Brief Announcement: Multicoordinated Paxos[*]

Lásaro Camargos[†⋆]
lasaro@unicamp.br

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

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

[†] Institute of Computing
State University of Campinas,
Brazil

[‡] School of Computer and
Communication Sciences
École Polytechnique Fédérale
de Lausanne, Switzerland

[⋆] Faculty of Informatics
University of Lugano,
Switzerland

Categories and Subject Descriptors:
C.2.4 [Distributed Systems]: Distributed Applications

General Terms: Algorithms, Design.

Keywords: Atomic Broadcast, Consensus, Generalized, Multicoordinated, Paxos.

1. INTRODUCTION

In the consensus problem, processes must agree on a single value, given a set of proposals. It is used, for example, to implement state-machine replication [4], in which failure-independent processors are replicated to implement a reliable service. In this approach, the states of the replicas are consistently changed by applying deterministic commands from a sequence, each one agreed upon using a consensus instance.

Paxos [5] is an efficient and fault-tolerant consensus protocol originally intended for state-machine replication. It relies on a leader process to receive proposals from *proposer* processes (e.g., clients) and forward them to be agreed by the *acceptors*, which then notify the *learners* (e.g., replicas).

Even though Paxos provides very good performance, getting commands delivered by replicas in three communication steps, normal execution depends on the availability of the current leader. If the leader fails (or seems to have failed) a new leader must be elected, and this new leader has to synchronize with a quorum of acceptors before resuming normal execution. These actions take time and may introduce some temporary unavailability to the system. While this unavailability may not be significant when running a single instance of Paxos, it may cause a considerable slowdown on real systems, which use infinitely many instances of them [1].

Multicoordinated Paxos is an improved version of Paxos in which the failure of the leader does not slow down the protocol and the acceptance of new proposals. Our improvements neither increase Paxos latency nor require more acceptors to be contacted. In Multicoordinated Paxos, proposals are sent not to the leader, but to a set of *coordinator* processes, which act like the leader, and forward proposals to the acceptors. Acceptors, however, will only acknowledge values that have been sent by a quorum of coordinators.

Fast Paxos [7] is another improved variant of Paxos. In good runs of Fast Paxos, proposers send their proposals directly to the acceptors, reducing the minimum time to get a command learned to two communication steps. Since the leader can be bypassed, its unavailability is much less disruptive to the system, as in Multicoordinated Paxos. Nonetheless, this advantage comes at a price: acceptor quorums must be bigger than those in the original or in the Multicoordinated Paxos algorithm.

Fast Paxos can run either in the *classic* mode of the original Paxos protocol, or in the *fast* mode, switching between the modes to cope with dynamic changes in the system. Multicoordinated Paxos has a third mode, *multicoordinated*, more resilient to failures.

In the fast execution mode, since acceptors receive proposals directly from proposers, they may accept different commands for the same instance of consensus. In the worst scenario, no quorum of acceptors will accept the same command and learners will not be able to learn anything based on the received notifications. A similar situation can happen with our approach if coordinators forward different commands for the same instance of consensus, since acceptors will not be able to accept any value. This problem, called a *collision*, has many possible solutions, but all incur extra communication steps and, in the fast execution mode, extra disk writes.

In real applications, though, not all commands must be applied in the same order to all replicas. This notion of commutable commands can be used to alleviate the problem of collisions since commutable commands can be forwarded or accepted out of order. The Generalized Consensus [6] problem is a generalization of consensus that can take the application semantics into account, like the notion of commutable commands. In this problem, learners can augment their learned data structures and, thus, a single instance is enough to implement state-machine replication. In the full paper [2], we extend Generalized Paxos, a Generalized Consensus protocol similar to Fast Paxos, with multicoordinated execution; we also give rigorous correctness proofs of the resulting protocol.

Moreover, in the full paper, we also thoroughly review the hierarchy of algorithms ours depend upon (Paxos, Fast Paxos, and Generalized Paxos), and discuss practical issues like liveness, collision recovery, and how our protocols remove one disk write from runs with collisions. Summing up, our protocols either tolerate more acceptor and coordinator failures or write less frequently on stable storage than Fast and Classic Paxos, and similar protocolos.

2. MULTICOORDINATED PAXOS

Although the practical benefits of our approach are more evident when multiple commands are proposed and delivered, as in state machine replication, they are better understood starting from an isolated consensus instance. Hence, in the following, we present the multicoordinated version of the Fast Paxos consensus protocol.

Consensus. The safety requirements of the consensus problem can be described in terms of agreement among a set of *learner* processes, on values proposed by a set of *proposer* processes [8]. The liveness requirement is defined in terms of the set of *acceptor* processes. We call a *quorum* any finite set of acceptors that is large

[*]This work is supported by the NCCR-MICS grant 5005-67322.

enough to ensure liveness. The properties are the following:

Nontriviality: Any value learned must have been proposed.

Stability: A learner learns at most one value. (Usually omitted.)

Consistency: Two different learners cannot learn different values.

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.

We assume a crash-recovery asynchronous model in which processes communicate by exchanging messages that can be lost or duplicated, but not corrupted. Below we state the extra assumptions needed to circumvent the FLP result [3] and satisfy Liveness.

Fast and Classic Rounds. Like Fast Paxos [7], our protocol executes in rounds and assumes an unbounded number of them, totally ordered by a relation $<$. For simplicity, we assume here that the set of round numbers equals the set of natural numbers. The execution of rounds need not be ordered, and actions in different rounds may even interleave.

Each round i has its own set of quorums of acceptors, called i-quorums. In each round i, an acceptor can "accept" at most one value, and we say that a value has been *chosen* if accepted by an i-quorum in some round i; a chosen value is ready to be learned.

To orchestrate round executions, we assume a set of *coordinator* processes, organized in quorums per round. We refer to a quorum of coordinators for round i as an i-coordquorum, and say that c is a coordinator of i if c belongs to some i-coordquorum. Rounds can be classic or fast. In classic rounds, proposers send their proposals to the coordinators, so they can pick one and forward to the acceptors. Acceptors can only accept a value that has been forwarded by an i-coordquorum. Due to the following requirement, acceptors are guaranteed not to accept different values in the same classic round.

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

In fast rounds, acceptors may accept values coming directly from proposers, if they are told by the coordinators of the round to do so. As different values could then be accepted in the same round, the assumption below is needed for the algorithm to ensure liveness.

Fast Quorum Assumption 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 $Q \cap R \cap S \neq \emptyset$.

A round is divided into two phases, each one involving a number of actions. Briefly, during the first phase of round i, coordinators query an i-quorum about their latest accepted values and make them change their rounds to i, preventing acceptors from accepting values in any round $j < i$. Based on the values received during the first phase and given that acceptors will not accept values on lower-numbered rounds, coordinators are able to identify the value that has been or might be chosen at smaller rounds and can pick it. Coordinators then start the second phase of i by sending such value to the acceptors, which will accept it unless they have moved to a higher-numbered round. If the coordinators found no value to pick, they send any proposed value to acceptors, or have the proposers send their proposals directly.

The algorithm. We now present the complete algorithm as atomic actions executed by *proposers*, *coordinators*, *acceptors*, and *learners*. A TLA+ specification of this algorithm is given in the full paper [2]. The actions are the following:

$Propose(p, v)$ Executed by proposer p to propose value v. In the action, p sends a $\langle$"`propose`"$, v\rangle$ message to all coordinators and acceptors.

$Phase1a(c, i)$ Executed by any coordinator c of round i to start round i. In the action, c sends a message $\langle$"`1a`"$, i\rangle$ to each acceptor a asking a to take part in round i.

$Phase1b(a, i)$ Executed by acceptor a upon reception of a message $\langle$"`1a`"$, i\rangle$, if i is greater than any other round a has ever heard of.[1] In this case, a sends a message $\langle$"`1b`"$, i, vval, vrnd\rangle$ to all the coordinators of round i, where $vrnd$ is the highest-numbered round in which a has accepted a value (or an invalid round number if no value has been accepted by a) and $vval$ is the value it accepted in $vrnd$. The pre-condition of this action makes sure that after it is executed for round i, acceptor a will not execute it for a round j such that $j < i$. As we show in action $Phase2b$, this action also prevents a from accepting a value for a round j lower than i. This is a guarantee to the coordinators of i that the pair $vval$ and $vrnd$ will remain consistent as the information about the latest value accepted by a for a round number lower than i.

$Phase2a(c, i)$ Executed by any coordinator c of round i upon reception of $\langle$"`1b`"$, i, vval, vrnd\rangle$ from any i-quorum Q. In this action, c sends a $\langle$"`2a`"$, i, val\rangle$ message to the acceptors, where val is the value to be accepted by the acceptors. The value is picked based on the "`1b`" messages c received: (i) If none of the "`1b`" messages has a valid round number, then no value has been chosen in any round $j < i$, and c can pick any proposed value. Otherwise, let k be the greatest round number $vrnd$ received amongst the "`1b`" messages. (ii) If there exists a value v such that, for some k-quorum R, a message $\langle$"`1b`"$, k, v\rangle$ was received from every acceptor in $R \cap Q$, then c picks v. If such a value does not exist, (iii) c can pick any proposed value. If any value can be picked and i is a fast round, then c can pick the special value Any, that tell acceptors to accept proposals directly from proposers.

$Phase2b(a, i)$ Executed by acceptor a, for round i. This action is enabled if a has not heard of a round greater than i and has received a message $\langle$"`2a`"$, i, val\rangle$ coming from all coordinators in some i-coordquorum with the same value val. If i is a fast round and $val = Any$, then a can accept any value sent in a "`propose`" message. If i is classic and, therefore, $val \neq Any$, then a accepts val. After accepting value v, a sends the message $\langle$"`2b`"$, i, v\rangle$ to all learners.

$Learn(l)$ Executed by learner l when it receives a $\langle$"`2b`"$, i, val\rangle$ message from each acceptor in an i-quorum. The messages imply that val has been chosen and l can learn it.

Briefly, liveness is ensured if a quorum of acceptors and at least one coordinator is alive. More details can be found in [2].

3. REFERENCES

[1] M. Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proc. of the OSDI'06*, Nov. 2006.

[2] L. Camargos, R. Schmidt, and F. Pedone. Multicoordinated Paxos. Technical report, EPFL, 2006.

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

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

[5] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998.

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

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

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

[1] We say that a has heard of j if actions $Phase1b(a, j)$ or $Phase2b(a, j)$ have been executed.