Fast Genuine Generalized Consensus
Pierre Sutra, Marc Shapiro

To cite this version:

Pierre Sutra, Marc Shapiro. Fast Genuine Generalized Consensus. 2010. <hal-00476885v3>

HAL Id: hal-00476885
https://hal.archives-ouvertes.fr/hal-00476885v3
Submitted on 2 Sep 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
Abstract

Consensus is a central primitive for building replicated systems, but its latency constitutes a bottleneck. 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, the latency of Generalized Paxos equals six communication steps, which is higher 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.
1 Introduction

The consensus primitive enables a set of replicas to agree on a common order in which they will execute commands. As consensus is central to fault-tolerance computing, much of the literature is concerned with improving its latency (measured in steps, i.e., one-way message delays). Generalized Consensus [21] improves latency in two but common cases, observing that: (i) Commands are often ordered spontaneously by the network, and (ii) Commuting commands need not be mutually ordered. Lamport’s recent Generalized Paxos [21] takes only two communication steps in these cases.\(^1\) However, under contention, replicas may receive non-commuting commands in different orders, called a collision. Generalized Paxos takes four extra steps to recover.

We propose a new generalized consensus algorithm, FGGC, that recovers from collisions in a single step, yet uses the optimal number of replicas \((2f+1)\). FGGC is genuine, i.e., it requires only two steps when commands commute.\(^2\) FGGC uses the following techniques: (i) We distinguish read and write quorums. (ii) We define a centered ballot, i.e., one where the coordinator alone forms a read quorum. (iii) We show that in centered ballots where the coordinator remains the coordinator of the next ballot, it can transfer information between ballots and execute Phase 1 locally, reducing recovery to two steps. (iv) We then show that, under reasonable conditions, recovery can be done in one step. (v) As long as no fault occurs, FGGC accesses the same \(f+1\) replicas - this contrast with previous genuine generalized consensus algorithms which require \(3f+1\) replicas and access at least \(2f+1\) of them.

In order to recover in one step when an acceptor detects a collision, FGGC assumes that learners hear from a majority of acceptors before they learn, and both learners and acceptors receive Paxos “2B messages”. If 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 that it accepted (by receiving commands directly from proposers), takes the largest such prefix that is compatible with the 2B message, and picks their least upper bound. This saves all the communication steps that would have been required to accept commands in the next ballot.

Roadmap Section 2 recalls some basic definitions. In Section 3, we refine Generalized Paxos with read and write quorums. Then Section 4 refines this, first to two-step collision recovery, then to one. We survey related work in Section 5. A discussion follows in Section 6. We close this paper in Section 7. To ease reading, correctness is discussed only briefly along this paper; full proofs are available in the Appendix.

2 The generalized consensus problem

Consensus algorithms such as Paxos agree on a single sequence of commands. The problem addressed by generalized consensus is to agree on an equivalence class of sequences. To support this, Lamport defined a very general abstraction, the command structure [21]. This section recapitulates the main concepts from Lamport [21], and defines the fast and genuine properties. We follow closely Lamport’s definitions, with small differences that we examine in Section 6.

\(^1\)In “nice” runs, i.e., during which replicas do not crash and the system behaves synchronously.

\(^2\)A non-genuine algorithm, one that orders commands even when they commute, is obviously correct but incurs higher latency.
2.1 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. $\sigma = (C_1, \ldots, C_n)$. We note $\circ$ the usual concatenation operator. When a command $C$ appears in a $c$-seq $\sigma$, we say that $\sigma$ contains $C$, denoted $C \in \sigma$. Given a set of commands $\mathcal{C}$, the set $cseq(\mathcal{C})$ contains all the command sequences constructed with commands from $\mathcal{C}$. In the following, 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 $(CStruct, \bot, \bullet)$.

We extend the operator $\bullet$ inductively to command sequences as follows:

$$u \bullet (C_1, \ldots, C_n) = \begin{cases} u & \text{if } n = 0 \\ (u \bullet C_1) \bullet (C_2, \ldots, C_n) & \text{otherwise} \end{cases}$$

A $c$-struct $u$ contains a $c$-seq $\sigma$ iff $\bot \bullet \sigma$ equals $u$. If there exists a $c$-seq in $u$ that contains a command $C$, then $u$ contains $C$. We note $cmd(u)$ the commands contained in $u$. In the following, for some set of commands $\mathcal{C}$, we note $Str(\mathcal{C})$ all the $c$-structs constructable with the commands in $\mathcal{C}$, that is the set $\{ \bot \bullet \sigma : \sigma \in cseq(\mathcal{C}) \}$.

We write $\subseteq$ the pre-order over $CStruct$ induced by $\bullet$, i.e., $u \subseteq v \triangleq \exists \sigma \in CSeq, u \bullet \sigma = v$. A $c$-struct $u$ prefixes a $c$-struct $v$ when there exists a $c$-seq $\sigma$ such that $v = u \bullet \sigma$.

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 $\bigcap \mathcal{U}$. 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. By extension, we say that $\mathcal{U}$ is $k$-compatible iff $k \leq |\mathcal{U}|$, and for every subset $U$ of $\mathcal{U}$, if $|U| \leq k$, then $U$ 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 $\bigcup \mathcal{U}$. For the sake of simplicity, given two $c$-structs $u$ and $v$, we shall write $u \sqcup v$ (respectively $u \sqcap v$) instead of $\bigcup \{u,v\}$ (resp. $\bigcap \{u,v\}$).

In the sequel of this paper, we assume that $(CStruct, \bullet, \bot)$ satisfies the following assumptions:

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

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

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

(CS3.1) $\forall v \in CStruct, \forall u \in \mathcal{U}, v \subseteq u \Rightarrow v \in Str(\mathcal{C})$

(CS3.2) $\mathcal{U}$ compatible $\Rightarrow \bigcup \mathcal{U} \in Str(\mathcal{C})$

(CS3.3) $\mathcal{U}$ 2-compatible $\Rightarrow \mathcal{U}$ compatible

CS4. $\forall C \in Cmd, \forall \mathcal{U} \subseteq CStruct, (\mathcal{U}$ compatible $\land \forall u \in \mathcal{U}, C \in u) \Rightarrow C \in \bigcap \mathcal{U}$
2.2 Generalized consensus

We consider a global time model. Generalized consensus is a distributed system that consists of two finite sets of processes: Proposers and Learners.

A proposer \( p \) holds a variable \( \text{proposed}_p \) that contains the commands \( p \) proposes to generalized consensus. It repeatedly executes action \( \text{propose}(C) \) that adds a command \( C \) to \( \text{proposed}_p \). Hereafter, we note \( \text{prpCmd} \) the set \( \bigcup_{p \in \text{Proposers}} \text{proposed}_p \). A learner \( l \) holds a c-struct \( \text{learned}_l \), and learns a new c-struct \( u \) by executing \( \text{learn}(u) \). This action assigns \( u \) to \( \text{learned}_l \). We say that learner \( l \) learns command \( C \) when \( u \) contains \( C \).

A run is a finite sequence of actions executed by one or more processes, e.g. the sequence \( \langle \text{propose}_{p_1}(C), \text{learn}_{l_2}(C), \text{learn}_{l_1}(C) \rangle \), where by \( \text{action}_i(\text{args}) \) we denote that process \( i \) executes \( \text{action} \) with arguments \( \text{args} \).

Processes may fail (crash) during a run; a process that does not crash is said correct. We note \( \text{Faulty} \) the set of processes that crash and \( \text{Correct} \) the set of correct processes (the current run will be clear from context). Initially, for every proposer \( p \), variable \( \text{proposed}_p \) equals \( \{\} \), and for every learner \( l \), \( \text{learned}_l \) equals \( \bot \). Runs of generalized consensus satisfy the following properties:

**Non-triviality.** For every learner \( l \), \( \text{learned}_l \in \text{Str}(\text{prpCmd}) \) always holds.

**Stability.** It is always the case that \( \text{learned}_l = v \) implies \( v \sqsubseteq \text{learned}_l \) at all later times, for any learner \( l \) and c-struct \( v \).

**Consistency.** The set \( \{\text{learned}_l : l \in \text{Learners}\} \) 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 instantiations of the command structure abstraction correspond to different distributed tasks. For instance if we define a c-struct as a set of commands, and the operator \( u \cdot C \) as \( u \cup \{C\} \), this is identical to uniform reliable broadcast [17]. If a command structure is a singleton or the empty set, and:
\[
\text{if } u \text{ equals } \{\} \text{, then } C \text{, otherwise } u
\]
we obtain consensus [8]. We construct uniform atomic broadcast [17] when a c-struct is a c-seq and:
\[
\text{if } c \in u \text{, then } u \cdot C \triangleq u \cdot C \text{, otherwise } u \circ C
\]

To transform generalized consensus into uniform generic broadcast [25][3] we instantiate \( C\text{Struct} \) as a Mazurkiewicz trace [10]: Let \( \triangleright \) be a binary, symmetric and irreflexive relation over \( \text{Cmd} \), 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 \( \text{Cmd} \), and \( <_u \) is a partial order over \( \mathcal{E}_u \). We let \( \bot \) be the empty graph, and we define the operator \( \cdot \) by:
\[
\text{if } C \in u \text{, then } u \cdot C \triangleq u \cup \{C\}, <_u \cup \{(D,C) : D \in \mathcal{E}_u \land D \triangleright C\} \text{, otherwise }
\]

\[3\text{In [25] the authors define non-uniform generic broadcast. Modifying their definition to enforce uniformity is straightforward.}\]
Properties of generalized consensus. An algorithm $\mathcal{A}$ implementing generalized consensus in a message-passing distributed system is genuine when during nice runs of $\mathcal{A}$, every command is learned in two communication steps if $CStruct$ is compatible. Algorithm $\mathcal{A}$ is fast when (i) during nice runs of $\mathcal{A}$ if all the processes receives messages in the same order, then every command is learned in two steps and (ii) during nice runs of $\mathcal{A}$, every command is learned in at most three steps.\(^4\)

3 GPaxos

Along this section we describe GPaxos, an algorithm à la generalized Paxos [21] to solve generalized consensus in message-passing distributed systems. GPaxos is similar to generalized Paxos, but novel by the distinction it makes between read and write quorums. The subsequent section refines 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. When an action’s preconditions (keyword $\text{pre}$) are true, the action’s operations (keyword $\text{action}$) may execute atomically. A comment in square brackets indicates the role of the process, either a proposer ($\text{Proposers}$), an acceptor ($\text{Acceptors}$), a coordinator ($\text{Coordinators}$), or learner ($\text{Learners}$). 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.

3.1 Ballots and quorums

GPaxos executes an unbounded sequence of asynchronous rounds or ballots. We associate a ballot to a ballot number, or balnum in $\text{BalNum}$ 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 $m++$. The highest balnum smaller than $m$ is denoted $m--$.\(^5\) Hereafter, 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 $\text{Acceptors}$, 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: $w\text{quorum}(k)$, and a set of read quorums: $r\text{quorum}(k)$. An element in $w\text{quorum}(m)$ is a write quorum of $m$, or for short a $m$-wquorum. Similarly an element in $r\text{quorum}(m)$ 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 $\text{coord}(m)$ in $\text{Coordinators}$.

In addition, we assume hereafter that:

Q1. Given a ballot $m$, two $m$-wquorums $W$ and $W'$ and a $m$-rquorum $R$, $W \cap W' \neq \emptyset$ and $R \cap W \neq \emptyset$.

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

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,

---

\(^4\)We formally define fastness and genuineness in Appendix A.5.

\(^5\)By Zorn’s lemma such a balnum exists.
Algorithm 1 GPaxos : generalized consensus à la generalized Paxos - code at process $i$

1: \textbf{propose}(C) \ [\text{proposer}]
2: \textbf{pre}: \ C \in \text{Cmd}
3: \textbf{action}: \ \text{send (propose,C) to Acceptors} \cup \text{Coordinators}
4: 
5: \textbf{phase1A}(m) \ [\text{coordinator}]
6: \textbf{pre}: \ \text{maxStart}_i < m
7: \textbf{action}: \ \text{receive (1A,m) from coord(m)}
8: \textbf{phase1B}(m) \ [\text{acceptor}]
9: \textbf{pre}: \ \text{bal}_i < m
10: \textbf{action}: \ \text{send (1B,m,cmd,eval) to coord(m)}
11: 
12: \textbf{phase2Start}(m,R,k) \ [\text{coordinator}]
13: \textbf{pre}: \ \text{maxTried}_i = \text{none}
14: \textbf{action}: \ \text{send (2A,m,maxTried) to Acceptors}
15: 
16: \textbf{phase2AClassic}(m,C) \ [\text{coordinator}]
17: \textbf{pre}: \ \text{maxTried}_i \neq \text{none}
18: \textbf{action}: \ \text{send (2A,m,maxTried) to Acceptors}
19: 
20: \textbf{phase2BClassic}(m,u) \ [\text{acceptor}]
21: \textbf{pre}: \ \text{received (2A,m,u) from coord(m)}
22: \textbf{action}: \ \text{send (2B,m,cmd,eval) to Learners}
23: 
24: \textbf{phase2BFast}(C) \ [\text{acceptor}]
25: \textbf{pre}: \ \text{isFast}(%\text{cbal})
26: \textbf{action}: \ \text{send (2B,m,cmd,eval) to Learners}
27: 
28: \textbf{learn}(m,W,u) \ [\text{learner}]
29: \textbf{pre}: \ \text{W} \in \text{wQuorum(m)}
30: \textbf{action}: \ \text{learned}_i := \cup \{\text{learned}_i, u\}
and a balnum is a natural integer, then the following is a valid assignment: Ballot $m$ is assigned to the $m^{th}$ coordinator (modulo $\|\text{Coordinators}\|$). A ballot $m$ is fast iff $m$ is even. For every ballot $m$, the read quorums of $m$ are all the majorities of acceptors. If $m$ is classic, the write quorums of $m$ are all the majorities of acceptors; otherwise $m$ is fast, and the write quorums of $m$ are the set $Q$ of acceptors such that $|Q| > \frac{3m}{4}$.

### 3.2 GPaxos details

To propose a command $C$, an acceptor executes $\text{propose}(C)$. This action sends a $\text{propose}$ 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 $cval_a$.

At the beginning of ballot $m$, $\text{coord}(m)$ tries to convince acceptors to join $m$. If enough acceptors participate in $m$, $\text{coord}(m)$ suggests one or more c-structs. Coordinator $c$ stores the latest ballot it started: $bal_c$, and the latest c-struct it suggested at ballot $cval_c$. If no c-struct was suggested so far, $\text{maxTried}_c$ equals $\perp$ (an element that is not in $\text{CStruct}$).

In the initial state, every acceptor has joined ballot 0 and accepted $\perp$, and every learner has learned $\perp$; in other words: (i) For every acceptor $a$, both $bal_a$ and $cbal_a$ equal 0, and $cval_a$ equals $\perp$, (ii) For every learner $l$, $\text{learned}_l$ equals $\perp$, (iii) For every coordinator $c$, $\text{maxTried}_c$ equals 0, and (iv) if $c = \text{coord}(0)$, then $\text{maxTried}_c$ equals $\perp$, otherwise $\text{maxTried}_c$ equals $\perp$.

A c-struct $a$ is chosen at some ballot $m$, when there exists an $m$-quorum 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 $a$ 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 $\text{GPaxos}$ 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 $\text{GPaxos}$ maintains these invariants by detailing how it executes a classic ballot:

- **phase1A($m$)**: When it start a ballot $m$, the coordinator of $m$ sends a 1A message labelled $m$ 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 of ballot $m$ (line 16).

- **phase2Start($m,R,k$)**: The coordinator of ballot $m$ executes this action once there exists a ballot $k$ and a quorum $R$ such that (i) $\text{coord}(m)$ has received a 1B message labelled $m$ from every acceptor in $R$, (ii) $k$ is the highest ballot $\text{coord}(m)$ 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 Generalized Consensus and Paxos any two quorums intersect. Because our assumptions on quorums are weaker (see Q1 and Q2), we need condition (iii).
We explain the reason why by proving informally that when \textit{coord}(m) executes either line 26 or 29, \textit{maxTried}_{\text{coord}(m)} is safe at ballot \(m\):

**Proof (sketch).** Consider that a c-struct \(v\) is choosable at some ballot \(n\) smaller than \(m\), and assume that GPSafety-2 holds for every ballot prior to \(m\). First, we observe that there exists an acceptor \(a\) in \(R\) such that \(a\) has accepted some c-struct at ballot \(k\), and \textit{coord}(m) received a 1B message from \(a\) (line 22). Then we consider the two following cases:

**Case** \(n < k\): By invariant GPSafety-2, every c-struct accepted at ballot \(k\) by acceptors in \(R\) suffixes \(v\). If now \(W\) is empty (line 25), then \(u\) suffixes \(v\). Otherwise, for every quorum \(W\), \(\cap W\) suffixes \(v\) (line 28). Hence in both cases \textit{maxTried}_{m} suffixes \(v\) (lines 26 and 29).

**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 \(W \cap R \neq \emptyset\). It follows that at least one acceptor in \(R\) has accepted a c-struct at ballot \(n\). This contradicts the definition of \(k\).

\(\square\)

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

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

- **phase2BCClassic**(\(m,u\)) : 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 \(cval\_a\) 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 \textit{maxTried}_{\text{coord}(m)}, every c-struct accepted at ballot \(m\) prefixes \textit{maxTried}_{\text{coord}(m)} (invariant GPSafety-1). Moreover \textit{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 ballot \(m\) (lines 58 and 59). To learn c-struct \(u\), learner \(l\) assigns to \(\text{learned}_l\) the value of \(\sqcup \{\text{learned}_l,u\}\). This maintains the stability invariant of generalized consensus.

At first glance, \textit{GPaxos} has a latency of five steps in a classic ballot, as this is the length of the causal path between \textit{propose}, 1A, 1B, 2A and 2B messages. However, as long as \textit{coord}(m) does not crash and no coordinator starts a ballot higher than \(m\), \textit{coord}(m) may suggest new commands within \(m\ ad eternam\). As a consequence, if ballot 0 is classic, every command is learned in three communication steps during a nice run.

### 3.3 Fast ballots, collisions and recovery

To further reduce latency, acceptors execute action \textit{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 $cval_a \cdot C$ (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 $\text{coord}(m)$ at ballot $m$. If $a_1$ and $a_2$ receive two commands $C$ and $D$ in the same order, i.e., the network spontaneously orders $C$ before $D$, then both $a_1$ and $a_2$ extend $u$ in $v = (u \cdot C) \cdot D$. As a consequence, $v$ becomes 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 $w = (u \cdot D) \cdot C$, 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. More precisely, a process $i$ detects that a collision occurs at a ballot $m$ when the following predicate holds at $i$:

$$\text{collide}(m) \triangleq \exists W \in \text{wquorum}(m), \begin{cases} \forall a \in W, \text{received } (2B,m,u) \text{ from } a \\ \neg\{(u : \exists a \in W, \text{received } (2B,m,u) \text{ from } a \} \text{ compatible} \end{cases}$$

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.

### 4 Fast recovery from collisions in generalized consensus

In this section we refine GPaxos to recover, first in two steps, then in one step.

#### 4.1 About coordinated recovery

Most of the time, when a collision occurs in a nice run at ballot $m$, the coordinator of ballot $m++$ is also the coordinator of ballot $m$. After detecting the collision it starts ballot $m++$. Lamport observes that when this type of situation happens in Fast Paxos, if $\text{coord}(m)$ 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 $\text{Coordina tors} \subseteq \text{Acceptors}$. If a collision occurs at ballot $m$, $\text{coord}(m++)$ considers 2B messages received during ballot $m$ as 1B messages for ballot $m++$.

With the coordinated recovery technique, $\text{coord}(m++)$ skips phase one of ballot $m++$. This saves two communications steps when a collision occurs in Fast Paxos.
The coordinated recovery technique cannot be applied to GPaxos. Indeed, an acceptor of GPaxos continuously accepts newly proposed commands during fast ballots. This implies that when a collision occurs at ballot \( m \), \( \text{coord}(m++) \) 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 \( u = (\bot \cdot C) \cdot D \) at ballot \( m \), while \( a_2 \) accepts \( v = (\bot \cdot D) \cdot C \). Ballot \( m \) collides. If \( \text{coord}(m++) \) starts ballot \( m++ \), then \( \sqcap \{u, v\} \), which equals \( \bot \), should be safe. However concurrently to the beginning of ballot \( m++ \), acceptors \( a_1 \) and \( a_2 \) accept command \( E \). It follows that \( \bot \cdot E \) is now chosen at ballot \( m \). The coordinated recovery violates invariant GPSafety-2.

To maintain invariant GPSafety-2 \( \text{coord}(m++) \) must know what was chosen at ballot \( m \). Since acceptors accept new commands at will, \( \text{coord}(m++) \) cannot skip the first phase of ballot \( m++ \). In 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.

### 4.2 Recovery in two steps

To recover in two steps we introduce the concept of a centered ballot. We say that ballot \( m \) is *centered* when every \( m \)-wquorum contains \( \text{coord}(m) \):

\[
centered(m) = \forall W \in \text{wquorum}(m), \text{coord}(m) \in W
\]

If \( m \) is centered, observe that \( \{\text{coord}(m)\} \) 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 \( \text{Coordinators} \subseteq \text{Acceptors} \). For every fast ballot \( m, m \) is centered and \( \{\text{coord}(m)\} \) is a \( m \)-rquorum.

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

### 4.3 Recovery in one step: Fast Genuine Generalized Consensus

Algorithm 2 depicts the code of FGGC, a fast genuine generalized consensus algorithm. It adds action \( \text{recover} \) 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 \( \text{coord}(m) \) accepted at ballot \( m \). More precisely, our algorithm works as follows:

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

1: // Actions propose, phase1B, phase2Start, phase2Start, phase2BClassic, phase2BFast and learn are identical to the ones in Algorithm 1.

2: recover($u$) [acceptor]

3: pre: collide($cbal_i$)

4: isFast($cbal_i$)++

5: $\exists W \in wquorum(cbal_i)$, $a \in W$

6: $cbal_i = bal_i$

7: $\exists u \in CStruct$, received ($2B,u,cbal_i$) from coord($cbal_i$)

8: action: let $V = \{v \in CStruct : v \subseteq cval_i \wedge \{u,v\}$ compatible\}

9: let $safe \uplus \{u,\uplus V\}$

10: $bal_i := bal_i++$

11: $cbal_i := bal_i$

12: $cval_i := safe$

13: send ($2B,cval_i$) to Learners

14: 

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($u$), and assume that every ballot prior or equal to $m$ is safe. We 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, we must show that $v$ prefixes $safe$. We first observe that $u$ prefixes $cval_{coord}(m)$ (line 8). Then we consider the two 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 $V$ 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 $cval_a$. With a similar reasoning we conclude that $v$ prefixes $cval_{coord}(m)$. This implies that $\{u,v\}$ is compatible. Hence $V$ contains $v$, which implies that $safe$ suffixes $v$.

□

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 crashes and some synchrony assumptions. Appendix A.4 provides a proof of progress in the unreliable failure detectors model.

Latency. If FGGC continuously executes fast ballots using the same $m$-wquorum $W$, every proposed command is learned in at most three steps (see Appendix A.5). Nevertheless, if some crash occurs and $W$ is no longer available, FGGC must be able to execute classic ballots. To satisfy this requirement, we should 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 \(|\text{Acceptors}|\)). 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 of acceptors is always a read quorum of \( m \), and if \( m \) is fast \( \{\text{coord}(m)\} \) is also a read quorum of \( m \). For some ballot \( m = (i,j) \), \( m++ \) equals \((i,j+1)\). The ordering relation \(<\) over \( \text{BalNum} \), is defined given two ballots \( m = (i,j) \) and \( n = (k,l) \), by: \( m < n \triangleq i < k \lor (i = k \land j < l) \).

Definition above ensures that (i) \( FGGC \) execute 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. We prove in Appendix A.5.3), that when \( \text{BalNum} \) follows this definition, \( FGGC \) is both fast and genuine.\(^6\)

5 Related work

State machine replication [31] is a fundamental technique to construct wait-free linearizable shared data types [18] in message-passing distributed systems. In this section we briefly review algorithms to implement state machine replication in message-passing distributed systems where processes halt by crashing. Table 1 summarizes our survey.

The classical approach to implement state machine replication is to execute successive instances of consensus. The seminal FLP paper [14] shows that is not possible to solve consensus in an asynchronous system using a deterministic algorithm when a single process crashes. 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. Notably Dwork et al. [13] and Paxos [20] solve consensus deterministically for \( f < n/2 \) under weak synchrony assumptions. Both algorithms preserve the agreement property of consensus, even if the system behaves asynchronously. This behaviour is called indulgence [15] in the context of failure detectors.

Metrics for evaluating the performance of a consensus algorithm include: (i) best-case time complexity, or latency degree, and (ii) the number of failures it tolerates, or resiliency degree. In particular, latency should be low during nice runs. Both Paxos and Chandra et al.’s algorithm [7] solve consensus in three communication steps. The early-deciding algorithm of Schiper [29] improves this by solving consensus in two steps, which is optimal in the general case [23]. These algorithms have optimal resilience, i.e., 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. [4] and Pedone et al. [26] solve consensus in one step, in nice runs where all processes propose the same command. Deciding in one step during stable runs, i.e., runs during which all faults are initial and the system behaves synchronously, requires \( f < n/3 \) [4]. Optimal resilience requires the consensus algorithm to decide in one step during stable runs only if at most \( f < n/4 \) processes crash, as in Lamport’s Fast Paxos [22]. When every process is at a same time a proposer, an acceptor and a learner, and every

---

\(^6\)With this definition, once a classic ballot has been joined, no fast ballots will ever be executed. This is not acceptable in practice. One can modify Example 3 to remove this undesirable phenomenon, e.g. by considering that ballot \((i,j)\) is fast when \( i \) is even. The definition given at Example 3 is nevertheless of interest because it ensures that \( FGGC \) is fast and genuine and ease proving liveness.
process proposes the same command, $FGGC$ manages to decide in one step with only a majority of processes. This does not contradict Lamport’s result, as $FGGC$ decides only if a privileged set $[16]$ of processes is alive (precisely the unique centered quorum of ballot 0).

When a collision occurs during a fast ballot, the latency of Fast Paxos may exceed that of Paxos. A recent algorithm by Charron-Bost et al. $[9]$ tentatively executes a classic ballot during a fast ballot. Their algorithm solves consensus in one step during nice runs, if every process proposes the same command, and two otherwise; it has optimal resilience. When c-structs follows definition 1, ballot 0 is fast, the single write quorum of ballot $O$ contains all acceptors, and every higher ballot is classic, $FGGC$ is identical to this solution.

Both $FGGC$ and $[9]$ trade decision in one-step during all stable runs for zero-degradation, i.e., fast recovery when a collision occurs. Deciding in one-step during all stable runs is a property of interest for repeated consensus as if a fault occurs during an instance $i$, instances greater than $i$ still decide quickly. However this property is of few interest for generalized consensus because the primitive is executed only once to implement state machine replication.

Both Pedone et al. $[25]$ and Lamport $[21]$ observe that since replicas might execute commuting commands in different orders, it is unnecessary to order all commands. The algorithm of Pedone et al. tolerates $f < n/3$ crashes and does not leverage the spontaneous ordering of the network. More recently, Zieliński’s Generic Optimistic Broadcast algorithm $[32]$ requires two steps when concurrent commands either commute, or were spontaneously ordered by the network, and three steps otherwise, and tolerates $f < n/3$ faults. However, the author himself concedes that this algorithm is not practically implementable.

Multicoordinated Paxos uses multiple processes to coordinate a ballot $[5]$. This increases dependability at the cost of a higher probability of collision.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Time complexity</th>
<th>resilience optimality</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>everything commute</td>
<td>spontaneous order</td>
</tr>
<tr>
<td>Dwork et al. [13]</td>
<td>$3n+1$</td>
<td>$3n+1$</td>
</tr>
<tr>
<td>Chandra et al. [7]</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Paxos. [20]</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Schiper [29]</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Brasileiro et al. [4]</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>Lamport [22]</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>Charron-Bost et al. [9]</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>Pedone et al. [26]</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Lamport [21]</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Camargos et al. [5]</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Zieliński [32]</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>$FGGC$</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 1: Complexity of the surveyed algorithms when implementing state machine replication.
6 Discussion and perspectives

6.1 Generalized consensus and uniformity

Our liveness clause of generalized consensus (see Section 2.2) requires that every correct learner learns every command learned by some learner. This enforces uniform agreement on learned commands. In our opinion, because we focus on state machine replication, uniformity is a desired property for clients. In [21], the liveness clause of generalized consensus nevertheless does not enforce uniformity. This is not an issue as non-trivial agreement tasks are uniform in partial-synchrony systems. However the two specifications differ in the general case.

6.2 Informal proof of Zieliński’s conjecture

Aguilera et al. [2] define a generic broadcast algorithm to be thrifty regarding an oracle, if it queries the oracle only if conflicting messages are broadcast. Recently Zieliński [32] conjectured that there is no thrifty indulgent generic broadcast, that tolerates $f < n/3$ faults and delivers messages during stable runs in two steps if all messages commute, or are spontaneously ordered by the network, and three steps otherwise. Assume that such an algorithm $A$ exists, and construct on top of $A$ a consensus algorithm $A'$ by submitting to $A$ commands that are proposed, and learning the first command delivered. We observe that $A'$ satisfies the following properties during stable runs: (i) if all processes propose the same command, then, since a command commute with itself, all correct processes decide in at most two steps, and (ii) every command is learned in at most three steps. Property (i) is called one-step [11], and property (ii) is named zero-degradation [12], both properties being expressed in the consensus model of Lamport, i.e., with the distinction between proposers and learners. Dobre and Suri recently [11] prove that every one-step consensus algorithm which tolerates $f < n/3$ faults has a stable run during which some process learns after three communication steps. Algorithm $A'$ contradicts this result. Thus no such algorithm $A$ exists.

6.3 On assumption CS3.1

In Lamport [21] assumption CS3.1 is written as follows: $\forall C \subseteq Cmd, \forall U \in Str(C), \cap U \in Str(C)$. This definition is too weak to preserve non-triviality. We illustrate this claim below:

Example 4: Consider that $Cmd = \{A,B,C\}$. Note $u = \bot \bullet A$, $v = \bot \bullet B$ and $w = (\bot \bullet B) \bullet C$. and assume that $u$ equals $w$, i.e., applying $u$ or $w$ leads to the same state. Consider a system with a single proposed command $A$ and a single acceptor $a$ such that at ballot 0 acceptor $a$ accepts c-struct $u$. Observe that since $v \bullet C = w$ and $w = u$, we have $v \subseteq u$. Thus $v$ is chosen at ballot 0, and consequently it might be learned. However $Str\{\{A\}\}$ does not contains $v$.

Our assumption CS3.1 is stronger than the original assumption appearing in [21]. This ensures that if acceptors accept only c-structs constructable with proposed commands, a chosen c-struct is constructable with proposed commands.

6.4 Beyond commutativity of commands

Assumption CS3.3 is contingent to the definition of generalized consensus. Indeed, this assumption is only required to prove the correctness of Generalized Paxos when it executes fast ballots having multiple write quorums. As a consequence, $FGGC$ solves generalized consensus without assumption
CS3.3. We briefly study below the command structure abstraction when it does not satisfy CS3.3. A detailed study is left for future work.

6.4.1 $k$-set agreement

We obtain (uniform) $k$-set agreement [28] for some natural $k$, if we let a c-struct be a subset of $Cmd$ and define operator $\bullet$ as follows:

$$u \bullet C \triangleq \begin{cases} u & \text{if } |u| = k \text{ or } C \in u, \\ u \cup \{C\} & \text{otherwise} \end{cases}$$

If $k \geq 2$, $CStruct$ does not satisfies CS3.3. For instance if $k = 2$, the following set of c-structs $U = \{\{C\}, \{D\}, \{E\}\}$ is 2-compatible but not compatible.

6.4.2 Database replication

Consider that each command is a transaction optimistically executed, like in the Database State Machine approach [24], and that generalized consensus reconciles database replicas. To minimize abort rate, it is of interest that generalized consensus recognizes serializable histories [3]. For instance, consider two transactions $T_1 = w_1[x_1]$ and $T_2 = r_2[x_0].w_2[y_2]$ tentatively executed and concurrently submitted to generalized consensus. If $T_1$ commits, because $w_1[x_1].c_1.r_2[x_0].w_2[y_2].c_2$ is serializable, transaction $T_2$ should commit too. We now propose a definition of $CStruct$ to match this goal.

We let $Items$ be the set of replicated data items, and $Cmd$ be the set of transaction optimistically executed that access $Items$. Given a transaction $T$, we note $rs(T)$ (respectively $ws(T)$) the read set (resp. write set) of $T$. We note as usual $T_i \prec T_j$ when $T_i$ was executed tentatively after $T_j$, and $T_i \parallel T_j$ when neither $T_i \prec T_j$ nor $T_j \prec T_i$ holds. In addition, we consider the following binary relations over $Cmd$:

- Relation $T_i \rightarrow T_j$ (read “$T_i$ not after $T_j$”) expresses that $T_i$ must be ordered before $T_j$.

$$T_i \rightarrow T_j \triangleq \left( \forall T_j \mid T_i \parallel T_j \land (rs(T_i) \cap ws(T_j)) \neq \{\} \right)$$

- Relation $T_i \nprec T_j$ (read “$T_i$ non-commute with $T_j$”) captures that transactions $T_i$ and $T_j$ does not commute.

$$T_i \nprec T_j \triangleq T_i \parallel T_j \land ws(T_i) \cap ws(T_j) \neq \{\}$$

- Relation $T_i \triangleright T_j$ (read “$T_i$ must-have $T_j$”) defines that transaction $T_i$ needs transaction $T_j$ to execute.

$$T_i \triangleright T_j \triangleq T_j \parallel T_i \land (rs(T_i) \cap ws(T_j)) \neq \{\}$$

For instance, if transaction $T_i$ is submitted to generalized consensus, and $T_i$ reads optimistically a data item written by $T_j$, the conjunction $T_i \rightarrow T_i \land T_i \triangleright T_j$ expresses the causal dependency between $T_i$ and $T_j$, and that if $T_j$ is aborted, then $T_i$ has to be aborted too.
We now define $CStruct$ accordingly to the relations we gave above. We let a c-struct $u$ be a serialization graph $(\mathcal{E}_u, <_u)$, and we construct operator $\bullet$ as follows:

$$u \bullet T_i = \begin{cases} 
    u & \text{if } T_i \in \mathcal{E}_u \\
    u & \text{if } \exists T_j \in \mathcal{E}_u, T_i \rightarrow T_j \\
    u & \text{if } \exists T_j \in \text{Cmd}, T_i \nabla T_j \land T_j \not\in \mathcal{E}_u \\
    (\mathcal{E}_u \cup \{T_i\}, <_u \cup \{(T_j,T_i) \in (\rightarrow \cup \nabla) : T_j \in \mathcal{E}_u\}) & \text{otherwise}
\end{cases} \quad (4)$$

Defining $CStruct$ with Equation 4 implies that generalized consensus recognizes serializable histories: Relation $\rightarrow$ captures all the read-from dependencies between committed transaction. By Equation 4 these dependencies are acyclic. Relation $\nabla$ tracks cascading aborts since if transaction $T$ reads-from transaction $T'$, by Equation 4 committing $T$ implies committing $T'$. Finally, relation $\nabla$ requires that there exists a version-order compatible with the read-from dependencies.

Notice that the definition of $CStruct$ given at Equation 4 does not satisfy CS3.3 as soon as there are three data items. Nevertheless, because assumption CS3.3 is not needed when there is a single fast quorum per fast ballot, $FGGC$ solves the resulting generalized consensus problem.

Both generic broadcast for transaction commit [27] and the generalized consensus resulting from Equation 4 ensure serializability. However, because we order concurrent read/write conflicts (definition of $\rightarrow$), our problem allows faster solutions. For instance, if a transaction $T_1$ reads $x$ concurrently to a transaction $T_2$ that writes $x$, generic broadcast needs 3 steps to decide $T_1$ and $T_2$ (without spontaneous ordering) whereas our $FGGC$ algorithm needs only two. Besides, our definition is more generic since it entails the problem solved by replicated database systems that handle cascading aborts.

7 Conclusion

This paper presents $FGGC$, a fast genuine generalized consensus algorithm. $FGGC$ tolerates $f < n/2$ replicas crashes, does not order commuting commands, and leverages the spontaneous ordering of the network. During failure-free runs, $FGGC$ decides in two steps when commands commute, and three otherwise. $FGGC$ achieves a higher resiliency and/or a lower latency than previous work. $FGGC$ has been implemented and is currently undergoing experimental evaluation.

Acknowledgment The authors thank Julien Sopena for his comments on an earlier version of this paper.

References


A Proof of correctness

Along this section we prove the correctness of FGGC. As we will see, FGGC might only execute classic ballots to solve generalized consensus. This implies that action recover is unnecessary to progress, and thus proving FGGC gives us also a proof of GPaxos.

Outline of the proof: We first introduce the ballot array abstraction to model runs of FGGC. We then explain how to extract a safe c-struct from a ballot array to model action phase2Start. Further, we construct an abstract fast genuine generalized consensus algorithm, and show that every run of this algorithm produces a “correct” ballot array. From the properties of a correct ballot array, we deduce that the abstract algorithm preserves the non-triviality, stability and consistency invariants of generalized consensus. We then prove that FGGC implements the abstract algorithm, and show that under some synchrony assumptions, FGGC is live. We conclude this proof by proving that FGGC is both fast and genuine.7

A.1 Preliminaries

Considering the notation introduced in Section 2.2, we formally define generalized consensus, denoted GC hereafter, as follows:

Non-triviality. For every learner $l$, learned$_l \in Str(prpCmd)$ always holds.

$$\forall l \in Learners, \Box(learned_l \in Str(prpCmd))$$

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 $v$.

$$\forall v \in CStruct, \forall l \in Learners, \Box((learned_l = v) \Rightarrow \Box(v \sqsubseteq learned_l))$$

Consistency. The set \{learned$_l : l \in Learners$\} is always compatible.

$$\Box\{learned_l : l \in Learners\} \text{ compatible}$$

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 $l$ eventually learns $C$.

$$\forall C \in Cmd, \forall l \in Learners, \begin{cases} l \in Correct, \\ \exists p \in Proposers \cap Correct, \Diamond (C \in proposed_p) \\ \forall \exists l' \in Learners, \Diamond (C \in learned_{l'}) \end{cases} \Rightarrow \Diamond (C \in learned_l)$$

The state of generalized consensus is externally visible in the sense of [1]. In the following we call weakly-terminating generalized consensus, denoted GC*, the problem specified by the conjunction of non-triviality, consistency and stability. The abstract algorithm we introduce later in the proof solves this weaker form of generalized consensus.

7Our proof schema is similar to Lamport [21]. We borrow some ideas from Camargos PhD [5].
A.1.1 Ballot array

This section introduces the ballot array abstraction [21] with two modifications: First, since we distinguish read from write quorums, we precise what type of quorums we are considering in the ballot array abstraction. Second, in order to prove the correctness of action recover, we consider a definition of “choosable at” that is more restrictive than the original one.

Let none be an element that is not in CStruct, we extend \(\subseteq\) to CStruct \(\cup\) \{none\} by defining \(v \subseteq w\) to be true if both \(v\) and \(w\) equals none, and to be false if either \(v\) or \(w\) equals none (but not both).

**Definition 1** (ballot array). A ballot array is a mapping that assigns to each acceptor \(a\), a ballot \(\hat{\beta}_a\), and to each acceptor \(a\) and ballot \(m\), a value \(\beta_a[m]\) in CStruct \(\cup\) \{none\}, such that:

\[
\forall a \in \text{Acceptors}, \left\{ \begin{array}{l}
\beta_a[O] = \bot \\
\forall m > \hat{\beta}_a, \beta_a[m] = \text{none}
\end{array} \right.
\]

A ballot array models a run of an algorithm solving generalized consensus: \(\hat{\beta}_a\) is the highest ballot to which acceptor \(a\) has participated, and each c-struct entry \(\beta_a[m]\) models the fact that \(a\) has accepted \(\beta_a[m]\) at ballot \(m\).

A chosen c-struct is a c-struct that has been accepted by a write quorum of acceptors, and that can be learned:

**Definition 2** (chosen at). A c-struct \(u\) is chosen at a ballot \(m\) in ballot array \(\beta\), iff there exists a \(m\)-wquorum \(W\) such that \(u \subseteq \beta_a[m]\) for every acceptor \(a\) in \(W\).

We note hereafter \(\text{chosen}(m,\beta)\) the c-structs chosen at ballot \(m\) in \(\beta\), and \(\text{chosen}(\beta)\) all c-structs chosen (at least at one ballot) in \(\beta\).

**Definition 3** (choosable at). A c-struct \(u\) is choosable at ballot \(m\) in ballot array \(\beta\), iff there exists a \(m\)-wquorum \(W\) such that for every acceptor \(a\) in \(W\), (i) if \(\hat{\beta}_a > m\) then \(u \subseteq \beta_a[m]\), and (ii) if \(\beta_a[m] \neq \text{none}\) then \(u\) and \(\beta_a[m]\) are compatible.

A c-struct is choosable at some ballot \(m\) if it is, or it might be, chosen at \(m\). In Paxonian algorithms, an acceptor stops its participation to a ballot \(m\) as soon as \(\hat{\beta}_a > m\). Part (i) of the definition of “choosable at” models this behavior: \(u\) is choosable at ballot \(m\) if there exists a \(m\)-wquorum \(W\) such that every acceptor in \(W\) have accepted (at least) \(u\) at ballot \(m\), or might later accept \(u\) at ballot \(m\). This is the original definition of Lamport in “Generalized Consensus and Paxos”. Part (i) is however not sufficient to prove the correctness of action recover. It is necessary to observe in addition that if for every \(m\)-wquorum, there exists some acceptor \(a\) which has accepted a c-struct which is not compatible with \(u\), then \(u\) is not, and it will never be, chosen at \(m\). This is part (ii) of our definition. In the following, \(\text{choosable}(m,\beta)\) denotes the c-structs choosable at ballot \(m\) in \(\beta\).

**Definition 4** (safe at). A c-struct \(u\) is safe at ballot \(m\) in ballot array \(\beta\), iff for every c-struct \(v\) choosable at some ballot \(k < m\), \(v \subseteq u\).

**Definition 5** (safe ballot array). A ballot array \(\beta\) is safe, iff for every ballot \(m\), for every acceptor \(a\), if \(\beta_a[m]\) is a c-struct, then \(\beta_a[m]\) is safe at ballot \(m\).
Definition 6 (conservative ballot array). A ballot array $\beta$ is conservative, iff for every classic ballot $m$, the set of c-structs accepted at ballot $m$ is compatible.

The proposition that follows is essential to understand the correctness of GPaxos and FGGC. It models the fact that invariant GPSafety-2 implies the consistency invariant of generalized consensus.

**Proposition 1.** If $\beta$ is safe, the set of c-structs chosen in $\beta$ is compatible.

**Proof**

Let: $\beta$ a safe ballot array.

**Fact 1:** $\text{chosen}(\beta)$ is 2-compatible

Let: $u_1$ and $u_2$ two c-structs chosen respectively at ballot $m_1$ and $m_2$ in $\beta$.

Let: for $i$ in $[1,2]$, note $W_i$ a $m_i$-wquorum s.t. for every acceptor $a$ in $W_i$, $u_i$ prefixes $\beta_a[m_i]$.

Proof: For $i$ in $[1,2]$, quorum $W_i$ exists since $u_i$ is chosen at ballot $m_i$ in $\beta$.

**Case:** $m_1 < m_2$

Let: $a \in W_2$

Proof: A quorum is a non-empty set of acceptors.

**Fact 2:** $u_2 \subseteq \beta_a[m_2]$

Proof: $a$ belongs to $W_2$.

**Fact 3:** $u_1 \subseteq \beta_a[m_2]$

**Fact 4:** $u_1 \in \text{chosen}(m_1,\beta)$

**Fact 5:** $m_1 < m_2$

**Fact 6:** $\beta_a[m_2] \in \text{CStruct}$

**Fact 7:** $u_2 \subseteq \beta_a[m_2]$

**Fact 8:** $u_2 \in \text{CStruct}$

QED.

Proof: Definition of none.

**Fact 9:** $\beta$ safe

QED.

QED.

**Case:** $m_1 = m_2$

Let: $a \in W_1 \cap W_2$

Proof: By assumption Q1.

**Fact 10:** $u_1 \subseteq \beta_a[m]$

**Fact 11:** $u_2 \subseteq \beta_a[m]$

QED.

**Case:** $m_1 > m_2$

QED.

Proof: The case $m_1 > m_2$ is symmetric to $m_1 < m_2$ and thus omitted.

QED.

Proof: Assumption CS3.3 concludes.
Proposition 2 below models that GPaxos and FGGC preserves the non-triviality invariant of generalized consensus.

**Proposition 2.** Let $C$ be a set of commands, and assume that for every acceptor $a$ and every ballot $m$, if $\beta_a[m]$ is a c-struct, then $\beta_a[m]$ belongs to $\text{Str}(C)$. Then, every c-struct chosen in $\beta$ belongs to $\text{Str}(C)$.

**Proof**

Let: $u$ a c-struct chosen in $\beta$.

Let: $m$ a ballot and $W$ a $m$-wquorums s.t. for every acceptor $a$ in $W$, $u \subseteq \beta_a[m]

Proof: Definition of $u$ chosen in $\beta$.

Let: $a$ an acceptor in $W$

Fact 1: $u$ is a c-struct

Fact 2: $u \subseteq \beta_a[m]$

Fact 3: $\beta_a[m] \neq \text{none}$

Proof: Facts 1 and 2, and definition of $\text{none}$.

Fact 4: $\beta_a[m] \in \text{Str}(C)$

Proof: Fact 3, definition of a ballot array and our hypothesis.

QED.

Proof: Facts 2 and 4, and assumption CS3.1.

---

**A.1.2 Extracting a safe c-struct**

We now explain how to extract a safe c-struct from a ballot array to models action $\text{phase2Start}$ (line 18 in Algorithm 1). Procedure $\text{provedSafe}(m,R,k,\beta)$, see Algorithm 3, takes as input a ballot $m$, a quorum $R$, a ballot $k$, and a safe and conservative ballot array $\beta$. Proposition 3 below shows that if $m, k, R$ and $\beta$ satisfy the precondition of $\text{provedSafe}$, then $\text{provedSafe}(m,R,k,\beta)$ outputs a c-struct safe at ballot $m$ in $\beta$.

**Proposition 3.** Given two ballots $m$ and $k$, a quorum $R$, and a ballot array $\beta$, if $m, k, R$ and $\beta$ satisfy the preconditions of $\text{provedSafe}$, then $\text{provedSafe}(m,R,k,\beta)$ returns a c-struct safe at ballot $m$ in $\beta$.

**Proof**

Let: $m, k, R$ and $\beta$ satisfying the preconditions of $\text{provedSafe}$.

Prove: $\text{provedSafe}(m,R,k,\beta)$ returns an element in $\text{CStruct}$. 

---
Algorithm 3 \textit{provedSafe}(m,R,k,\beta)

1: \textbf{pre:} \quad \forall a \in R, \beta_a \geq m
2: \quad m > 0
3: \quad k = \max \{ n \in \text{BalNum} : n < m \land \exists a \in R, \beta_a[n] \neq \text{none} \}
4: \quad \forall n \in \text{BalNum}, (k \leq n < m) \Rightarrow R \in \text{rquorum}(n)
5: \quad \beta \text{ safe}
6: \quad \beta \text{ conservative}
7: \textbf{action:} \quad \text{Let } W = \{ W \in \text{wquorum}(k) : \forall a \in W \cap R, \beta_a[k] \neq \text{none} \}
8: \quad \text{if } W = \{ \}
9: \quad \text{return some element in } \{ \beta_a[k] \neq \text{none} : a \in R \}
10: \quad \text{else}
11: \quad \text{Let } \gamma(W) = \sqcap \{ \beta_a[k] \neq \text{none} : a \in R \cap W \}
12: \quad \text{return } \Gamma = \sqcup \{ \gamma(W) : W \in \mathcal{W} \}

Let: \( W \) as defined at line 7 in Algorithm 3.

Case: \( W = \{ \} \)

Fact 1: \( \forall a \in \text{Acceptors}, \beta_a[k] \neq \text{none} \Rightarrow \beta_a[k] \in \text{CStruct} \)

Proof: \( \beta \) is a ballot array.

Fact 2: \( \exists a \in R, \beta_a[k] \neq \text{none} \)

Proof: \( k \) satisfies the precondition at line 3 in Algorithm 3, \( m > 0 \) (precondition at line 2 in Algorithm 3), and for every acceptor \( a, \beta_a[0] \) equals \( \bot \).

QED.

Case: \( W \neq \{ \} \)

Fact 3: \( \{ \gamma(W) : W \in \mathcal{W} \} \) compatible

Case: \( k \) is classic

Fact 4: \( \{ \beta_a[k] \neq \text{none} : a \in R \} \) compatible

Proof: \( \beta \) is conservative and \( k \) is classic.

QED.

Proof: Fact 4 and the definition of \( \gamma(W) \) for some \( W \) in \( \mathcal{W} \) as defined at line 11 in Algorithm 3.

Case: \( k \) is fast

Fact 5: \( \{ \gamma(W) : W \in \mathcal{W} \} \) 2-compatible

Let: \( W_1 \) and \( W_2 \) in \( \mathcal{W} \).

Let: \( a \) in \( W_1 \cap W_2 \cap R \).

Proof: Assumption Q2.

Fact 6: \( \gamma(W_1) \sqsubseteq \beta_a[k] \)

Proof: Definition of \( \gamma(W_1) \).

Fact 7: \( \gamma(W_2) \sqsubseteq \beta_a[k] \)

Proof: Definition of \( \gamma(W_2) \).

QED.

QED.

Proof: Assumption CS3.3 concludes.

Fact 8: \( \text{CStruct} = \text{Str}(\text{cmd}(\text{CStruct})) \)

Proof: Assumption CS1.
Fact 9: $\Gamma \in CStruct$
Fact 10: $\forall W \in W, \gamma(W) \in CStruct$
Proof: Fact 8 and Assumption CS3.1
QED.

Proof: Definition of $\Gamma$ at line 12 in Algorithm 3, assumption CS3.2 and fact 8.
QED.

**Prove:** provedSafe$(m,R,k,\beta)$ safe at ballot $m$ in $\beta$

Let: $u$ be a c-struct choosable at some ballot $n < m$ in $\beta$.
Let: $W_u$ be a $n$-wquorum s.t. for every acceptor $a$ in $W_u$, if $\bar{\beta}_a > n$, then $u$ prefixes $\beta_a[n]$.

Proof: By definition of $u$ choosable at ballot $n$ in $\beta$.

**Prove:** $u \subseteq provedSafe(m,R,k,\beta)$

**Case:** $k < n < m$

Fact 11: $\forall W \in wquorum(n), W \cap R \neq \{\}$

Fact 12: $\forall n' \in \mathbb{N}, (k \leq n' < m) \Rightarrow R \in rquorum(n')$
Proof: $R$ satisfies the precondition at line 4 in Algorithm 3.

Fact 13: $\forall n' \in \mathbb{N}, (k \leq n' < m) \Rightarrow (\forall W \in wquorum(n'), W \cap R \neq \{\})$
Proof: From Q1 and fact 12.

QED.

Let: $a$ in $R \cap W_u$
Proof: Fact 11 and $W_u$ is a $n$-wquorum.

Fact 14: $\bar{\beta}_a > n$
Proof: Line 1 in Algorithm 3.

Fact 15: $\beta_a[n] \neq none$
Proof: By construction.

Contradiction
Proof: Fact 15 contradicts the definition of $k$ at line 3 in Algorithm 3.

**Case:** $n < k$

Fact 16: $\forall a \in R, \beta_a[k] \neq none \Rightarrow u \subseteq \beta_a[k]$
Proof: $\beta$ safe.

**Case:** $W = \{\}$
QED.
Proof: Fact 16 and line 9 in Algorithm 3.

**Case:** $W \neq \{\}$

Fact 17: $\forall W \in W, u \subseteq \gamma(W)$
Proof: Fact 16 and the definition of $\gamma(W)$ for some $W$ in $W$ at line 11 in Algorithm 3.

Fact 18: $u \subseteq \Gamma$
Proof: Fact 17 and the definition of $\Gamma$ at line 12 in Algorithm 3.
QED.

**Case:** $n = k$

23
Fact 19: \( \forall a \in W_u \cap R, u \subseteq \beta_a[k] \)

Fact 20: \( \forall a \in W_u \cap R, \beta_a > n \)

Fact 21: \( \forall a \in W_u \cap R, \beta_a \geq m \)

Proof: Line 1 in Algorithm 3.

Fact 22: \( k < m \)

Proof: Line 3 in Algorithm 3.

Fact 23: \( n = k \)

QED.

Fact 24: \( n \in \text{choosable}(n, \beta) \)

Fact 25: \( \beta \text{ safe} \)

QED.

Fact 26: \( W_u \in W \)

Proof: Fact 18.

Fact 27: \( u \subseteq \cap \gamma(W_u) \)

Proof: Fact 19 and line 11 in Algorithm 3.

Fact 28: \( u \subseteq \Gamma \)

Proof: Fact 27 and line 12 in Algorithm 3.

QED.

The proposition below models the fact that the c-struct initially suggested by the coordinator is always constructable with proposed commands.

**Proposition 4.** Assume some set of commands \( C \). Consider two ballots \( m \) and \( k \), a quorum \( R \), and a ballot array \( \beta \) such that for every acceptor \( a \) and every ballot \( m \), if \( \beta_a[m] \) is a c-struct, then \( \beta_a[m] \) belongs to \( Str(C) \). If \( m \), \( k \), \( R \) and \( \beta \) satisfy the preconditions of \( \text{provedSafe} \), then \( \text{provedSafe}(m, R, k, \beta) \) returns a c-struct that belongs to \( Str(C) \).

**Proof**

The case where \( W \) is empty is obvious. Now if \( W \) contains some \( m \)-wquorum, by assumption CS3.1 for any \( W \) in \( W \), \( \gamma(W) \) belongs to \( Str(C) \). Assumption CS3.2 concludes.

□

### A.2 An abstract fast genuine generalized consensus algorithm

The heart of our proof is a non-distributed algorithm: \( \text{abstractFGGC} \).\(^8\) Along this section we prove that \( \text{abstractFGGC} \) implements weakly-terminating generalized consensus, i.e., the problem defined by generalized consensus without the liveness property. To prove that \( \text{abstractFGGC} \) implements \( GC^* \) we proceed as follows: Section A.2.1 defines variables and actions of \( \text{abstractFGGC} \), as well as a set of invariants. Sections A.2.3 and A.2.2 prove that \( \text{abstractFGGC} \) maintains these invariants. Based on these invariants, we affirm in Section A.2.4 the existence of a refinement mapping [1] from \( \text{abstractFGGC} \) to \( GC^* \).

\(^8\)This algorithm acts as a bridge between generalized consensus and \( FGGC \). Nevertheless, we believe that it is not difficult to modify \( \text{abstractFGGC} \) in order to use it in a shared-memory distributed system.
A.2.1 Variables and invariants

Algorithm 4 depicts the code of abstractFGGC which uses the following variables: Proposed is an array mapping each proposer p to a value Proposed[p] in 2^{\text{Cmd}}, Learned is an array mapping each learner l to a value Learned[l] in CStruct, MaxTried and MinTried are two arrays mapping every ballot to a value in CStruct\cup \{none\}, and \beta is a ballot array. Variables Learned and Proposed are externally visible. To ease notations, we denote hereafter PrpCmd the value: \bigcup_{p \in \text{Proposers}} Proposed[p].

Following what we have said in Section 4, we assume hereafter that (FGGC1) Coordinators \subseteq Acceptors \subseteq Learners, and that (FGGC2) every fast ballot is centered and associated to a unique write quorum, i.e. \forall m \in \text{BalNum}, isFast(m) \Rightarrow (\text{centered}(m) \land \text{wquorum}(m)) = 1.

An initial state of abstractFGGC is a state where (i) for every proposer p, Proposed[p] is empty, (ii) for every learner l, Learned[l] equals \bot, (iii) for every acceptor a, \hat{\beta}_a equals 0 and \beta_a[0] equals \bot, and (iv) MaxTried[0] and MinTried[0] both equals \bot, and for every ballot m > 0, MaxTried[m] and MinTried[m] both equals none.

The type correctness of abstractFGGC is trivial for Proposed, Learned, MaxTried and MinTried. In addition, we observe that for some acceptor a and ballot m, if abstractFGGC modifies \beta_a[m], then m equals \hat{\beta}_a. Since \hat{\beta}_a only increases during every run of abstractFGGC, variable \beta is always a ballot array.

Our refinement’s mapping from abstractFGGC to GC^* relies on the following invariants:

- **NT1.** \forall m \in \text{BalNum}, \Box (\forall a \in \text{Acceptors}, \beta_a[m] \neq \text{none} \Rightarrow \beta_a[m] \in \text{Str}(\text{PrpCmd}) \land (\text{MaxTried}[m] \neq \text{none} \Rightarrow \text{MaxTried}[m] \in \text{Str}(\text{PrpCmd})))

- **NT2.** \forall l \in \text{Learners}, \Box (\forall U \subseteq \text{chosen}(\beta), \text{learned}_l = \bigcup U)

- **STB1.** \forall v \in \text{CStruct}, \forall l \in \text{Learners}, \Box ((\text{Learned}[l] = v) \Rightarrow \Box (v \subseteq \text{Learned}[l]))

- **CONS1.** \Box (\beta \text{ conservative})

- **CONS2.** \Box (\beta \text{ safe})

Our first step is consequently to prove that abstractFGGC effectively maintains those invariants. To show that abstractFGGC preserves an invariant INV, we assume a state of abstractFGGC where all the invariants above plus INV holds, and we prove that every step taken by abstractFGGC in that state preserves INV. As usual, when we consider a state s of abstractFGGC and we investigate the state s’ of abstractFGGC after the step (s,s’), for some variable var of abstractFGGC, we note var’ the value of var in state s’. In particular, for an acceptor a, \beta_a[m]’ is the value of \beta_a[m] in state s’.

A.2.2 CONS1 and CONS2

Our proof relies on the auxiliary invariants below:

- **CONS\text{a}.** \forall k \in \text{BalNum}, \forall a \in \text{Acceptors}, \Box (\hat{\beta}_a = k \Rightarrow \Box (\hat{\beta}_a \geq k))

- **CONS\text{b}.** \forall u \in \text{CStruct}, \forall m \in \text{BalNum}, \forall a \in \text{Acceptors}, \Box (\beta_a[m] = u \Rightarrow (\Box (u \subseteq \beta_a[m] \land \exists W \in \text{wquorum}(m), a \in W))

- **CONS\text{c}.** \forall u \in \text{CStruct} \cup \{\text{none}\}, \forall m \in \text{BalNum}, \forall a \in \text{Acceptors}, \Box (\beta_a[m] = u \Rightarrow \Box (\beta_a[m] = u))

- **CONS\text{d}.** \forall U \subseteq \text{CStruct}, \forall m \in \text{BalNum}, \Box (\forall (\text{choosable}(m,\beta) = U) \Rightarrow U \subseteq \text{choosable}(m,\beta))
Algorithm 4 Abstract Fast Genuine Generalized Consensus

1: Propose(\(C,p\))
2: \(\text{pre: } p \in \text{Proposers}\)
3: \(C \in \text{Cmd}\)
4: \(\text{action: } \text{Proposed}[p] := \text{Proposed}[p] \cup \{C\}\)

5: Learn\((m,u,l)\)
6: \(\text{pre: } l \in \text{Learners}\)
7: \(u \in \text{chosen}(m,\beta)\)
8: \(\text{action: } \text{Learned}[l] := \text{Learned}[l] \cup u\)

9: JoinBallot\((m,a)\)
10: \(\text{pre: } a \in \text{Acceptors}\)
11: \(\hat{\beta}_a < m\)
12: \(\hat{\beta}_a := m\)

13: StartBallot\((m,R,k,c)\)
14: \(\text{pre: } c = \text{coord}(m)\)
15: \(\text{MaxTried}[m] = \text{none}\)
16: \(\forall a \in R, \hat{\beta}_a \geq m\)
17: \(m > 0\)
18: \(k = \text{max}\ \{n \in \text{BalNum} : n < m \land \exists a \in R, \beta_a[n] \neq \text{none}\}\)
19: \(\forall a \in \text{BalNum}, k \leq n < m \Rightarrow R \in \text{rquorum}(n)\)
20: \(\beta \text{ safe}\)
21: \(\beta \text{ conservative}\)
22: \(\text{action: } \text{MinTried}[m] := \text{provedSafe}(m,R,k,\beta)\)
23: \(\text{MaxTried}[m] := \text{MinTried}[m]\)

24: Suggest\((m,C,c)\)
25: \(\text{pre: } c = \text{coord}(m)\)
26: \(C \in \text{PrpCmd}\)
27: \(\text{MaxTried}[m] \neq \text{none}\)
28: \(\text{action: } \text{MaxTried}[m] := \text{MaxTried}[m] \bullet C\)

29: ClassicVote\((u,a)\)
30: \(\text{pre: } a \in \text{Acceptors}\)
31: \(\text{MaxTried}[\hat{\beta}_a] \neq \text{none}\)
32: \(\text{MinTried}[\hat{\beta}_a] \subseteq u \subseteq \text{MaxTried}[\hat{\beta}_a]\)
33: \(\beta_a[\hat{\beta}_a] = \text{none} \lor \beta_a[\hat{\beta}_a] \subseteq u\)
34: \(\exists W \in \text{wquorum}(m), a \in W\)
35: \(\text{action: } \beta_a[\hat{\beta}_a] := u\)

36: FastVote\((C,a)\)
37: \(\text{pre: } a \in \text{Acceptors}\)
38: \(C \in \text{PrpCmd}\)
39: \(\beta_a[\hat{\beta}_a] \neq \text{none}\)
40: \(\text{isFast}(\hat{\beta}_a)\)
41: \(\text{action: } \beta_a[\hat{\beta}_a] := \beta_a[\hat{\beta}_a] \bullet C\)

42: Recover\((u,a)\)
43: \(\text{pre: } a \in \text{Acceptors}\)
44: \(\text{isFast}(\hat{\beta}_a++)\)
45: \(\exists W \in \text{wquorum}(\hat{\beta}_a++), a \in W\)
46: \(\beta_a[\hat{\beta}_a] \neq \text{none}\)
47: \(\beta_{\text{coord}}(\hat{\beta}_a)[\hat{\beta}_a] \neq \text{none}\)
48: \(u \subseteq \beta_{\text{coord}}(\hat{\beta}_a)[\hat{\beta}_a]\)
49: \(\text{action: } \text{let } V = \{v \in \text{CStruct} : v \subseteq \beta_a[\hat{\beta}_a] \land \{u,v\} \text{ compatible}\}\)
50: \(\text{let safe} = u \cup V\)
51: \(\hat{\beta}_a := \hat{\beta}_a++\)
52: \(\beta_a[\hat{\beta}_a] := \text{safe}\)
CONSe. ∀m ∈ BalNum, ∀a ∈ Acceptors, ¬isFast(m) ⇒ □(βa[m] ≠ none) ⇒ MinTried[m] ⊆ βa[m] ⊆ MaxTried[m]

CONSf. ∀m ∈ BalNum, □(β safe 
                  β conservative 
                  MinTried[m] ≠ none) ⇒ MinTried[m] safe at ballot m in β

CONSg. ∀m ∈ BalNum, ∀a ∈ Acceptors, □(βa[m] ∈ CStruct ⇒ (∃W ∈ wquorum(m), a ∈ W))

Invariant CONSa tells us that when an acceptor joins a ballot k, it cannot join later a ballot lower than k. CONSb expresses that when βa[m] is a c-struct, it grows monotonically along time. Invariant CONSc encapsulates the fact that when an acceptor joins a ballot k, it cannot change its vote at a ballot lower than k. Those three invariants are classical Paxonians invariants, and follow directly from the code of abstractFGGC (see actions JoinBallot, ClassicVote, FastVote and Recover). Proving these invariants is left to the reader.

When a c-struct is choosable at some point in time, i.e., it is chosen or might be chosen later, then it was always choosable previously (invariant CONSd). This invariant is essential to ensure that when a c-struct is safe at some point in time, it is always safe at some later time. Lemma 1 below proves that abstractFGGC maintains this invariant. We proceed by contradiction.

**Lemma 1.** Algorithm 4 maintains invariant CONSd.

**Proof**

Let: u ∈ CStruct

Prove: ¬(u ∈ choosable(m,β′) ∧ u /∈ choosable(m,β))

Let: W0 such that: ∀a ∈ W0, (βa′ < m ∨ u ⊑ βa[m]′) ∧ (βa[m]′ ≠ none ⇒ {u,βa[m]′} compatible).

Proof: u ∈ choosable(m,β′)

Let: a in W0 such that: (βa > m ∧ u ∉ βa[m]) ∨ (βa[m] ≠ none ∧ ¬ {u,βa[m]} compatible).

Proof: u /∈ choosable(m,β)

Assume: βa′ < m

Case: βa > m

Fact 1: βa′ < m ⇒ βa < m

Proof: Invariant CONSa.

Contradiction

Case: βa[m] ≠ none

Fact 2: βa′ < m ⇒ βa < m

Proof: Invariant CONSa.

Fact 3: βa < m ⇒ βa[m] = none

Proof: Definition of a ballot array.

Contradiction

Assume: u ⊑ βa[m]′
Case: \( \hat{\beta}_a > m \land u \not\subseteq \beta_a[m] \)

Fact 4: \( \hat{\beta}_a > m \land u \not\subseteq \beta_a[m] \Rightarrow u \not\subseteq \beta_a[m]' \)

Proof: Invariant CONSc.

Contradiction

Case: \( \beta_a[m] \neq \text{none} \land \neg \{u, \beta_a[m]\} \text{ compatible} \)

Fact 5: \( \forall v \in CStruct, \beta_a[m] \subseteq v \Rightarrow u \not\subseteq v \)

Proof: \( \neg \{u, \beta_a[m]\} \text{ compatible} \)

Fact 6: \( \beta_a[m] \subseteq \beta_a[m]' \)

Proof: Invariant CONSb.

Contradiction

\[ \square \]

Invariant CONSe tells us that when \( m \) is classic, for every acceptor \( a \), if \( \beta_a[m] \) is a c-struct, then \( \text{MinTried}[m] \subseteq \beta_a[m] \subseteq \text{MaxTried}[m] \) always holds.

**Lemma 2.** Algorithm 4 maintains invariant CONSe.

**Proof**

Let: \( m \) a classic ballot

Let: \( a \) an acceptor s.t. \( \beta_a[m]' \neq \text{none} \)

Prove: \( \text{MinTried}[m]' \subseteq \beta_a[m]' \subseteq \text{MaxTried}[m]' \)

Fact 1: \( \text{MinTried}[m] \subseteq \beta_a[m] \subseteq \text{MaxTried}[m] \)

Case: \( \text{StartBallot}(m, -, -, -) \)

Fact 2: \( \beta_a[m] \neq \text{none} \)

Proof: Invariant CONSb and definition of \( \text{none} \).

Fact 3: \( \text{MinTried}[m] \neq \text{none} \)

Proof: Definition of \( \text{none} \).

Fact 4: \( \text{MaxTried}[m] \neq \text{none} \)

Proof: Definition of \( \text{none} \).

Contradiction

Proof: \( \text{MaxTried}[m] = \text{none} \) is a precondition of \( \text{StartBallot} \) : line 18 in Algorithm 4.

Case: \( \text{Suggest}(m,C,-) \)

Fact 5: \( \text{MaxTried}[m]' = \text{MaxTried}[m] \cdot C \)


Unchanged: \( \beta_a[m], \text{MinTried} \)

QED.

Case: \( \text{ClassicVote}(u,a) \land \hat{\beta}_a = m \)
Fact 6: $\text{MinTried}[m] \sqsubseteq u \sqsubseteq \text{MaxTried}[m]$
   Proof: This is a precondition of \textit{ClassicVote}: line 37 in Algorithm 4.

Fact 7: $\beta_a[m]' = u$

Unchanged: \text{MinTried}, \text{MaxTried}

QED.

A direct consequence of CONSe, is that $\beta$ is conservative (invariant CONS1).

\textbf{Proposition 5.} \textit{Algorithm 4 maintains invariant CONS1.}

\textit{Proof}

Let: $m$ a classic ballot

Prove: \{ $\beta_a[m] \neq \text{none} : a \in \text{Acceptors}$ \} compatible

Let: $a \in \text{Acceptors}, \beta_a[m] \neq \text{none}$

Fact 1: $\beta_a[m] \sqsubseteq \text{MaxTried}[m]$
   Proof: Since ballot $m$ is classic and $\beta_a[m]$ is a c-struct, we may apply invariant CONSe.

QED.

If ballot $m$ is classic and $\beta$ is both safe and conservative, the c-struct initially suggested by the coordinator of $m$ is safe.

\textbf{Lemma 3.} \textit{Algorithm 4 maintains invariant CONSf.}

\textit{Proof}

Let: $m$ a ballot

Assume: $\beta$ safe $\land$ $\beta$ conservative

Prove: $\text{MinTried}[m]'$ safe at ballot $m$ in $\beta$

Case: \textit{StartBallot}(m,R,k,−)

Fact 1: $\text{provedSafe}(m,R,k,\beta)$ safe at ballot $m$ in $\beta$

Fact 2: $\forall a \in R, \hat{\beta}_a \geq m$

Fact 3: $m > 0$

29
Fact 4: \( k = \max \{ n \in \text{BalNum} : n < m \land \exists a \in R, \beta_a[n] \neq \text{none} \} \)

Fact 5: \( \forall n \in \text{BalNum}, k \leq n < m \Rightarrow R \in \text{rquorum}(n) \)

Fact 6: \( \beta \text{ safe} \land \beta \text{ conservative} \)
Proof: By hypothesis

QED.

Proof: Proposition 3.

Fact 7: \( \text{MinTried}[m]' = \text{provedSafe}(m,R,k,\beta) \)

QED.

QED.

When an acceptor \( a \) accepts some c-struct \( u \), \( a \) belongs to some write quorum of \( m \): invariant CONSg. This invariant clearly follows from the preconditions at lines 39 and 52 on actions \text{ClassicVote} and \text{Recover}.

We are now able to prove invariant CONS2, i.e., that \( \beta \) is always safe.

Proposition 6. Algorithm 4 maintains invariant CONS2

Proof

Let: \( m \) a ballot, \( a \) an acceptor such that \( \beta_a[m]' \neq \text{none} \), \( n \) a ballot such that \( n < m \), and \( v \) a c-struct belonging to \text{choosable}(n,\beta').

Assume: \( \beta \text{ safe} \land \beta \text{ conservative} \)
Prove: \( v \subseteq \beta_a[m]' \)

Fact 1: \( v \in \text{choosable}(n,\beta) \)
Proof: \( v \in \text{choosable}(n,\beta') \) and invariant CONSd.

Case: \( \neg \text{isFast}(m) \land \beta_a = m \land \text{ClassicVote}(u,a) \)

Fact 2: \( v \subseteq \text{MinTried}[m] \)

Fact 3: \( \text{MinTried}[m] \text{ safe at ballot } m \text{ in } \beta \)
Fact 4: \( \beta \text{ safe} \land \beta \text{ conservative} \)
Proof: By hypothesis

Fact 5: \( \text{MinTried}[m] \neq \text{none} \)

QED.

Proof: Invariant CONSf.

Fact 6: \( v \in \text{choosable}(n,\beta) \)
Proof: Fact 1.
Fact 7: \( n < m \)
QED.

Fact 8: \( \text{MinTried}[m] \subseteq u \)

Fact 9: \( \beta_a[m]' = u \)
QED.

Case: \( \text{isFast}(m) \land \hat{\beta}_a = m \land \text{FastVote}(C,a) \)

Fact 10: \( v \subseteq \beta_a[m] \)

Fact 11: \( \beta_a[m] \) safe at ballot \( m \) in \( \beta \)
Fact 12: \( \beta_a[m] \neq \text{none} \)

Fact 13: \( v \in \text{choosable}(n,\beta) \)
Fact 14: \( n < m \)
Fact 15: \( \hat{\beta}_a = m \)
Fact 16: \( \beta \) is safe.
QED.

Fact 17: \( \beta_a[m]' = \beta_a[m] \cdot C \)
QED.

Case: \( \text{isFast}(m) \land \hat{\beta}_a++ = m \land \text{Recover}(u,a) \)

Fact 18: \( v \subseteq \beta_a[\hat{\beta}_a] \)

Case: \( n < \hat{\beta}_a \)

Fact 19: \( \beta_a[\hat{\beta}_a] \) safe at ballot \( \hat{\beta}_a \) in \( \beta \)
Fact 20: \( \beta_a[\hat{\beta}_a] \neq \text{none} \)
Fact 21: \( \beta \) is safe.
QED.

Fact 22: \( v \in \text{choosable}(n,\beta) \)
Proof: Fact 1.
Fact 23: \( n < \hat{\beta}_a \)
QED.

Case: \( \hat{\beta}_a = n \)
Let: \( W \in wquorum(n), \forall b \in W, \hat{\beta}_b' > n \Rightarrow v \subseteq \beta_b'[n]' \)
Proof: Part (i) of the definition of \( v \) choosable at ballot \( n \) in \( \beta' \).

Fact 24: \( a \in W \)
Fact 25: \( \exists W \in wquorum(n), a \in W \)
Proof: Line 53 in Algorithm 4 implies that \( \beta_a[n] \neq \text{none} \); invariant CONSg concludes.
Fact 26: $|\text{wquorum}(n)| = 1$
Proof: Assumption FGGC2.
QED.

Fact 27: $\hat{\beta}_a' > n$
Fact 28: $\hat{\beta}_a' = \hat{\beta}_a++$
Fact 29: $\hat{\beta}_a = n$
QED.

QED.

Fact 30: $\{v, \beta_c[\hat{\beta}_a]\}$ compatible
Case: $n < \hat{\beta}_a$

Fact 31: $\beta_c[\hat{\beta}_a]$ safe at ballot $\hat{\beta}_a$ in $\beta$
Fact 32: $\beta_c[\hat{\beta}_a] \neq \text{none}$
Proof: Line 54 in Algorithm 4.
Fact 33: $\beta$ is safe.
QED.

Fact 34: $v \in \text{choosable}(n, \beta)$
Proof: Fact 1.
Fact 35: $n < \hat{\beta}_a$
QED.

Case: $\hat{\beta}_a = n$

Let: $W \in \text{wquorum}(n), \forall b \in W, \beta_c[n] \neq \text{none} \Rightarrow \{\beta_c[n], v\}$ compatible
Proof: Part (ii) of the definition of $v$ choosable at ballot $n$ in $\beta'$.

Fact 36: $c \in W$

Fact 37: $\exists W \in \text{wquorum}(n), c \in W$
Proof: Line 54 in Algorithm 4 and invariant CONSg.
Fact 38: $|\text{wquorum}(n)| = 1$
Proof: Assumption FGGC2.
QED.

Fact 39: $\beta_c[\hat{\beta}_a] \neq \text{none}$
Proof: Line 54 in Algorithm 4.
QED.

Fact 40: $v \in V$

Fact 41: $\{u, v\}$ compatible

Fact 42: $u \subseteq \beta_c[\hat{\beta}_a]$

Fact 43: $\{v, \beta_c[\hat{\beta}_a]\}$ compatible
Proof: Fact 30.
QED.
Fact 44: \( v \subseteq \beta_a[\beta_a] \)
Proof: Fact 18.
QED. Line 56 in Algorithm 4.

Fact 45: \( v \subseteq \text{safe} \)
Proof: Fact 40 and line 57 in Algorithm 4.
QED.

Proof: Line 59 concludes.

\[ \square \]

A.2.3 NT1, NT2 and STB1

Proposition 7 and 8 below show that every c-struct accepted by some acceptor (NT1) or suggested by the leader (NT2), is always constructable with proposed commands. We then prove in proposition 9 that learned c-structs are stable along time (STB1).

Proposition 7. Algorithm 4 maintains invariant NT1.

Proof

Let: \( m \) a ballot and \( a \) an acceptor such that: \( \beta_a[m]' \neq \text{none} \land \text{MaxTried}[m]' \neq \text{none} \).

Prove: \( \beta_a[m]' \in \text{Str}(\text{PrpCmd}') \land \text{MaxTried}[m]' \in \text{Str}(\text{PrpCmd}') \)

Case: Propose \((-, -)\)

Fact 1: \( \text{PrpCmd}' \subseteq \text{PrpCmd} \)

Unchanged: \( \text{maxTried}, \beta \).

QED.

Case: StartBallot \((m, -, -, -)\)

Fact 2: \( \text{MinTried}[m]' \in \text{Str}(\text{PrpCmd}) \)

Fact 3: \( \text{MinTried}[m]' = \text{provedSafe}(m, R, k, \beta) \)
Proof: Line 25 in Algorithm 4

Fact 4: \( \text{provedSafe}(m, R, k, \beta) \) returns a c-struct in \( \text{Str}(\text{PrpCmd}) \)
Proof: Invariant NT1 and proposition 4.

QED.

Fact 5: \( \text{MaxTried}[m]' = \text{MinTried}[m]' \)

Unchanged: \( \beta, \text{PrpCmd} \).

QED.

Case: Suggest \((m, C, -)\)
Fact 6: $C \in PrpCmd$

Fact 7: $MaxTried[m] \in Str(PrpCmd)$

Fact 8: $MaxTried[m] \neq none$

QED.
Proof: Invariant NT1

Fact 9: $MaxTried[m]' = MaxTried[m] \cdot C$

Unchanged: $\beta, PrpCmd$.
QED.

Case: $ClassicVote(u,a)$

Assume: $\hat{\beta}_a = m$

Fact 10: $MaxTried[m] \in Str(PrpCmd)$
Proof: Line 36 in Algorithm 4 and invariant NT1.

Fact 11: $u \subseteq MaxTried[m]$

Fact 12: $u \in Str(PrpCmd)$
Proof: Facts 10 and 11, and assumption CS3.1.

Fact 13: $\beta_a[m]' = u$

Unchanged: $maxTried, PrpCmd$.
QED.

Case: $FastVote(C,a)$

Assume: $\hat{\beta}_a = m$

Fact 14: $C \in PrpCmd$

Fact 15: $\beta_a[m] \in Str(PrpCmd)$
Proof: Line 45 in Algorithm 4, and invariant NT1.

Fact 16: $\beta_a[m]' = \beta_a[m] \cdot C$

Unchanged: $maxTried, PrpCmd$.
QED.

Case: $Recover(u,a)$

Assume: $\hat{\beta}_a = m$

Fact 17: $\forall v \in V, v \in Str(PrpCmd)$
Fact 18: $\beta_a[m] \in \text{Str}(PrpCmd)$
Proof: Invariant NT1.

Fact 19: $\forall v \in \mathcal{V}, v \sqsubseteq \beta_a[m]$
QED.

Fact 20: $\beta_{\text{coord}(m)}[m] \in \text{Str}(PrpCmd)$
Proof: Line 54 in Algorithm 4, and invariant NT1.

Fact 21: $\text{safe} \in \text{Str}(PrpCmd)$
Proof: Line 57 in Algorithm 4 and assumption CS3.2.

Unchanged: $\text{maxTried, PrpCmd}$. QED.

\[ \square \]

Proposition 8. Algorithm 4 maintains invariant NT2.

Proof

Let: $l$ a learner.

Prove: $\exists V \subseteq \text{chosen}(\beta'), \text{Learned}[l]' = \sqcup V$

Fact 1: $\text{chosen}(\beta) \subseteq \text{chosen}(\beta')$
Proof: This is implied by invariant CONSb.

Case: $\text{Learn}(-,l,u)$

Let: $\mathcal{U}$ a subset of $\text{chosen}(\beta)$ s.t. $\text{Learned}[l] = \sqcup \mathcal{U}$
Proof: Invariant NT2.

Let: $V = \mathcal{U} \cup \{u\}$

Fact 2: $u \in \text{chosen}(\beta)$

Fact 3: $\text{Learned}[l]' = \sqcup V$
QED.

QED.

\[ \square \]


Proof

35
Let: \( l \) a learner.

Prove: \( \text{Learned}[l] \sqsubseteq \text{Learned}[l]' \)

Case: \( \text{Learn}(-,l,u) \)

Fact 1: \( \text{Learned}[l]' = \sqcup \{u, \text{Learned}[L]\} \)
QED.

A.2.4 From abstractFGGC to \( GC^* \)

To prove that abstractFGGC implements \( GC^* \), we consider the following refinement’s mapping \( R \):
For every process \( p \) in Proposers, we map \( \text{Proposed}[p] \) in abstractFGGC to \( \text{proposed}_p \) in \( GC^* \); that is; \( \forall p \in \text{Proposers}, \text{proposed}_p \overset{R}{=} \text{Proposed}[p] \). For every learner \( l \) in Learners, we map \( \text{Learned}[l] \) in abstractFGGC to \( \text{learned}_l \) in \( GC^* \), i.e., \( \forall l \in \text{Learners}, \text{learned}_l \overset{R}{=} \text{Learned}[l] \).

To prove the correctness of our refinement’s mapping we have to show that it satisfies the following properties: (i) for every state \( s \) of abstractFGGC, the external state of \( R(s) \), is an external state of \( GC^* \), (ii) for every initial state \( s_0 \) of abstractFGGC, \( R(s_0) \) is an initial state of \( GC^* \), (iii) for every transition \( (s,s') \) of abstractFGGC, \( (R(s),R(s')) \) is a transition of \( GC^* \) or \( R(s) = R(s') \), and (iv) for every run \( r \) of abstractFGGC, \( r \) satisfies the liveness property of \( GC^* \).

**Proposition 10.** For every state \( s \) of abstractFGGC, \( R(s) \) satisfies the non-triviality and consistency invariant of \( GC^* \).

**Proof**

- Non-triviality

  Fact 1: \( \forall m \in \text{BalNum}, \text{chosen}(m,\beta) \subseteq \text{Str}(\text{PrpCmd}) \)

    Let: \( m \) a ballot and \( u \) a c-struct chosen at ballot \( m \) in \( \beta \).

  Fact 2: \( u \in \text{Str}(\text{PrpCmd}) \)

    Proof: Invariant NT1 and proposition 2.

    QED.

  Fact 3: \( \forall l \in \text{Learners}, \text{Learned}[l] \in \text{Str}(\text{PrpCmd}) \)

    Let: \( l \) a learner

    Let: \( U \) a subset of \( \text{chosen}(\beta) \) s.t. \( \text{Learned}[l] = \sqcup U \).

    Proof: Invariant NT2.

  Fact 4: \( \forall u \in U, u \in \text{Str}(\text{PrpCmd}) \)

    Proof: Fact 1.
Fact 5: $\mathcal{U}$ compatible
Proof: Invariants CONS1, CONS2 and proposition 1.
QED.
Proof: Fact 4 and 5, and assumption CS3.2.

Fact 6: $\forall l \in \text{Learners}, \text{learned}_l \sqsupseteq \text{Learned}[l]$

Fact 7: $\text{prpCmd} = \text{PrpCmd}$

Fact 8: $\text{prpCmd} = \bigcup_{p \in \text{Proposers}} \text{proposed}_p$

Fact 9: $\text{PrpCmd} = \bigcup_{p \in \text{Proposers}} \text{Proposed}[p]$

Fact 10: $\forall p \in \text{Proposers}, \text{proposed}_p \sqsupseteq \text{Proposed}[p]$
QED.

Fact 11: $\forall l \in \text{Learners}, \text{learned}_l \sqsubseteq \text{Str}(\text{prpCmd})$
Proof: Facts 3, 6 and 7.
QED.

- Consistency

Fact 12: $\{\text{Learned}[l] : l \in \text{Learners}\}$ compatible
Let: Let $v$ be a c-struct s.t. for every c-struct $u$ in chosen($\beta$), $v$ suffixes $u$.
Proof: By invariants CONS1 and CONS2 and proposition 1, chosen($\beta$) is always compatible.

Fact 13: $\forall l \in \text{Learners}, \text{Learned}[l] \sqsubseteq v$
Let: $l$ a learner
Let: Pick $\mathcal{U}_l \subseteq \text{chosen}(\beta)$ s.t. $\text{Learned}[l] = \bigvee \mathcal{U}_l$.
Proof: Invariant NT2.

Fact 14: $v$ is an upper bound of $\mathcal{U}_l$.
Proof: $v$ is an upper bound of chosen($\beta$), and $\mathcal{U}_l$ is a subset of chosen($\beta$).

Fact 15: $\bigvee \mathcal{U}_l$ is the lower upper bound of $\mathcal{U}_l$.
Fact 16: $\bigvee \mathcal{U}_l \sqsubseteq v$
Proof: Facts 14 and 15.
QED.
QED.

Fact 17: $\forall l \in \text{Learners}, \text{learned}_l \sqsupseteq \text{Learned}[l]$
QED.

Proposition 11. Let $s$ and $s'$ be two states of abstractFGGC such that $(s,s')$ is a transition of abstractFGGC. The pair $(\mathcal{A}(s),\mathcal{A}(s'))$ is a transition of $\text{GC}^*$ or $\mathcal{A}(s)$ equals $\mathcal{A}(s')$. In both cases, the resulting state $\mathcal{A}(s')$ preserves the stability invariant of $\text{GC}^*$. 

□
Proof

Observe that if action \textit{StartBallot}, \textit{JoinBallot}, \textit{ClassicVote}, or \textit{FastVote} is executed in state \( s \), then \( \mathcal{R}(s) \) equals \( \mathcal{R}(s') \) since none of these actions modifies \textit{Proposed} or \textit{Learned} (stuttering).

If now we consider that for some proposer \( p \), and for some command \( C \), action \textit{Propose}(\( C,p \)) is executed, since \( \text{proposed}_p \) equals \( \text{Proposed}[p] \) by \( \mathcal{R} \), it is easy to see that \( \mathcal{R}(s') \) is the state of \( GC^* \) after the execution of \( \text{propose}_p(C) \) in state \( \mathcal{R}(s) \).

Suppose now that for some ballot \( m \), some c-struct \( u \) and some learner \( l \), state \( s' \) results from the execution of \textit{Learn}(\( m,u,l \)) at state \( s \). Dixit line 3 in Algorithm 4, \( \text{Learned}[l] \) in state \( s' \) equals \( \text{Learned}[l] \cup u \). Thus, in state \( \mathcal{R}(s') \), \( \text{learned}_l \) equals \( \text{Learned}[l] \cup u \). Hence, executing \textit{Learn}(\( m,u,l \)) implements action \textit{learn}_l(\( \text{learned}_l \cup u \)). This operation preserves the stability invariant of \( GC^* \).

We deduce from proposition 10 and proposition 11, \( \mathcal{R} \) satisfies properties (i) and (iii). According to the way we defined an initial state of \textit{abstractFGGC} in Section A.2.1, and the refinement’s mapping \( \mathcal{R} \), properties (ii) is immediate. Property (iv) of \( \mathcal{R} \) comes from the fact that there is no liveness property for \( GC^* \). We conclude that \textit{abstractFGGC} implements \( GC^* \).

A.3 Correctness of \textit{FGGC} 

We first show the existence of a refinement mapping from \textit{FGGC} to \textit{abstractFGGC}. This implies that \textit{FGGC} implements \( GC^* \). We then prove that under some synchrony assumptions, that we will express in the unreliable failure detectors model, \textit{FGGC} satisfies the liveness property of generalized consensus and is both fast and genuine.

A.3.1 From \textit{FGGC} to \textit{abstractFGGC} 

To model the underlying message-passing system, we add variable \textit{Msg} to \textit{FGGC}. This variable is shared among all processes in the system and initially equals \( \{\} \). Condition “received \textit{msg} from \textit{j}” is true at process \( i \) iff the triple \((j,\textit{msg},i)\) is in \textit{Msg}. Recall that in Section 3 we assume that links may lost messages. To model this, we consider that operation “send \textit{msg} to \textit{j}” for some message \textit{msg} and some process \textit{j}, may add or not the triple \((i,\textit{msg},j)\) to \textit{Msg}.

\textbf{History variables} In order to construct our refinement mapping from \textit{FGGC} to \textit{abstractFGGC}, we add history variables \([1]\) as well as some additional operations to the actions of \textit{FGGC}. We describe them below:

\( h\text{MinTried} \) and \( h\text{MaxTried} \): We add to each coordinator \( c \) two unbounded arrays: \( h\text{MinTried}_c \) and \( h\text{MaxTried}_c \). Each of these arrays maps to every ballot \( k \) a value \( h\text{MinTried}_c[k] \) in \( \text{CStruct} \cup \{\text{none}\} \). Initially, both \( h\text{MinTried}_c[0] \) and \( h\text{MaxTried}_c[0] \) equals \( \bot \), and for every ballot \( m \) greater than 0, \( h\text{MinTried}_c[m] \) and \( h\text{MaxTried}_c[m] \) equals \( \text{none} \). After \( c \) executes line 29 in Algorithm 1, \( c \) sets both \( h\text{MinTried}_c[\text{maxStart}_c] \) and \( h\text{MaxTried}_c[\text{maxStart}_c] \) to the value of \( \text{maxTried}_c \). After \( c \) executes line 37 in Algorithm 1, \( c \) sets \( h\text{MaxTried}_c[\text{maxStart}_c] \) to the value of \( \text{maxTried}_c \).

\footnote{Modeling non-reliable links is \textit{de facto} non-deterministic. We do not however assume non-deterministic processes.}
hcval: We augment the state of each acceptor \( a \) with the history variable \( hcval_a \). This variable is an unbounded array that maps every ballot \( m \) to a value \( hcval_a[m] \) in \( CStruct \cup \{ \) none \( \} \). Initially, \( hcval_a[0] \) equals \( cval_a \), i.e. \( \bot \), and for every ballot \( m \) greater than \( 0 \), \( hcval_a[m] \) equals \( \) none \( \). After \( a \) executes lines 45 and 54 in Algorithm 1, and line 13 in Algorithm 2, acceptor \( a \) sets \( hcval_a[cbal_a] \) to the value of \( cval_a \).

The refinement mapping We consider the following refinement mapping \( \mathfrak{R} \) from the states of \( FGGC \) (augmented with the history variables we defined above) to the states of \( abstractFGGC \):

- \( \forall p \in Proposers, \) Proposed\( [p] \mathfrak{R} \{ C \in Cmd : \exists i \in Acceptors \cup Coordinators, (p,(propose,C),i) \in Msg \} \)
- \( \forall l \in Learners, Learned[l] \mathfrak{R} learned_l \)
- \( \forall m \in BalNum, MaxTried[m] \mathfrak{R} hMaxTried_{coord(m)}[m] \)
- \( \forall m \in BalNum, MinTried[m] \mathfrak{R} hMinTried_{coord(m)}[m] \)
- \( \forall a \in Acceptors, \forall m \in BalNum, \beta_a[m] \mathfrak{R} hcval_a[m] \)
- \( \forall a \in Acceptors, \hat{\beta}_a \mathfrak{R} bal_a \)

A.3.2 Correctness of \( \mathfrak{R} \)

The correctness of \( \mathfrak{R} \) relies on the following invariants:

PAX1. \( \forall a \in Acceptors, \forall m \in BalNum, \Box (cbal_a \leq bal_a \land cbal_a = m \Rightarrow o(cbal_m \geq m) \land bal_a = m \Rightarrow o(bal_m \geq m)) \)

PAX2. \( \forall a \in Acceptors, \Box (cval_a \neq none) \)

MSG1. \( \forall c \in Coordinators, \forall m \in BalNum, \Box ((c,(1A,m),\bot) \in Msg \Rightarrow m > 0) \)

MSG2. \( \forall a \in Acceptors, \forall m \in BalNum, \Box ((a,(1B,m),\bot) \in Msg \Rightarrow \Box (bal_a \geq m)) \)

MSG3. \( \forall c \in Coordinators, \forall m \in BalNum, \Box ((\bot,(1B,m,\bot),c) \inMsg \Rightarrow c = coord(m)) \)

MSG4. \( \forall m \in BalNum, \forall u \in CStruct \cup \{ none \}, \Box ((\bot,(2A,m,u),\bot) \in Msg \Rightarrow (hMinTried_{coord(m)}[m] \subseteq u \subseteq hMaxTried_{coord(m)}[m] \land u \in CStruct)) \)

MSG5. \( \forall a \in Acceptors, \forall u \in CStruct \cup \{ none \}, \forall m \in BalNum, \Box ((a,(2B,m,u),\bot) \in Msg \Rightarrow (u \subseteq hcval_a[m] \land u \in CStruct)) \)

HST1. \( \forall c \in Coordinators, \forall m \in BalNum, \Box (\Box (maxStart_c = m \Rightarrow (hMaxTried_c[m] = maxTried_c)) \)

HST2. \( \forall a \in Acceptors, \forall m \in BalNum, \Box (cbal_a = m \Rightarrow (hcval_a[m] = cval_a)) \)

HST3. \( \forall a \in Acceptors, \forall m \in BalNum, \Box (cbal_a = m \Rightarrow (\forall n > m, hcval_a[n] = none)) \)

HST4. \( \forall a \in Acceptors, \forall m \in BalNum, \forall u \in CStruct \cup \{ none \}, \Box ((bal_a > m \land hcval_a[m] = u) \Rightarrow \Box (hcval_a[m] = u)) \)
Proving these invariants is left to the reader.

**Proposition 12.** For every initial state $s_0$ of FGGC, $\mathcal{F}(s_0)$ is an initial state of abstractFGGC.

**Proof**

Let: $s_0$ an initial state of FGGC.

Fact 1: $\forall p \in \text{Proposers}, \text{Proposed}[p] = \emptyset$

Proof: $\text{Msg}$ is initially empty (see Section A.3.1).

Fact 2: $\forall l \in \text{Learners}, \text{Learned}[l] = \bot$

Proof: Section 3.2 tells us that for every learner $l$, learned$_l$ equals initially $\bot$.

Fact 3: $\forall a \in \text{Acceptors}, \hat{\beta}_a = 0 \land \beta_a[0] = \bot \land (\forall m \in \text{BalNum}, m > 0 \Rightarrow \beta_a[m] = \text{none})$

Proof: Section 3.2 tells us that initially, for every acceptor $a$, bal$_a$ equals 0 and cval$_a$ equals $\bot$. Section A.3.1 tells us that initially, for every acceptor $a$ and every ballot $m$ greater than 0, hcval$_a[m]$ equals none.

Fact 4: $(\text{MaxTried}[0] = \text{MinTried}[0] = \bot) \land (\forall m > 0, \text{MaxTried}[m] = \text{MinTried}[m] = \text{none})$

Proof: Section A.3.1 tells us that $h$MinTried$_{\text{coord}}(0)$ and $h$MaxTried$_{\text{coord}}(0)$ equals initially $\bot$, and that for every ballot $m$ greater than 0, $h$MinTried$_{\text{coord}}(m)$ and $h$MaxTried$_{\text{coord}}(m)$ equals none.

QED.

Proof: Facts 1 to 4 imply that $\mathcal{F}(s_0)$ is an initial state of abstractFGGC.

**Proposition 13.** For every step $(s,s')$ of FGGC, $(\mathcal{F}(s),\mathcal{F}(s'))$ is a (possibly stuttering) step of abstractFGGC.

**Proof**

Let: $s$ a state of FGGC, and $(s,s')$ a step taken by FGGC in state $s$ by a process $i$.

Assume: $(s,s') = \text{propose}_i(C)$

Prove: $(\mathcal{F}(s),\mathcal{F}(s')) = \text{Propose}(C,i) \lor \mathcal{F}(s) = \mathcal{F}(s')$

Fact 1: $i \in \text{Proposers}$

Proof: Line 1 in Algorithm 1.

Fact 2: $C \in \text{Cmd}$

Proof: Line 2 in Algorithm 1.

Unchanged: Learned, $\beta$, MinTried and MaxTried.

Case: $\text{Msg'} = \text{Msg}$

QED.

Case: $\exists J \subseteq \text{Acceptors} \cup \text{Coordinators}, \text{Msg'} = \text{Msg} \cup \{(i,(\text{propose},C),j) : j \in J\}$
Fact 3: \( \text{Proposed}[i]' = \text{Proposed}[i] \cup \{C\} \)

Proof: From the definition of \( \text{Proposed} \) by \( \mathfrak{F} \).

QED.

QED.

Proof: Line 3 in Algorithm 1 is the single operation of \textit{propose} that modifies \( \text{Proposed} \).

Assume: \((s,s') = \text{phase1A}_i(m)\)

Prove: \( \mathfrak{F}(s) = \mathfrak{F}(s') \)

Unchanged: \( \text{Proposed, Learned, } \beta, \text{MinTried and MaxTried} \).

QED.

Assume: \((s,s') = \text{phase1B}_i(m)\)

Prove: \( (\mathfrak{F}(s),\mathfrak{F}(s')) = \text{JoinBallot}(m,i) \)

Fact 4: \( i \in \text{Acceptors} \)

Proof: Line 12 in Algorithm 1.

Fact 5: \( \hat{\beta}_i < m \)

Fact 6: \( \hat{\beta}_i \equiv bal_i \)

Fact 7: \( bal_i < m \)

Proof: Line 13 in Algorithm 1.

QED.

Fact 8: \( \hat{\beta}_i' = m \)

Fact 9: \( bal_i' = m \)

Proof: Line 15 in Algorithm 1.

Fact 10: \( \hat{\beta}_i' \equiv bal_i' \)

QED.

Fact 11: \( \forall m \in \text{BalNum}, \beta_i[m]' = \beta_i[m] \)

Fact 12: \( \forall a \in \text{Acceptors}, a \neq i \Rightarrow (\hat{\beta}_a' = \hat{\beta}_a \land (\forall m \in \text{BalNum}, \beta_a[m]' = \beta_a[m])) \)

Unchanged: \( \text{Proposed, Learned, MinTried and MaxTried} \).

QED.

Assume: \((s,s') = \text{phase2Start}_i(m,R,k)\)

Prove: \( (\mathfrak{F}(s),\mathfrak{F}(s')) = \text{StartBallot}(m,R,k,i) \)

Fact 13: \( i = \text{coord}(m) \)

Fact 14: \( i \) received a message \((1B,m,\ldots,\ldots)\) from some acceptor \( a \).

Proof: Line 21 in Algorithm 1.

QED.

Proof: Invariant MSG3.

Fact 15: \( \text{MaxTried}[m] = \text{none} \)

Fact 16: \( h\text{MaxTried}_i[m] = \text{none} \)
Fact 17: \( \text{maxTried}_i = \text{none} \)
Proof: Line 19 in Algorithm 1.

Fact 18: \( \text{maxStart}_i = m \)
Proof: Line 20 in Algorithm 1.

QED.
Proof: Invariant HST1.

Fact 19: \( \text{MaxTried}[m] = h\text{MaxTried}_i[m] \)
Fact 20: \( \text{MaxTried}[m] \overset{\Delta}{=} h\text{MaxTried}_{\text{coord}(m)}[m] \)
Fact 21: \( \text{coord}(m) = i \)

QED.
Proof: Invariant MSG2.

Fact 22: \( \forall a \in R, \hat{\beta}_a \geq m \)

Fact 23: \( i \) received a message \((1B, m, \ldots, \ldots)\) from every acceptor \(a\) in \(R\)
Proof: Line 21 in Algorithm 1.

QED.

Fact 24: \( m > 0 \)

QED.
Proof: Initially for every acceptor \(a\), \( \text{bal}_a \) equals 0, and the precondition at line 13 in Algorithm 1.

Fact 25: \( k = \max \{n \in \text{BalNum} : n < m \land \exists a \in R, \beta_a[n] \neq \text{none} \} \)
Let: \( a \) an acceptor in \( R \).
Let: \( n_a \) a natural such that \( i \) received a message \((1B, m, n_a, \ldots)\) from \(a\)
Proof: Line 21 of Algorithm 1

Fact 26: \( n_a < m \)
Proof: Acceptor \( a \) sends a message \((1B, m, n_a, \ldots)\) only if \( a \) executes line 16 with \( c\text{bal}_a = n_a \). Precondition at line 13 in Algorithm 1 ensures that at that time, \( \text{bal}_a \) is smaller than \( m \). Invariant PAX1 implies that \( n_a < m \).

Fact 27: \( \beta_a[n_a] \neq \text{none} \)

Fact 28: \( \text{hcval}_a[n_a] \neq \text{none} \)
Proof: Acceptor \( a \) sends a message \((1B, m, n_a, \ldots)\) only if \( a \) executes line 16 with \( c\text{bal}_a = n_a \). From HST2 we know that \( \text{hcval}_a[n_a] \) equals at that time \( \text{cval}_a \).
Invariant PAX2 tells us that \( \text{cval}_a \) is always a c-struct.

Fact 29: \( \beta_a[n_a] \overset{\Delta}{=} \text{hcval}_a[n_a] \)

QED.

Fact 30: \( \forall n \in \text{BalNum}, (n > n_a \land n < m) \Rightarrow \beta_a[n] = \text{none} \)
Proof: Acceptor \( a \) sends a message \((1B, m, n_a, \ldots)\) only if \( a \) executes line 16 with \( c\text{bal}_a = n_a \). Invariant HST3 implies that at that time, for every ballot \( n \) greater than \( n_a \), \( \text{hcval}_a[n] \) equals \( \text{none} \). Since when \( a \) executes line 16, \( \text{bal}_a \) is greater than \( m \), invariant HST4 implies that if in addition \( n \) is smaller than \( m \), \( \text{hcval}_a[n] \) remains equals to \( \text{none} \) at all later time.
QED.
Proof: Line 21 in Algorithm 1.

Fact 31: \( \forall n \in \text{BalNum}, k \leq n < m \Rightarrow R \in \text{rquorum}(n) \)
QED.
Proof: Line 22 in Algorithm 1.

Fact 32: \( \beta_{\text{safe}} \land \beta_{\text{conservative}} \)
QED.

Proof: This is true at state \( \mathfrak{F}(s) \).

Fact 33: \( \text{MinTried}[m]' = \text{provedSafe}(m,R,k,\beta) \land \text{MinTried}[m]' = \text{MaxTried}[m]' \)

Fact 34: \( \forall a \in \text{acceptorSet}, \text{received } (1B,m,k,u) \text{ from } a \Rightarrow u = \beta_a[k] \)
Let: \( a \) an acceptor and \( u \) a value in \( CStruct \cup \{\text{none}\} \) such that \( i \) received a message \( (1B,m,k,u) \) from \( a \).

Fact 35: \( u = \text{hcval}_a[k] \)
Proof: From the definition of \( Msg \), message \( (1B,m,k,u) \) comes from the execution of \( \text{phase1B}(m) \) by acceptor \( a \). At this time, \( \text{bal}_a \) is strictly greater than \( m \) and \( u \) equals \( \text{hcval}_a[k] \). Invariant HST4 concludes.

Fact 36: \( \beta_a[m] \overset{3}{=} \text{hcval}_a[k] \)
QED.

Fact 37: \( \text{MinTried}[m] = h\text{MinTried}_{m[c]}' \)
Fact 38: \( \text{MinTried}[m] \overset{3}{=} h\text{MinTried}_{\text{coord}(m)}[m] \)
Fact 39: \( \text{maxStart}_i = m \)
Fact 40: \( i = \text{coord}(m) \)
Fact 41: \( \text{maxStart}_i' = \text{maxStart}_i \)
QED.

Fact 42: \( \text{MaxTried}[m] = h\text{MaxTried}_{m[c]}' \)
Proof: Similar to Fact 37.
Fact 43: \( h\text{MinTried}_{m[c]}' = h\text{MaxTried}_{m[c]}' \)
Proof: Section A.3.1.
QED.

Proof: Facts 24 and that lines 24 to 29 of Algorithm 1 match lines 7 to 12 of Algorithm 3 imply that \( h\text{MinTried}_{m[c]}' \) equals \( \text{provedSafe}(m,R,k,\beta) \). Facts 42 and 43 concludes.

Unchanged: \( \text{Proposed, Learned and } \beta \)
QED.

Assume: \( (s,s') = \text{phase2AClassic},_i(m,C) \)

Prove: \( (\mathfrak{F}(s),\mathfrak{F}(s')) = \text{Suggest}(C,i,) \)

Fact 44: \( i \in \text{Acceptors} \)
Proof: Line 32 in Algorithm 1.

Fact 45: \( \text{MaxTried}[m] \neq \text{none} \)
**Fact 46:** \( \text{maxTried} \neq \text{none} \)

*Proof:* Line 33 in Algorithm 1.

**Fact 47:** \( h\text{MaxTried}_{\text{coord}(m)}[m] = \text{maxTried} \)

**Fact 48:** \( \text{maxStart}_i = m \)

*Proof:* Line 34 in Algorithm 1.

**Fact 49:** \( i = \text{coord}(m) \)

*Proof:* Since \( \text{maxTried} \) is a c-struct, \( i \) executed \( \text{phase2Start}(m,\_\_\_) \) previously. Hence \( i \) is the coordinator of ballot \( m \).

**QED.**

**Fact 50:** \( \text{MaxTried}[m] \overset{\Delta}{=} h\text{MaxTried}_{\text{coord}(m)}[m] \)

**QED.**

**Fact 51:** \( C \in \text{PrpCmd} \)

*Proof:* Line 35 in Algorithm 1.

**Fact 52:** \( \neg \text{isFast}(m) \)

*Proof:* Line 36 in Algorithm 1.

**Fact 53:** \( \text{MaxTried}[m'] = \text{MaxTried}[m] \cdot C \)

**Fact 54:** \( \text{MaxTried}[m] = h\text{MaxTried}_{\text{coord}(m)}[m] \)

**Fact 55:** \( \text{MaxTried}[m'] = h\text{MaxTried}_{\text{coord}(m)}[m'] \)

**Fact 56:** \( \text{maxStart}_i' = \text{maxStart}_i \)

**Fact 57:** \( \text{maxStart}_i = m \)

**Fact 58:** \( \text{maxTried}_i' = \text{maxTried}_i \cdot C \)

*Proof:* Line 37 in Algorithm 1.

**QED.**

**Unchanged:** Proposed, Learned, MinTried and \( \beta \)

**QED.**

**Assume:** \( (s,s') = \text{phase2BClassic}_i(m,u) \)

**Prove:** \( \tilde{\gamma}(s),\tilde{\gamma}(s') = \text{ClassicVote}(u,i) \)

**Fact 59:** \( i \in \text{Acceptors} \)

*Proof:* Line 40 in Algorithm 1.

**Fact 60:** \( \tilde{\beta}_i = m \)

**Fact 61:** \( \text{bal}_i = m \)

*Proof:* Line 42 in Algorithm 1.

**Fact 62:** \( \text{bal}_i \overset{\Delta}{=} \tilde{\beta}_i \)

**QED.**

**Fact 63:** \( \text{MinTried}[\tilde{\beta}_i] \subset u \subset \text{MaxTried}[\tilde{\beta}_i] \)

**Fact 64:** \( h\text{MinTried}_{\text{coord}(m)}[m] \subset u \subset h\text{MaxTried}_{\text{coord}(m)}[m] \)

**Fact 65:** \( i \) received a message \( (i, (2A, u), \text{coord}(m)) \)

*Proof:* Line 41 in Algorithm 1.
QED.
Proof: Invariant MSG4.
Fact 66: $h\text{MinTried\_coord}(m)[m] \triangleq \text{MinTried}[m]$
Fact 67: $h\text{MaxTried\_coord}(m)[m] \triangleq \text{MaxTried}[m]$
QED.

Proof: Fact 60.

Fact 68: $\exists W \in w\text{quorum}(m), a \in W$
Proof: Line 43 in Algorithm 1.

Fact 69: $\beta_i[\hat{\beta}_i] = \text{none} \lor \beta_i[\hat{\beta}_i] \sqsubseteq u$

Fact 70: $\beta_i[\hat{\beta}_i] \triangleq \text{hcval}_i[\hat{\beta}_i]$
Case: $bal_i \neq cbal_i$
Fact 71: $bal_i > cbal_i$
Proof: Invariant PAX1 and since $bal_i \neq cbal_i$.
Fact 72: $\text{hcval}_i[bal_i] = \text{none}$
Proof: Invariant HST3 and fact 70.
Fact 73: $\text{hcval}_i[\hat{\beta}_i] = \text{none}$
Proof: $\beta_i \triangleq bal_i$ and fact 71.
QED.
Proof: Facts 69 and 72.

Case: $bal_i = cbal_i$
Fact 74: $\text{cval}_i \sqsubseteq u$
Proof: Line 44 in Algorithm 1.
Fact 75: $\text{hcval}_i[cbal_i] = \text{cval}_i$
Proof: $bal_i = cbal_i$ and invariant HST2.
Fact 76: $bal_i \triangleq \hat{\beta}_i$
QED.

Fact 77: $\beta_i[\hat{\beta}_i]' = u$
Fact 78: $\beta_i[\hat{\beta}_i]' \triangleq \text{hcval}_i[\hat{\beta}_i]'$
Fact 79: $\text{hcval}_i[cbal_i]' = u$
Proof: Section A.3.1.
Fact 80: $\hat{\beta}_i' = cbal_i'$
Fact 81: $\hat{\beta}_i' \triangleq bal_i'$
Fact 82: $bal_i' = cbal_i'$
Proof: Lines 42 and 47 in Algorithm 1.
QED.

QED.

Fact 83: $\forall m \in \text{BalNum}, \beta_i[m]' = \beta_i[m]$
Fact 84: $\forall a \in \text{Acceptors}, a \neq i \Rightarrow (\hat{\beta}_a' = \hat{\beta}_a \land (\forall m \in \text{BalNum}, \beta_a[m]' = \beta_a[m]))$

Unchanged: Proposed, Learned, MinTried, MaxTried
QED.

Assume: \((s, s') = \text{phase2BFast}_u(m)i\)

Prove: \((\hat{s}(s), \hat{s}(s')) = \text{FastVote}(u, i)\)

Fact 85: \(i \in \text{Acceptors}\)
Proof: Line 50 in Algorithm 1.

Fact 86: \(C \in \text{PrpCmd}\)
Proof: Line 53 in Algorithm 1.

Fact 87: \(\beta_i[\hat{\beta}_i] \neq \text{none}\)

Fact 88: \(cval_i \neq \text{none}\)
Proof: Invariant PAX2.

Fact 89: \(hcval_i[\hat{bal}_i] = cval_i\)
Fact 90: \(hcval_i[cbal_i] = cval_i\)
Proof: Invariant HST2.

Fact 91: \(bal_i = cbal_i\)
Proof: Line 52 in Algorithm 1.

QED.

Fact 92: \(\beta_i[\hat{\beta}_i] = \text{isFast} (\hat{\beta}_i)\)
Proof: Line 51 in Algorithm 1.

Fact 93: \(\beta_i[\hat{\beta}_i] = hcval_i[\hat{\beta}_i]\)

Fact 94: \(\hat{\beta}_i \hat{s} \hat{bal}_i\)
QED.

Fact 95: \(\beta_i[\hat{\beta}_i]' = \beta_i[\hat{\beta}_i] \cdot C\)

Fact 96: \(\beta_i[\hat{\beta}_i]' = \beta_i[\hat{\beta}_i] \cdot C\)

Fact 97: \(\beta_i[\hat{\beta}_i] = cval_i\)

Fact 98: \(\beta_i[\hat{\beta}_i] \hat{s} hcval_i[\hat{\beta}_i]\)
Fact 99: \(\hat{\beta}_i = cbal_i\)
Proof: Since \(\hat{\beta}_i \hat{s} bal_i\) and line 52 in Algorithm 1 tells us that \(bal_i = cbal_i\).

Fact 100: \(hcval_i[\hat{bal}_i] = cval_i\)
Proof: From line 52 in Algorithm 1 we know that \(bal_i = cbal_i\). Invariant HST2 concludes.

QED.

Fact 101: \(\beta_i[\hat{\beta}_i]' = cval_i'\)

Fact 102: \(\beta_i[\hat{\beta}_i]' \hat{s} hcval_i[\hat{\beta}_i]'\)
Fact 103: \(\hat{\beta}_i = cbal_i'\)
Proof: Fact 99 and \(cbal_i' = cbal_i\).

Fact 104: \(hcval_i[\hat{bal}_i]' = cval_i'\)
Proof: \(bal_i' = cbal_i'\) and invariant HST2.
Fact 105: $cval_i' = cval_i \bullet C$

Proof: Line 54 in Algorithm 1.

QED.

Fact 106: $\forall m \in BalNum, \beta_i[m]' = \beta_i[m]$

Fact 107: $\forall a \in Acceptors, a \neq i \Rightarrow (\hat{\beta}_a' = \hat{\beta}_a \land (\forall m \in BalNum, \beta_a[m]' = \beta_a[m]))$

Unchanged: Proposed, Learned, MinTried and MaxTried

QED.

Assume: $(s, s') = recover_i(u)$

Prove: $(\mathfrak{g}(s), \mathfrak{g}(s')) = \text{Recover}(u,i)$

Fact 108: $i \in Acceptors$

Proof: Line 3 in Algorithm 2.

Fact 109: $\hat{\beta}_i = cbal_i$

Fact 110: $\hat{\beta}_i \overset{\Delta}{=} bal_i$

Fact 111: $bal_i = cbal_i$

Proof: Line 7 in Algorithm 2.

QED.

Fact 112: $\text{isFast}(\hat{\beta}_i++)$

Fact 113: $\text{isFast}(cbal_i++)$

Proof: Line 5 in Algorithm 2.

Fact 114: $\hat{\beta}_i = cbal_i$


QED.

Fact 115: $\beta_i[\hat{\beta}_i] \neq \text{none}$

Fact 116: $\beta_i[\hat{\beta}_i] = cval_i$

Fact 117: $\beta_i[\hat{\beta}_i] \overset{\Delta}{=} hcval_i[\hat{\beta}_i]$

Fact 118: $hcval_i[cbal_i] = cval_i$

Fact 119: $\hat{\beta}_i = cbal_i$


QED.

Fact 120: $cval_i \neq \text{none}$

Proof: Invariant PAX2.

QED.

Fact 121: $\beta_{\text{coord}(\hat{\beta}_i)}[\hat{\beta}_i] \neq \text{none}$

Fact 122: $\beta_{\text{coord}(\hat{\beta}_i)}[\hat{\beta}_i] = \beta_{\text{coord}(cbal_i)}[cbal_i]$


Fact 123: $hcval_{\text{coord}(i)}[cbal_i] \neq \text{none}$

Proof: Invariant HST5.
Fact 124: $i$ received a message $(i,(2B,cbal_i,u),coord(cbal_i))$
Proof: Line 8 in Algorithm 2.

Fact 125: $u \sqsubseteq hcval_{coord(i)}[cbal_i]$
Proof: Invariant HIST5.

Fact 126: $u \neq none$
Proof: Invariant HST5.

QED.

Fact 127: $u \sqsubseteq \beta_{coord(\hat{\beta})}[\hat{\beta}]

Fact 128: $\beta_{coord(\hat{\beta})}[\hat{\beta}] = \beta_{coord(cbal_i)}[cbal_i]$

Fact 129: $i$ received a message $(i,(2B,cbal_i,u),coord(cbal_i))$
Proof: Line 8 in Algorithm 2.

Fact 130: $u \sqsubseteq hcval_{coord(i)}[cbal_i]$
Proof: Invariant HIST5.

QED.

Fact 131: $\hat{\beta}_i' = \hat{\beta}_i ++$

Fact 132: $\hat{\beta}_i \triangleright eq bal_i$

Fact 133: $\hat{\beta}_i' \triangleright eq bal_i'$

Fact 134: $bal_i' = bal_i ++$

QED.

Let: $U = \{v \in CStruct : v \sqsubseteq \beta_i[\hat{\beta}_i] \wedge \{u,v\} \text{ compatible}\}$

Fact 135: $\beta_i[\hat{\beta}_i]' = \sqcup \{u, \sqcup U\}$

Fact 136: $U = \{v \in CStruct : v \sqsubseteq cval_i \wedge \{u,v\} \text{ compatible}\}$

Fact 137: $cval_i' = \sqcup \{u, \sqcup U\}$
Proof: Lines 10 and 13 in Algorithm 2.

Fact 138: $\beta_i[\hat{\beta}_i]' = cval_i'$

Fact 139: $\beta_i[\hat{\beta}_i]' \triangleright eq hcval_i[\hat{\beta}_i]'$

Fact 140: $\hat{\beta}_i' = cbal_i'$
Proof: Facts 130 and line 12 in Algorithm 2.

Fact 141: $hcval_i[cbal_i]' = cval_i'$
Proof: Definition of $hcval_i$.

QED.

Fact 142: $\forall a \in Acceptors, a \neq i \Rightarrow (\hat{\beta}_a' = \hat{\beta}_a \wedge (\forall m \in BalNum, \beta_a[m]' = \beta_a[m]))$

Unchanged: Proposed, Learned, MinTried and MaxTried

QED.

Assume: $(s,s') = learn_i(m,W,u)$
Prove: $\mathcal{F}(s), \mathcal{F}(s') = \text{Learn}(m, u, i)$

Fact 143: $i \in \text{Learners}$
Proof: Line 57 in Algorithm 1.

Fact 144: $u \in \text{chosen}(m, \beta)$

Fact 145: $W \in wquorum(m)$
Proof: Line 58 in Algorithm 1.

Fact 146: $\forall a \in W, u \subseteq \beta_a[m]$  
Let: $a$ in $W$  
Let: $v$ a $c$-struct s.t. $i$ received a message $(2B, m, v)$ from $a$ and $u \subseteq v$.
Proof: Line 59 in Algorithm 1.

Fact 147: $v \subseteq hcval_a[m]$  
Proof: Invariant MSG5.

Fact 148: $\beta_a[m] \not\subseteq hcval_a[m]$  
QED.

QED.

Fact 149: $\text{Learned}[i]' = \text{Learned}[i] \sqcup u$

Fact 150: $\text{learned}' = \text{learned} \sqcup u$
Proof: Line 60 in Algorithm 1.

Fact 151: $\text{Learned}[i] \not\supseteq \text{learned}_i$.
QED.

Unchanged: $\text{Proposed}$, $\beta$, $\text{MinTried}$ and $\text{MaxTried}$.

QED.

Both variables $\text{Proposed}$ and $\text{Learned}$ are defined by the refinement mapping $\mathcal{F}$. Hence, $\mathcal{F}$ preserves the externally visible state component of $\text{abstractFGGC}$. Propositions 12 and 13 tell us that $\mathcal{F}$ is a refinement mapping from $\text{FGGC}$ to $\text{abstractFGGC}$. Section A.2 depicts $\mathcal{R}$, a refinement mapping from $\text{abstractFGGC}$ to $\text{GC}^*$. We conclude that $\text{FGGC}$ implements $\text{GC}^*$.

### A.4 Liveness

We now prove in the unreliable failure detectors model [7] that $\text{FGGC}$ is live when processes crash and links are reliable.

**Roadmap.** Section A.4.1 presents our assumptions and modifications on $\text{FGGC}$ to make progress in the unreliable failure detectors model. In particular, $\text{FGGC}$ is modified as follows: (i) we use $\Diamond S$ and $\Omega$, (ii) we add action $\text{skipBallot}$ to $\text{FGGC}$, and (iii) we add a precondition to action $\text{phase1A}$. Section A.4.2 contains the heart of our proof. We study liveness in runs during which $\text{FGGC}$ executes a bounded number of ballots, then in runs during which an unbounded number of ballots is executed.
A.4.1 Preliminaries

The model. We consider along this section that processes communicate through reliable links, i.e., if \( i \) executes "send \( msg \) to \( j \)" for some message \( msg \) and some process \( j \), this action adds the triple \( (i, msg, j) \) to \( Msg \). As usual, when arguing about the liveness of \( FGGC \), we assume that \( FGGC \)'s runs are fair, i.e., if an action is enabled infinitely often at some correct process \( i \), \( i \) executes it infinitely often. Moreover, we assume that the asynchronous message-passing distributed system between acceptors is augmented with an eventually strong failure detector \([7]\): \( \diamond S \). Failure detector \( \diamond S \) outputs at each acceptor \( a \) a list of acceptors \( \diamond S_a \) that are suspected to have crashed. This failure detector ensures that during every run:

\[
\forall a \in \text{Acceptors} \cap \text{Correct}, \forall b \in \text{Acceptors} \cap \text{Faulty}, \diamond \Box (b \in \diamond S_a)
\]

\[
\exists a_0 \in \text{Acceptors} \cap \text{Correct}, \forall a \in \text{Acceptors} \cap \text{Correct}, \diamond \Box (a_0 \notin \diamond S_i)
\]

\( FGGC \) in the unreliable failure detectors model makes also use of an eventual leader oracle \([6]\): \( \Omega \). This failure detector outputs at each acceptor \( a \) an eventual correct leader for \( \text{Acceptors} \):

\[
\exists a_0 \in \text{Acceptors} \cap \text{Correct}, \forall a \in \text{Acceptors} \cap \text{Correct}, \diamond \Box (\Omega_a = a_0)
\]

This failure detector is equivalent (in the sense of \([7]\)) to \( \diamond S \), hence augmenting the system with \( \Omega \) to not modify our previous synchrony assumptions. It has been shown that \( \Omega \) is the weakest failure detector to solve consensus \([6]\). Thus, our synchrony assumptions in the unreliable failure detectors model model are minimal.

Addenda to \( FGGC \). The construction below ensures that \( FGGC \) makes progress despite faults:\(^{10}\)

- \( BalNum \) satisfies the definition we gave in Example 3 at Section 4.

- First we add to \( FGGC \) action \( \text{skipBallot} \) (see Algorithm 5). Acceptors execute this action to alert a coordinator that there are not participating to a ballot it coordinates. For instance, when an acceptor \( a \) joins a ballot \( n > m \) not coordinated by \( \text{coord}(m) \), \( a \) sends a \( \text{SKIP} \) message to \( \text{coord}(m) \).

- Second, we add the following preconditions to action \( \text{phase1A}(m) \):\(^{11}\)

\[
\neg \text{isFast}(m)
\]

\[
i = \Omega_i
\]

\[
\left( \exists a \in \text{Acceptors}, \text{received} \ (\text{SKIP}, \max\text{Start}_i) \text{ from } a \right)
\]

\[
\lor \left( \text{isFast}(\max\text{Start}_i) \land \forall W \in w\text{quorum}(\max\text{Start}_i), \exists a \in W \cap \diamond S_i \right)
\]

These preconditions state that a coordinator \( i \) starts only classic ballots. Moreover \( i \) may start ballot \( m \) only if \( i \) is the leader of \( \text{Acceptors} \), and \( i \) suspects that \( \max\text{Start}_i \) do not progress.

\(^{10}\)The construction that follows eases proving liveness, and is sufficient to ensure that \( FGGC \) is both genuine and fast. It is not a guideline for an implementation.

\(^{11}\)Recall that Assumption FGGC1 in Section 4.3 requires that every coordinator is an acceptor. As a consequence, coordinators can access the failure detectors we defined previously.
Algorithm 5 Action \textit{skipBallot} - code at process $i$

\begin{enumerate}
\item \textbf{skipBallot($m$) [acceptor]}
\item \textbf{pre:} \ $c_{bal_i} > m$
\item \textbf{coord}($c_{bal_i}$) $\neq$ coord($m$)
\item \textbf{action:} send (\textit{SKIP},$m$) to coord($m$)
\end{enumerate}

**Started, joined and live ballots.** A ballot is started when at least one acceptor joins it. We note \textit{Started} the balnums of started ballots:

\[ \text{Started} \triangleq \{ m \in \text{BalNum} : \exists a \in \text{Acceptors}, \diamond (c_{bal_a} = m) \} \]

Similarly, a ballot is \textit{joined} when some \textit{m}-wquorum joins it:

\[ \text{Joined} \triangleq \{ m \in \text{BalNum} : \exists W \in \text{wquorum} (m), \forall a \in W, \diamond (c_{bal_a} = m) \} \]

A ballot is live when, as we will see in Proposition 19, it ensures that \textit{FGGC} satisfies the liveness clause of generalized consensus:

\[ \text{live}(m) \triangleq \{ \text{coord}(m) \in \text{Correct} \exists W \in \text{wquorum} (m), \forall a \in W, a \in \text{Correct} \land \Box (c_{bal_a} = m) \} \]

**A.4.2 \textit{FGGC} is live**

Proposition 14 bellow follows from the notions we introduced in the previous section. A rigorous proof is left to the reader.

**Proposition 14.** If Started is bounded, then for every coordinator $i$ there exists a ballot $m$ such that $\maxStart_i = m$ holds forever.

**Proposition 15.** Consider a coordinator $c$ such that $c$ satisfies Equation 6, a ballot $m$ such that eventually $\maxStart_c = m$ holds forever. Coordinator $c$ never receives a message (\textit{SKIP},$m$).

**Proof**

Let: $c$ s.t. $c$ satisfies Equation 6.

Let: $m$ a balnum s.t. $m$ satisfies $\diamond \Box (\maxStart_c = m)$.

Let: $n \in \text{BalNum}$ s.t. $(\text{coord}(n) = c \land n > m \land \neg \text{isFast}(n))$

Proof: Assumption Bal3.

Assume: $\exists a \in \text{Acceptors}, \diamond (c \text{ received } \text{ (SKIP},m \text{) from } a)$

Fact 1: $c$ executes \textit{phase1A$(n)$}.

QED.

Proof: Action \textit{phase1A$(n)$} is eventually always enabled at $c$ because (i) $c$ satisfies Equation 6 (ii) $c$ receives a message (\textit{SKIP},$m$), (iii) $n$ is classic and coordinated by $c$, and (iv) $n > m$ holds. The fact that $c$ is correct concludes.
Contradiction

Proof: Since \( n > m \) and \( c \) executes phase1A\((n)\), \( \diamond \square (\maxStart_c > m) \) holds. This contradicts the definition of \( m \).

\[ \square \]

**Proposition 16.** Consider a coordinator \( c \) such that \( c \) satisfies Equation 6, and a ballot \( m \) such that eventually \( \maxStart_c = m \) holds forever. If \( m \) is classic, then \( m \) is live.

**Proof**

Let: \( c \) a coordinator s.t. \( c \) satisfies Equation 6.

Let: \( m \) a classic ballnum s.t. \( m \) satisfies \( \diamond \square (\maxStart_c = m) \).

Let: \( W \in wquorum(m) \) s.t. \( W \subseteq Correct \).

Proof: Since there are \( 2f + 1 \) acceptors, there exists a correct majority set. The fact that \( m \) is classic and Assumption Bal2 concludes.

**Fact 1:** \( \forall a \in W, \diamond \square (cbal_a = m) \)

Let: \( a \) s.t. \( \square \diamond (cbal_a \neq m) \)

**Fact 2:** \( \diamond (a \text{ received } (1A,m) \text{ from } c) \)

Proof: Since initially \( \maxStart_c = 0 \) and ballot 0 is fast, \( c \) executes phase1A\((m)\). Thus \( c \) executes line 10 in Algorithm 1. Because \( a \) is a correct acceptor and the link from \( c \) to \( a \) is reliable, \( a \) eventually receives a message \( (1A,m) \) from \( c \).

**Fact 3:** \( \diamond \square (cval_a > m \land \text{coord}(cval_a) \neq c) \)

**Fact 4:** \( \diamond \square (cval_a > m) \)

Proof: \( \diamond (cval_a \neq m) \), Fact 2 and Invariant PAX1.

**Fact 5:** \( \diamond \square (\neg \text{isFast}(cval_a)) \)

Proof: From the definition of BalNum, every classic ballot is higher than every fast ballot. Fact 2, the precondition of action phase1B concludes and Invariant PAX1 conclude.

QED.

Proof: \( \diamond \square (\maxStart_c = m) \), Fact 4 and Fact 5.

**Fact 6:** \( a \) executes \( \text{skipBallot}(m) \)

Proof: Fact 3 implies that \( \text{skipBallot}(m) \) is always enabled at acceptor \( a \). Because \( a \) is correct, it eventually executes this action.

**Fact 7:** \( \diamond (c \text{ received } (\text{SKIP},m) \text{ from } a) \)

Proof: Fact 6 tells us that acceptor \( a \) executes Line 4 in Algorithm 5. The facts that \( c \) is correct and the link from \( a \) to \( c \) is reliable conclude.

Contradiction

Proof: Proposition 15.

QED.
Proposition 17. Consider a coordinator $c$ such that $c$ satisfies Equation 6, and a ballot $m$ such that eventually $\maxStart_c = m$ holds forever. If $m$ is fast and Started is bounded, then there exists a live ballot.

Proof

Let: $c$ a coordinator s.t. $c$ satisfies Equation 6.

Let: $m$ a fast balnum s.t. $m$ satisfies $\Diamond \Box (\maxStart_c = m)$.

Fact 1: $\exists W \in wquorum(m), W \subseteq Correct$.

Assume: $\exists a \in W, a \in \text{Faulty}$

Let: $a \in W \cap \text{Faulty}$

Let: $n \in \text{BalNum}$ s.t. $n > m \land \text{coord}(n) = c \land \neg \text{isFast}(n)$

Proof: Assumption Bal3.

Fact 2: $c$ executes $\text{phase1A}(n)$.

Fact 3: $\Diamond \Box (a \in \Diamond S_c)$

Proof: Equation 5 and $a \in \text{Faulty}$.

QED.

Proof: Action $\text{phase1A}(n)$ is eventually always enabled at $c$ because (i) $c$ satisfies Equation 6 (ii) $m$ is fast (iii) Fact 3, and (iv) $n > m$ holds. Since $c$ is correct, it eventually executes this action.

Contradiction

Proof: Since $n > m$ and $c$ executes $\text{phase1A}(n)$, $\Diamond \Box (\maxStart_c > m)$ holds. This contradicts the definition of $m$.

Let: $W \in wquorum(m)$ s.t. $W \subseteq Correct$.

Proof: Fact 1.

Let: $a \in W$

Let: $n \in \text{BalNum}, \Diamond \Box (c\text{bal}_a = n)$

Proof: Started is bounded

Fact 4: $\text{coord}(n) = c$

Assume: $\text{coord}(m) \neq c$

Fact 5: $m < n$

Proof: $m$ is fast, thus if $\text{coord}(n) \neq c$, $n$ is classic. The result follows from the definition of $\text{BalNum}$.

Fact 6: $a$ executes $\text{skipBallot}(m)$

Proof: Fact 5 implies that $\text{skipBallot}(m)$ is always enabled at acceptor $a$. Because $a$ is correct, it eventually executes this action.
Fact 7: $\Diamond(c \text{ received } (\text{SKIP},m) \text{ from } a)$

Proof: Fact 6 tells us that acceptor $a$ executes Line 4 in Algorithm 5. The facts that $c$ is correct and the link from $a$ to $c$ is reliable conclude.

Contradiction Proof:

Proof: Proposition 15.

Fact 8: $\text{isFast}(n)$

Fact 9: $c$ never executes $\text{phase1A}(m')$ for some classic ballot $m'$.

Proof: From the definition of $\text{BalNum}$, every classic ballot is higher than every fast ballot. Since $\Diamond\Box(\text{maxStart}_c = m)$ holds, precondition at Line 6 in Algorithm 1 implies that $c$ never starts a classic ballot.

QED.

Proof: Fact 9 implies that acceptor $a$ joins ballot $n$ by executing $\text{recover}$. The precondition at Line 5 and the assignment at Line 11 in Algorithm 2 conclude.

Fact 10: $\forall b \in W, \Diamond\Box(c_{bal}_b = n)$

Let: $b$ an acceptor in $W$
Let: $n'$ a balnum s.t. $\Diamond\Box(c_{bal}_b = n')$
Case: $n' > n$

Fact 11: $\forall b' \in W, b \text{ received } (2B,n'--\Box) \text{ from } b'$

Proof: Fact 8 implies that $n'$ is fast. Since a coordinator cannot start a fast ballot, $b$ joins ballot $n'$ by executing $\text{recover}$. Line 4 in Algorithm 2 and the definition of $\text{collide}(m)$ concludes.

Fact 12: $\Diamond(c_{bal}_a = n'--)$

Proof: Fact 11.

Fact 13: $n = n'--$

Fact 14: $n'-- \geq n$

Proof: From $n' > n$.

Fact 15: $n'-- \leq n$

Proof: From the conjunction of $\Diamond(c_{bal}_a = n'--)$, $\Diamond\Box(c_{bal}_a = n)$ and Invariant PAX1.

QED.

Fact 16: $\Diamond(c_{bal}_a = n')$

Fact 17: $\Diamond\Box(c_{bal}_a = n'--)$

Fact 18: $\text{collide}(n'--)$ is eventually true at acceptor $a$.

Proof: Since all the acceptors in $W$ are correct, eventually $a$ receives all the $2B$ messages received by $b$ at ballot $n'--$.

Fact 19: $W \in \text{wquorum}(n')$

Proof: Definition of $\text{BalNum}$.

QED.

Proof: Due to Facts 17 to 19, $a$ eventually executes $\text{recover}$ at ballot $n'--$ and joins ballot $n'$.
Contradiction
Case: \( n' < n \)
Contradiction

Proof: This case is symmetric to the previous case, and thus omitted.

QED.

Since there are \( f + 1 \) acceptors, there exists a coordinator that satisfies Equation 6. We deduce from propositions 16 and 17 that:

**Proposition 18.** If \( \text{Started} \) is bounded, then there exists a live ballot.

The existence of a live ballot ensures that \( \text{FGGC} \) makes progress. We prove this fact below:

**Proposition 19.** If there exists a live ballot, then \( \text{FGGC} \) satisfies the liveness clause of generalized consensus.

**Proof**

Let: \( m \in \text{BalNum} \) s.t. \( m \) is live.

Let: \( c = \text{coord}(m) \)

Let: \( W \) a \( m \)-wquorum s.t. \( \forall a \in W, a \in \text{Correct} \wedge \Diamond (\text{cbal}_a = m) \)

Proof: \( m \) is live.

Let: \( C \) a command proposed by a correct proposer \( p \) or learned by some learner \( l' \).

Let: \( l \) a correct learner.

Prove: \( \Diamond (C \in \text{learned}_l) \)

Case: \( C \) proposed by \( p \)

Fact 1: \( \forall a \in W, \Diamond (a \text{ received } (\text{propose}, C)) \text{ from } p \)

Fact 2: \( \forall a \in W, \exists u \in \text{CStruct}, C \in u \wedge \Diamond (l \text{ received } (2B, m, u) \text{ from } a) \)

Case: \( \text{isFast}(m) \)

Fact 3: for every acceptor \( a \in W, a \) executes \( \text{phase2BFast}(C) \) with \( \text{cbal}_a = m \)

QED.

Case: \( \neg \text{isFast}(m) \)

Fact 4: \( \text{phase2AClassic}_{\text{coord}(m)}(m, C) \)

Fact 5: \( \exists u \in \text{CStruct}, C \in u \wedge \forall a \in W, \text{phase2BClassic}_a(m, u) \)

QED.

Let: for \( a \in W \) define \( u_a \in \text{CStruct} \) s.t. \( C \in u_a \wedge \Diamond (l \text{ received } (2B, m, u_a) \text{ from } a) \).

Fact 6: \( C \in \bigcap \{u_a : a \in W\} \)
Fact 7: \( \{u_a : a \in W \} \) compatible

QED.

Proof: Assumption CS4

Fact 8: \( l \) eventually executes \( \text{learn}(m,W,\cap \{u_a : a \in W \}) \).

QED.

Case: \( C \) learned by some learner \( l' \)

Let: \( n \) the ballot at which \( l' \) learns \( C \), i.e., it executes \( \text{learn}(u,m,\bot) \) with \( C \in u \) for the first time.

Case: \( n < m \)

QED. Consider an acceptor \( a \) in \( W \). Because \( hcvla[m] \) is always safe. \( cvla \) contains \( C \) at ballot \( m \). Since ballot \( m \) never collides, \( l \) learns \( C \) at balllot \( m \).

Case: \( n = m \)

QED. If \( m \) is classic, then \( \text{coord}(m) \) suggested \( C \) during \( m \). Because \( \text{coord}(m) \) is correct, every acceptor in \( W \) eventually accepts command \( C \). Hence eventually \( C \) is learned by \( l \). If now \( m \) is fast, because all acceptors in \( W \) are correct and ballot \( m \) never collides, \( l \) eventually learns \( C \).

\( \Box \)

It remains to show that if an unbounded number of ballots are started, \( \text{FGGC} \) is still live. To this goal, we first prove that in such a case, a finite number of classic ballot is started: Proposition 20. It follows that an unbounded number of fast ballots are started: Corollary A.4.2. From the definition of \( \text{BalNum} \) we deduce in Proposition 21 that \( \text{Joined} \) equals exactly \( \{(0,i) : i \in \mathbb{N}\} \). We then show that if \( \text{Started} \) is unbounded, then all the ballots in \( \text{Joined} \) collide: Proposition 22. Proposition 23 concludes.

A rigorous proof of propositions 20 to 22 is left to the reader.

**Proposition 20.** A bounded number of classic ballots is started.

**Corollary.** If \( \text{Started} \) is unbounded, then an unbounded number of fast ballots is started.

**Proposition 21.** If \( \text{Started} \) is unbounded, then \( \text{Joined} \) equals exactly the balnums \( \{(0,i) : i \in \mathbb{N}\} \).

**Proposition 22.** If \( \text{Started} \) is unbounded, then all the ballot in \( \text{Joined} \) collide.

**Proposition 23.** If \( \text{Started} \) is unbounded, \( \text{FGGC} \) is live.

**Proof**

Let: \( W \) a \( m \)-wqorum s.t. \( \forall m \in \text{Joined}, \{W\} = \text{wqorum}(m) \).

Proof: Proposition 21 and the definition of \( \text{BalNum} \).

Let: \( c \) a coordinator s.t. \( \forall m \in \text{Joined}, \text{coord}(m) = m \)

Proof: Proposition 21 and the definition of \( \text{BalNum} \).

Let: \( C \) a command proposed by a correct proposer \( p \) or learned by some learner \( l' \).
Let: \( l \) a correct learner.

Prove: \( \Diamond \Box (C \in \text{learn}_l) \)

Case: \( C \) proposed by \( p \)

Fact 1: \( \forall a \in W, \Diamond (a \text{ received } (\text{propose},C) \text{ from } p) \)

Fact 2: for every acceptor \( a \in W, a \text{ executes } \text{phase2BFast}(C) \)

Let: \( m \) a ballot s.t. when \( c \) executes \( \text{phase2BFast}(C) \), \( cbal_c \) equals \( m \).

Fact 3: \( \exists u \in \text{CStruct}, C \in u \land \forall a \in W, \Box (cbal_a = n \Rightarrow u \subseteq cval_a) \)

Let: \( u \) a c-struct s.t. \( \forall a \in W, \Box (cbal_a = n \Rightarrow u \subseteq cval_a) \)

Fact 4: \( l \) eventually executes \( \text{learn}(n,W,u) \).

QED.

Case: \( C \) learned by some learner \( l' \)

Let: \( n \) the ballot at which \( l' \) learns \( C \), i.e., it executes \( \text{learn}(u,m,_) \) with \( C \in u \) for the first time.

QED. Trivial since \( W \) is the single \( m \)-wquorum and all the acceptors in \( W \) are correct.

\( \square \)

A.5 On the latency of \( \text{FGGC} \)

In this section we prove that \( \text{FGGC} \) satisfies the two performance criteria we defined in Section 2, i.e., fastness and genuineness. To this goal, we consider the unreliable failure detectors model, and the variant of \( \text{FGGC} \) we depicted in Section A.4. First, we introduce the latency degree [30]. According to this metrics, when the problem solved is consensus, i.e., \( \text{CStruct} \) satisfies Equation 2, \( \text{FGGC} \) is optimal. Then, we introduce a novel metrics: the delivery degree. The delivery degree extends the latency degree to generalized consensus. It measures the number of communication steps to learn commands during the worst nice run. \( \text{FGGC} \) is both fast and genuine according to this metrics.

A.5.1 The latency degree

The latency degree [30] is the usual metrics to measure the latency of a consensus algorithm. For some consensus algorithm \( \mathcal{A} \), the latency degree of \( \mathcal{A} \) is the number of message delays for all processes to decide during \( \mathcal{A} \)'s best run. We give a more formal treatment below:

- Given a consensus algorithm \( \mathcal{A} \), and a run \( r \) of \( \mathcal{A} \), an event \( e \) in \( r \) is the occurrence of an action of \( \mathcal{A} \) during \( r \). We note \( r_{|e} \) the prefix of \( r \) up to event \( e \) (excluded).
- Given a run \( r \), we note \( e \prec_r f \) when event \( e \) happens-before event \( f \) in \( r \) [19], that is when \( e \prec_r f \) and either \( e \) and \( f \) occur on the same process, or there exists some message \( m \) such that \( e \) contains an operation \( \text{send}(m) \), and \( f \) contains an operation \( \text{recv}(m) \).
A path \( \rho \) (from event \( e_1 \) to event \( e_n \)) in a run \( r \), is a sequence \( \langle e_1, \ldots, e_n \rangle \) such that for every \( i \) in \([1, n]\), \( e_i \prec r e_{i+1} \). Path \( \rho \) is causal if in addition for every \( i \) in \([1, n]\), there is no event \( e \) such that \( e_i \prec r e \prec r e_{i+1} \).

- The length of a path \( \rho \): \( \text{len}(\rho) \), is the number of pairs of events \((e, f)\) in \( \rho \) for which there exists some message \( m \) such that \( e \) contains an operation \( \text{send}(m) \), and \( f \) contains an operation \( \text{recv}(m) \).

- The latency of \( A \) during \( r \): \( \text{lat}(r) \), is the length of the longest causal path in \( r \) from some event \( \text{propose}() \) to some event \( \text{decide}() \).

- The latency degree of \( A \) is the latency of \( A \) during its best run:

\[
\Delta_A \triangleq \min \{ \text{lat}(r) : r \in \text{runs}(A) \}
\]

where \( \text{runs}(A) \) denotes the runs of \( A \).

Proposition 24 below proves that the latency degree of \( \text{FGGC} \) equals 2. As a corollary of the hyperfast learning lemma of [23], \( \text{FGGC} \) is optimal.

**Proposition 24.** If \( \text{CStruct} \) follows definition 2, then \( \Delta_{\text{FGGC}} = 2 \).

**Proof** Let \( r \) be a run of \( \text{FGGC} \) during which all acceptors receive \( \text{propose} \) messages in the same order. Name \( C \) the first command received, and assume in addition that when an acceptor receives \( C \), it executes \( \text{phase2BFast}(C) \) (this is possible because ballot 0 is fast). It follows that every acceptor accepts at ballot 0 \( \text{cstruct} u = \bot \cdot C \) As a consequence, every correct learner \( l \) receives a \( 2B \) message containing \( u \) at ballot 0. Hence \( l \) learns \( C \) at ballot 0. The length of a causal path between events \( \text{propose}(C) \) and \( \text{learn}_l(u \in C) \) with \( C \in u \) equals two: it contains exactly messages \((\text{propose}, C)\) and \((2B, 0, u)\). Hence \( \Delta_{\text{FGGC}} = 2 \).

\( \square \)

The latency degree takes only into account the best run. Considering the best run is of interest because it removes side-effects (e.g. asynchronism) that may prevent algorithms to progress for some arbitrary long period of time, and are not relevant in average. However, the best run may not reflect accurately average latency. For instance, both Fast Paxos [22] and \( \text{FGGC} \) leverage the spontaneous ordering of commands by the network to solve consensus in two steps. Such a situation is nevertheless rare in a wide-area network.

A more accurate figure is given by the maximum number of communication steps during all nice runs. However, such a metrics does not work for tasks having multiple inputs. To illustrate this claim, consider Figure 1 depicting a run of a broadcast primitive. Because the reception of \( m_1 \) is unrelated with the reception of \( m_2, m_3 \) and \( m_4 \), the latency of \( m_1 \) in this run should equal 1: this is the path in blue from event \( \text{send}(m_1) \) to event \( \text{recv}(m_1) \). Nevertheless, because we measure the length of the longest causal path between these two events, the latency of \( m_1 \) equals 3: path in red.

In the next section we introduce a metrics to report the latency of generalized consensus. We then formally define genuineness and fastness, and prove that \( \text{FGGC} \) matches these two performance criteria.
A.5.2 The delivery degree

Given a generalized consensus algorithm $\mathcal{A}$, we define the delivery degree of $\mathcal{A}$ as follows:

- Consider a run $r$ of $\mathcal{A}$, and two events $e$ and $f$ in $r$ such that $e \prec f$. We say that $f$ is necessary to $e$ in $r$ if $r_e \circ f$ is not a run of $\mathcal{A}$.
- A path $\rho = \langle e_1, \ldots, e_n \rangle$ is accurate if for every $i$ in $\llbracket 1, n \rrbracket$, event $e_i$ is necessary to event $e_{i+1}$.
- The latency of $\mathcal{A}$ during a run $r$: $lat(r)$, is the length of the longest accurate path from some event $propose_p(C)$ to some event $learn_l(u)$ with $C \in u$, $p \in Proposers$ and $l \in Learners$.
- The delivery degree of $\mathcal{A}$ during the set of runs $\mathcal{R}$ is the highest latency of $\mathcal{A}$ during all runs in $\mathcal{R}$.

$$\Lambda_\mathcal{R}^\mathcal{A} \triangleq \max \{ lat(r) : r \in \mathcal{R} \}$$

Our metrics takes into account events that are necessary to learn a command $C$ and prunes others. This gives a precise number of the communication steps to learn a command $C$ in a nominal case. For instance in Figure 1, the delivery time of $m_1$ equals 1 because events $send(m_2)$, $recv(m_2)$, $send(m_3)$, ..., are contingent to the reception of $m_1$. Another motivating example is the implementation of state machine replication on top of consensus. Suppose that consensus is solved in three steps, and consider a run during which a command $C$ is proposed at an instance $i$ and decided at an instance $j > i$. If we consider the longest causal path, then $C$ is decided in $3 \times (i - j)$ steps. Our metrics does not take into account instances of consensus prior to $j$ because none of them is necessary to decide $C$.

A.5.3 FGGC is fast and genuine

Given a generalized consensus algorithm $\mathcal{A}$, we denote $so(\mathcal{A})$ the nice runs of $\mathcal{A}$ during which all processes receive messages addressed to them in the same order. A generalized consensus algorithm $\mathcal{A}$ is genuine if $\Lambda_\mathcal{A}^{nice(\mathcal{A})} = 2$ when $CStruct$ is compatible. Algorithm $\mathcal{A}$ is fast when $\Lambda_\mathcal{A}^{nice(\mathcal{A})} = 3$ and $\Lambda_\mathcal{A}^{so(\mathcal{A})} = 2$., We prove below that FGGC matches these two criteria.

Proposition 25. During a nice run, no classic ballot is started.

Proof

During a nice run, there is no fault and all failure detectors behave perfectly. As a consequence, failure detector $\Diamond S$ always returns $\{\}$ at all processes. This implies that no coordinator executes
action $\text{phase1A}$: precondition $\text{isFast}(\text{maxStart}_i) \land \forall W \in \text{wquorum}(\text{maxStart}_i), \exists a \in W \cap \diamond S_i$ is always false. Since initially $\text{cbal}_a$ equals 0 for every acceptor $a$, ballot 0 is fast and action $\text{recover}$ cannot be executed to join a classic ballot, we conclude that no classic ballot is started.

\[ \square \]

**Corollary.** During a nice run, there exist a coordinator $c$ and a quorum $W$ such that for every started ballot $m$, $c$ equals $\text{coord}(m)$ and $W$ is the single $m$-wquorum of $m$.

The corollary above follows immediately from proposition 25 and the definition of $\text{BalNum}$. In the sequel we refer to this particular coordinator and this particular quorum using notations $c$ and $W$ respectively.

**Proposition 26.** If $\text{CStruct}$ is compatible or all processes receive messages addressed to them in the same order, then 0 is the unique started ballot.

**Proof**
For every acceptor $a$, $\text{cbal}_a$ equals initially 0. It follows that ballot 0 is started. Assume by contradiction the existence of an acceptor $a$ such that $\text{cbal}_a = m$ with $m > 0$, holds at some point in time. Since no classic ballot is started (proposition 24), $a$ cannot execute action $\text{phase1B}$ to join ballot $m$. Thus it executes action $\text{recover}(\cdot)$. As a consequence ballot $m$---collides. This contradicts the fact that $\text{CStruct}$ is compatible or (by a short induction) that processes receive messages addressed to them in the same order.

\[ \square \]

**Proposition 27.** $\text{FGGC}$ is genuine.

**Proof**
Consider a nice run $r$ of $\text{FGGC}$, and a command $C$ proposed by a proposer $p$ and learned by a learner $l$ during $r$. Since $l$ learns $C$, proposition 26 implies that $l$ receives from every acceptor $a$ in $W$ a message $(2B,0,u_a)$ with $C \in u_a$. Because $a$ executes only the action $\text{phase2BFast}$ during $r$ and ballot 0 does not collide, we observe that $\text{phase2BFast}_a(C)$ is the only action executed by $a$ that is necessary for $l$ to learn $C$. This action only requires that proposer $p$ executes action $\text{propose}(C)$ previously. We conclude that every accurate path from event $\text{propose}_p(C)$ to event $\text{learn}_l(C)$ is of the form $(\text{propose}_p(C),\text{phase2BFast}_a(C),\text{learn}_l(C))$. The length of this path equals two. It follows that the latency of $\text{FGGC}$ equals two during $r$. We conclude that $\text{FGGC}$ is genuine.

\[ \square \]

**Proposition 28.** The delivery degree of $\text{FGGC}$ equals two in runs during which all processes receive messages addressed to them in the same order, i.e., $\lambda_{\text{FGGC}} = 2$.

**Proof**
The proof is identical to the proof of proposition 27 and is thus omitted.
**Proposition 29.** The delivery degree of FGGC equals three in nice runs.

**Proof**
Consider a nice run \( r \) of FGGC, and a command \( C \) proposed by a proposer \( p \) and learned by a learner \( l \) during \( r \). Since \( l \) learns \( C \), there exists a ballot \( m \) such that \( l \) receives from every acceptor \( a \) in \( W \) a message \( (2B,m,u,\epsilon) \) with \( C \in u, \epsilon \). If we assume that every acceptor in \( W \) executes \( \text{phase2BFast}(C) \) at ballot \( m \), then since \( \text{propose}_p(C) \) is the only action which is necessary to this action, we conclude that the latency of FGGC during this run equals two. Otherwise there exists an acceptor \( a \) such that a collision occurs at ballot \( m \) and either (i) \( a \) receives a message \( (2B,m,\epsilon,u) \) with \( C \in u \) from \( c \), or (ii) \( cval_a \) contains \( C \) at ballot \( m \). In the first case, according to the code of action \( \text{recover} \), action \( \text{propose}_p(C) \) is the only action that is necessary to obtain \( C \in cval_c \) at ballot \( m \). In both cases we conclude that the length of the longest accurate path from event \( \text{propose}_p(C) \) to event \( \text{learn}_l(u) \) equals three.

**Proposition 30.** FGGC is fast.

**Proof**
From propositions 28 and 29.