# Brief Announcement: A Family of Leaderless Generalized-Consensus Algorithms


Giuliano Losa
ECE, Virginia Tech
glosa@vt.edu

Sebastiano Peluso
ECE, Virginia Tech
peluso@vt.edu

Binoy Ravindran
ECE, Virginia Tech
binoy@vt.edu


## 1. INTRODUCTION

Most distributed command-ordering algorithms inspired by Paxos and MultiPaxos [4] rely on a unique leader process which enforces an ordering on commands during fault-free periods. In practice, the leader has more work to do than other nodes, causing a *load balancing problem* and forming a performance bottleneck. Moreover, in geo-replicated systems, where bandwidth and latency vary considerably between sites, using a unique leader that every site has to contact results in uneven performance across sites, causing a *fairness problem*.

Recent ordering algorithms like S-Paxos [1] and Mencius [5] address the load-balancing problem: S-Paxos decouples payload distribution, which can be done without any undue burden on the leader, from ordering, while Mencius periodically rotates the role of leader among nodes. However, S-Paxos and Mencius do not solve the fairness problem because nodes have to contact other predetermined nodes independently of the quality of communication links.

To solve the fairness problem, a node must be allowed to choose which other nodes to communicate with based on the performance of its network links, as in the EPaxos [6] and Alvin [7] algorithms, both of which do not employ a unique leader, and allow any node to choose the closest (in terms of latency) quorum of nodes to reach agreement. EPaxos and Alvin are based on a novel but intricate algorithmic idea first introduced in EPaxos, which is difficult to generalize to devise other algorithms with different performance characteristics.

In this paper we show that the core idea underlying EPaxos can be captured in a *generic leaderless generalized-consensus algorithm* that uses two new abstractions: a *dependency-set algorithm*, which suggests dependencies for commands, and a *map-agreement algorithm*, which ensures that, for each submitted command, processes agree on a dependency set. Both abstractions lend themselves well to implementations that avoid the use of a unique leader and let nodes choose which set of nodes to communicate with for command ordering. Our generic algorithm gives rise to a family of algorithms whose members are obtained by using concrete dependency-set and map-agreement algorithms.




*PODC'16 July 25-28, 2016, Chicago, IL, USA*





ACM ISBN 978-1-4503-3964-3/16/07.

DOI: http://dx.doi.org/10.1145/2933057.2933072


On top of enabling modular correctness proofs of algorithms like EPaxos, we expect that the modular structure of our generic leaderless algorithm will allow a principled theoretical and empirical evaluation of the trade-offs that can be achieved by different implementations of our two abstractions. For example, some implementations may perform better in a cluster, while others in a wide area network; similarly, implementations of our abstractions may be optimized for different metrics, such as agreement latency, impact of failures or conflicts, tolerance to slow processes, load balancing, quorum size, etc.

A formalization of our work in TLA+ is available at http://losa.fr/research/leaderless.

## 2. LEADERLESS GENERALIZED-CONSENSUS ALGORITHMS

We consider a set of processes $P$ which are subject to crash-stop faults and which communicate by message-passing in an asynchronous network. Processes in $P$ must solve the Generalized Consensus problem [3], in which each process receives proposals for commands of the form $\mathsf{GC\text{-}propose}(c)$, must produce decisions of the form $\mathsf{GC\text{-}decide}(\sigma)$, where $\sigma$ is a *c-struct*, and the sequence of calls observed must satisfy the non-triviality, consistency, stability, and liveness properties of generalized consensus. We consider a set $C$ whose members we call c-structs, including the c-struct $\perp$, with an operator $\bullet$, for appending commands to a c-struct, such that any c-struct is of the form $\perp \bullet cs$ for some sequence of commands $cs$. Intuitively, a c-struct represents a set of sequences of commands that are all equivalent up to the ordering of commutative commands. We say that two commands *commute* when $\sigma \bullet c_1 \bullet c_2 = \sigma \bullet c_2 \bullet c_1$, for every c-struct $\sigma$. C-structs are partially ordered: $\sigma_1 \sqsubseteq \sigma_2$ if and only if there exists a sequence of commands $s$ such that $\sigma_2 = \sigma_1 \bullet s$. Moreover, $\perp \sqsubseteq \sigma$ for any $\sigma$. A c-struct contains a command $c$ when it is of the form $\perp \bullet cs$ where $c$ appears in the sequence of commands $cs$. Two c-structs are compatible when they have a common upper bound that is constructible from the commands contained in the two c-structs. The *non-triviality* property of generalized consensus requires that any decided c-struct be of the form $\perp \bullet \mathcal{D}$, where $\mathcal{D}$ is a sequence of proposed commands; *consistency* requires that any two decided c-structs $\sigma_1$ and $\sigma_2$ be compatible; *stability* requires that when a process $p$ decides a c-struct $\sigma_1$ at time $t_1$ and $\sigma_2$ at time $t_2$, then $t_1 \leq t_2$ implies that $\sigma_1 \sqsubseteq \sigma_2$; finally, *liveness* requires that if a command keeps being proposed, then a c-struct $\sigma$ containing the command is eventually decided. We refer the reader to Lamport [3] for a thorough discussion

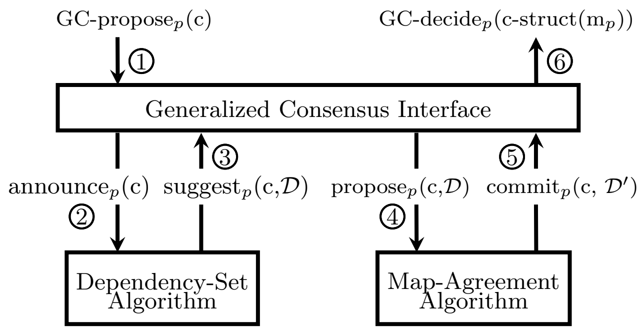


**Figure 1: Control flow of a process $p$ using our abstractions; numbers describe the order of events.**

```
upon init_p()
    cmds_p ← ∅
    ∀c, deps_p[c] ← ∅
    ∀c, heard_p[c] ← ∅
    ∀c, suggested_p[c] ← false
upon announce_p(c)
    cmds_p ← cmds_p ∪ {c}
    ∀q ∈ P send ⟨cmd | c⟩ to q
upon receive ⟨cmd | c⟩ from q
    D ← {d : d ∈ cmds_p ∧ d ≠ c ∧ d ≍ c}
    cmds_p ← cmds_p ∪ {c}
    send ⟨deps | c, D⟩ to q
upon receive ⟨deps | c, D⟩ from q
    deps_p[c] ← deps_p[c] ∪ D
    heard_p[c] ← heard_p[c] ∪ {q}
upon ∃c : heard_p[c] ∈ quorums
        ∧ suggested_p[c] = false
    suggested_p[c] ← true
    call suggest_p(c, deps_p[c])
```

**Figure 2: Example of dependency-set algorithm implementation.**

of c-structs and generalized consensus.

Our leaderless generalized consensus algorithm orchestrates a dependency-set algorithm and map-agreement algorithm to implement Generalized Consensus, as shown in fig. 1. A process $p$ maintains two local variables $m_p$, a map from commands to sets of commands, and $\sigma_p$, a c-struct. Upon receiving a command $c$, $p$ proceeds in three phases. First, it calls the *dependency-set* algorithm (Section 2.1) to determine a set of commands $\mathcal{D}$ on which the command $c$ may depend. Second, $p$ proposes the mapping $c \mapsto \mathcal{D}$ to the *map-agreement* algorithm (Section 2.2); when the map-agreement algorithm commits a mapping $c \mapsto \mathcal{D}'$ for $c$, $p$ inserts the mapping $c \mapsto \mathcal{D}'$ in $m_p$, and we say that $\mathcal{D}'$ is the dependency set of $c$. We say that $c$ is *executable* in $m_p$ when every command associated to $c$ in the transitive closure of $m_p$ is in the domain of $m_p$. Finally, when $c$ is executable in $m_p$, then $p$ may *locally* run the *graph-processing* algorithm described in Section 2.3 on $m_p$, obtaining a c-struct $\sigma$, and then call $\mathsf{GC\text{-}decide}(\sigma)$.

Any other process can start and run the three phases for the same command in parallel with the initial proposer of the command, and eventually execute the command, so that a command will be eventually included in a decided c-struct despite failures.

## 2.1 Computing Potential Dependency Sets

A dependency-set algorithm exposes the following interfaces at each process $p$: the input interface $\mathsf{announce}_p(c)$, to announce a command $c$; the output interface $\mathsf{suggest}_p(c, \mathcal{D})$, to suggest a set of dependencies $\mathcal{D}$ for $c$; and the input interface $\mathsf{commit}(c, \mathcal{D})$, to observe which command is committed by the map-agreement abstraction. Moreover, a dependency-set algorithm must ensure the following properties:

- Safety: (S1), for any call $\mathsf{suggest}_p(c, \mathcal{D})$, every command in $\{c\} \cup \mathcal{D}$ must have been announced before; (S2), for any two calls $\mathsf{suggest}_p(c_1, \mathcal{D}_1)$ and $\mathsf{suggest}_q(c_2, \mathcal{D}_2)$, if $c_1$ and $c_2$ do not commute, then $c_1 \in \mathcal{D}_2 \vee c_2 \in \mathcal{D}_1$.
- Liveness: (L1) If a call $\mathsf{announce}_p(c)$ was made, then eventually a suggestion $\mathsf{suggest}(c, \mathcal{D})_p$ is made, and (L2), if a suggestion $\mathsf{suggest}_p(c, \mathcal{D})$ has been made and no input $\mathsf{commit}_q(c, \mathcal{D}')$ is observed, then eventually a new suggestion $\mathsf{suggest}_r(c, \mathcal{D}'')$ for $r \neq p$ is made.

## 2.2 Agreeing on Dependency Sets

A map-agreement algorithm exposes the following interface at each process p: the input interface $\mathsf{propose}_p(c, \mathcal{D})$, to propose a set of dependencies $\mathcal{D}$ for a command $c$; the output interface $\mathsf{commit}_p(c, \mathcal{D})$, to commit $\mathcal{D}$ for $c$. Moreover, a map-agreement algorithm must ensure that:

- Safety: (S3), for any commit $\mathsf{commit}_p(c, \mathcal{D})$, then $\mathcal{D}$ has been proposed for $c$ at an earlier time; (S4), for any two calls $\mathsf{commit}_p(c, \mathcal{D}_1)$ and $\mathsf{commit}_q(c, \mathcal{D}_2)$, $\mathcal{D}_1$ is equal to $\mathcal{D}_2$.
- Liveness: (L3) if a proposal $propose_p(c, \mathcal{D})$ is made, then eventually a decision $\mathsf{commit}_p(c, \mathcal{D})$ is made.

## 2.3 Local Dependency-Graph Processing

The graph processing algorithm is executed locally by a process when a new command becomes executable. Based on the map $m_p$, it computes a c-struct $\sigma$ that is then used to call $\mathsf{GC\text{-}decide}(\sigma)$. The map $m_p$ defines a directed graph describing dependencies among commands, and that may contain cycles. The graph processing algorithm breaks cycles so as to obtain a partially ordered set of commands that uniquely determines a c-struct. Relying on the properties of the dependency-set and map-agreement algorithm expressed in **Lemma 2**, the graph processing algorithm ensures the non-triviality, consistency, and stability properties of Generalized Consensus.

For each set of commands $\mathcal{D}$, we assume that processes initially agree on a total order $<_{\mathcal{D}}$ on $\mathcal{D}$. In practice, if each commands is attached a unique identifier taken from a totally ordered set, it is easy to defined and compute $<_{\mathcal{D}}$.

The local variable $m_p$ denotes a directed graph $g_p$ whose set of vertices $V(g_p)$ is the executable commands of $m_p$ and whose edges $E(g_p)$ are such that there is an edge from $c_1$ to $c_2$ if and only if $c_2 \in m_p[c_1]$ (i.e., $c_1$ depends on $c_2$). For example, if $m_p = [c_1 \mapsto \{c_2, c_4\}, c_2 \mapsto \{c_3\}, c_3 \mapsto \{c_2\}]$ then $V(g_p) = \{c_2, c_3\}$ and $E(g_p) = \{(c_2, c_3), (c_3, c_2)\}$.

A directed graph $g$ *induces a partial order* $\preceq_g$ on its vertices defined such that $c_1 \preceq_g c_2$ if and only if there is a path from $c_2$ to $c_1$ and none from $c_1$ to $c_2$. For example, consider $h$ where $V(h) = \{c_1, c_2, c_3, c_4\}$ and $E(h) = \{(c_1, c_2), (c_2, c_1), (c_1, c_3)\}$. We have that $\preceq_h = \{(c_3, c_1)\}$. We say that a total order $\ll$ on $V(g)$ is a *linearization* of $g$ when for every $c_1, c_2 \in V(g)$, $c_1 \preceq_g c_2$ implies $c_1 \ll c_2$, and if $c_1$ and $c_2$ belong to the same strongly connected component $\mathcal{D}$ and $c_1 <_{\mathcal{D}} c_2$ hold, then $c_1 \ll c_2$. For example, assuming that $c_1 <_{\{c_1, c_2\}} c_2$, the linearizations of $h$ are $\langle c_3, c_1, c_2, c_4 \rangle$,

$\langle c_3, c_1, c_4, c_2 \rangle$, $\langle c_3, c_4, c_1, c_2 \rangle$, and $\langle c_4, c_3, c_1, c_2 \rangle$.

Let us define graph intersection such that $V(g_1 \cap g_2) = V(g_1) \cap V(g_2)$ and $E(g_1 \cap g_2) = E(g_1) \cap E(g_2)$, and graph union such that $V(g_1 \cup g_2) = V(g_1) \cup V(g_2)$ and $E(g_1 \cup g_2) = E(g_1) \cup E(g_2)$. Then, we say that $g$ is a vertex-induced subgraph of $g'$ if and only if $V(g) \subseteq V(g')$, and for every $e \in E(g')$, if both endpoints of $e$ are in $V(g)$, then $e \in E(g)$. Define two graphs $g_1$ and $g_2$ as *compatible* if and only if $g_1 \cap g_2$ is a vertex-induced subgraph of $g_1 \cup g_2$.

**Lemma 1.** *Assume that $l_1$ and $l_2$ are linearizations of two compatible dependency graphs $g_1$ and $g_2$, respectively, and that if $c_1, c_2 \in V(g_2)$ are a pair of non-commutative commands, then either $(c_1, c_2) \in E(g_2)$ or $(c_2, c_1) \in E(g_2)$. Then we have that: (a), $\perp \bullet l_1$ and $\perp \bullet l_2$ are compatible c-structs; (b), if $c_1 = c_2$ then $\perp \bullet l_1 = \perp \bullet l_2$.*

**Definition 1.** $\mathsf{c\text{-}struct}(m_p) = \perp \bullet l$*, where $l$ is a linearization of $g_p$.*

By lemma 1(b), $l$ exists and is unique, therefore $\mathsf{c\text{-}struct}(m_p)$ is well-defined.

## 2.4 Correctness

**Lemma 2.** *$g_p^1$, the graph $g_p$ at any time $t_1$, and $g_q^2$, the graph $g_q$ at any time $t_2$, are compatible.*

**Theorem 1.** *The consistency property of generalized consensus always holds.*

From lemma 2 and property S2, $g_p^1$ and $g_q^2$ satisfy the premises of lemma 1. Therefore we get that $\mathsf{c\text{-}struct}(\mathsf{m_p})$ at time $t_1$ and $\mathsf{c\text{-}struct}(m_q)$ at time $t_2$ are compatible.

**Theorem 2.** *If a command $c$ which is GC-proposed is then repeatedly GC-proposed, then a c-struct containing $c$ is eventually decided.*

The properties L1 and L2 ensure that dependency sets are repeatedly proposed for $c$ to the map-agreement algorithm as long as a commit is not observed. Moreover, the map-agreement algorithm ensures that when dependency sets are repeatedly proposed for $c$, then a dependency set will eventually be committed for $c$. Similarly, any dependency of $c$ is eventually committed and $c$ becomes executable, at which points it is included in the next decided c-struct.

## 2.5 Implementing the Abstractions

Both the dependency-set and map-agreement abstractions are well suited for leaderless implementations.

**The Dependency-Set Abstraction.** The dependency-set abstraction can be implemented as shown in fig. 2. Due to space constraints, our example focuses on satisfying the safety property only. A process $p$ announcing a command $c$ asks all the processes for the set of commands $c'$ that they have seen so far and that do not commute with $c$ (noted $c' \asymp c$). Then, it suggests the union of the dependency sets received from a *quorum* of processes. Quorums are sets of processes such that the intersection of any two quorums is not empty. Since $p$ receives sets of commands seen by other processes, the implementation ensures S1. Moreover, consider two suggestions $\mathsf{suggest}(c_1, \mathcal{D}_1)$ and $\mathsf{suggest}(c_2, \mathcal{D}_2)$ where $c_1 \asymp c_2$. There are two quorums $Q_1$ and $Q_2$ such that $\mathcal{D}_1$ was computed from $Q_1$ and $\mathcal{D}_2$ was computed from $Q_2$. By the definition of quorums, there is a process $q$ belonging to both $Q_1$ and $Q_2$. This process $q$ received either $c_1$ before $c_2$, in which case $c_1 \in \mathcal{D}_2$, or $c_2$ before $c_1$, in which case $c_2 \in \mathcal{D}_1$. Therefore, property S2 holds.

**The Map-Agreement Abstraction.** A state-machine replication algorithm like MultiPaxos [4] can implement the map-agreement algorithm by uniquely associating commands with positions in its sequence of consensus instances. However, MultiPaxos provides unnecessary guarantees on the ordering among instances. Instead, one can uniquely associate one incarnation of MultiPaxos with each process, which is initially the leader of its MultiPaxos incarnation. We assume that a command is associated with a unique natural number $\mathsf{id}(c)$ and with the process who first received $c$, noted $\mathsf{pid}(c)$. A process receiving a proposal $\mathsf{propose}_p(c, \mathcal{D})$ proposes $c \mapsto \mathcal{D}$ in instance $\mathsf{id}(c)$ of the MultiPaxos incarnation of process $\mathsf{pid}(c)$. Upon a MultiPaxos decision $c \mapsto \mathcal{D}'$, $p$ calls $\mathsf{commit}_p(c, \mathcal{D}')$. Note that the initial proposer of a command, being the MultiPaxos leader of its MultiPaxos incarnation, is free to choose any quorum of processes to reach agreement with on a dependency set. Therefore, in a geo-replication setting, the initial proposer can choose a quorum of processes excluding those nodes that are too far away. In a failure-free case, a command is committed in one round-trip with the chosen quorum.

EPaxos improves upon this scheme by using an algorithm similar to Fast Paxos [2] instead of MultiPaxos, and by combining the dependency-set algorithm of fig. 2 (except that it uses larger quorums), with a fast round of Fast Paxos: the reception of identical dependency sets from a fast quorum of processes (after an $\mathsf{announce}_p(c)$) acts as a single-round-trip decision of Fast Paxos.


## Acknowledgments

This material is based upon work supported by the National Science Foundation under grant CNS-1523558, and by the US Air Force Office of Scientific Research under grant FA9550-15-1-0098.


## References


[1] Martin Biely et al. "S-Paxos: Offloading the Leader for High Throughput State Machine Replication". In: *SRDS*. IEEE Computer Society, 2012, pp. 111–120.

[2] Leslie Lamport. "Fast Paxos". In: *Distributed Computing* 19.2 (2006), pp. 79–103.

[3] Leslie Lamport. *Generalized Consensus and Paxos*. https://research.microsoft.com/en-us/um/people/lamport/pubs/pubs.html#generalized. 2005.

[4] Leslie Lamport. "Paxos made simple". In: *ACM Sigact News* 32.4 (2001), pp. 18–25.

[5] Yanhua Mao et al. "Mencius: Building Efficient Replicated State Machine for WANs". In: *OSDI*. Ed. by Richard Draves and Robbert van Renesse. USENIX Association, 2008, pp. 369–384.

[6] Iulian Moraru et al. "There is more Consensus in Egalitarian Parliaments". In: *SOSP*. Ed. by Michael Kaminsky and Mike Dahlin. ACM, 2013, pp. 358–372.

[7] Alexandru Turcu et al. "Be General and Don't Give Up Consistency in Geo-Replicated Transactional Systems". In: *OPODIS*. Ed. by Marcos K. Aguilera et al. Vol. 8878. LNCS. Springer, 2014, pp. 33–48.