from proposers to be accepted by the acceptors in the second phase. Hence, a proposal requires at least three communication steps to enter a chosen prefix and be learned in a classic round: one step to reach the coordinator, and two to finish the second phase of the round. Figure 1(a) illustrates the these steps.

In *fast* rounds, after executing phase one the coordinator can inform the acceptors about which c-hist may already be chosen and tell them to extend this c-hist themselves with proposals received directly from the proposers and accept them. This way, the latency to get a command chosen and learned is reduced to two communication steps. Figure 1(b) illustrates the second phase of a fast round. Since acceptors can extend the c-hist identified in the first phase in different ways, they may end up accepting incompatible extended c-hists in fast rounds. Assumption 1 ensures that no two incompatible extensions will be chosen, but it is not enough allow identifying if one of them has been. Figure 1(e) shows a round in which a learner cannot determine if a or b have been chosen after having received the acceptance information from just two acceptors. If the third acceptor crashes before sending this information, then no agreement can be reached. To avoid this case we need a stronger assumption regarding quorum intersections. To state such an assumption, we must define quorums per round of the algorithm; a quorum for round i an is i-quorum.

Assumption 2 (Fast Quorum Requirement) *For any rounds i and j:*

(i) If Q is an i-quorum and S is a j-quorum, then $Q \cap S \neq \emptyset$.

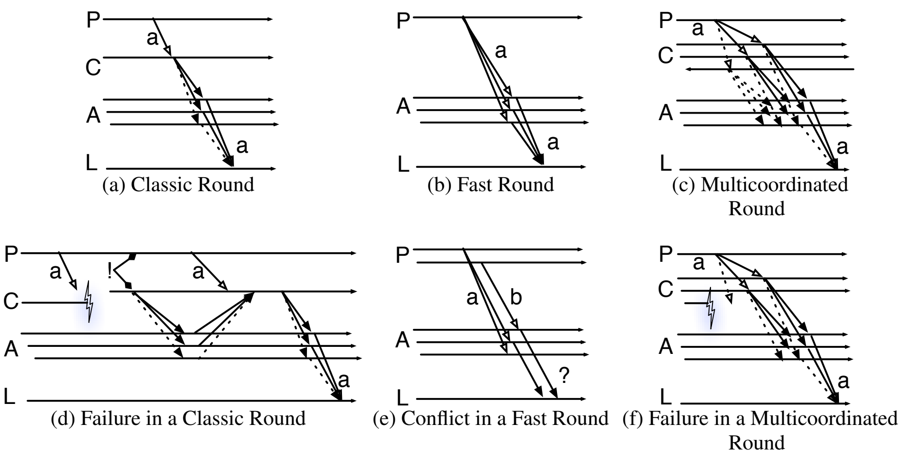
(ii) If Q is an i-quorum, R and S are j-quorums, and j is a fast round, then $Q \cap R \cap S \neq \emptyset$.

In the general case, this stronger assumption requires bigger quorums. These constraints are achieved, for example, if every set of $\lceil (2n+1)/3 \rceil$ acceptors is a quorum for fast and classic rounds, or classic and fast quorums, for short. If classic quorums are defined to be any majority of acceptors, fast quorums must be as big as $\lceil (3n+1)/4 \rceil$ acceptors. It has been shown, however, that any asynchronous consensus protocol that allows a decision to be reached in two communication steps must satisfy similar quorum requirements [15] (Fast Learning Theorem). Hence, the same is valid for Generic Broadcast, as previously noted by Pedone and Schiper [19].

In *multicoordinated* rounds, multiple coordinators cooperate in orchestrating the execution, grouped in quorums of coordinators per round. For simplicity, we call a quorum of coordinators for round i an i-coordquorum and say that c is a coordinator of round i iff it belongs to an i-coordquorum. As in classic rounds, coordinators pre-accept commands by extending the c-hists they had forwarded to acceptors, and forward the extended histories to the acceptors. Acceptors can only accept an extended c-hist if it is a prefix of the c-hists coming from a quorum of coordinators for that round so that no two acceptors will accept two incompatible sequences in the same multicoordinated round. For this to be guaranteed, the following requirement is needed.

Assumption 3 (Coord-quorum Requirement) *For any classic round i, if L and P are i-coordquorums, then $L \cap P \neq \emptyset$.*

Figures 1(c) and (f) show a round with three coordinators, where any two of them form a coordquorum. It is easy to see that a classic round is a multicoordinated round with a single coordquorum of cardinality one.



P
C
A
L
a
b
?
!
(a) Classic Round
(b) Fast Round
(c) Multicoordinated Round
(d) Failure in a Classic Round
(e) Conflict in a Fast Round
(f) Failure in a Multicoordinated Round

Figure 1. Message patterns in different round types. P, C, A, and L respectively stand for *Proposer*, *Coordinator*, *Acceptor*, and *Learner*. The lightning symbol and the exclamation mark represents a crash and its detection. "a" and "b" are proposals sent in the messages.

Figure 2 shows a more complex run, in which we have merged some messages not to clutter the picture. Consider that commands with the same shape *or* color do commute. In the run, the first three commands ($\Box, \triangledown, \circ$) are chosen and learned without conflicts. The fourth command, ▼, conflicts with the first and second ones (▼ $\asymp \circ$ and ▼ $\asymp \Box$). Because both the first and second coordinators had seen $\Box$, they append ▼ after $\Box$ in their c-hists. However, because the first coordinator has not seen $\circ$, its c-hist will not be compatible with the c-hist sent by the second coordinator to the acceptors, which has $\circ$ before ▼. (The $\triangledown$ command is not important here, since it commutes with ▼.) Had both coordinators heard of $\circ$, their c-hists would be compatible and accepted by the acceptors. Since that is not the case, a new round to solve the conflict is needed; in the example, we have shown a classic round being started. Observe how learners learn different but compatible prefixes. The dashed arrows show the leader polling the acceptors for their accepted c-hists.

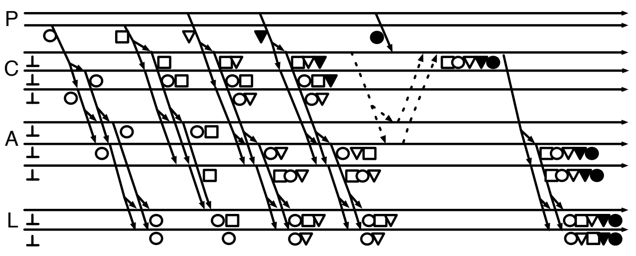


Figure 2. Multicoordinated round followed by a classic round.

3.2. The Algorithm

MGB is presented in Algorithm 1. Before explaining it we must note two things. First, to simplify the presentation, we have defined quorums in terms of their cardinalities. Let n be the number of acceptors in the system, F be the maximum number of acceptor failures that does not prevent progress, and E be the maximum number of acceptor failures that still allows fast termination. Acceptor quorums are defined as any set of at least $n - F$ acceptors, and fast acceptor quorums are defined as any set of at least $n - E$ acceptors. As explained in Section 3.1, as long as $2E + F < n$, Assumptions 1 and 2 are satisfied [15]. As for coordinators, we let any set with a majority of coordinators be a coordquorum, which trivially satisfies Assumption 3. Second, we have considered generic implementations of c-hists. In [5] we provide an implementation of c-hists and the $\bullet$ operator for this implementation.

The algorithm is given as the set of atomic actions executed by process p. Actions are associated to roles and can only be executed by processes playing such a role; action *Propose*, for example, is executed by p if it is a proposer. Each action defines a set of pre-conditions (cond) and a set of sub-actions (sub). Sub-actions are only executed if all pre-conditions are satisfied. Moreover, we assume that if an action is enabled, *i.e.*, all of its pre-conditions are true, then it is eventually executed. An empty set of pre-conditions is always satisfied. We explain each action below.

Processes keep their state in a set of variables defined *per-role*. For now, we assume that all variables are stably stored. A coordinator c keeps two variables:

$crnd[c]$ is the current round of c, initially 0.

$cval[c]$ is the latest c-hist c has sent in a phase "2a" message (sent in round $crnd[c]$). It is initially $\perp$ for all coordinators.

An acceptor a keeps three variables:

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

$vrnd[a]$ is the round at which a has accepted a value for the last time. Initially 0.

$vval[a]$ is the c-hist a has accepted at round $vrnd[a]$. Initially the empty c-hist $\perp$.

According to the acceptors' initialization, learners will always learn command histories with $\perp$ as the smallest element. Therefore, $learned[l]$ initially equals $\perp$, for every learner l in the system.

Proposers execute action $Propose(C)$ to propose a new command C. This is done by sending the message $\langle$"propose", $C\rangle$ to acceptors and coordinators.

If a coordinator wants to start a round i, bigger than its current round and such that it belongs to an i-coordquorum, then it executes action $Phase1a(i)$. The action sends a message $\langle$"1a", $i\rangle$ to all acceptors asking them to take part in round i.

Acceptors execute action $Phase1b(i)$ to join a round i. The action is enabled iff the acceptor is currently at a lower round and has received a message $\langle$"1a", $i\rangle$ from a coordinator. In response, the acceptor joins the round by setting $rnd[p]$ to i and sending a message $\langle$"1b", $i, vval[p], vrnd[p]\rangle$ to all coordinators of the round. The pre-condition of this action ensures that after being executed for round i, the same acceptor will not execute it for any round $j \leq i$.

Any coordinator can execute action $Phase2Start(i)$ if its current round is smaller than i and it has received an "1b" message for round i from all acceptors in a quorum Q of $n - F$ acceptors. In the action, the coordinator selects a c-hist val to forward to the acceptors using the function $SelectVal()$. val is selected based on the histories already accepted by the acceptors in Q and informed in their "1b" messages. val is chosen such that it extends any c-hist that may have been decided in a round $j < i$. The exact implementation of $SelectVal$ is discussed in Section 3.3.

$Phase2Start$ sets $cval[p]$ to the selected c-hist and $crnd[p]$ to i and, due to the action's pre-condition, is executed only once per *round*. The coordinator then sends a message $\langle$"2a", $i, cval[p]\rangle$ to all acceptors.

After executing $Phase2Start$, if a coordinator receives new commands from proposers, it can extend its c-hist for the current round and ask the acceptors to accept this extended history. It does so by executing action

Algorithm 1 Multicoordinated Generic Broadcast Protocol of process p.

```
$Propose(C)$ [Proposer]
  act: • send $\langle$"propose", $C\rangle$ to acceptors and coordinators
$Phase1a(i)$ [Coordinator]
  cond: • $p$ is in some $i$-coordquorum and
        • $crnd[p] < i$.
  act:  • send $\langle$"1a", $i\rangle$ to acceptors
$Phase1b(i)$ [Acceptor]
  cond: • $rnd[p] < i$.
        • $p$ has received a message $\langle$"1a", $i\rangle$
  act:  • $rnd[p] \leftarrow i$
        • send $\langle$"1b", $i, vval[p], vrnd[p]\rangle$ to coords of round $i$.
$Phase2Start(i)$ [Coordinator]
  cond: • $crnd[p] < i$ and
        • $c$ received a "1b" message for
          round $i$ from all acceptors
          in a set $Q$ of $n - F$ acceptors.
  act:  • $cval[p] \leftarrow SelectVal()$
        • $crnd[p] \leftarrow i$
        • send $\langle$"2a", $i, cval[p]\rangle$ to the acceptors
$Phase2aClassic()$ [Coordinator]
  cond: • $p$ has received a $\langle$"propose", $C\rangle$ message.
  act:  • $cval[p] \leftarrow cval[p] \bullet C$
        • send $\langle$"2a", $crnd[p], cval[p]\rangle$ to acceptors
$Phase2bClassic(i)$ [Acceptor]
  cond: • $rnd[p] \leq i$,
        • $\exists i$-coordquorum $L$ such that:
           $\forall c \in L$, received a "2a" message for round $i$ from $c$:
             if $i$ is multicoordinated
               then $L$ has a majority of the coordinators
             if $i$ is fast or classic
               then $L$ has size one
        • $vrnd[p] < i$ or $vval[p]$ is compatible with $\sqcap L2aVals$,
          where $L2aVals = \{v : p$ received $\langle$"2a", $i, v\rangle$
          from $c$ and $c \in L\}$
  act:  • if $vrnd[a] = i$ then $vval[a] = vval[a] \sqcup (\sqcap L2aVals)$
                             else $vval[a] = \sqcap L2aVals$.
        • $vrnd[a] \leftarrow i$
        • $rnd[a] \leftarrow i$
        • send $\langle$"2b", $i, vval[a]\rangle$ $vval[a]$ to learners
$Phase2bFast()$ [Acceptor]
  cond: • $rnd[p]$ is a fast round,
        • $rnd[p] = vrnd[a]$, and
        • $a$ has received a $\langle$"propose", $C\rangle$ message.
  act:  • $vval[p] \leftarrow vval[p] \bullet C$
        • send $\langle$"2b", $vrnd[p], vval[p]\rangle$ to learners
$Learn()$ [Learner]
  cond: • $p$ has received "2b" messages for some round $i$ from
          all acceptors in some set $Q$ of acceptors:
           if $i$ is classic or multicoordinated
             then $Q$ has cardinality $n - F$
           if $i$ is fast
             then $Q$ has cardinality $n - E$
  act:  • $learned[p] \leftarrow learned[p] \sqcup (\sqcap Q2bVals)$
          where $Q2bVals$ is the set of values received in the
          "2b"messages received from acceptors in $Q$.
```

$Phase2aClassic()$, which sets $cval[p]$ to $cval[p] \bullet C$ and sends a $\langle$"2a", $crnd[p], cval[p]\rangle$ message to the acceptors.

By executing action $Phase2bClassic(i)$, acceptors accept c-hists forwarded by the coordinators in round i. This action is enabled iff the acceptor is not in a higher round and has received a "2a" message for round i from all coordinators in i-coordquorum L. That is, if i is a classic round, then L has a majority of the coordinators; if i is a fast or classic round, then L has size one. Moreover, either the acceptor has not accepted anything in round i yet, or its previously accepted value can be extended to a prefix of the received command histories, that is, the accepted value is compatible with the glb of the received c-hists. After accepting a c-hist, acceptors inform the learners with a "2b" message so that they can extend what they have learned.

When in a fast round, acceptors can also extend their accepted c-hists with commands received directly from proposers. This is done in action $Phase2bFast()$ which simply appends the c-hists with a command and informs the learners, as in action $Phase2bClassic$.

Finally, learners execute action $Learn()$ when they have received "2b" messages for some round i from all acceptors in some quorum Q of acceptors; Q has cardinality $n - F$ if i is a classic or multicoordinated round and cardinality $n - E$ if i is a fast round. When it executes this action, learner p extends its learned c-hist $learned[p]$ to $learned[p] \sqcup (\sqcap Q2bVals)$, where $Q2bVals$ is the set of values received in the "2b" messages from acceptors in Q.

3.3. Selecting val in $Phase2a$

We now define how a coordinator c picks the value to be forwarded to acceptors on action $Phase2Start$ in some round i. In the description of the procedure, we refer to the third and fourth fields of a message m =$\langle$"1b", $i, vval[a], vrnd[a]\rangle$ sent by acceptor a for round i as $m.vval$ and $m.vrnd$. We make the following definitions.

- Q is a set of acceptors of cardinality $n - F$, such that c has received a "1b" message from each acceptor in Q, for round i.
- $1bQ$ is the set of "1b" messages received by c, from all acceptors in Q, in round i.
- k is the maximum element in the set $\{m.vrnd : m \in 1bQ\}$ of round numbers in the received "1b" messages.
- $kacceptors$ is the set of acceptors in Q from which c has received a message m such that $m.vrnd = k$.
- $vals(S)$ is the set $m.vval$ for all messages $m \in 1bQ$ received from all acceptors in a set S.

If k is a classic or multicoordinated round, then any subset of $kacceptors$ with $n - 2F$ elements could combine with the acceptors from which messages were not received to form quorum R. In this case, the acceptors in R could have chosen any prefix of the values accepted by the acceptors in $R \cap Q$. Let $InterAtk$ be the set of all such subsets, that is, subsets of $kacceptors$ with cardinality $n - 2F$. If $InterAtk$ is empty, then c can choose any message m from an acceptor in $kacceptors$, and forward any extension of $m.vval$ to the acceptors. Otherwise, c must forward an extension of $\sqcup\Gamma$, where Γ is the longest prefix shared by acceptors in the subsets of $InterAtk$, i.e., $\Gamma = \{\sqcap vals(e) : e \in InterAtK\}$. In the case of simple majority quorums, that is, $F = \lfloor (n-1)/2 \rfloor$, $n - 2F = 1$ and Γ will equal the set of values in messages received from all acceptors in $kacceptors$, and the calculation of glbs can be skipped.

If k is a fast round, then the subsets of $kacceptors$ in $InterAtk$ must have cardinality $n - 2E$, as this is the minimum size of an intersection of two fast quorums. The rest of the procedure to pick a sequence to be extended remains the same.

3.4. Correctness

We have formally specified the Multicoordinated Generic Broadcast in the TLA$^+$ specification language and tested it for small setups with the TLC checker. For the general cases we devised detailed correctness proofs manually. Due to space constraints, we do not include neither the specification nor the proofs here. They are available in the extended version of this paper [5].

4. Practical Aspects

4.1. Collisions

In multicoordinated rounds, a collision happens when commands proposed concurrently arrive at the coordinators in different orders, leading to the forwarding of incompatible c-hists. If no quorum forwards c-hists whose glb can extend the values previously accepted by the acceptors, the round is stuck and no new command can get accepted.

This is a different type of collision than the one that may occur in fast rounds. In fast rounds, a collision happens when acceptors accept incompatible c-hists that cannot further extend the values learned so far. In this case, acceptors pay the price of accepting commands that will never be learned, which does not happen in collisions of multicoordinated rounds. This difference has important implications since acceptors must write on stable storage every time they accept a value but coordinators do not have to, as we explain in Section 4.3. In the following, we show a simple mechanism to deal with collisions in multicoordinated rounds.

First, collision identification must be done by the acceptors when they receive the phase "2a" messages from the coordinators. If two coordinators of the same i-coordquorum send "2a" messages for round i with incompatible c-hists, acceptors execute action $Phase1b(a, i+1)$ as if they had received a phase "1a" message for round $i+1$.

If round $i+1$ is classic or multicoordinated and enough acceptors identify the collision, which will normally happen if messages are not lost and processes do not crash, then the coordinators of round $i+1$ will execute action $Phase2Start(i+1)$ based on the received messages, followed by one or more executions of action $Phase2aClassic()$. Thus, the collision in round i will be resolved with only two extra communication steps (as compared to the usual three of a classic round). Clearly, to avoid that another collision happens when the coordinators start round $i+1$, it is advisable to have it as a single-coordinated round. After some time of normal execution, if conflicting commands stop being proposed, the coordinator of round $i+1$ can start a multicoordinated round again. This approach is a variation of the *coordinated recovery* presented in [14].

4.2. Liveness

Message losses are the first problem to be dealt with in order to ensure liveness. The solution to that is to have processes keep on re-sending their last message, which can be optimized as described in Section 2.4.1 of [14].

The possibility of starting new rounds allows the algorithm to progress if a round does not succeed because of coordinator crashes or collisions. However, their continuous initialization could prevent values from being accepted. In Classic Paxos and Fast Paxos this is prevented by using some (unreliable) leader election algorithm that eventually elects a single correct leader which will be responsible for starting a higher-numbered round under its coordination. In Multicoordinated Paxos, we use the same strategy to prevent the continuous initialization of new rounds.

If the current leader starts a new classic round (of which it is the only coordinator), liveness is ensured as long as the leader does not crash and other coordinators do not wrongly think they are the leader and try to start a higher-numbered round. If other coordinators interfere, the leader must be notified. This is done by extending the algorithm so that an acceptor replies to phase "1a" or "2a" messages with a round number lower than its current one. This notification tells the coordinator that its current round number is too low to get values accepted. When a coordinator that thinks to be the leader receives such a notification message from an acceptor, it can start a higher-numbered round.

If the leader starts a fast round, liveness is ensured as long as the leader does not crash during the execution of phase 1 and collisions do not happen during the rest of the round execution. Collisions can be resolved by starting a new classic single-coordinated round or using the techniques described in [14].

In multicoordinated rounds, liveness is ensured if the leader does not crash during the execution of phase 1, collisions do not happen, and some quorum of coordinators is available during the rest of the round execution. The failure of the leader is not a problem since another correct leader is eventually selected which will make sure that a new round starts. As for collisions, the mechanism presented in Section 4.1 can be used—the only restriction we make is that the leader must be one of the coordinators for the following round, otherwise the leader might think the round change is an interference and try a higher-numbered round. Last, to cope with the failure of coordinators, the leader must start a new round if it believes that other coordinators have failed. Their possible failure can be assessed by monitoring their "2a" messages or some external failure detection mechanism. When the leader suspects that there are not enough coordinators in the current round to ensure progress, it starts a new higher-numbered round with a different set of coordinator quorums.

4.3. Reducing Stable Storage Writes

We have assumed so far that processes keep their variables in stable storage to allow recovery from crashes. The most common type of stable storage is the magnetic disk, which has access times orders of magnitude bigger than volatile memory. Hence, it is very important to reduce the access to stable storage as much as possible. We show now how to reduce the stable storage requirements of our protocol.

Assumption 3 imposes no restriction between coordinator quorums of different rounds. If it is always possible to start new rounds with any set of coordinator quorums, coordinators are not required to store their variables on stable storage. A coordinator that crashes and later recovers could just be seen as a new coordinator in the system, which is easily implemented by having an "incarnation" counter associated with its identifier. In the following we explain how new rounds can be created with any set of coordinator quorums.

Round numbers can be constructed as a record of the form $\langle Count, Id, RType, S\rangle$, where $Count$ is a natural number, Id is a coordinator's unique identifier, $RType$ is a natural number, and S is a set of coordinator quorums. Counter $Count$ always allows the creation of a new higher-numbered round, Id identifies the coordinator that created the round, $RType$ tells the round type (fast if 0, classic or multicoordinated otherwise), and S identifies all valid coordinator quorums for this round. A round is uniquely identified by the first three fields, and the total order relation is given by comparing them lexicographically. Field S is merely informative and is not taken into consideration when comparing two rounds. Using this approach, when the current leader wants to start a new round, it can simply define the four fields according to its current knowledge.

Since Assumption 2 requires that quorums of different rounds intersect, acceptors cannot lose their state after a crash and assume a different identity upon recovery. This happens because the values accepted by acceptors cannot be forgotten, or the algorithm's safety would be compromised. Therefore, these values must always be stored in stable storage, incurring a disk write (or equivalent operation) whenever an acceptor executes a $Phase2b$ action. As a result, acceptors are not as easily replaceable as coordinators and more complex strategies must be used [12, 17].

Action $Phase1b$ also changes the internal state of an acceptor and, at a first sight, this seems to imply that $Phase1b$ must also write on stable storage. However, an acceptor a may store $rnd[a]$ only in volatile memory as long as, after recovering from a crash, it manages to initialize $rnd[a]$ with a higher value than the previous one. This is done as follows: Field $Count$ described previously in this section can be composed of a major and a minor component, $MCount$ and $mCount$. When an acceptor executes $Phase1b$ for some round, if $MCount$ equals the previous value in $rnd[a]$, it changes $rnd[a]$ in memory only; otherwise, it writes it on stable storage. During recovery, the acceptor simulates the reception of a "`1a`" message with an $MCount$ higher than the one it has on stable storage. To get values accepted by the recovered acceptor, coordinators will be forced to use higher rounds. In the normal case, acceptors write $rnd[a]$ on stable storage only once, when they are started. In the presence of failures, this strategy results in one extra stable write at each acceptor, per recovery. In this case, progress can only be ensured as long as acceptors do not crash and recover forever.

Learners do not have to write on stable storage because they can always ask the acceptors to resend their latest "`2b`" messages. Hence, there could be infinitely many of them in the system.

5. Generalized Consensus and other Related Works

Generalized Consensus is a generalization of the consensus problem defined in terms of *command structure*, or simply *c-struct* [13]. By using different c-structs, Generalized Consensus reduces do different agreement problems. As explained in [13], one can define c-structs for traditional consensus, total order broadcast, generic broadcast, etc. The command histories presented in Section 2 are, in fact, an instance of c-structs and were first presented in [13].

Generalized Paxos [13] is an extension of Fast Paxos that solves Generalized Consensus. Generalized and Fast Paxos are the only other agreement protocols we are aware of that have classic and fast modes. Generalized Paxos can be further extended with multicoordinated mode to improve availability [5]. We call the resulting protocol *Multicoordinated Paxos*. The Generic Broadcast algorithm that we presented in Section 3.2 is a simplified version of Multicoordinated Paxos.

There are three generic broadcasts algorithms in the literature to which we can compare our protocol: GB+ [18], AGB [1], and OptGB [22]. In specific, GB+ is roughly equivalent to the fast mode only version of our protocol. We say roughly because the fast mode always ignores conflicts between messages that have been spontaneously ordered, while GB+ only ignores conflicts of messages delivered distant in time. Likewise, AGB may be able to ignore some conflicts but, in general it falls back to atomic broadcast when conflicting proposals happen. Differently from GB+, AGB tolerates the crash of any minority of acceptors and, hence, requires three steps to deliver messages in the absence of conflicts and three more otherwise. The authors of AGB have also described a variation of the algorithm that tolerates less than one third of failures but decides in two or four steps.) The protocol of [22] combines the fast and the single-coordinated mode to terminate in two steps. It also combines the drawbacks of both rounds, requiring bigger acceptor quorums and relying on a correctly elected leader. In general, these protocols consider the crash stop, do not have a generalized version, cannot switch execution modes, and all resort to Consensus or Atomic Broadcast to solve conflicts. What is more, Generalized and Multicoordinated Paxos may resort to coordinated recovery and, in the case of Multicoordinated Paxos, an uncoordinated classic recovery which may gradually lower the requirement for spontaneous ordering until the instance is decided.

6. Conclusion

In this paper we discussed how to use multiple coordinators concurrently to devise multicoordinated rounds for agreement protocols. Multicoordinated rounds have better availability than classic and fast rounds. We exemplified the use of multicoordinated rounds with a Generic Broadcast algorithm, which uses a conflict relation between proposes values to mitigate the problem of collisions. Our protocol implements both multicoordinated, classic, and fast modes and allow switching between these modes to adapt to changes in the environment. We have also discussed more practical aspects of our protocol as how to plan for switching round types during runtime to adapt to environment changes and how to minimize disk access. In future works, we will explore the load balancing features of multicoordinated rounds.

Finally, we discussed how our Generic Broadcast protocol solves, in fact, the broader Generalized Consensus problem and its many instances (i.e., Consensus, Reliable Broadcast, Atomic Broadcast, Generic Broadcast, etc).

References

[1] M. K. Aguilera, C. Delporte-Gallet, H. Fauconnier, and S. Toueg. Thrifty generic broadcast. In *Proceedings of the 14th International Symposium on Distributed Computing (DISC)*, number 1914, pages 268–282, Toledo, Spain, 2000. Springer-Verlag.

[2] M. Ben-Or. Another advantage of free choice (extended abstract): Completely asynchronous agreement protocols. In *PODC '83: Proceedings of the second annual ACM symposium on Principles of distributed computing*, pages 27–30, New York, NY, USA, 1983. ACM Press.

[3] M. Burrows. The chubby lock service for loosely-coupled distributed systems. In *USENIX'06: Proceedings of the 7th conference on USENIX Symposium on Operating Systems Design and Implementation*, pages 24–24, Berkeley, CA, USA, 2006. USENIX Association.

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

[5] L. Camargos, R. Schmidt, and F. Pedone. Multicoordinated Paxos. Technical report, EPFL and University of Lugano, 2006.

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

[7] T. D. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. *Journal of the ACM*, 43(4):685–722, 1996.

[8] W. Chen, S. Toueg, and M. K. Aguilera. On the quality of service of failure detectors. *IEEE Transactions on Computers*, 51(5):561–580, May 2002.

[9] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, April 1985.

[10] V. Hadzilacos and S. Toueg. *Fault-tolerant broadcasts and related problems*, pages 97–145. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2 edition, 1993.

[11] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[12] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.

[13] L. Lamport. Generalized consensus and paxos. Technical Report MSR-TR-2005-33, MSR, 2004.

[14] L. Lamport. Fast paxos. *Distributed Computing*, 19(2):79–103, October 2006.

[15] L. Lamport. Lower bounds for asynchronous consensus. *Distributed Computing*, 19(2):104–125, October 2006.

[16] B. W. Lampson. How to build a highly available system using consensus. In Babaoglu and Marzullo, editors, *10th International Workshop on Distributed Algorithms (WDAG 96)*, volume 1151, pages 1–17. Springer-Verlag, Berlin Germany, 1996.

[17] M. Massa and L. Lamport. Cheap paxos. In *Proc. of the 2004 Intl. Conference on Dependable Systems and Networks*, June 2004.

[18] F. Pedone and A. Schiper. Handling message semantics with generic broadcast protocols. *Distributed Computing*, 15(2):97–107, April 2002.

[19] F. Pedone and A. Schiper. Brief announcement: on the inherent cost of generic broadcast. In *PODC '04: Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing*, pages 401–401, New York, NY, USA, 2004. ACM.

[20] F. Pedone, A. Schiper, P. Urbán, and D. Cavin. Solving agreement problems with weak ordering oracles. In *EDCC-4: Proceedings of the 4th European Dependable Computing Conference on Dependable Computing*, pages 44–61, London, UK, 2002. Springer-Verlag.

[21] M. O. Rabin. Randomized byzantine generals. In *Proc. of the 24th Annu. IEEE Symp. on Foundations of Computer Science*, pages 403–409, 1983.

[22] P. Zielinski. Optimistic generic broadcast. In *Proceedings of the 19th International Symposium on Distributed Computing*, pages 369–383, Kraków, Poland, 2005.