Fast Genuine Generalized Consensus

Pierre Sutra, Marc Shapiro
INRIA Paris-Rocquencourt and LIP6, Université Pierre et Marie Curie, Paris, France

***Abstract*—Consensus (agreeing on a sequence of commands) is central to the operation and performance of distributed systems. A well-known solution to consensus is Fast Paxos. In a recent paper, Lamport enhances Fast Paxos by leveraging the commutativity of concurrent commands. The new primitive, called Generalized Paxos, reduces the collision rate, and thus the latency of Fast Paxos. However if a collision occurs, Generalized Paxos needs four communication steps to recover, which is slower than Fast Paxos. This paper presents FGGC, a novel consensus algorithm that reduces recovery delay when a collision occurs to one. FGGC tolerates $f < n/2$ replicas crashes, and during failure-free runs, processes learn commands in two steps if all commands commute, and three steps otherwise; this is optimal. Moreover, as long as no fault occurs, FGGC needs only $f + 1$ replicas to progress.**

I. Introduction

The consensus primitive enables a set of replicas to agree on a common order of update commands to execute. As consensus is so central to distributed fault-tolerance, it is important to improve its performance. Generalized consensus extends this problem to agreeing on *equivalent* sequences, enabling speculative optimisations. For instance, when commands commute, all orders are equivalent: a replica may speculatively accept any command as the next one, predicting that it commutes with concurrent ones. A similar speculation is possible when the network usually delivers commands in the same order to all replicas.

Almost all applications would benefit from these optimisations if they are successful. Typically, the majority of operations commute: any read operation commutes with any other; reads and writes to independent items commute; and high-level operations such adding and removing items of a set, commute logically (even though they might conflict at a low level). Typical local-area networks, interconnection networks and networks-on-chip usually deliver messages in the same order to all destinations (in contrast to a WAN).

However, if replicas mis-predict, a so-called *collision* occurs; the protocol must *recover*, which may be so costly as to eclipse the above gains.

Thus, Lamport's Generalized Paxos protocol [12] has a latency of two communication steps, during "nice" runs, when the algorithm predicts correctly (i.e., either the commands are already ordered by the network or they commute). However, recovery costs an extra four communication steps.

Our experimental evaluation confirms that previous generalized consensus algorithms suffer from a number of performance issues. The optimisations of Fast Paxos [4] and Generalized Paxos are overshadowed by the cost of recovery from a collision, and by a high computational overhead. Furthermore, Generalized Paxos is wasteful of computational resources: to tolerate f faults, it requires $3f + 1$ replicas and accesses at least $2f + 1$ of them.

This paper describes Fast Genuine Generalized Consensus (FGGC), a new generalized consensus algorithm that includes a number of improvements. FGGC recovers from collisions in a single step. FGGC is *genuine*, i.e., it takes more than two steps only when commands are non-commuting. FGGC remains fault-tolerant and uses resources sparingly: to tolerate f crashes, FGGC uses only $2f + 1$ replicas (the optimum), and accesses the same $f + 1$ replicas as long as no fault occurs. Its decreased theoretical complexity translates into actual performance gains: *(1)* On a LAN, Paxos outperforms the other algorithms; however, FGGC performs best of the alternatives and remains close to Paxos. *(2)* In a WAN, the performance of FGGC is comparable to Paxos when collisions are frequent, and outperforms all other algorithms by a wide margin, when commands commute even with low probability.

The general outline of our approach is the following: *(1)* We distinguish *read* and *write quorums*, and define a *centered ballot* (one where the coordinator alone forms a read quorum). *(2)* If the coordinator of a centered ballot remains the same at the next ballot, it can transfer information between ballots and execute Phase 1 locally, reducing recovery to two steps. *(3)* In addition, if centered ballots contain a unique write quorum, this can be further reduced to one step: When an acceptor detects a collision, it waits until it receives a "2B message" from the ballot coordinator. The acceptor then looks at the prefixes of the commands it accepted by receiving commands from proposers, takes the largest such prefix that is compatible with the "2B message", and picks their least upper bound. *(4)* Furthermore, we propose a number of optimisations such as an improved least-upper-bound computation, batching commands, and avoiding redundant messages.

Technically, this requires relatively small changes to the specification of Generalized Paxos. However, the reasoning is far from trivial. To understand them and argue that they are correct in the general case requires a rather subtle theoretical understanding. Therefore, in the first few sections, we explain the theory and sketch the proofs (we refer to Sutra's PhD thesis [19] for the full proofs): basic definitions in Section II; a detailed description of Generalized Paxos in Section III; and finally, improvements to recover, first in two steps, then in one (Section IV).

After this, we describe our implementation and further optimisations (Section V), and evaluate the results experimentally (Section VI).

We survey related work in Section VII. We conclude in Section VIII.

II. The generalized consensus problem

Consensus algorithms such as Paxos [11] agree on a single sequence of commands. In contrast, generalized consensus aims to agree on an *equivalence class* of sequences. In order to reason about equivalence classes, Lamport defines a very general abstraction, the command structure (CStruct). This section recapitulates the main concepts from Lamport [12], then defines the *fast* and *genuine* properties.

A. Command structures

Let Cmd be a set of operations or *commands*. A command sequence, or *c-seq* hereafter, is a finite sequence of commands, e.g. $\sigma = \langle C_1, \ldots, C_{n\geq 0}\rangle$. We note $\circ$ the usual concatenation operator. When a c-seq σ contains a command C, we shall note it $C \in \sigma$. Given a set of commands $\mathcal{C}$, the set $cseq(\mathcal{C})$ contains all the command sequences constructed with commands from $\mathcal{C}$. We shall write $CSeq$ the set $cseq(Cmd)$.

A command structure set is a triple $(CStruct, \bullet, \bot)$ where *(i)* $CStruct$ is a set of command structures, or *c-structs* hereafter, *(ii)* $\bullet$ is an operator from $CStruct \times Cmd$ to $CStruct$ that *appends* a command to a c-struct and produces a new c-struct, and *(iii)* $\bot$ is an element in $CStruct$ called the null c-struct. When it is clear from the context, we shall write in the following $CStruct$ instead of $(CStruct, \bullet, \bot)$.

We extend the operator $\bullet$ inductively to command sequences such that $u \bullet \langle C_1, \ldots, C_{n\geq 0}\rangle$ equals u if n equals 0, and $(u \bullet C_1) \bullet \langle C_2, \ldots, C_n\rangle$, otherwise.

A c-struct u contains a c-seq σ iff $\bot \bullet \sigma$ equals u. By extension, u contains a command C if there exists a c-seq in u that contains C. In the following, for some set of commands $\mathcal{C}$, we shall note $Str(\mathcal{C})$ all the c-structs constructable with commands in $\mathcal{C}$, i.e., the set $\{\bot \bullet \sigma : \sigma \in cseq(\mathcal{C})\}$.

We write $\sqsubseteq$ the pre-order over $CStruct$ induced by $\bullet$, and we say that u *prefixes* v when $u \sqsubseteq v$ holds, i.e., $u \sqsubseteq v = \exists\ \sigma \in CSeq, u \bullet \sigma = v$.

Given a set of c-structs $\mathcal{U} \subseteq CStruct$, a lower bound of $\mathcal{U}$ is a c-struct v such that for all u in $\mathcal{U}$, v prefixes u. A c-struct w is the greatest lower bound (glb) of $\mathcal{U}$ if for every lower bound v of $\mathcal{U}$, v prefixes w. When the glb of $\mathcal{U}$ exists, we write it $\sqcap\, \mathcal{U}$. Similarly an upper bound of $\mathcal{U}$ is a c-struct v such that for all u in $\mathcal{U}$, u prefixes v. When there exists an upper bound of $\mathcal{U}$, we say that $\mathcal{U}$ is *compatible*. A least upper bound (lub) of $\mathcal{U}$ is a c-struct w such that for every upper bound v of $\mathcal{U}$, w prefixes v. When the lub of $\mathcal{U}$ exists, we note it $\sqcup\, \mathcal{U}$. For the sake of simplicity, given two c-structs u and v, we shall write $u \sqcup v$ (respectively $u \sqcap v$) instead of $\sqcup\, \{u,v\}$ (resp. $\sqcap\, \{u,v\}$).

We make the following assumptions about $CStruct$:

CS1. $CStruct = \{\bot \bullet \sigma : \sigma \in CSeq\}$

CS2. $\sqsubseteq$ is a reflexive partial order over $CStruct$.

CS3. $\forall \mathcal{C} \subseteq Cmd, \forall \mathcal{U} \subseteq Str(\mathcal{C})$,

- $\forall v \in CStruct, \forall u \in \mathcal{U}, v \sqsubseteq u \Rightarrow v \in Str(\mathcal{C})$
- $\mathcal{U}$ *compatible* $\Rightarrow \sqcup\, \mathcal{U} \in Str(\mathcal{C})$
- $\forall u, v \in \mathcal{U}, \{u,v\}$ *compatible* $\Rightarrow \mathcal{U}$ *compatible*

CS4. $\forall C \in Cmd, \forall \mathcal{U} \subseteq CStruct$,
$(\mathcal{U}$ *compatible* $\wedge\, \forall u \in \mathcal{U}, C \in u) \Rightarrow C \in \sqcap\, \mathcal{U}$

B. Generalized consensus

Generalized consensus is a distributed system that consists of two finite sets of processes: $Proposers$ and $Learners$. A proposer p holds a variable $proposed_p$ that contains the commands p *proposes* to generalized consensus. It repeatedly executes action $propose(C)$ that adds a command C to $proposed_p$. Hereafter, we note $prpCmd$ the set $\cup_{p \in Proposers} proposed_p$. A learner l holds a c-struct $learned_l$, and *learns* a new c-struct u by executing $learn(u)$. This action assigns u to $learned_l$; we say that learner l learns a command C when $C \in u$.

Processes may fail (*crash*) during a run; a process that does not crash is said *correct*. Initially, for every proposer p, variable $proposed_p$ equals $\{\}$, and for every learner l, $learned_l$ equals $\bot$. Runs of generalized consensus satisfy the following properties:

- **Non-triviality.** For every learner l, $learned_l \in Str(prpCmd)$ always holds.
- **Stability.** It is always the case that $learned_l = v$ implies $v \sqsubseteq learned_l$ at all later times, for any learner l and c-struct v.
- **Consistency.** The set $\{learned_l : l \in Learners\}$ is always compatible.
- **Liveness.** For any command C and any learner l, if l is correct and either (i) a correct proposer proposes C, or (ii) some learner learns C, then eventually l learns C.

Different definitions of $(CStruct, \bullet, \bot)$ correspond to different distributed tasks. For instance, if c-struct is a set of commands, and the operator $u \bullet C$ equals $u \cup \{C\}$. This is equivalent to uniform reliable broadcast [10]. If now a command structure is a singleton or the empty set, and $u \bullet C$ equals C if u is empty, or u otherwise, then we this is the classical consensus problem. We construct uniform atomic broadcast [10] when a c-struct is a c-seq and $u \bullet C$ equals u if u contains C, and $u \circ C$ otherwise. To transform generalized consensus into uniform generic broadcast [15] we instantiate c-structs as Mazurkiewicz traces [5]: Let $\asymp$ be a binary, symmetric and irreflexive relation over Cmd, modeling that two commands are dependent or *non-commuting*. A command history u is a digraph $(\mathcal{E}_u, <_u)$ where $\mathcal{E}_u$ is a subset of Cmd, and $<_u$ is a partial order over $\mathcal{E}_u$. We let $\bot$ be the empty graph. The cstruct $v = u \bullet C$ equals u if v

contains C, otherwise, $\mathcal{E}_v$ equals $\mathcal{E}_u \cup \{C\}$, and $<_v$ equals $<_u \cup \{(D,C) : D \in \mathcal{E}_u \wedge D \asymp C\}$.

C. Properties

Consider an algorithm $\mathcal{A}$ implementing generalized consensus in a message-passing distributed system. A run ρ of $\mathcal{A}$ is *nice* when no fault occurs during ρ, and the system behaves synchronously. Algorithm $\mathcal{A}$ is *genuine* when during nice runs every command is learned in two communication steps if $CStruct$ is compatible. Algorithm $\mathcal{A}$ is *fast* when during nice runs (i) if all the processes receive messages in the same order, every command is learned in two steps and (ii) every command is learned in at most three steps.

III. GPaxos

This section describes GPaxos, an algorithm *à la* Generalized Paxos [12] to solve generalized consensus in message-passing distributed systems. GPaxos is identical to Generalized Paxos, except that it distinguishes between read and write quorums. The later sections will improve GPaxos to ensure fast recovery.

We assume a partially-synchronous message-passing distributed system of deterministic processes. Links between processes are unreliable, and processes may crash. Algorithm 1 specifies GPaxos as a set of actions. A comment in square brackets indicates the role of the process, either a proposer ($Proposers$), an acceptor ($Acceptors$), a coordinator ($Coordinators$), or learner ($Learners$). GPaxos makes use of $2f+1$ acceptors and $f+1$ coordinators where $f > 0$ is the maximum number of crashes GPaxos tolerates to be live. A set of $f+1$ acceptors is called a *majority set*.

A. 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 totally ordered by a relation $<$. Given a balnum m, we assume the existence of a smallest balnum higher than m, noted $m{+}{+}$. and a highest balnum smaller than m, noted $m{-}{-}$. In the remainder of this paper, we identify a ballot by its balnum.

During a ballot, a learner attempts to learn one or more c-structs containing proposed commands. GPaxos relies on quorums of acceptors, i.e. non-empty subsets of $Acceptors$, to remember the c-structs learned during a ballot. Quorums are constructed as follows: We map to each ballot m a set of write quorums: $wquor(k)$, and a set of read quorums: $rquor(k)$. An element in $wquor(m)$ is a write quorum of m, or for short a m-wquorum. Similarly an element in $rquor(m)$ is a read quorum of m, abbreviated in m-rquorum. A ballot m is either *fast* or *classic*, and is associated with a unique coordinator $coord(m)$ in $Coordinators$. In addition, we assume hereafter that:

Q1. Given a ballot m, two m-wquorums W and W' and a m-rquorum R, it holds that: $W \cap W' \neq \{\}$ and $R \cap W \neq \{\}$.

Q2. Given a fast ballot m, two m-wquorums W and W', and a m-rquorum R, $W \cap W' \cap R \neq \{\}$.

As in previous Paxonian algorithms, processes know *a priori* the mapping of ballots to quorums and to coordinators. For instance, if every process is at the same time an acceptor and a coordinator, and a balnum is a natural integer, then the following is a valid assignment: Ballot m is assigned to the m^{th} coordinator (modulo $|Coordinators|$). A ballot m is fast iff m is even. For every ballot m, the read quorums of m are all the majorities sets. If m is classic, the write quorums of m are all the majorities sets; otherwise m is fast, and the write quorums of m are the set Q of acceptors such that $|Q| > \frac{3n}{4}$.

B. GPaxos details

To propose a command C, an acceptor executes $propose(C)$. This action sends a propose message to all the acceptors and to all the coordinators in the system (line 3).

Acceptors constitute the stable memory of the system. They successively join ballots, and vote during them. Each acceptor a maintains three variables: its current ballot: bal_a, the latest ballot during which it voted for, or *accepted*, a c-struct: $cbal_a$, and the c-struct it accepted at ballot $cbal_a$: $cval_a$.

At the beginning of ballot m, $coord(m)$ tries to convince acceptors to join m. If enough acceptors participate in m, $coord(m)$ suggests one or more c-structs. To this goal, a coordinator c stores the latest ballot it started: bal_c, and the latest c-struct it has suggested at ballot bal_c: $maxTried_c$. If no c-struct was suggested so far, $maxTried_c$ equals $none$ (an element that is not in $CStruct$).

In the initial state, every acceptor has joined ballot 0 and accepted $\bot$, and every learner has learned $\bot$. In other words: *(i)* for every acceptor a, both bal_a and $cbal_a$ equal 0, and $cval_a$ equals $\bot$, *(ii)* for every learner l, $learned_l$ equals $\bot$, *(iii)* for every coordinator c, $maxTried_c$ equals 0; and *(iv)* if $c = coord(0)$, then $maxTried_c = \bot$, otherwise $maxTried_c = none$.

A c-struct u is *chosen* at some ballot m, when there exists a m-wquorum of acceptors W, such that for every acceptor a in W, u prefixes the c-struct accepted by a at ballot m. A c-struct u is *choosable* at some ballot m, if it is chosen at m, or it might later be chosen at m. Once a c-struct is chosen, it might be learned by learners.

Two key invariants of GPaxos ensure that learners never learn incompatible c-structs:

GPSafety-1 If two c-structs u_1 and u_2 are accepted at some classic ballot m, then $\{u_1, u_2\}$ is compatible.

GPSafety-2. If an acceptor a accepts a c-struct u at some ballot m, then u is *safe*, i.e., for every c-struct v choosable at some ballot $n < m$, c-struct v prefixes u.

We explain how GPaxos maintains these invariants by detailing how it executes a classic ballot:

- $phase1A(m)$: When it start a ballot m, the coordinator of m, denoted hereafter c, sends a 1A message labelled

Algorithm 1 GPaxos generalized consensus algorithm, as executed by some process i

```
1: propose(C) [proposer]
2:   pre: C ∈ Cmd
3:   eff: send (propose,C) to Acceptors ∪ Coordinators
4:
5: phase1A(m) [coordinator]
6:   pre: maxStart_i < m
7:        i = coord(m)
8:   eff: maxTried_i := none
9:        maxStart_i := m
10:       send (1A,m) to Acceptors
11:
12: phase1B(m) [acceptor]
13:   pre: bal_i < m
14:       rcv_coord(m) (1A,m)
15:   eff: bal_i := m
16:       send (1B, m,cbal_i, cval_i) to coord(m)
17:
18: phase2Start(m,R,k) [coordinator]
19:   pre: maxTried_i = none
20:       maxStart_i = m
21:       ∀a ∈ R, rcv_a (1B,m,_,_)
22:       k = max{n < m : ∃a ∈ R, rcv_a (1B,m,n,_)}
23:       ∀n ∈ ℕ, k ≤ n < m ⇒ R ∈ rquor(n)
24:   eff: W := {W ∈ wquor(k) : ∀a ∈ W ∩ R, rcv_a (1B,m,k,_)}
25:       if W = {}
26:         maxTried_i := u s.t. ∃a ∈ Acceptors, rcv_a (1B,m,k,u)
27:       else
28:         Let γ(W) ≜ ⊓{u : ∃a ∈ R ∩ W, rcv_a (1B,m,k,u)}
29:         maxTried_i := ⊔{γ(W) : W ∈ W}
30:       send (2A,m,maxTried_i) to Acceptors
31:
32: phase2AClassic(m,C) [coordinator]
33:   pre: maxTried_i ≠ none
34:       maxStart_i = m
35:       ∃p ∈ Proposers, rcv_p (propose,C)
36:       ¬isFast(m)
37:   eff: maxTried_i := maxTried_i • C
38:       send (2A,m,maxTried_i) to Acceptors
39:
40: phase2BClassic(m,u) [acceptor]
41:   pre: rcv_coord(m) (2A,m,u)
42:       bal_i ≤ m
43:       ∃W ∈ wquor(m), a ∈ W
44:       cbal_i ≠ bal_i ∨ cval_i ⊏ u
45:   eff: cval_i := u
46:       bal_i := m
47:       cbal_i := m
48:       send (2B,m,cval_i) to Learners
49:
50: phase2BFast(C) [acceptor]
51:   pre: isFast(cbal_i)
52:       bal_i = cbal_i
53:       ∃p ∈ Proposers, rcv_p (propose,C)
54:   eff: cval_i := cval_i • C
55:       send (2B,cbal_i,cval_i) to Learners
56:
57: learn(m,W,u) [learner]
58:   pre: W ∈ wquor(m)
59:       ∀a ∈ W, ∃v ∈ CStruct, rcv_a (2B,m,v) ∧ u ⊑ v
60:   eff: learned_i := ⊔ {learned_i, u}
61:
```

m to the acceptors (line 10).

- $phase1B(m)$: 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 16).

- $phase2Start(m,R,k)$: The coordinator c executes this action once there exists a ballot k and a quorum R such that (i) $coord(m)$ has received a 1B message labelled m from every acceptor in R, (ii) k is the highest ballot $coord(m)$ has heard of in the 1B messages it has received from the acceptors in R, and (iii) for every ballot n such that $k \leq n < m$, R is a read quorum of n. In Lamport [12] any two quorums intersect. Because our assumptions on quorums are weaker than Lamport's (see Q1 and Q2), we need condition (iii). We explain the reason why by proving that when $coord(m)$ executes either line 26 or 29, $maxTried_{coord(m)}$ is safe at ballot m:

  Proof (sketch). Assume that a c-struct v is choosable at some ballot $n \leq m$, and suppose that GPSafety-2 holds for every ballot prior to m. First, observe that there exists an acceptor a in R such that a has accepted some c-struct at ballot k, and $coord(m)$ received a 1B message from a (line 22). Then consider the following cases:

  Case $n < k$: By invariant GPSafety-2, every c-struct accepted at ballot k by acceptors in R suffixes v. If now $\mathcal{W}$ is empty (line 25), then u suffixes v. Otherwise, for every quorum W, $\gamma(W)$ suffixes v (line 28). Hence, $maxTried_{coord(m)}$ suffixes v.

  Case $k \leq n < m$: R is a read quorum of n (line 23). Thus, for every n-wquorum W, assumption Q1 tells us that $W \cap R \neq \{\}$. It follows that at least one acceptor in R has accepted a c-struct at ballot n. This contradicts the definition of k.

  □

  Once $coord(m)$ has extracted a c-struct that is safe at ballot m and stored it in $maxTried_{coord(m)}$, $coord(m)$ suggests it to the acceptors in a 2A message (line 30).

- $phase2AClassic(m,C)$: When $maxTried_{coord(m)}$ differs from $none$, by construction $maxTried_{coord(m)}$ is safe at m. If m is classic, $coord(m)$ appends newly proposed commands to $maxTried_{coord(m)}$ and suggests the resulting c-struct to the acceptors (line 38).

- $phase2BClassic(m,u)$: When an acceptor a belonging to a m-wquorum receives a 2A message containing a c-struct u and a can join ballot m, or joined it previously, a accepts u by assigning u to $cval_a$ (line 45). Acceptor a then updates $cbal_a$ and bal_a to the value of m (lines 46 and 47), and sends a 2B message containing $cval_a$ to the learners (line 48). Since c-struct u extends $maxTried_{coord(m)}$, every c-struct accepted at ballot m prefixes $maxTried_{coord(m)}$ (invariant GPSafety-1). Moreover $maxTried_m$ is safe at ballot m, thus very accepted c-struct is safe at ballot m (invariant GPSafety-2).

- $learn(m,W,u)$: A learner l learns a c-struct u, once l knows that u is chosen at m (lines 58 and 59).

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 steps in a classic ballot, as this is the length of the causal path between propose, 1A, 1B, 2A and 2B messages. 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, since $coord(0)$ can skip phase one of ballot 0 ($\bot$ is *de facto* safe at ballot 0), every command is learned in three communication steps during a nice run.

C. 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 51), and accepted the safe c-struct suggested by the coordinator (line 52), 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 54), then sends a 2B message containing the new value of $cval_a$ to the learners (line 55).

Commands accepted during a fast ballot are learned in two steps: the causal path contains a propose message and 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 1: *Let a_1 and a_2 be two acceptors that joined a fast ballot m. Suppose that a_1 and a_2 form a m-wquorum, and note 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 in $v = (u \bullet C) \bullet D$. As a consequence, v is chosen at ballot m. If now C and D are received in different orders, e.g. a_1 extends u in v and a_2 extends u in $w = (u \bullet D) \bullet C$, then if C and D commute v equals 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 at i:

$$collide(m) \triangleq \exists W \in wquor(m), \begin{cases} \forall a \in W, rcv_a\ (\mathsf{2B}, m, \_) \\ \neg\,(\{u : \exists a \in W, rcv_a\ (\mathsf{2B}, m, u)\}\ compatible) \end{cases}$$

When a collision occurs at ballot m, 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). Reducing this delay is the subject of the next section.

IV. Fast recovery from collisions

Most of the time, when a collision occurs at some ballot m, the coordinator of ballot $m{+}{+}$ is also the coordinator of ballot m, and after detecting the collision it starts ballot $m{+}{+}$. Lamport [14] observes that when this type of situation happens in Fast Paxos, if $coord(m)$ receives a 2B message at ballot m from an acceptor a, it knows the same information as if it had received a 1B message from a at ballot $m{+}{+}$, i.e., (i) since there is no ballot between m and $m{+}{+}$, a will not join a ballot smaller than $m{+}{+}$, and (ii) m is the highest ballot at which a voted for some value. As a consequence, we may apply the following optimization :

Coordinated recovery. We require that $Coordinators \subseteq Acceptors$. If a collision occurs at ballot m, $coord(m{+}{+})$ considers the 2B messages sent during ballot m, as 1B messages for ballot $m{+}{+}$.

With the coordinated recovery technique, $coord(m{+}{+})$ skips phase one of ballot $m{+}{+}$. This saves two communication steps when a collision occurs in Fast Paxos.

An acceptor of GPaxos continuously accepts newly proposed commands during fast ballots. This implies that when a collision occurs at ballot m, $coord(m{+}{+})$ cannot use the 2B messages it received at ballot m to pick a safe c-struct; we illustrate this below:

Example 2: *Consider three commands C, D and E such that C and D are non-commuting, and E commutes with both C and D. Let m be a fast ballot, $W = \{a_1, a_2\}$ be a m-wquorum, and suppose that a_1 accepts $u = (\bot \bullet C) \bullet D$ at ballot m, while a_2 accepts $v = (\bot \bullet D) \bullet C$. Ballot m collides. If $coord(m{+}{+})$ starts ballot $m{+}{+}$, then $u \sqcap v$, which equals $\bot$, should be safe. However, concurrently to the beginning of ballot $m{+}{+}$, acceptors a_1 and a_2 accept command E. Hence, $\bot \bullet E$ is now chosen at ballot m. This violates invariant GPSafety-2.* □

To maintain invariant GPSafety-2 $coord(m{+}{+})$ must know what was chosen at ballot m. Since acceptors accept at will new commands, $coord(m{+}{+})$ cannot skip phase one, and thus we cannot employ the coordinated recovery technique. In the remaining of this section we first present a simple variant of GPaxos that reduces latency to four steps when a collision occurs. Then, we depict our complete solution to recover in one step.

A. Recovery in two steps

To recover in two steps we introduce the concept of a *centered* ballot. We say that a ballot m is centered when every m-wquorum contains $coord(m)$, i.e., $centered(m) = \forall W \in wquor(m), coord(m) \in W$. If m is centered, observe that $\{coord(m)\}$ is by construction a read quorum of m. Consider a ballot m that collides, and assume that the coordinator of ballot m is the coordinator of ballot $m{+}{+}$. Then, the coordinator may execute phase one of ballot $m{+}{+}$ locally:

Two-step recovery. We require that $Coordinators \subseteq Acceptors$. For every fast ballot m, m is centered and $\{coord(m)\}$ is a m-rquorum.

We now further refine this recovery technique to recover in one step when a collision occurs.

B. Recovery in one step

Algorithm 2 depicts the code of FGGC, a fast genuine generalized consensus algorithm. It adds action *recover* to the code of Algorithm 1. To recover in one step when a collision occurs at ballot m, FGGC allows acceptors to spontaneously accept, at ballot $m{+}{+}$, a c-struct built from what $coord(m)$ accepted at ballot m. In more detail, our algorithm works as follows:

One-step recovery. We assume that (**FGGC1**) $Coordinators \subseteq Acceptors$ and $Acceptors \subseteq Learners$, and that (**FGGC2**) every fast ballot is centered and associated to a unique write quorum. When acceptor a detects that ballot $cbal_a$ collides (line 4), a executes $recover(u)$ provided that: ballot $cbal_a{+}{+}$ is fast (line 5), a belongs to some write quorum of $cbal_a{+}{+}$ (line 6), a did not join a higher ballot than $cbal_a$ (line 7), and a received a 2B message from the coordinator of $cbal_a$ containing u (line 8). When acceptor a executes $recover(u)$, it joins ballot $cbal_a{+}{+}$, and spontaneously accepts at this ballot the least upper bound of the set consisting of u and the greatest prefix of $cval_a$ compatible with u.

We prove informally below that executing action *recover* maintains invariant GPSafety-2:

Proof (sketch). Note m the value of bal_a when a executes $recover(u)$, assume that GPSafety-2 holds for ballots prior or equal to m. and let v be a c-struct choosable at some ballot $n \leq m$. Variable $safe$ is the c-struct initially accepted by a at ballot $m{+}{+}$ (line 13), hence to satisfy GPSafety-2 at ballot $m{+}{+}$, we must show that v prefixes $safe$. We first observe that c-struct u prefixes $cval_{coord(m)}$ (line 8), then we consider the following cases:

Case $n < m$: Since $cval_a$ is safe at ballot m, and $n < m$, v prefixes $cval_a$. By a similar reasoning, we obtain that v prefixes $cval_{coord(m)}$ which implies that $\{u,v\}$ is compatible. Thus by construction $\mathcal{V}$ contains v. We conclude that $safe$ suffixes v.

Case $n = m$: If a accepted some c-struct at ballot m, then a belongs to a m-wquorum (lines 43 and 52 in Algorithm 1 and line 6 in Algorithm 2). Besides, ballot m collides, thus it is fast. Our algorithm requires that if m is fast, there is a single m-wquorum (assumption FGGC2). As a consequence, if v is chosen at m, v prefixes $cval_a$. By a similar reasoning we conclude that v prefixes $cval_{coord(m)}$. This implies that $\{u,v\}$ is compatible. Hence $\mathcal{V}$ contains v, which implies that $safe$ suffixes v.

□

Algorithm 2 Fast Genuine Generalized Consensus, as executed at process i

```
1: // Actions propose, phase1B, phase2Start, phase2Start,
   phase2BClassic, phase2BFast and learn: unchanged from
   Algorithm 1.
2:
3: recover(u) [acceptor]
4:   pre: collide(cbal_i)
5:        isFast(cbal_i++)
6:        ∃W ∈ wquor(cbal_i++), a ∈ W
7:        cbal_i = bal_i
8:        ∃u ∈ CStruct, rcv_{coord(cbal_i)} (2B,u,cbal_i)
9:   eff: let V = {v ∈ CStruct : v ⊑ cval_i ∧ {u,v} compatible}
10:       let safe = ⊔ {u, ⊔ V}
11:       bal_i := bal_i++
12:       cbal_i := bal_i
13:       cval_i := safe
14:       send (2B,cbal_i,cval_i) to Learners
15:
```

Liveness: FGGC fulfills the liveness clause of generalized consensus if for every correct learner l and every command C, either proposed by some correct proposer or learned by some learner, there exists a ballot m, a m-wquorum W, and a c-struct u containing C, such that l executes $learn(m,W,u)$. This requires that at most f acceptors crash, and some synchrony assumptions. In Sutra [19], we provide a proof of progress in the unreliable failure detector model.

Latency: To ensure that all commands are learned in at most three steps, FGGC needs to continuously execute fast ballots using the same m-wquorum W. Nevertheless, if some crash occurs and W is not available anymore, FGGC must be able to execute classic ballots. To satisfy this requirement, we must construct $BalNum$ carefully. We give such a construction below:

Example 3: *A balnum m is a couple $m = (i,j)$ where $i \in \{0,1\}$ and $j \in \mathbb{N}$. If i equals 0, m is fast and coordinated by the first acceptor; otherwise m is classic and coordinated by the j^{th} acceptor (modulo $|Acceptors|$). Ballot 0 equals (0,0). If m is fast, the single write quorum of m is some majority set containing the first coordinator. If m is classic, every majority set is a write quorum of m. A majority set is always a read quorum, and if ballot m is fast $\{coord(m)\}$ is also a read quorum of m. For some ballot $m = (i,j)$, $m{+}{+}$ equals $(i,j+1)$. Relation $<$ over $BalNum$, is defined given two ballots $m = (i,j)$ and $n = (k,l)$, by: $m < n \stackrel{\triangle}{=} i < k \vee (i = k \wedge j < l)$.* □

The definition above ensures that *(i)* FGGC executes fast ballots $(0,0), (0,1), \ldots$, switching from one ballot to a higher ballot only if a collision occurs, and *(ii)* FGGC is able to switch to a classic ballot if a crash occurs.

V. Implementation

We implemented FGGC in the Daisylib group communication library [2]. Our implementation is written in Java. It

Algorithm	*CStruct*	FASTBALLOTS?	Recovery
Paxos	singleton	*false*	DEFAULT
Fast Paxos[a]	singleton	*true*	TWOSTEP
[4]	singleton	*true*	ONESTEP
Paxos[b]	c-seq	*false*	DEFAULT
Fast Paxos[b]	c-seq	*true*	TWOSTEP
[4][b]	c-seq	*true*	ONESTEP
GPaxos	trace	*true*	DEFAULT
Section IV-A	trace	*true*	TWOSTEP
FGGC	trace	*true*	ONESTEP

[a]With the coordinated recovery technique of Section IV
[b]Repeated in order to implement state machine replication

Table I
EXECUTION MODES

has 5,200 lines of code, of which 1,100 for the command structure abstraction.

The core of our implementation is GPaxos with $Coordinators = Acceptors \subseteq Learners \subseteq Proposers$. The different blocks: proposer, acceptor, coordinator and learner, are executed in isolation. They communicate via a message-passing interface on top of TCP/IP. To track learned c-structs and collisions, each block maintains the c-structs accepted at each ballot, and updates it every time a 2B message is received. When a c-struct u is learned at some ballot m, ballots prior to m are removed to bound memory consumption.

Coordinators use an eventual leader service and an eventual failure detector service running between acceptors to time the start of a new ballot. More precisely, a coordinator c starts a new ballot when c is the leader of $Acceptors$, and either (i) every write quorum of $maxStart_i$ contains a faulty process, or (ii) $maxStart_c$ is fast, a collision happens at that ballot and the recovery technique is set to DEFAULT or TWOSTEP (these flags are explained shortly).

Our implementation is very flexible and allows multiple execution modes. Three parameters defines an execution mode:

- The definition of *CStruct*. A c-struct can be either a singleton, a c-seq or a trace (see Section II-B).
- The mapping from balnums to read/write quorums. When the flag FASTBALLOTS is *false*, every balnum is classic; otherwise every balnum is fast.
- The recovery technique applied when a collision occurs. The value DEFAULT indicates that the recovery technique of GPaxos is used, TWOSTEP stands for the technique covered in Section IV-A, and ONESTEP indicates the technique described in Section IV-B.

Each execution mode corresponds to a particular Paxonian algorithm. Table I summarizes the execution modes, and the algorithm implemented by each mode.

To make our framework efficient, we also introduced several optimizations that we describe below:

Computing with c-structs: When a c-struct is either a c-seq or a trace, computing the glb (or the lub) of a set of c-structs is expensive. For instance, if each c-struct is a c-seq containing n commands, and there are m such c-structs, computing the glb costs $O(nm)$ operations. When c-structs are traces, complexity raises to $O(n^2m)$. We applied two techniques to reduce this complexity. First, we always use a single write quorum per ballot. With a single write quorum, there is no need to execute line 28 in Algorithm 1, and only the computation of the lub is necessary. Our second optimization is to cache the glb and to update it incrementally when the process receives a 2B message.

Batching commands: When generalized consensus is under high load, processing one command at a time is expensive. To reduce this cost, we batch proposed commands. More precisely, when a proposer proposes multiple commands, we group them in a particular command, called *command array*. Two command arrays commute if the commands they contain commute.

Reducing redundancy: When a process sends a monotonically growing c-struct, e.g., u, $u \bullet A$, $u \bullet A \bullet B$, ..., much of this information is redundant. To reduce redundancy, we leverage First-In First-Out links. The first time a process i sends a c-struct u, i sends it entirely. Then, each time i appends a new command A to u, i sends only A. This optimization is used when a coordinator executes $phase2AClassic$, and when an acceptor executes actions $phase2BClassic$ and $phase2BFast$.

VI. EXPERIMENTAL EVALUATION

In this section, we compare Paxos, the improved Fast Paxos of Charron-Bost and Schiper [4], GPaxos and FGGC. We make our comparison using the workload of Mencius [**?**]. This workload consists of commands to randomly read or write from a set of read/write registers. As the number of register increases, so does the probability that two commands will commute. A client submits commands in closed loop, i.e., it submits a command, waits until it returns, and immediately submits a new one.

A. Experimental settings

A command consists of the following fields: operation type (read or write, 1 byte); register name (2 bytes); the sequence number (2 bytes); and client ID (2 bytes).

We evaluated the protocols using a fully inter-connected cluster of bi-processor AMD-64 computers, running at 2.4 Ghz with 2 GB of memory; each processor has two cores. A gigabit Ethernet links the nodes. The bandwidth and message delay of our local network, measured using netperf and ping, are 940 Mbps and 0.05 ms respectively.

A physical node runs either all four roles, or a proposer, a learner and a certain amount of clients. In the former case we call it a *server node*, and in the latter a *client node*. We always run three client nodes and three server nodes ($f = 1$). The number of clients per client node varies from 10 to 400.

When a learner learns a command C, it executes C, sends a reply to the client that submitted it, and logs the first four fields of C. Upon receiving the reply from a learner, a

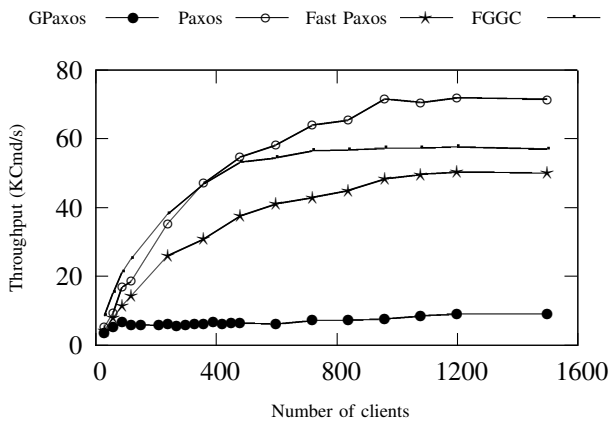
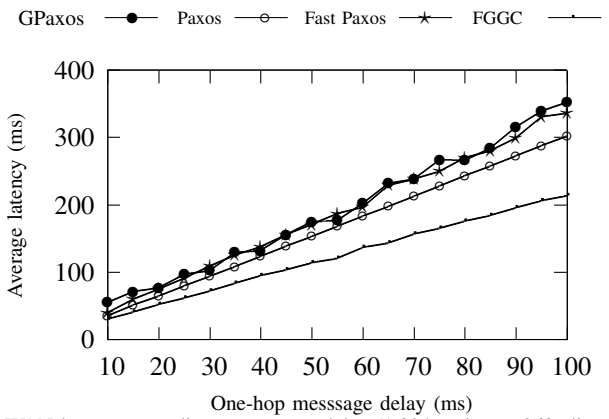
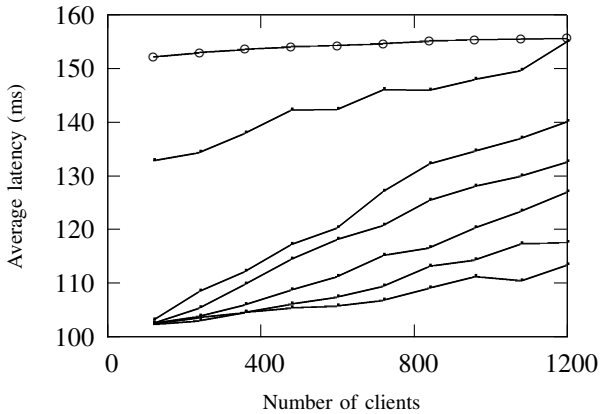


Figure 1. LAN performance



(a) WAN latency according to message delay (1,024 registers, 360 clients)



(b) WAN latency decreases as commutativity increases. Top to bottom: Paxos; FGGC with 1, 1 K, 2 K, 4 K, 8 K and 16 K registers. (Message delay = 50 ms)

Figure 2. WAN performance

client computes and logs the latency of the request. We use the client-side log to analyze experiment results. During an experiment, each client submits 3,000 commands. The first and last 1,000 commands are not taken into account.

In our experiments, we vary the latency of links between nodes to emulate both LAN and WAN topologies. For the former, nodes use the underlying gigabit network directly. In the WAN topology, we use the Linux traffic shaper to force a one-way message delay of 50 ms, except in Section VI-C1 where latency varies from 10 to 100 ms. Bandwidth is limited to 20 MB/s in all WAN experiments.

B. LAN performance

Figure 1 displays the throughput of Paxos, Fast Paxos, FGGC and GPaxos in a LAN setting (higher is better). Clients access 1,024 registers in this experiment.

Observe that the performance of GPaxos is an order of magnitude worse than other algorithms. The explanation for this result is that when a collision occurs in GPaxos, the coordinator sends a 2A message and *a priori* we cannot use fifoness of links to reduce the amount of transmitted information.

The peak performance of Paxos is 72 KCmd/s. Under max load, a command takes 15 ms to execute with a standard deviation of 10 ms. The maximum throughput of FGGC and Fast Paxos is 57 KCmd/s (around 80% of Paxos highest throughput). When both algorithms reach their maximum throughput, the latency of a command is on average 25 ms with a standard deviation of 10 ms.

The most expensive operation in the critical path for FGGC and Fast Paxos is to compute if a set of c-structs is compatible or not. Recovery remains somewhat expensive in our current implementation, as it requires copying two c-structs, and computing the lower upper bound of a set of c-structs. In a future version, we plan to compute incrementally the result of *recover* each time a 2B message is received from the coordinator.

C. WAN performance

We now compare different algorithms in a simulated WAN environment.

1) Latency: Figure 2(a) shows the latency to execute commands as the one-hop message delay between nodes varies (lower is better). In this experiment, we set the number of registers to 1,024, and we run 360 clients.

Observe that the average latency of GPaxos and Paxos are similar. Nevertheless, their variance is very different: Paxos always delivers messages in three communication steps, whereas GPaxos needs between two and six communication steps. For instance, when the one-hop message delay equals 50 ms, Paxos needs 156 ms to execute a command with a standard deviation of 4 ms, whereas GPaxos takes 175 ms with a standard deviation of 130 ms. This shows that the recovery technique of GPaxos is expensive.

As soon as latency is significant, FGGC outperforms the other algorithms. For instance when message delay is 50 ms, FGGC executes commands in 120 ms, with a standard deviation of 24 ms. Notice that, with 360 clients and 1,024

Algorithm	Time complexity			optimal resilience?
	everything commutes	spontaneous order	general case	
[7]	4n	4n	4n	*yes*
[3]	4	4	4	*yes*
[11]	4	4	4	*yes*
[17]	3	3	3	*yes*
[1]	3	2	3	*yes*
[14]	3	2	6	*yes*
[4]	3	2	3	*yes*
[16]	2	3	3	*no*
[12]	2	2	6	*yes*
[20]	2	2	3	*no*
FGGC	2	2	3	*yes*

Table II
COMPLEXITY AND RESILIENCE OF CONSENSUS ALGORITHMS FOR STATE-MACHINE REPLICATION

registers, collisions are not rare: during this experiment both FGGC and GPaxos started around 1,600 ballots.

2) Leveraging commutativity: Figure 2(b) evaluates the influence of commutativity on latency in FGGC (lower is better). In this experiment, we set the message delay to 50 ms. For comparison purposes, we report the latency of Paxos (top curve), and we vary both the number of clients (x axis), and the number of registers (the other curves).

The figure shows that, even with a single register, taking into account the commutativity of reads improves performance with respect to Paxos by 13%. As more registers are added, commutativity increases, which reduces the average latency of FGGC. When fewer than 800 clients access the system concurrently and there are at least 4 K registers, ballots rarely collide, and FGGC is close to the theoretical lower bound. As more clients are added, or fewer registers are available, the performance of FGGC degrades.

The peak throughput of both algorithms is obtained with 1,200 clients. In this setting, both Paxos and FGGC deliver 7,200 commands/s when there is a single register. When there are 16,384 registers, FGGC outputs 9,600 commmands/s with an average latency of 115 ms.

VII. RELATED WORK

State machine replication [18] is a fundamental technique for implementing wait-free linearizable shared data in a message-passing distributed system. This section briefly reviews state machine replication and consensus algorithms when processes may halt by crashing. Table II summarizes our survey.

The classical approach to implement state machine replication is to execute successive instances of consensus. The seminal FLP paper [8] shows that consensus cannot be solved in an asynchronous system using a deterministic algorithm when processes may crash. Fortunately, as real systems behave synchronously most of the time, there has been major progress in fault-tolerant consensus. Notably, both Dwork et al. [7] and Paxos [11] solve consensus deterministically for $f < n/2$ under weak synchrony assumptions.

Metrics for evaluating the performance of a consensus algorithm include: *(i)* best-case time complexity, or *latency*, and *(ii)* the number of failures it tolerates, or *resiliency*. As most runs are "nice" , latency should remain low in this common case. Both Paxos and Chandra et al.'s algorithm [3] solve consensus in three communication steps. The early-deciding algorithm of Schiper [17] improves this, solving consensus in two steps, which is optimal in the general case [13]. These algorithms have optimal resilience: they tolerate $f < n/2$ faults.

Empirically, it is observed that the network often spontaneously delivers broadcast messages in the same order to all receivers. Leveraging this, Brasileiro et al. [1] and Pedone et al. [16] solve consensus in one step during nice runs where all processes propose the same command. Deciding in one step during stable runs requires $f < n/3$ [1]. Optimal resilience requires to decide in one step during stable runs only if at most $f < n/4$ processes crash, as in Lamport's Fast Paxos [14]. Although FGGC decides in one step with only a majority of processes, this does not contradict Lamport's result, as FGGC decides only if a privileged set [9] of processes (precisely, the centered quorum of ballot 0) is alive.

When a collision occurs during a fast ballot, Fast Paxos may have higher latency than Paxos. A recent algorithm by Charron-Bost and Schiper [4] improves Fast Paxos by tentatively executing a fast ballot during a classic ballot. (This is the variant of Fast Paxos used in our performance evaluation.) During nice runs, this algorithm solves consensus in one step if every process proposes the same command, and two otherwise; it has optimal resilience. When a c-struct is a singleton, ballot 0 is fast, the single write quorum of ballot 0 contains all acceptors, and every higher ballot is classic, FGGC behaves similarly, with the subtle difference that, where Charron-Bost and Schiper [4] picks the coordinator of a ballot "on the fly," FGGC assigns it *a priori*. This allows Charron-Bost and Schiper [4] to be zero-degrading [6], i.e., to decide in at most two steps when all crashes occurs initially and the system behaves synchronously (called a stable run). It worth noticing nevertheless that properties of generalized consensus during stable runs are of few interest because this primitive is executed only once to implement state machine replication.

Both Pedone et al. [15] and Lamport [12] observe that since replicas might execute commuting commands in different orders, it is unnecessary to order all commands. The algorithm of Pedone and Schiper [15] tolerates $f < n/3$ crashes and does not leverage the spontaneous ordering of the network. More recently, Zieliński [20] proposed an algorithm to solves generic broadcast in two steps when concurrent commands either commute or were spontaneously ordered by the network, and three steps otherwise. As the author himself concedes, his algorithm is not practical.

Multicoordinated Paxos of **?**] uses more than one coordinator per ballot. 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 c-structs suggested by coordinators collide. In such a case, a higher ballot is started. Multicoordinated Paxos increases dependability at the cost of a higher probability that collisions occur.

VIII. Conclusion

This paper presents FGGC, a fast genuine generalized consensus algorithm. FGGC tolerates $f < n/2$ faults and makes use of $f + 1$ processes to progress. During nice runs, FGGC decides in two communication steps when commands either commute or are spontaneously ordered by the network, and three otherwise. Evaluation in a WAN shows that (i) even if clients access the same objects, FGGC achieves results comparable to Paxos, and (ii) if commands commute with good probability, FGGC outperforms Paxos.

References

[1] Francisco V. Brasileiro, Fabíola Greve, Achour Mostéfaoui, and Michel Raynal. Consensus in one communication step. In *Int. Conf. on Parallel Arch. and Compilation Tech. (PaCT)*, pages 42–50, Barcelona, Spain, 2001. Springer-Verlag.

[2] Lásaro J. Camargos, Nicolas Schiper, Pierre. Sutra, and Marcin Wieloch. Daisylib http://sourceforge.net/projects/daisylib/.

[3] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, 1996.

[4] Bernadette Charron-Bost and André Schiper. Improving Fast Paxos: being optimistic with no overhead. In *Pacific Rim Dependable Computing (PRDC)*, pages 287–295, Riverside, CA, USA, 2006. IEEE Comp. Society.

[5] Volker Diekert. *The Book of Traces*. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1995. ISBN 9810220588.

[6] Partha Dutta and Rachid Guerraoui. Fast indulgent consensus with zero degradation. In *European Dependable Computing Conference (EDCC)*, pages 191–208, Toulouse, France, 2002. Springer-Verlag. ISBN 3-540-00012-7. URL http://portal.acm.org/citation.cfm?id=645333.649854.

[7] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, 1988. ISSN 0004-5411.

[8] Michael J. Fischer, Nancy A. Lynch, and Michael S. Patterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985. doi: http://doi.acm.org/10.1145/3149.214121.

[9] Rachid Guerraoui and Michel Raynal. The information structure of indulgent consensus. *IEEE Transactions on Computers*, 53(4):453–466, 2004.

[10] Vassos Hadzilacos and Sam Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical report, Ithaca, NY, USA, 1994.

[11] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998. ISSN 0734-2071.

[12] Leslie Lamport. Generalized consensus and Paxos. Technical Report MSR-TR-2005-33, Microsoft, March 2005.

[13] Leslie Lamport. Lower bounds for asynchronous consensus. *Distributed Computing*, 19(2):104–125, October 2006. ISSN 0178-2770.

[14] Leslie Lamport. Fast Paxos. *Distributed Computing*, 19(2):79–103, October 2006. ISSN 0178-2770.

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

[16] Fernando Pedone, André Schiper, Peter Urban, and David Cavin. Solving agreement problems with weak ordering oracles. In *European Dependable Computing Conference (EDCC)*, pages 44–61, Toulouse, France, 2002. Springer-Verlag.

[17] A. Schiper. Early Consensus in an Asynchronous System with a Weak Failure Detector. *Distributed Computing*, 10(3):149–157, 1997. doi: NA.

[18] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Computing Surveys*, 22(4):299–319, 1990. ISSN 0360-0300.

[19] Pierre Sutra. *Efficient Protocols for Generalized Consensus and Partial Replication*. PhD thesis, UPMC, November 2010.

[20] Piotr Zieliński. Optimistic generic broadcast. In *Int. Conf. on Distributed Computing (DISC)*, pages 369–383, Cracow, Poland, 2005.