2017 IEEE 31st International Conference on Advanced Information Networking and Applications

On Making Generalized Paxos Practical

Tuanir F. Rezende*, Pierre Sutra †, Rodrigo Q. Saramago*, Lasaro Camargos*‡
* Universidade Federal de Uberlândia, Faculdade de Computação
Email: {tuanir, rod}@comp.ufu.br, lasaro@ufu.br
‡ Hedvig Inc.
† Télécom SudParis, Département d'informatique
Email: pierre.sutra@telecom-sudparis.eu

***Abstract*—Generalized Paxos (GPaxos) is a recent solution to Generalized Consensus, a distributed problem to which several key agreement problems reduce. We envision that GPaxos may unify within a single and novel Agreement-as-a-Service infrastructure multiple distributed protocols. To date this potential is however not fully unleashed, due to the steep learning curve of the protocol and the high complexity of its implementation. Moreover, before GPaxos reaches a real world usage, several computationally expensive operations have to be optimized and simplified. This paper aims at closing this gap between theory and practice. To this end, we first provide a concise tour of Generalized Paxos, hardly found elsewhere. Then, we assess the versatility of the Generalized Consensus problem by presenting a variation of GPaxos that solves the lease coordination problem. Our last contribution consists in three optimizations that apply to the critical phases of the algorithm: *(i)* a method to quickly start a new round, *(ii)* a novel approach to execute a checkpoint, and *(iii)* a data structure that speeds-up the detection of an agreement.**

I. Introduction

State machine replication (SMR) is a classical technique to implement a fault-tolerant service, by creating several copies of the service and ordering the client commands across replicas. Some recent works [1, 2] observe that replicas need to order only non-commuting commands, and thus that commuting commands may execute in two message delays, sidestepping the time complexity lower bound on consensus [3]. Lamport [2] names the task of agreeing on this partial order, the Generalized Consensus problem.

Generalized Paxos (GPaxos) is a protocol that solves Generalized Consensus with optimal time latency in the best-case scenario. GPaxos is also known for its versatility in the sense that, as different agreement problems reduce to Generalized Consensus, they are all solved by the same algorithm. Despite such appealing qualities, GPaxos has not been adopted in real world systems. We believe that this essentially comes from *(i)* the complexity of the specification given in [2] which is concisely expressed yet hard to implement, and *(ii)* the small set of reductions made from other problems to Generalized Consensus.

In this paper, we address these points by firstly presenting a full pseudo-code description and explanation of GPaxos that serves as an auxiliary material to the original TLA+ version given in [2]. Second, we attest the versatility of Generalized Consensus by presenting an elegant reduction of the lease coordination problem. Third, we detail a set of optimizations to improve the performance of GPaxos during the first phase of the protocol, when an agreement occurs, and during the computation of a checkpoint.

Outline. Section II provides the necessary background to define Generalized Consensus, then presents a detailed pseudo-code description of GPaxos. Section III introduces our new command-structure set for the lease coordination problem, as well as a brief discussion regarding the advantages of GPaxos over Generic Broadcast, an algorithm commonly regarded as simpler but as powerful as GPaxos. Section IV describes our optimizations to the protocol. We review the related work in Section V, then close in Section VI.

This work partially supported by project PVE CAPES 88881.062190/2014-01, CNPq and Fapemig.

II. Background

A. System Model

We assume a system model similar to the timed asynchronous model defined in [4] with a finite and fixed set of processes $\Pi = \{p_1, p_2, \ldots, p_n\}$, each process p_i making progress at its own speed.

Processes abide to the crash-recovery failure model, in which a host may lose its non-persistent state when crashing then re-join the system afterwards.

Communication happens through message passing over reliable asynchronous channels, which may delay the messages for a bounded (yet unknown) amount of time. Messages are neither duplicated nor have their content altered.

B. Generalized Consensus

Generalized Consensus is an extension of the ordinary Consensus problem, defined in terms of a data structure called *command structure*, or simply *c-struct*. Depending on the c-structs in use, Generalized Consensus reduces to various problems, such as Consensus, Total Order Broadcast or Generic Broadcast [1]. In what follows, we present the Generalized Consensus problem and the notion of c-struct using the original framework of Lamport [2].

C. C-Struct Sets

A c-struct set *CStruct* is defined in terms of an element $\perp$, a set of commands *Cmd*, an operator $\bullet$ that appends a

1550-445X/17 $31.00 © 2017 IEEE
DOI 10.1109/AINA.2017.94

command to a c-struct, as well as a set of axioms listed later. A c-struct is a very general data structure. For instance, we define the ordinary Consensus problem as follows: each c-struct is a singleton, $\bot$ is the empty set, and $v \bullet C$ equals C if $v = \bot$, and v otherwise. Another example of interest is when c-structs are partially ordered sets, $\bot$ is the empty set, and $v \bullet C$ is an operation that extends the partially ordered set v with the command C, making C succeeds (with respect to the partial order) any conflicting command in v, given some external conflicting relation over *Cmd*. This last c-struct set captures the notion of commutable commands, and is named *command histories* in [2]. It corresponds to the Generic Broadcast problem [1].

Before we present the four axioms of a c-struct set, some definitions are necessary. We write a finite sequence of commands, or **c-seq** hereafter, as $\langle C_1, C_2, \ldots, C_m \rangle$. The set of all finite sequences whose commands belong to $S \subseteq Cmd$ is denoted $Seq(S)$. Notice here that several repetitions of some element C may appear in a sequence $\sigma \in Seq(S)$. We define the operator $\bullet$ over the sequences of commands as follows:

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

We say that a c-struct w **extends** a c-struct v, or equivalently that v **prefixes** w, iff there exists a c-seq σ such that $w = v \bullet \sigma$. This fact is denoted as $v \sqsubseteq w$. Given a set T of c-structs, we say that v is a **lower bound** of T iff $v \sqsubseteq w$ for all w in T. A **greatest lower bound (glb)** of T is a lower bound v of T such that $w \sqsubseteq v$ for every lower bound w of T; we represent it with $\sqcap T$. Similarly, we say that v is an **upper bound** of T iff $w \sqsubseteq v$ for all w in T. A **least upper bound (lub)** of T is an upper bound v of T such that $v \sqsubseteq w$ for every upper bound w of T. We note it $\sqcup T$. Two c-structs v and w are defined to be **compatible** iff they have a common upper bound, and a set S of c-structs is compatible iff its elements are pairwise compatible.

We say that a c-struct v is **constructible from** a set P of commands if $v = \bot \bullet \sigma$, for some c-seq σ containing all the elements of P. A c-struct v **contains** some command C when v is constructible from some set P of commands such that $C \in P$. We define $Str(P)$ as the set of all c-structs constructible from subsets of P for some set P of commands—that is, $Str(P) \triangleq \{\bot \bullet \sigma : \sigma \in Seq(P)\}$.

A c-struct set *CStruct* must satisfy axioms CS1-CS4 below. CS1-CS2 are basic requirements to satisfy the properties discussed above. As we shall see shortly, CS3 and CS4 are necessary for Generalized Paxos and similar algorithms to ensure the safety and liveness properties of Generalized Consensus.

CS1. $CStruct = Str(Cmd)$

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

CS3. For any $P \subseteq Cmd$ and any c-structs u,v, and w in $Str(P)$:
- $\sqcap\{v, w\}$ exists and is in $Str(P)$.
- If v and w are compatible, then $\sqcup\{v, w\}$ exists and is in $Str(P)$.
- If $\{u, v, w\}$ is compatible, then u and $\sqcup\{v, w\}$ are compatible.

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

We can now generalize the original definition of consensus to deal with c-structs instead of single absolute values. This new problem is defined in terms of a c-struct set *CStruct* that includes a $\bot$ value, a set of commands *Cmd*, and an operator $\bullet$. Proposers propose commands in *Cmd* and given some learner l, we let $learned[l]$ be the current learned c-struct (initially $\bot$). Generalized Consensus is defined by the following properties:

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

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

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

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

D. Generalized Paxos

This section details GPaxos, Lamport's solution to the Generalized Consensus problem[2]. We provide a pseudo-code description of this algorithm in Algorithm 1, where we describe GPaxos as a set of atomic actions. For each action, its effects (**eff**) are guarded by one or more preconditions (**pre**). A comment in square brackets indicates the role of the process, either a proposer (*Proposers*), an acceptor (*Acceptors*), a coordinator (*Coordinators*), or a learner (*Learners*).

1) Ballots and quorums: GPaxos executes an unbounded sequence of asynchronous rounds, or *ballots*. We associate a ballot to a ballot number, or *balnum*, picked in *BalNum* that uniquely identifies it. Balnums are unbounded and they form a well-ordered set for some relation $<$, where 0 denotes the smallest element. In what follows, we identify a ballot with its balnum.

During a ballot, a learner attempts to learn one or more c-structs containing proposed commands. To this end, GPaxos relies on quorums of acceptors, i.e. non-empty subsets of *Acceptors*.

Quorums are constructed as follows: We map each ballot m to a set of quorums $quorum(k)$. An element in $quorum(m)$ is a quorum of m, or for short an m-quorum. A ballot m is either *fast* or *classic* and is associated with a unique coordinator $coord(m)$ in *Coordinators*. We consider hereafter that:

Q1. For any two quorums Q and Q', $Q \cap Q' \neq \{\}$ holds.

Q2. Given a fast ballot m, two m-quorums Q_1 and Q_2, and some n-quorum Q, it holds that: $Q_1 \cap Q_2 \cap Q \neq \{\}$.

Processes participating to GPaxos can compute locally the mapping of ballots to quorums and to coordinators. Such a computation may take the following form: Every process is at the same time an acceptor and a coordinator. A balnum is an integer, and ballot m is coordinated by the m^{th} coordinator (modulo $|Coordinators|$). A ballot m is fast iff m is even. If

Algorithm 1 Generalized Paxos – code at process i

1: *propose*(C) [proposer]
2: **pre:** $C \in Cmd$
3: **eff:** *send* $(\mathsf{propose}, C)$ *to* $Acceptors \cup Coordinators$
4: *phase1A*(m) [coordinator]
5: **pre:** $maxStart_i < m$
6: $i = coord(m)$
7: **eff:** $maxTried_i \leftarrow none$
8: $maxStart_i \leftarrow m$
9: *send* $(\mathsf{1A}, m)$ *to Acceptors*
10: *phase1B*(m) [acceptor]
11: **pre:** $bal_i < m$
12: $rcv_{coord(m)}$ $(\mathsf{1A}, m)$
13: **eff:** $bal_i \leftarrow m$
14: *send* $(\mathsf{1B}, m, cbal_i, cval_i)$ *to* $coord(m)$
15: *phase2Start*(m, Q, k) [coordinator]
16: **pre:** $maxTried_i = none$
17: $maxStart_i = m$
18: $Q \in quorum(m)$
19: $\forall a \in Q : rcv_a$ $(\mathsf{1B}, m, -, -)$
20: $k = max\{n < m \mid \exists a \in Q : rcv_a$ $(\mathsf{1B}, m, n, -)\}$
21: **eff:** $\mathscr{R} \leftarrow \{R \in quorum(k) \mid \forall a \in R \cap Q : rcv_a$ $(\mathsf{1B}, m, k, -)\}$
22: **if** $\mathscr{R} = \{\}$
23: $maxTried_i \leftarrow u$, s.t. $\exists a \in Acceptors : rcv_a$ $(\mathsf{1B}, m, k, u)$
24: **else**
25: Let $\gamma(R) \triangleq \sqcap\{u \mid \exists a \in R \cap Q : rcv_a$ $(\mathsf{1B}, m, k, u)\}$
26: $maxTried_i \leftarrow \sqcup\{\gamma(R) \mid R \in \mathscr{R}\}$
27: *send* $(\mathsf{2A}, m, maxTried_i)$ *to Acceptors*
28: *phase2AClassic*(m, C) [coordinator]
29: **pre:** $maxTried_i \neq none$
30: $maxStart_i = m$
31: $\exists p \in Proposers : rcv_p$ $(\mathsf{propose}, C)$
32: $\neg isFast(m)$
33: **eff:** $maxTried_i \leftarrow maxTried_i \bullet C$
34: *send* $(\mathsf{2A}, m, maxTried_i)$ *to Acceptors*
35: *phase2BClassic*(m, u) [acceptor]
36: **pre:** $rcv_{coord(m)}$ $(\mathsf{2A}, m, u)$
37: $bal_i \leq m$
38: $\exists Q \in quorum(m) : i \in Q$
39: $cbal_i \neq bal_i \vee cval_i \sqsubset u$
40: **eff:** $cval_i \leftarrow u$
41: $bal_i \leftarrow m$
42: $cbal_i \leftarrow m$
43: *send* $(\mathsf{2B}, m, cval_i)$ *to Learners*
44: *phase2BFast*(C) [acceptor]
45: **pre:** $isFast(cbal_i)$
46: $bal_i = cbal_i$
47: $\exists p \in Proposers : rcv_p$ $(\mathsf{propose}, C)$
48: **eff:** $cval_i \leftarrow cval_i \bullet C$
49: *send* $(\mathsf{2B}, cbal_i, cval_i)$ *to Learners*
50: *learn*(m, Q, u) [learner]
51: **pre:** $Q \in quorum(m)$
52: $\forall a \in Q : \exists v \in CStruct : rcv_a$ $(\mathsf{2B}, m, v) \wedge u \sqsubseteq v$
53: **eff:** $learned_i \leftarrow \sqcup \{learned_i, u\}$

now m is classic, the quorums of m are all the majorities sets. Otherwise, and following [2], m is fast and a quorum Q should satisfy $|Q| > \frac{3}{4} \times |Acceptors|$.

2) Variables and further definitions: To propose a command C, a process packs C in a propose message and sends it to all acceptors and coordinators in the system (at line 3 in Algorithm 1).

Acceptors ensure the long-term memory of the system. They successively join ballots, and vote during them. Each acceptor a maintains three variables: the current ballot (bal_a), the latest ballot during which it *accepted* (voted for) a c-struct ($cbal_a$), and the c-struct it accepted at that ballot ($cval_a$). Initially for every acceptor a, $bal_a = cbal_a = 0$, and $cval_a = \bot$.

At the beginning of ballot m, $coord(m)$ tries to convince acceptors to join m. If enough acceptors participate to m, $coord(m)$ suggests one or more c-structs. A coordinator c stores the latest ballot it started ($maxStart_c$), and the latest c-struct it suggested at that ballot ($maxTried_c$). If no c-struct was suggested so far in $maxStart_c$, then $maxTried_c$ equals $none \notin CStruct$. At the start of the algorithm, $maxTried_c = 0$ and $maxStart_c$ equals $\bot$ if $c = coord(0)$, and $none$ otherwise.

Considering some ballot m and a c-struct u, we shall say that *(i)* u is **chosen** at m, when there exists an m-quorum of acceptors Q, such that for every acceptor $a \in Q$, u prefixes the c-struct accepted by a at ballot m; *(ii)* u is **choosable** at m if u is chosen at m, or it might later be chosen at m; and *(iii)* u is **safe** at m when it suffixes all the c-struct choosable at m. Based on these definitions, GPaxos ensures three key invariants:

S0 If a c-structs u is learned, then u is chosen at some ballot.
S1 If two c-structs u_1 and u_2 are accepted at some classic ballot m, then $\{u_1, u_2\}$ is compatible.
S2. If an acceptor accepts a c-struct u, then u is safe at some ballot.

Invariants S0 and S2 together with assumption Q1 on quorums imply that learned c-structs are compatible. As a consequence, GPaxos satisfies the consistency requirement of Generalized Consensus.

3) Algorithmic Details: We now detail how GPaxos executes a classic ballot to maintain invariants S0-S2.

- *phase1A*(m): When it start a ballot m, the coordinator of m, denoted hereafter c, sends a 1A message labelled m to the acceptors (line 9).
- *phase1B*(m): When an acceptor a receives a 1A message labelled m, and bal_a is strictly smaller than m, *a joins* ballot m by setting bal_a to m. Then, acceptor a sends a 1B message labelled with m containing $cbal_a$ and $cval_a$ to the coordinator c (line 14).
- *phase2Start*(m, R, k): Coordinator c executes this action when there exist a ballot k and an m-quorum Q such that $coord(m)$ received a 1B message labelled m from every acceptor in Q, and k is the highest ballot mentioned in such messages. If this holds, the coordinator c extracts a c-struct which is safe at ballot m. Then, c stores this c-struct in $maxTried_c$, before suggesting it to the acceptors in a 2A message (line 27).
- *phase2AClassic*(m, C): When $maxTried_c$ differs from $none$, by construction this c-struct is safe at m. If m is classic, c appends newly proposed commands to $maxTried_c$ and suggests the resulting c-struct to the acceptors (line 34).
- *phase2BClassic*(m, u): When an acceptor a belonging to an m-quorum receives a 2A message containing a c-struct u and a can join ballot m (or has joined it previously), a accepts u by assigning u to $cval_a$ (line 40). Acceptor a then updates $cbal_a$ and bal_a to the value of m (lines 41

and 42), then sends a 2B message containing $cval_a$ to the learners (line 43).
Since c-struct u extends $maxTried_c$, every c-struct accepted at ballot m prefixes $maxTried_c$. Moreover, as $maxTried_c$ is safe at ballot m, every accepted c-struct is safe at ballot m. These two properties ensure respectively invariants S1 and S2.

– $learn(m,Q,u)$: A learner l learns a c-struct u once l knows that u is chosen at m (lines 51 and 52). To learn u, learner l assigns to $learned_l$ the value of $\sqcup\ \{learned_l, u\}$. This maintains the stability invariant of generalized consensus.

At first glance, GPaxos has a latency of five message delays. This is the length of the causal path leading from a propose to a 2B message. However, as long as $coord(m)$ does not crash and no coordinator starts a ballot higher than m, $coord(m)$ may suggest new commands within m. As a consequence, the common case is that every command is learned in three communication steps.

4) Fast ballots, collisions and recovery : To further reduce latency, acceptors execute action *phase2BFast* during fast ballots:

– *phase2BFast*(C): Once an acceptor a has joined a fast ballot (line 45), and accepted the safe c-struct suggested by the coordinator (line 46), a tries to extend it with newly proposed commands. More precisely, when a receives a propose message containing a command C, it sets $cval_a$ to $cval_a \bullet C$ (line 48), then sends a 2B message containing the new value of $cval_a$ to the learners (line 49).

Commands accepted during a fast ballot are learned in two steps: the causal path contains a propose message followed by a 2B message. A fast ballot leverages both the spontaneous ordering of the messages by the network and the compatibility of c-structs, as we illustrate below:

– Example: Let a_1 and a_2 be two acceptors that joined a fast ballot m. Suppose that a_1 and a_2 form an m-quorum, and accept u, the c-struct suggested by $coord(m)$ at ballot m. If a_1 and a_2 receive two commands C and D in this order, i.e., the network spontaneously orders C before D, then both a_1 and a_2 extend u to $v = (u \bullet C) \bullet D$. As a consequence, v is chosen at ballot m. Now, in the case where C and D are received in different orders, e.g. a_1 extends u to $v = (u \bullet C) \bullet D$ and a_2 extends u to $w = (u \bullet D) \bullet C$, then if C and D commute $v = w$, and v is still chosen at m.

However, if the set of c-structs accepted by the acceptors is not compatible, a collision occurs. A process i detects that a collision occurs at a ballot m when the following predicate holds:

$$collide(m) \triangleq \exists Q \in quorum(m) : \begin{cases} \forall a \in Q : rcv_a\ (2B, m, -) \\ \neg(\{u \mid \exists a \in Q : rcv_a\ (2B, m, u)\}\ compatible) \end{cases}$$

When a collision occurs at a ballot, GPaxos starts a higher ballot. We call this a *recovery*. The latency of GPaxos equals six communication steps when a recovery occurs: two messages during the fast ballot that collides (propose, 2B), plus four messages to recover (1A, 1B, 2A, 2B).

As pointed out previously, Generalized Consensus may model different distributed tasks using the c-struct abstraction. We may solve all these problems using the GPaxos algorithm. Following this idea, we propose in the next section a new definition for the lease coordination problem.

III. A C-STRUCT FOR LEASE COORDINATION

Modern large-scale distributed systems rely on fault-tolerant mechanisms to coordinate accesses to shared resources, such as files. One of such mechanisms is the notion of lease [5] that solves the critical section problem. Leases are, for instance, at core of the Google Chubby lock service [6]. They are also closely related to the principle of ephemeral znode in Apache ZooKeeper [7].

In this section, we present a novel *CStruct* definition that allows GPaxos to solve the lease coordination problem. Our solution is surprisingly simple and easy to implement. To the best of our knowledge, this is the first time GPaxos is used to solve a variation of the lease coordination problem in a distributed system, and one of the few applications of Generalized Consensus outside of the set initially documented by Lamport [2].

A. Problem statement

We consider that each process has access to a local (hardware) clock. Following [4], the clocks are loosely synchronized. This means that the clock drift between any two clocks is bounded by some constant ε.

When a process p_i requires the access to a critical section, it sends a message to the proposers containing a command with the following structure:

$$C = \langle CS, p_i, T_{begin}, T_{end} \rangle$$

The first value of the tuple identifies the critical section (CS), the second identifies the process (p_i), and the last two respectively the beginning (T_{begin}) and the end (T_{end}) of the lease. The *duration* of the lease is defined as $T_{end} - T_{begin}$. To guarantee progress [5], we assume that no lease has a duration smaller than ε.

We expect that the lease issued by a process represents a need that can be postponed to a later date. In particular, if the critical section is already in use, the process can wait after the corresponding lease expires. The lease duration is, however, not altered. This semantics of a lease request is the most common, and it guarantees that each process eventually gains access to the critical section for the duration of its original request.

B. Notion of CLease

We now define a set of c-structs, denoted hereafter *CLease*, that is well-suited for the lease coordination problem.

Each c-struct in *CLease* is a map. The empty c-struct, $\perp$, corresponds to the empty map. For one such c-struct *LeaseMap*, a key in *LeaseMap* identifies some critical section,

say *CS*, and the value $LeaseMap[CS]$ is the queue of requests that access *CS*.

In order to append a command C to a c-struct *LeaseMap*, a process executes Algorithm 2. To ease the presentation, we treat a missing key as one with an empty queue in this algorithm.

Algorithm 2 *CLease* append algorithm

```
1:  function LeaseMap • C
2:      let C = ⟨CS, p, T_begin, T_end⟩
3:      if LeaseMap[CS] = ⟨⟩ then
4:          nLease ← ⟨p, T_begin, T_end⟩
5:      else
6:          last_elem ← LeaseMap[CS].last()
7:          nTbegin ← last_elem.T_end + ε
8:          nT ← T_end − T_begin
9:          nTend ← nTbegin + nT
10:         nLease ← ⟨p, nTbegin, nTend⟩
11:     LeaseMap[CS].enqueue(nLease)
```

At the light of the above definitions, we can show that *CLease* satisfies the following properties:

- $LeaseMap \sqsubseteq LeaseMap'$ holds iff for every key $CS \in LeaseMap$, the queue $LeaseMap[CS]$ prefixes the queue $LeaseMap'[CS]$;
- The set $\{LeaseMap, LeaseMap'\}$ is compatible iff for any key $CS \in LeaseMap \cap LeaseMap'$, either $LeaseMap[CS]$ prefixes $LeaseMap'[CS]$, or the converse holds.

In GPaxos, upon learning a command mentioning its identifier, the client is free to access the resource within the time boundaries defined by the lease duration.

C. The case for Generalized Consensus

It is commonly believed that GPaxos and Generic Broadcast (GBCast) [1] solve the same problem. In particular, Lamport notes that "the difference in efficiency between the two algorithms is insignificant" [8]. As a consequence, one might conclude that the performance gains of Generalized Consensus stem from instantiating c-structs as command histories. In which case, GBCast would work just as well.

A variation of *CLease* shows that in fact this is not an accurate picture. With more details, while Generic Broadcast looks only at pairwise conflict relations, the append operator of a c-struct set can look at the set of accepted commands before appending a new one. For example, one could devise a k-mutual exclusion [9] algorithm based on *CLease*, in which a request is accepted only if it will not cause more than k leases to be ongoing at the same time. Such an approach is however not possible with GBCast.

IV. Optimizations to Generalized Paxos

In [2], Lamport explores the expressiveness of TLA+ to provide a concise specification for GPaxos. Naively translating such a specification results in an inefficient code. In particular, it requires to solve a harder problem than required, and leads to the computation from scratch of intrinsically incremental data structures. This section discusses some of these limitations and proposes ways to alleviate them.

A. Action phase2Start

As described in Section II-D, if the coordinator fails or a collision occurs, a new ballot is started by re-executing actions *phase1A*, *phase1B* then *phase2Start*. In particular, *phase2Start* determines for some m-quorum Q, the set $\mathscr{R}$ of k-quorums R such that a 1B message was received from every $a \in Q \cap R$.

We observe that the usual approach to define the m-quorums is to fix some value $qsize(m)$, and require that every set of $qsize(m)$ acceptors is an m-quorum. Under such an assumption, and while enumerating all the k-quorums to check if they belong to $\mathscr{R}$ works, this is not efficient. Instead, we propose to enumerate the intersections between Q and all the possible values of R. Since several quorums intersect with Q similarly, this set of intersections is smaller than $\mathscr{R}$, and thus cheaper to compute. We detail such an approach below.

Following the preconditions of action *phase2Start*, we consider a ballot m and an m-quorum Q. We let k be the highest ballot mentioned in the 1B messages received from Q and define $Q' = \{a \in Q \mid rcv_a\ (1\mathsf{B}, m, k, -)\}$. Our observation is that every k-quorum R intersecting with Q' has a minimum size of $qsize(m) + qsize(k) - |Acceptors|$. Thus, we may replace $\mathscr{R}$ with $\mathscr{I} = \{I \subseteq Q' \ :\ |I| \geq qsize(k) + qsize(m) - |Acceptors|\}$ in Algorithm 1.

B. Horizontal checkpointing

GPaxos saves c-structs in variables and sends them in messages. Over time, such c-structs include the whole system history, and it becomes prohibitively expensive to handle them. To solve this issue, Lamport suggests to use checkpoints and multiple GPaxos instances. This is similar to what is done in conventional state machine replication with the so-called Multi-Paxos protocol [12]: when the c-structs exchanged during an instance are too large, a checkpoint command is proposed and a higher instance is started.

Differently from Lamport, we propose in this section to run a single instance of Generalized Consensus and to successively checkpoint it over time, a technique we call **horizontal checkpointing**. Horizontal checkpointing leverages the observation that once a c-struct is learned, it is by definition stable and, therefore, can be forgotten. Nonetheless, as failures may occur, discarding part of a c-struct should be done with care.

In a nutshell, our technique achieves this as follows: When a process appends a checkpoint command to some c-struct, it trims the c-struct up to the previous checkpoint command. Two c-structs are comparable (using relation $\sqsubseteq$) if they suffix the same checkpoint. Below, we present the horizontal checkpointing mechanism more formally, argue about its correctness, and detail its benefits.

1) Checkpoint command: Following [2], we note $\circ$ the binary operator to concatenate two sequences of commands. A command C is a **checkpoint** when it satisfies the predicate below.

$$\forall \rho, \sigma, \tau \in Str(Cmd) : \bot \bullet \rho \bullet C \bullet \sigma = \bot \bullet \tau : \exists \eta \in Str(Cmd) : \begin{cases} \tau = \eta \circ \langle C \rangle \circ \sigma \\ \bot \bullet \eta = \bot \bullet \rho \end{cases}$$

Let us notice here that $\bot \bullet \tau = (\bot \bullet \eta) \bullet (\bot \bullet C \bullet \rho)$. As a consequence, if we trim the c-struct $\bot \bullet \tau$ up to command C, this predicate tells us that we may still reconstruct it using η.

In the case where $CStruct = Str(Cmd)$, $\eta = \rho$ satisfies the above definition for any command C. Hence, a checkpoint command can be any command. On the other hand, if c-structs are command histories, a checkpoint command cannot commute with any other command. To see this, consider for the sake of contradiction that D is commuting with C. Defining $\tau = \langle C, D \rangle$, $\rho = \langle D \rangle$ and $\sigma = \langle\rangle$, the c-seq η should satisfy $\tau = \eta \circ C$, which is not possible.

2) Notions of k-checkpoint: In what follows, we number each checkpoint command using some natural $k > 0$, and we note H_k the checkpoint command numbered k. A c-struct u is a **k-checkpoint** if either $k = 0$ and $u = \bot$, or there exists a c-seq σ such that $H_k \notin \sigma$ and $u = \bot \bullet (\sigma \circ H_k)$.

Accordingly to the previous definitions, we observe that once a k-checkpoint is chosen, it is unique, and it prefixes $cval_a$ at $f+1$ acceptors. We prove such a claim below:

Proof: Let u and v be two k-checkpoints. The case $k = 0$ being trivial, we assume hereafter that $k > 0$. Note ρ and τ two c-seqs such that:

$$u = \bot \bullet (\rho \circ H_k) \quad \text{with } H_k \notin \rho$$
$$v = \bot \bullet (\tau \circ H_k) \quad \text{with } H_k \notin \tau$$

Consider that u and v are chosen respectively at ballots m and n. First of all, assume that $m > n$. As c-struct u is chosen at ballot m, it is safe by invariant S2. Consequently, we have $v \sqsubseteq u$. The case $n > m$ is symmetric and leads to $u \sqsubseteq v$. Now if both u and v are chosen at the same ballot, invariant $Q1$ implies that for some acceptor a we have $u \sqsubseteq cval_a$ and $v \sqsubseteq cval_a$.

The above reasoning tells us that there exit two c-seqs σ and ϕ such that $u \bullet \sigma = v \bullet \phi$. From which, we deduce that $\bot \bullet (\rho \circ H_k \circ \sigma) = \bot \bullet (\tau \circ H_k \circ \phi)$. By definition of a checkpoint command, there exists a c-seq η such that: $\eta \circ H_k \circ \sigma = \tau \circ H_k \circ \phi$, with $\bot \bullet \eta = \bot \bullet \rho$. As $H_k \notin \rho$, we observe that $H_k \notin \bot \bullet \rho$, leading to $H_k \notin \eta$. From $H_k \notin \tau$, we then deduce $\eta = \tau$ and $\sigma = \phi$. As a consequence, we obtain $u = v$. ■

At the light of this result, we shall note hereafter $(v_k)_k$ the set of k-checkpoints chosen during an execution.

3) The horizontal checkpointing technique: Our horizontal checkpointing technique assumes that there is a *single* quorum per ballot. Because a coordinator may always jump to a higher ballot whose quorum is responsive, such an assumption does not harm liveness (see [15] for more details).

Let $|\sigma|$ be the length of a c-seq σ, and assume some fixed threshold chksize. A process may propose H_{k+1} only if there exists a c-struct u and a c-seq σ such that: *(i)* u is a k-checkpoint; *(ii)* $|\sigma| >$ chksize; and *(iii)* $u \bullet \sigma$ is learned.

We define the **checkpoint number** of a c-struct u, denoted hereafter $H(u)$, as $max(\{k : H_k \in u\} \cup \{0\})$. For some c-struct u accepted at a ballot m, if $H(u) > 0$ holds, we notice that $v_{H(u)-1} \sqsubseteq u$ is true. At the light of this observation, when sending a c-struct $u = v_{k-1} \bullet w$ in Algorithm 1, an acceptor simply sends w instead.

In Algorithm 1, we augment the state of every acceptor with a variable *chk* storing for each natural either a c-struct, or *none*. Initially, $chk[k] = none$ for every $k > 0$, and $chk[0] = \bot$. When an acceptor a notices that $(v_k \sqsubseteq cval_a \ \wedge \ H(cval_a) = k+1)$ holds, a assigns v_k to $chk[k]$. This situation occurs when a is executing either $phase2BFast(H_{k+1})$, or $phase2BClassic(-, u)$ with $H(u) = k+1$.

We complete our horizontal checkpointing technique with a catch-up mechanism defined as an action $catchup(k)$ added to Algorithm 1. Action $catchup(k)$ triggers at learner l when l receives a message $(\mathsf{2B}, -, v)$ with $k = H(learned_l) + 1 < H(v)$. In such a case, process l sends a message $(\mathsf{catchup}, k)$ to all acceptors. An acceptor a receiving such a message and satisfying locally $chk[k] \neq none$ sends a message $(\mathsf{catchup}, k, chk[k])$ back to l. Upon the reception of such a message, l assigns $learned_l$ to $\sqcup \{learned_l, chk[k]\}$.

4) Benefits: There are several advantages in using the horizontal checkpointing technique over multiple instances of GPaxos. First of all, our approach fully embraces the genericity brought by the c-struct abstraction, requiring a single protocol stack to solve Generalized Consensus. Second, as the horizontal checkpointing technique uses a single instance of GPaxos, our technique reduces the memory footprint. Third, in the base approach multiple instances have to run in sequence one after the other, which requires a stop-the-world mechanism: when a checkpoint command operation is decided, a new instance of the protocol has to start. On the contrary, our checkpoint technique solely necessitates to agree on a the checkpoint command. This saves two message delays.

C. Learner's optimized quorum detection

The computation of lubs and glbs is the most expensive part of GPaxos. In what follows, we show that in particular during the learning phase this cost is high and we develop a solution to decrease it.

1) An expensive computation: When a 2B message from some acceptor a is received at a learner (lines 51-53 in Algorithm 1), a straightforward implementation of the *learn* action goes as follows:

(**Construction**) Find all the possible quorums of acceptors Q with $a \in Q$, and check if the c-structs accepted by Q are compatible. If this is the case, calculate the glb of these c-structs to find the greatest common prefix u, then assign $\sqcup \{learned_i, u\}$ to $learned_i$.

Hence, we need to calculate a glb followed by a lub for each quorum including a. The cost of this computation depends on the definition of *CStruct*. Considering that we have m c-structs with n elements each, this costs varies from $O(nm)$ when c-structs are c-seqs to $O(n^2m)$ for c-histories.[1] Hence, in the worth case, the complexity of the learning phase is $O(qn^2m)$, where q is the number of possible quorums.

Each time a new 2B message is received, a new quorum may be formed that extends the learned c-struct even further.

[1]It takes $O(n)$ operations to append a new command to a c-history already containing n nodes.

As the above computation takes place very often, it becomes a burden for the algorithm and may erase all the performance gains brought by Generalized Consensus. Such an issue was already observed in practice [15].

2) A Trie-like solution: To improve the learning phase of GPaxos, our solution is to store the accepted c-structs in a Trie-like data structure (or c-trie, for short) that grows over time. This data structure is used to speed-up the detection of an agreement.

In detail, our technique works as follows:

- When an acceptor forwards the c-struct u it accepted, it sends in addition a c-seq σ such that $u = \bot \bullet \sigma$.
- A learner that receives σ from some acceptor a executes Algorithm 3 to updates its c-trie and possibly detects that an agreement occurs.
- In Algorithm 3, the learner maintains the root in the c-trie (*root*) and a hash map ($find$) binding the c-structs to the nodes of the c-trie. That map provides constant time access to the prefixes. It is built over time with the help of a hash function (denoted *hash*). Each node of the c-trie maintains its children (*child*), a c-struct (*value*), and the number of acceptors that voted for that c-struct (*votes*).

For illustration purposes, consider 3 commands A, B and C, such that A commutes with C, and B conflicts with both A and C. Assume that the acceptors are in a fast ballot and that solely A, B and C are proposed. Figure 1 details a possible state of the c-trie during this scenario.

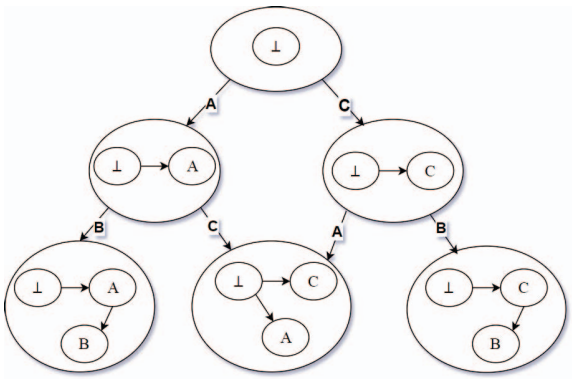

Fig. 1. Trie-like structure used for quorum detection

Consider that n is the length of the c-seq σ. In the worth case, the insertion of a c-struct in the c-trie requires $O(n^2)$ operation (due to line 11 in Algorithm 3). On the other hand, finding a c-struct that is accepted by a quorum takes $O(dm)$ time, where d is the depth of the c-trie and m the total number of acceptors (each level of the c-trie can have at most $|Acceptors|$). Hence, in total we execute $O(n^2 + dm)$ operations instead of $O(qn^2m)$ in the initial solution. Notice that in addition we may update the root of the c-trie, once we know that a c-struct prefixes all the c-structs accepted by the acceptors (typically at the start of a new ballot).

Algorithm 3 Trie-like insertion algorithm

```
1:  function insert(σ, a)
2:      node ← root                                   // root of the c-trie
3:      i ← 0
4:      n ← length(σ)
5:      while i < n do
6:          if node.child(σ[i]) ≠ ⊥ then
7:              node ← node.child(σ[i])
8:          else
9:              let u = node.value
10:             v ← u • σ[i]
11:             η ← find(v)                           // query hash map
12:             if η = ⊥ then
13:                 η = new node
14:                 η.value = v
15:                 find ← find ∪ {(hash(v), η)}
16:             node.child(σ[i]) ← η
17:             node ← η
18:         node.votes ← node.votes ∪ {a}
19:         i ← i + 1
```

V. Related work

The base approach to implement state machine replication (SMR) is to execute successive instances of consensus, each instance deciding upon a new command to apply.

The seminal FLP paper [10] shows that consensus is not solvable in an asynchronous system using a deterministic algorithm even if a single process may crash. Hopefully, real systems do not behave asynchronously all the time, and since FLP there have been major contributions to the problem of solving consensus in a fault-tolerant manner. In particular, Dwork et al. [11] and Lamport [12] solve consensus deterministically for $f < n/2$ failures and under weak synchrony assumptions. Both algorithms preserve the agreement property of consensus, even if the system behaves asynchronously. In the context of the failure detector abstraction, this behavior is called indulgence [13].

Paxos solves consensus in three message delays; this is optimal in the general case [3]. When a process executes solo, or the concurrent commands are commuting [1, 2, 14], this delay is reduced to two. However, if a collision occurs processes need to resolve the conflict by executing a recovery phase. In Generalized Paxos, the recovery takes four additional steps, leading to a total of six message delays.

FGGC [15] is a variation of Generalized Paxos that recovers in the optimal time of a single message delay. To this end, FGGC distinguishes read and write quorums, and uses a single write quorum per ballot. Each write quorum contains only $f+1$ processes (that is, a majority quorum).

Multicoordinated Paxos [16] uses more than one coordinator per ballot. This removes the downtime when the coordinator fails at the cost of a higher risk of collision. During a multicoordinated ballot m, an acceptor accepts a c-struct u only if u prefixes the c-structs received from some m-quorum of coordinators. A collision may occur during a multicoordinated ballot m when the c-structs suggested by coordinators collide. In such a case, a (higher) classic ballot is started.

EPaxos [17] is a multi-leader solution to the Generalized Broadcast problem, close to Mencius [18]. Similarly to [19],

the algorithm tracks conflicts and it delivers non-conflicting commands in two message delays. However, in the presence of conflicts, the protocol takes a slow path of four message delays.

M^2Paxos [20] uses a rotating ownership mechanism to avoid collisions and the related recovery cost. When the workload exhibits strong locality, that is each node accesses a partition of the shared state, M^2Paxos solves Generalized Consensus in two message delays.

A traditional use case of SMR is to orchestrate concurrent distributed processes with the help of a so-called coordination service. This type of service allows processes to elect a leader, define a participant group, or synchronize upon a datum (e.g., a file) to fulfill a parallel job. Well-known examples of coordination services include Google Chubby [21], Microsoft Azure's Lock Service [22], and Apache ZooKeeper [7].

In [23], the authors detail the internals of the Chubby distributed lock service. This service is built on top of a shared log. To construct this log, Chubby chains multiple instances of Paxos (also called Multi-Paxos [12]). The log is fully under control of Paxos but the format of a snapshot to truncate the log is specific to Chubby. Every read of the Chubby service goes through the Paxos coordinator. A lease mechanism avoids to read a stale content in situations where the coordinator rotates. In Section III, we explain how GPaxos solves this problem elegantly with the notion of *CLease*.

VI. Conclusion

Generalized Paxos (GPaxos) offers within the same framework an elegant and efficient solution to multiple distributed problems. However to date few implementations exist as the algorithm is difficult to understand and its applicability not well understood.

This paper makes a first step toward the direction of making Generalized Paxos practical. We first present a pseudo-code description that complements the original version given by Lamport [2]. Then, we introduce a variation for the lease coordination problem that underlines the versatility of the Generalized Paxos algorithm. Further, we show that several optimizations of the code are possible, regarding the first phase of the protocol, the detection of an agreement, and the checkpointing mechanism.

References

[1] F. Pedone and A. Schiper, "Handling message semantics with generic broadcast protocols," *Distributed Computing*, vol. 15, no. 2, pp. 97–107, April 2002.

[2] L. Lamport, "Generalized consensus and paxos," Microsoft Research, Tech. Rep. MSR-TR-2005-33, 2004.

[3] ——, "Lower bounds for asynchronous consensus," *Distributed Computing*, vol. 19, no. 2, pp. 104–125, 2006.

[4] F. Cristian and C. Fetzer, "The timed asynchronous distributed system model," *IEEE Transactions on Parallel and Distributed Systems*, pp. 642–657, June 1999.

[5] B. Kolbeck, M. Hogqvist, J. Stender, and F. Hupfeld, "Flease - lease coordination without a lock server," in *Proc. of the 2011 IEEE International Parallel & Distributed Processing Symposium*. Washington, DC, USA: IEEE Computer Society, 2011, pp. 978–988.

[6] M. Burrows, "The chubby lock service for loosely-coupled distributed systems," in *Proc. of the 7th conference on USENIX Symposium on Operating Systems Design and Implementation*. Berkeley, CA, USA: USENIX Association, 2006, pp. 24–24.

[7] F. P. Junqueira and B. C. Reed, "The life and times of a zookeeper," in *Proc. of the 28th ACM Symposium on Principles of Distributed Computing*. New York, NY, USA: ACM, 2009, pp. 4–4.

[8] L. Lamport, "The Writings of Leslie Lamport," , accessed: 2016-10-26.

[9] K. Raymond, "A distributed algorithm for multiple entries to a critical section," *Information Processing Letters*, vol. 30, no. 4, pp. 189 – 193, 1989.

[10] M. J. Fischer, N. A. Lynch, and M. S. Patterson, "Impossibility of distributed consensus with one faulty process," *J. ACM*, vol. 32, no. 2, pp. 374–382, Apr. 1985.

[11] C. Dwork, N. Lynch, and L. Stockmeyer, "Consensus in the presence of partial synchrony," *J. ACM*, vol. 35, no. 2, pp. 288–323, 1988.

[12] L. Lamport, "The part-time parliament," *ACM Trans. Comput. Syst.*, vol. 16, no. 2, pp. 133–169, May 1998.

[13] R. Guerraoui, "Indulgent algorithms (preliminary version)," in *PODC '00*. New York, NY, USA: ACM, 2000, pp. 289–297.

[14] L. Lamport, "Fast paxos," *Distributed Computing*, vol. 19, no. 2, pp. 79–103, 2006.

[15] P. Sutra and M. Shapiro, "Fast Genuine Generalized Consensus," in *Proc. of the 30th IEEE International Symposium on Reliable Distributed Systems*, Oct. 2011.

[16] L. Camargos, R. Schmidt, and F. Pedone, "Multicoordinated agreement protocols for higher availabilty," in *Proc. of the 7th IEEE International Symposium on Network Computing and Applications*. Washington, DC, USA: IEEE Computer Society, July 2008.

[17] I. Moraru, D. G. Andersen, and M. Kaminsky, "There is more consensus in egalitarian parliaments," in *Proc. of the 24th ACM Symposium on Operating Systems Principles*. New York, NY, USA: ACM, 2013, pp. 358–372.

[18] Y. Mao, F. P. Junqueira, and K. Marzullo, "Mencius: building efficient replicated state machines for wans," in *Proc. of the 8th USENIX conference on Operating systems design and implementation*. Berkeley, CA, USA: USENIX Association, 2008, pp. 369–384.

[19] P. Zielinski, "Optimistic generic broadcast," in *Proc. of the 19th International Symposium on Distributed Computing*, Kraków, Poland, 2005, pp. 369–383.

[20] S. Peluso, A. Turcu, R. Palmieri, G. Losa, and B. Ravindran, "Making fast consensus generally faster," in *Proc. of 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, June 2016, pp. 156–167.

[21] M. Burrows, "The chubby lock service for loosely-coupled distributed systems," in *Proc. of 7th symposium on Operating Systems Design and Implementation*, 2006.

[22] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold *et al.*, "Windows azure storage: A highly available cloud storage service with strong consistency," in *Proc. of the 23rd ACM Symposium on Operating Systems Principles*. New York, NY, USA: ACM, 2011, pp. 143–157.

[23] T. D. Chandra, R. Griesemer, and J. Redstone, "Paxos made live: An engineering perspective," in *Proc. of the 26th Annual ACM Symposium on Principles of Distributed Computing*. New York, NY, USA: ACM, 2007, pp. 398–407.