Generalized Paxos Made Byzantine (and Less Complex)

Miguel Pires[1], Srivatsan Ravi[2], and Rodrigo Rodrigues[1]

[1] INESC-ID and Instituto Superior Técnico (U. Lisboa)
miguel.pires@tecnico.ulisboa.pt, rodrigo.rodrigues@inesc-id.pt
[2] University of Southern California
srivatsr@usc.edu

Abstract. One of the most recent members of the *Paxos* family of protocols is *Generalized* Paxos. This variant of Paxos has the characteristic that it departs from the original specification of consensus, allowing for a weaker safety condition where different processes can have different views on a sequence being agreed upon. However, much like the original Paxos counterpart, Generalized Paxos does not have a simple implementation. Furthermore, with the recent practical adoption of Byzantine fault tolerant protocols, it is timely and important to understand how Generalized Paxos can be implemented in the Byzantine model. In this paper, we make two main contributions. First, we provide a description of Generalized Paxos that is easier to understand, based on a simpler specification and the pseudocode for a solution that can be readily implemented. Second, we extend the protocol to the Byzantine fault model.

1 Introduction

One of the fundamental challenges for processes participating in a distributed computation is achieving *consensus*: processes initially propose a value and must *eventually agree* on one of the proposed values [6]. Paxos [10], arguably, is one of the most popular protocols for solving the consensus problem among fault-prone processes. The evolution of the Paxos protocol represents a unique chapter in the history of Computer Science. It was first described in 1989 through a technical report [9], and was only published a decade later [10]. Another long wait took place until the protocol started to be studied in depth and used by researchers in various fields, namely the distributed algorithms [4] and the distributed systems [16] research communities. And finally, another decade later, the protocol made its way to the core of the implementation of the services that are used by millions of people over the Internet, in particular since Paxos-based state machine replication is the key component of Google's Chubby lock service [1], or the open source ZooKeeper project [7], used by Yahoo! among others. Arguably, the complexity of the presentation may have stood in the way of a faster adoption of the protocol, and several attempts have been made at writing more concise explanations of it [11,23].

More recently, several variants of Paxos have been proposed and studied. Two important lines of research can be highlighted in this regard. First, a series of papers hardened the protocol against malicious adversaries by solving consensus in a Byzantine fault model [19,14]. The importance of this line of research is now being confirmed as these protocols are now in widespread use in the context of cryptocurrencies and distributed ledger schemes such as blockchain [21]. Second, many proposals target improving the Paxos protocol by eliminating communication costs [13], including an important evolution of the protocol called Generalized Paxos [12], which has the noteworthy aspect of having lower communication costs by leveraging a more general specification than traditional consensus that can lead to a weaker requirement in terms of ordering of commands across replicas. In particular, instead of forcing all processes to agree on the same value (as with traditional consensus), it allows processes to pick an increasing sequence of commands that differs from process to process in that commutative commands may appear in a different order. The practical importance of such weaker specifications is underlined by significant research activity on the corresponding weaker consistency models for replicated systems [8,5].

In this paper, we draw a parallel between the evolution of the Paxos protocol and the current status of Generalized Paxos. In particular, we argue that, much in the same way that the clarification of the Paxos protocol contributed to its practical adoption, it is also important to simplify the

description of Generalized Paxos. Furthermore, we believe that evolving this protocol to the Byzantine model is an important task, since it will contribute to the understanding and also open the possibility of adopting Generalized Paxos in scenarios such as a blockchain deployment.

Concretely, this paper and the accompanying tech report make the following contributions to the Paxos family:

- We present a simplified version of the specification of Generalized Consensus, which is focused on the most commonly used case of the solutions to this problem, which is to agree on a sequence of commands;
- we extend the Generalized Paxos protocol to the Byzantine model;
- we present a description of the Byzantine Generalized Paxos protocol that is more accessible than the original description, namely including the respective pseudocode, in order to to make it easier to implement;
- we prove the correctness of the Byzantine Generalize Paxos protocol;
- and we discuss several extensions to the protocol in the context of relaxed consistency models and fault tolerance.

The remainder of the paper is organized as follows: Section 2 gives an overview of Paxos and its family of related protocols. Section 3 introduces the model and simplified specification of Generalized Paxos. Section 4 presents the Generalized Paxos protocol that is resilient against Byzantine failures. Section 5 concludes the paper with a discussion of several optimizations and practical considerations. Due to space constraints, this paper doesn't contain a description of the correctness proofs. However, these are included in a separate technical report [22]. Also included in the tech report is an extension to the protocol that implements a checkpointing feature. This extension allows the protocol to deal with the accumulation of state by safely discarding stored commands.

2 Background and related work

2.1 Paxos and its variants

The Paxos protocol family solves consensus by finding an equilibrium in face of the well-known FLP impossibility result [6]. It does this by always guaranteeing safety despite asynchrony, but foregoing progress during the temporary periods of asynchrony, or if more than f faults occur for a system of $N > 2f$ replicas [11]. The classic form of Paxos uses a set of proposers, acceptors and learners, runs in a sequence of ballots, and employs two phases (numbered 1 and 2), with a similar message pattern: proposer to acceptors (phase 1a or 2a), acceptors to proposer (phase 1b or 2b), and, in phase 2b, also acceptors to learners. To ensure progress during synchronous periods, proposals are serialized by a distinguished proposer, which is called the leader.

Paxos is most commonly deployed as Multi (Decree)-Paxos, which provides an optimization of the basic message pattern by omitting the first phase of messages from all but the first ballot for each leader [23]. This means that a leader only needs to send a *phase 1a* message once and subsequent proposals may be sent directly in *phase 2a* messages. This reduces the message pattern in the common case from five message delays to just three (from proposing to learning).

Fast Paxos observes that it is possible to improve on the previous latency (in the common case) by allowing proposers to propose values directly to acceptors [13]. To this end, the protocol distinguishes between fast and classic ballots, where fast ballots bypass the leader by sending proposals directly to acceptors and classic ballots work as in the original Paxos protocol. The reduced latency of fast ballots comes at the added cost of using a quorum size of $N - e$ instead of a classic majority quorum, where e is the number of faults that can be tolerated while using fast ballots. In addition, instead of the usual requirement that $N > 2f$, to ensure that fast and classic quorums intersect, a new requirement must be met: $N > 2e + f$. This means that if we wish to tolerate the same number of faults for classic and fast ballots (i.e., $e = f$), then the minimum number of replicas is $3f + 1$ instead of the usual $2f + 1$. Since fast ballots only take two message steps (*phase 2a* messages between a proposer and the acceptors, and *phase 2b* messages between acceptors and learners), there is the possibility of two proposers concurrently proposing values and generating a conflict, which must be resolved by falling back to a recovery protocol.

Generalized Paxos improves the performance of Fast Paxos by addressing the issue of collisions. In particular, it allows acceptors to accept different sequences of commands as long as non-commutative operations are totally ordered [12]. In the original description, non-commutativity

between operations is generically represented as an interference relation. In this context, Generalized Paxos abstracts the traditional consensus problem of agreeing on a single value to the problem of agreeing on an increasing set of values. *C-structs* provide this increasing sequence abstraction and allow the definition of different consensus problems. If we define the sequence of learned commands of a learner l_i as a *c-struct* $learned_{l_i}$, then the consistency requirement for generalized consensus can be defined as: $learned_{l_1}$ and $learned_{l_2}$ must have a *common upper bound*, for all learners l_1 and l_2. This means that, for any $learned_{l_1}$ and $learned_{l_2}$, there must exist some *c-struct* of which they are both prefixes. This prohibits interfering commands from being concurrently accepted because no subsequent *c-struct* would extend them both.

More recently, other Paxos variants have been proposed to address specific issues. For example, Mencius [18] avoids the latency penalty in wide-area deployments of having a single leader, through which every proposal must go through. In Mencius, the leader of each round rotates between every process: the leader of round i is process p_k, such that $k = n \ mod \ i$. Another variant is Egalitarian Paxos (EPaxos), which achieves a better throughput than Paxos by removing the bottleneck caused by having a leader [20]. To avoid choosing a leader, the proposal of commands for a command slot is done in a decentralized manner, taking advantage of the commutativity observations made by Generalized Paxos [12]. Conflicts between commands are handled by having replicas reply with a command dependency, which then leads to falling back to using another protocol phase with $f + \lfloor \frac{f+1}{2} \rfloor$ replicas.

2.2 Byzantine fault tolerant replication

Consensus in the Byzantine model was originally defined by Lamport et al. [15]. Almost two decades later, a surge of research in the area started with the PBFT protocol, which solves consensus for state machine replication with $3f + 1$ replicas while tolerating up to f Byzantine faults [3]. In PBFT, the system moves through configurations called *views*, in which one replica is the primary and the remaining replicas are the backups. The protocol proceeds in a sequence of steps, where messages are sent from the client to the primary, from the primary to the backups, followed by two all-to-all steps between the replicas, with the last step proceeding in parallel with sending a reply to the clients.

Zeno is a Byzantine fault tolerance state machine replication protocol that trades availability for consistency [24]. In particular, it offers eventual consistency by allowing state machine commands to execute in a *weak quorum* of $f + 1$ replicas. This ensures that at least one correct replica will execute the request and commit it to the linear history, but does not guarantee the intersection property that is required for linearizability.

The closest related work is Fast Byzantine Paxos (FaB), which solves consensus in the Byzantine setting within two message communication steps in the common case, while requiring $5f+1$ acceptors to ensure safety and liveness [19]. A variant that is proposed in the same paper is the Parameterized FaB Paxos protocol, which generalizes FaB by decoupling replication for fault tolerance from replication for performance. As such, the Parameterized FaB Paxos requires $3f + 2t + 1$ replicas to solve consensus, preserving safety while tolerating up to f faults and completing in two steps despite up to t faults. Therefore, FaB Paxos is a special case of Parameterized FaB Paxos where $t = f$. It has also been shown that $N > 5f$ is a lower bound on the number of acceptors required to guarantee 2-step execution in the Byzantine model. In this sense, the FaB protocol is tight since it requires $5f + 1$ acceptors to provide the same guarantees.

In comparison to FaB Paxos, our protocol, Byzantine Generalized Paxos (BGP), requires a lower number of acceptors than what is stipulated by FaB's lower bound. However, this does not constitute a violation of the result since BGP does not guarantee a 2-step execution in the Byzantine scenario. Instead, BGP only provides a two communication step latency when proposed sequences are commutative with any other concurrently proposed sequence. In other words, BGP leverages a weaker performance guarantee to decrease the requirements regarding the minimum number of processes. In particular, a proposed sequence may not gather enough votes to be learned in the ballot in which it is proposed, either due to Byzantine behaviour or contention between non-commutative commands. However, any sequence is guaranteed to eventually be learned in a way such that non-commutative commands are totally ordered at any correct learner.

3 Model

We consider an *asynchronous* system in which a set of $n \in \mathbb{N}$ processes communicate by *sending* and *receiving* messages. Each process executes an algorithm assigned to it, but may fail in two different ways. First, it may stop executing it by *crashing*. Second, it may stop following the algorithm assigned to it, in which case it is considered *Byzantine*. We say that a non-Byzantine process is *correct*. This paper considers the *authenticated* Byzantine model: every process can produce cryptographic digital signatures [25]. Furthermore, for clarity of exposition, we assume authenticated perfect links [2], where a message that is sent by a non-faulty sender is eventually received and messages cannot be forged (such links can be implemented trivially using retransmission, elimination of duplicates, and point-to-point message authentication codes [2].) A process may be a *learner*, *proposer* or *acceptor*. Informally, proposers provide input values that must be agreed upon by learners, the acceptors help the learners *agree* on a value, and learners learn commands by appending them to a local sequence of commands to be executed, $learned_l$. Our protocols require a minimum number of acceptor processes (N), which is a function of the maximum number of tolerated Byzantine faults (f), namely $N \geq 3f + 1$. We assume that acceptor processes have identifiers in the set $\{0, ..., N-1\}$. In contrast, the number of proposer and learner processes can be set arbitrarily.

Problem Statement. In our simplified specification of Generalized Paxos, each learner l maintains a monotonically increasing sequence of commands $learned_l$. We define two learned sequences of commands to be equivalent ($\sim$) if one can be transformed into the other by permuting the elements in a way such that the order of non-commutative pairs is preserved. A sequence x is defined to be an *eq-prefix* of another sequence y ($x \sqsubseteq y$), if the subsequence of y that contains all the elements in x is equivalent ($\sim$) to x. We present the requirements for this consensus problem, stated in terms of learned sequences of commands for a correct learner l, $learned_l$. To simplify the original specification, instead of using c-structs (as explained in Section 2), we specialize to agreeing on equivalent sequences of commands:

1. **Nontriviality.** If all proposers are correct, $learned_l$ can only contain proposed commands.
2. **Stability.** If $learned_l = s$ then, at all later times, $s \sqsubseteq learned_l$, for any sequence s and correct learner l.
3. **Consistency.** At any time and for any two correct learners l_i and l_j, $learned_{l_i}$ and $learned_{l_j}$ can subsequently be extended to equivalent sequences.
4. **Liveness.** For any proposal s from a correct proposer, and correct learner l, eventually $learned_l$ contains s.

4 Protocol

This section presents our Byzantine fault tolerant Generalized Paxos Protocol (or BGP, for short). Given our space constraints, we opted for merging in a single description a novel presentation of Generalized Paxos and its extension to the Byzantine model, even though each represents an independent contribution in its own right.

Algorithm 1 Byzantine Generalized Paxos - Proposer p

Local variables: $ballot\_type = \perp$

```
1: upon receive(BALLOT, type) do
2:     ballot_type = type;
3:
4: upon command_request(c) do
5:     if ballot_type == fast_ballot then
6:         SEND(P2A_FAST, c) to acceptors;
7:     else
8:         SEND(PROPOSE, c) to leader;
```

4.1 Overview

We modularize our protocol explanation according to the following main components, which are also present in other protocols of the Paxos family:

– **View-change** – The goal of this subprotocol is to ensure that, at any given moment, one of the proposers is chosen as a distinguished leader, who runs a specific version of the agreement subprotocol. To achieve this, the view-change subprotocol continuously replaces leaders, until one is found that can ensure progress (i.e., commands are eventually appended to the current sequence).

– **Agreement** – Given a fixed leader, this subprotocol extends the current sequence with a new command or set of commands. Analogously to Fast Paxos [13] and Generalized Paxos [12], choosing this extension can be done through two variants of the protocol: using either **classic ballots** or **fast ballots**, with the characteristic that fast ballots complete in fewer communication steps, but may have to fall back to using a classic ballot when there is contention among concurrent requests.

4.2 View-change

The goal of the view-change subprotocol is to elect a distinguished acceptor process, called the leader, that carries through the agreement protocol, i.e., enables proposed commands to eventually be learned by all the learners. The overall design of this subprotocol is similar to the corresponding part of existing BFT state machine replication protocols [3].

In this subprotocol, the system moves through sequentially numbered views, and the leader for each view is chosen in a rotating fashion using the simple equation *leader(view) = view mod N*. The protocol works continuously by having acceptor processes monitor whether progress is being made on adding commands to the current sequence, and, if not, they multicast a signed SUSPICION message for the current view to all acceptors suspecting the current leader. Then, if enough suspicions are collected, processes can move to the subsequent view. However, the required number of suspicions must be chosen in a way that prevents Byzantine processes from triggering view changes spuriously. To this end, acceptor processes will multicast a view-change message indicating their commitment to starting a new view only after hearing that $f + 1$ processes suspect the leader to be faulty. This message contains the new view number, the $f + 1$ signed suspicions, and is signed by the acceptor that sends it. In the pseudocode, signatures are created by signing data with a process' private key (e.g., $data_{priv_p}$) and validated by decrypting the data with its public key (e.g., $data_{pub_p}$). This way, if a process receives a view-change message without previously receiving $f + 1$ suspicions, it can also multicast a view-change message, after verifying that the suspicions are correctly signed by $f + 1$ distinct processes. This guarantees that if one correct process receives the $f + 1$ suspicions and multicasts the view-change message, then all correct processes, upon receiving this message, will be able to validate the proof of $f + 1$ suspicions and also multicast the view-change message.

Algorithm 2 Byzantine Generalized Paxos - Leader l

Local variables: $ballot_l = 0, maxTried_l = \bot, proposals = \bot, accepted = \bot, notAccepted = \bot, view = 0$

```
1:  upon receive(LEADER, view_a, proofs) from acceptor a do
2:      valid_proofs = 0;
3:      for p in acceptors do
4:          view_proof = proofs[p];
5:          if view_proof_{pub_p} == ⟨view_change, view_a⟩ then
6:              valid_proofs += 1;
7:      if valid_proofs > f then
8:          view = view_a;
9:
10: upon trigger_next_ballot(type) do
11:     ballot_l += 1;
12:     SEND(BALLOT, type) to proposers;
13:     if type == fast then
14:         SEND(FAST, ballot_l, view) to acceptors;
15:     else
16:         SEND(P1A, ballot_l, view) to acceptors;
17:
18: upon receive(PROPOSE, prop) from proposer p_i do
19:     if ISUNIVERSALLYCOMMUTATIVE(prop) then
20:         SEND(P2A_CLASSIC, ballot_l, view, prop);
21:     else
22:         proposals = proposals • prop;
23:
24: upon receive(P1B, ballot, bal_a, proven, val_a, proofs) from acceptor a do
25:     if ballot ≠ ballot_l then
26:         return;
27:
28:     valid_proofs = 0;
29:     for i in acceptors do
30:         proof = proofs[proven][i];
31:         if proof_{pub_i} == ⟨bal_a, proven⟩ then
32:             valid_proofs += 1;
33:
34:     if valid_proofs > N − f then
35:         accepted[ballot_l][a] = proven;
36:         notAccepted[ballot_l] = notAccepted[ballot_l] • (val_a \
    proven);
37:
38:         if #(accepted[ballot_l]) ≥ N − f then
39:             PHASE_2A();
40:
41: function PHASE_2A()
42:     maxTried = LARGEST_SEQ(accepted[ballot_l]);
43:     previousProposals = REMOVE_DUPLICATES(notAccepted[ballot_l]);
44:     maxTried = maxTried • previousProposals • proposals;
45:     SEND(P2A_CLASSIC, ballot_l, view, maxTried_l) to acceptors;
46:     proposals = ⊥;
47: end function
```

Finally, an acceptor process must wait for $N - f$ view-change messages to start participating in the new view, i.e., update its view number and the corresponding leader process. At this point, the acceptor also assembles the $N - f$ view-change messages proving that others are committing to the new view, and sends them to the new leader. This allows the new leader to start its leadership role in the new view once it validates the $N - f$ signatures contained in a single message.

4.3 Agreement protocol

The consensus protocol allows learner processes to agree on equivalent sequences of commands (according to our previous definition of equivalence). An important conceptual distinction between

the original Paxos protocol and BGP is that, in the original Paxos, each instance of consensus is called a ballot, whereas in BGP, instead of being a separate instance of consensus, ballots correspond to an extension to the sequence of learned commands of a single ongoing consensus instance. Proposers can try to extend the current sequence by either single commands or sequences of commands. We use the term *proposal* to denote either the command or sequence of commands that was proposed.

As mentioned, ballots can either be *classic* or *fast*. In classic ballots, a leader proposes a single proposal to be appended to the commands learned by the learners. The protocol is then similar to the one used by classic Paxos [10], with a first phase where each acceptor conveys to the leader the sequences that the acceptor has already voted for (so that the leader can resend commands that may not have gathered enough votes), followed by a second phase where the leader instructs and gathers support for appending the new proposal to the current sequence of learned commands. Fast ballots, in turn, allow any proposer to attempt to contact all acceptors in order to extend the current sequence within only two message delays (in case there are no conflicts between concurrent proposals). However, both types of ballots contain an additional round between the proposal phase and the voting phase called the verification phase. This additional round is an all-to-all broadcast between acceptors in which acceptors broadcast proofs indicating their committal to a sequence.

Algorithm 3 Byzantine Generalized Paxos - Acceptor a (view-change)

Local variables: $suspicions = \bot,\ new\_view = \bot,\ leader = \bot,\ view = 0, bal_a = 0,\ val_a = \bot,\ fast\_bal = \bot,\ checkpoint = \bot$

```
1:  upon suspect_leader do
2:      if suspicions[p] ≠ true then
3:          suspicions[p] = true;
4:          proof = ⟨suspicion, view⟩_priv_a;
5:          SEND(SUSPICION, view, proof);
6:
7:  upon receive(SUSPICION, view_i, proof) from acceptor i do
8:      if view_i ≠ view then
9:          return;
10:     if proof_pub_i == ⟨suspicion, view⟩ then
11:         suspicions[i] = proof;
12:     if #(suspicions) > f and new_view[view+1][p] == ⊥ then
13:         change_proof = ⟨view_change, view + 1⟩_priv_a;
14:         new_view[view + 1][p] = change_proof;
15:         SEND(VIEW_CHANGE, view+1, suspicions, change_proof);
16:
17: upon receive(VIEW_CHANGE, new_view_i, suspicions,
    change_proof_i) from acceptor i do
18:     if new_view_i ≤ view then
19:         return;
20:
21:     valid_proofs = 0;
22:     for p in acceptors do
23:         proof = suspicions[p];
24:         last_view = new_view_i − 1;
25:         if proof_pub_p == ⟨suspicion, last_view⟩ then
26:             valid_proofs += 1;
27:
28:     if valid_proofs ≤ f then
29:         return;
30:
31:     new_view[new_view_i][i] = change_proof_i;
32:     if new_view[view_i][a] == ⊥ then
33:         change_proof = ⟨view_change, new_view_i⟩_priv_a;
34:         new_view[view_i][a] = change_proof;
35:         SEND(VIEW_CHANGE, view_i, suspicions, change_proof);
36:
37:     if #(new_view[new_view_i]) ≥ N − f then
38:         view = view_i;
39:         leader = view mod N;
40:         suspicions = ⊥;
41:         SEND(LEADER, view, new_view[view_i]) to leader;
```

Next, we present the protocol for each type of ballot in detail. We start by describing fast ballots since their structure has consequences that implicate classic ballots.

4.4 Fast ballots

In contrast to classic ballots, fast ballots leverage the weaker specification of generalized consensus (compared to classic consensus) in terms of command ordering at different replicas, to allow for the faster execution of commands in some cases.

The basic idea of fast ballots is that proposers contact the acceptors directly, bypassing the leader, and then the acceptors send their vote for the current sequence directly to the learners, where this sequence now incorporates the proposed value. If a conflict exists and progress isn't being made, the protocol reverts to using a classic ballot. This is where generalized consensus allows for avoiding falling back to this slow path, namely in the case where the commands that are sequenced in a different order at different acceptors commute. However, this concurrency introduces safety problems even when a quorum is reached for some sequence due to the possibility of votes being delayed before reaching the learners. This situation coupled with the wrong proposal being sent in phase $2a$, could result in different sequences being learned by different learners. This is prevented by the introduction of a verification phase where acceptors gather proof of vote for $N - f$ acceptors such that they're able to send these proofs to the leader in the next classic ballot.

Next, we explain each of the protocol's steps for fast ballots in greater detail.

Step 1: Proposer to acceptors. To initiate a fast ballot, the leader informs both proposers and acceptors that the proposals may be sent directly to the acceptors. Unlike classic ballots, where the sequence proposed by the leader consists of the commands received from the proposers appended to previously proposed commands, in a fast ballot, proposals can be sent to the acceptors in the form

of either a single command or a sequence to be appended to the command history. These proposals are sent directly from the proposers to the acceptors.

Step 2: Acceptors to acceptors. Acceptors append the proposals they receive to the proposals they have previously accepted in the current ballot and broadcast the result to the other acceptors. This broadcast contains a signed tuple of the current ballot and the sequence being voted for and intuitively corresponds to a verification phase where acceptors gather proof that a sequence gathered enough support to be committed. This proof will be sent to the leader in subsequent ballots in order for it to pick a new sequence that preserves consistency. To ensure safety, correct learners must learn non-commutative commands in a total order. When an acceptor gathers $N - f$ proofs for equivalent values, it proceeds to the next phase. That is, sequences do not necessarily have to be equal in order to be learned since commutative commands may be reordered. Recall that a sequence is equivalent to another if it can be transformed into the second one by reordering its elements without changing the order of any pair of non-commutative commands. (Note that, in the pseudocode, proofs for equivalent sequences are being treated as belonging to the same index of the *proofs* variable, to simplify the presentation.) By requiring $N - f$ votes for a sequence of commands, we ensure that, given two sequences where non-commutative commands are differently ordered, only one sequence will receive enough votes even if f Byzantine acceptors vote for both sequences. Outside the set of (up to) f Byzantine acceptors, the remaining $2f + 1$ correct acceptors will only vote for a single sequence, which means there are only enough correct processes to commit one of them. Note that the fact that proposals are sent as extensions to previous sequences is critical to the safety of the protocol. In particular, since the votes from acceptors can be reordered by the network before being delivered to the learners, if these values were single commands it would be impossible to guarantee that non-commutative commands would be learned in a total order.

Step 3: Acceptors to learners. Phase $2b$ messages, which are sent from acceptors to learners, contain the current ballot number, the command sequence and the $N - f$ proofs gathered in the verification round. One could think that, since acceptors are already gathering proofs that a value will eventually be committed, learners are not required to gather $N - f$ votes and they can wait for a single phase $2b$ message and validate the $N - f$ proofs contained in it. However, this is not the case due to the possibility of learners learning sequences without the leader being aware of it. If we allowed the learners to learn after witnessing $N - f$ proofs for just one acceptor then that would raise the possibility of that acceptor not being present in the quorum of phase $1b$ messages. Therefore, the leader wouldn't be aware that some value was proven and learned. The only way to guarantee that at least one correct acceptor will relay the latest proven sequence to the leader is by forcing the learner to require $N - f$ phase $2b$ messages since only then will one correct acceptor be in the intersection of the two quorums.

Arbitrating an order after a conflict. When, in a fast ballot, non-commutative commands are concurrently proposed, these commands may be incorporated into the sequences of various acceptors in different orders, and therefore the sequences sent by the acceptors in phase $2b$ messages will not be equivalent and will not be learned. In this case, the leader subsequently runs a classic ballot and gathers these unlearned sequences in phase $1b$. Then, the leader will arbitrate a single serialization for every previously proposed command, which it will then send to the acceptors. Therefore, if non-commutative commands are concurrently proposed in a fast ballot, they will be included in the subsequent classic ballot and the learners will learn them in a total order, thus preserving consistency.

4.5 Classic ballots

Classic ballots work in a way that is very close to the original Paxos protocol [10]. Therefore, throughout our description, we will highlight the points where BGP departs from that original protocol, either due to the Byzantine fault model, or due to behaviors that are particular to the specification of Generalized Paxos.

In this part of the protocol, the leader continuously collects proposals by assembling all commands that are received from the proposers since the previous ballot in a sequence. (This differs from classic Paxos, where it suffices to keep a single proposed value that the leader attempts to reach agreement on). When the next ballot is triggered, the leader starts the first phase by sending phase $1a$ messages to all acceptors containing just the ballot number. Similarly to classic Paxos, acceptors reply with a phase $1b$ message to the leader, which reports all sequences of commands they voted for. In classic Paxos, acceptors also promise not to participate in lower-numbered ballots, in order

to prevent safety violations [10]. However, in BGP this promise is already implicit, given (1) there is only one leader per view and it is the only process allowed to propose in a classic ballot and (2) acceptors replying to that message must be in the same view as that leader.

As previously mentioned, phase $1b$ messages contain $N - f$ proofs for each learned sequence. By waiting for $N - f$ of such messages, the leader is guaranteed that, for any learned sequence s, at least one of the messages will be from a correct acceptor that, due to the quorum intersection property, participated in the verification phase of s. In the CFT version of the protocol, the leader could determine if some value had been chosen by a majority of acceptors by looking for $f + 1$ identical votes in the quorum of phase $1b$ messages. If some value had $f + 1$ votes, no other value could have been chosen since there are only other $2f$ acceptors in the system. However, the same isn't true for BGP because $f + 1$ identical votes for some value only attest that at least one correct process voted for that value. It's impossible to determine if some value was chosen by a majority unless the leader witnesses $2f + 1$ identical votes. Note that, since each command is signed by the proposed (this signature and its check is not explicit in the pseudocode), a Byzantine acceptor can't relay made-up commands. However, it can omit commands from its phase $1b$ message, which is why it's necessary for the leader to be sure that at least one correct acceptor in its quorum took part in the verification quorum of any learned sequence.

After gathering a quorum of $N - f$ phase $1b$ messages, the leader initiates phase $2a$ where it assembles a proposal and sends it to the acceptors. This proposal sequence must be carefully constructed to ensure both progress and consistency. Due to the importance and required detail of the leader's value picking rule, it will be described next in its own section. The acceptors reply to phase $2a$ messages by broadcasting their verification messages containing the proposed sequence and proof of their committal to that sequence. After receiving $N - f$ verification messages, an acceptor sends its phase $2b$ message to the learners, containing the ballot, the proposal from the leader and the $N - f$ proofs gathered in the verification phase. The behavior and reasoning behinds the learners actions is the same as in fast ballots, so we omit it in this description.

Algorithm 4 Byzantine Generalized Paxos - Acceptor a (agreement)

Local variables: $leader = \bot,\ view = 0, bal_a = 0,\ val_a = \bot,\ fast\_bal = \bot,\ proven = \bot$

```
1:  upon receive(P1A, ballot, view_l) from leader l do
2:      if view_l == view and bal_a < ballot then
3:          send(P1B, ballot, bal_a, proven, val_a, proofs[bal_a]) to leader;
4:          bal_a = ballot;
5:
6:  upon receive(FAST, ballot, view_l) from leader do
7:      if view_l == view then
8:          fast_bal[ballot] = true;
9:
10: upon receive(VERIFY, view_i, ballot_i, val_i, proof) from acceptor i do
11:     if proof_pub_i == ⟨ballot_i, val_i⟩ and view == view_i then
12:         proofs[ballot_i][val_i][i] = proof;
13:         if (#(proofs[ballot_i][val_i]) ≥ N − f or (#(proofs[ballot_i][val_i]) > f and isUniversallyCommutative(val_i))) and proofs[ballot_i][val_i][a] ≠ ⊥ then
14:             proven = val_i;
15:             send(P2B, ballot_i, val_i, proofs[ballot_i][value_i]) to learners;
16:
17: upon receive(P2A_CLASSIC, ballot, view, value) from leader do
18:     if view_l == view then
19:         phase_2b_classic(ballot, value);
20:
21: upon receive(P2A_FAST, value) from proposer do
22:     phase_2b_fast(value);
23:
24: function phase_2b_classic(ballot, value)
25:     univ_commut = isUniversallyCommutative(val_a);
26:     if ballot ≥ bal_a and !fast_bal[bal_a] and (univ_commut or proven == ⊥ or proven == subsequence(value, 0, #(proven))) then
27:         bal_a = ballot;
28:         if univ_commut then
29:             send(P2B, bal_a, value) to learners;
30:         else
31:             val_a = value;
32:             proof = ⟨ballot, val_a⟩_priv_a;
33:             proofs[ballot][val_a][a] = proof;
34:             send(VERIFY, view, ballot, val_a, proof) to acceptors;
35: end function
36:
37: function phase_2b_fast(ballot, value)
38:     if ballot == bal_a and fast_bal[bal_a] then
39:         if isUniversallyCommutative(value) then
40:             send(P2B, bal_a, value) to learners;
41:         else
42:             val_a = val_a • value;
43:             proof = ⟨ballot, val_a⟩_priv_a;
44:             proofs[ballot][val_a][a] = proof;
45:             send(VERIFY, view, ballot, val_a, proof) to acceptors;
46: end function
```

Leader value picking rule. Phase $2a$ is crucial for the correct functioning of the protocol because it requires the leader to pick a value that allows new commands to be learned, ensuring progress, while at the same time preserving a total order of non-commutative commands at different learners, ensuring consistency. The value picked by the leader is composed of three pieces: (1) the subsequence that has proven to be accepted by a majority of acceptors in the previous fast ballot, (2) the subsequence that has been proposed in the previous fast ballot but for which a quorum hasn't been gathered and (3) new proposals sent to the leader in the current classic ballot.

The first part of the sequence will be the largest sequence of the $N - f$ proven sequences sent in the phase $1b$ messages. The leader can pick such a value deterministically because, for any two proven sequences, they are either equivalent or one can be extended to the other. The leader is sure of this because for the quorums of any two proven sequences there is at least one correct acceptor that voted in both and votes from correct acceptors are always extensions of previous votes from

the same ballot. If there are multiple sequences with the maximum size then they are equivalent (by same reasoning applied previously) and any can be picked.

The second part of the sequence is simply the concatenation of unproven sequences or commands in an arbitrary order. Since these commands are guaranteed to not be learned at any learner, they can be appended to the leader's sequence in any order. Since $N - f$ phase $2b$ messages are required for a learner to learn a sequence and the intersection between the leader's quorum and the quorum gathered by a learner for any sequence contains at least one correct acceptor, the leader can be sure that if a sequence of commands is unproven in all of the gathered phase $1b$ messages, then that sequence wasn't learned and can be safely appended to the leader's sequence in any order.

The third part consists simply of commands sent by proposers to the leader with the intent of being learned at the current ballot. These values can be appended in any order and without any restriction since they're being proposed for the first time.

Algorithm 5 Byzantine Generalized Paxos - Learner l

Local variables: $learned = \bot, messages = \bot$

```
1:  upon receive(P2B, ballot, value, proofs) from acceptor a do
2:      valid_proofs = 0;
3:      for i in acceptors do
4:          proof = proofs[i];
5:          if proof_{pub_i} == ⟨ballot, value⟩ then
6:              valid_proofs += 1;
7:
8:      if valid_proofs ≥ N − f then
9:          messages[ballot][value][a] = proofs;
10:
11:         if #(messages[ballot][value]) ≥ N − f then
12:             learned = MERGE_SEQUENCES(learned, value);
13:
14: upon receive(P2B, ballot, value) from acceptor a do
15:     if ISUNIVERSALLYCOMMUTATIVE(value) then
16:         messages[ballot][value][a] = true;
17:         if #(messages[ballot][value]) > f then
18:             learned = learned • value;
19:
20: function MERGE_SEQUENCES(old_seq, new_seq)
21:     for c in new_seq do
22:         if !CONTAINS(old_seq, c) then
23:             old_seq = old_seq • c;
24:     return old_seq;
25: end function
```

5 Conclusion and discussion

We presented a simplified description of the Generalized Paxos specification and protocol, as well as its extension to be resilient against Byzantine faults. We now draw some lessons and outline some extensions to our protocol that present interesting directions for future work.

Handling faults in the fast case. A result that was stated in the original Generalized Paxos paper [12] is that to tolerate f crash faults and allow for fast ballots whenever there are up to e crash faults, the total system size N must uphold two conditions: $N > 2f$ and $N > 2e + f$. Additionally, the fast and classic quorums must be of size $N - e$ and $N - f$, respectively. This implies that there is a price to pay in terms of number of replicas and quorum size for being able to run fast operations during faulty periods. An interesting observation is that since Byzantine fault tolerance already requires a total system size of $3f + 1$ and a quorum size of $2f + 1$, we are able to amortize the cost of both features, i.e., we are able to tolerate the maximum number of faults for fast execution without paying a price in terms of the replication factor and quorum size.

Universally commutative commands. A downside of the use of commutative commands in the context of Generalized Paxos is that the commutativity check is done at runtime, to determine if non-commutative commands have been proposed concurrently. This raises the possibility of extending the protocol to handle commands that are universally commutative, i.e., commute with every other command. For these commands, it is known before executing them that they will not generate any conflicts, and therefore it is not necessary to check them against concurrently executing commands. This allows us to optimize the protocol by both bypassing the verification round and decreasing the number of phase $2b$ messages required to learn to a smaller $f+1$ quorum since $N-f$ votes aren't required to prevent learners from learning conflicting sequences. A quorum of $f+1$ is sufficient to ensure that a correct acceptor will eventually propagate the command to a quorum of $N-f$ acceptors. This optimization is particularly useful in the context of geo-replicated systems, since it can be significantly faster to wait for the $f+1$st message instead of the $N-f$th one. The link between Generalized Paxos and weak consistency models becomes clearer with the introduction of universally commutative commands. In the case of weakly consistent replication, weakly consistent requests can be executed as if they were universally commutative, even if in practice that may not be the case. E.g., checking the balance of a bank account and making a deposit do not commute since the output of the former depends on their relative order. However, some systems prefer to run both as weakly consistent operations [17].

Fast Byzantine Paxos comparison. In comparison to FaB Paxos, our Byzantine Generalized Paxos protocol requires a lower number of acceptors than what is stipulated by FaB's lower bound [19]. However, this does not constitute a violation of the result since BGP does not guarantee a two step execution in the Byzantine scenario. Instead, BGP only provides a two communication

step latency when proposed sequences are universally commutative with any other sequence. In the common case, BGP requires three messages steps for a sequence to be learned. In other words, Byzantine Generalized Paxos introduces an additional broadcast phase to decrease the requirements regarding the minimum number of acceptor processes. This may be a sensible trade-off in systems that target datacenter environments where communication between machines is fast and a high percentage of costs is directly related to equipment since fast communication links mitigate the latency cost of having an additional phase between the acceptors and the high cost of equipment and power consumption makes the reduced number of acceptor processes attractive.

Acknowledgements. This work was supported by the European Research Council (ERC-2012-StG-307732) and FCT (UID/CEC/50021/2013).

References

1. Burrows, M.: The chubby lock service for loosely-coupled distributed systems. In: Proc. 7th Symposium on Operating Systems Design and Implementation (2006)
2. Cachin, C., Guerraoui, R., Rodrigues, L.: Introduction to Reliable and Secure Distributed Programming (2nd ed.). Springer (2011)
3. Castro, M., Liskov, B.: Practical byzantine fault tolerance. In: Proc. 3rd Symposium on Operating Systems Design and Implementation (OSDI) (1999)
4. De Prisco, R., Lampson, B., Lynch, N.A.: Revisiting the Paxos algorithm. In: Proc. 11th Workshop on Distributed Algorithms. LNCS 1320, Springer (1997)
5. DeCandia et al., G.: Dynamo: Amazon's highly available key-value store. In: Proc. 21st Symposium on Operating Systems Principles (SOSP) (2007)
6. Fischer, M.J., Lynch, N.A., Paterson, M.S.: Impossibility of distributed consensus with one faulty process. J. ACM 32(2), 374–382 (Apr 1985)
7. Junqueira, F., Reed, B., Serafini, M.: Zab: High-performance broadcast for primary-backup systems. In: 41st Int. Conf. Dependable Systems and Networks (2011)
8. Ladin, R., Liskov, B., Shrira, L.: Lazy replication: Exploiting the semantics of distributed services. In: Proc. 9th Symp. Principles Distributed Computing (1990)
9. Lamport, L.: The part-time parliament. Tech. rep., DEC SRC (1989)
10. Lamport, L.: The Part-Time parliament. ACM Trans. on Computer Systems 16(2), 133–169 (May 1998)
11. Lamport, L.: Paxos made simple. SIGACT News 32(4), 18–25 (Dec 2001)
12. Lamport, L.: Generalized consensus and paxos. Tech. rep., MSR-TR-2005-33, Microsoft Research (2005)
13. Lamport, L.: Fast paxos. Distributed Computing 19(2), 79–103 (2006)
14. Lamport, L.: Byzantizing paxos by refinement. In: Distributed Computing - 25th International Symposium, DISC 2011. Proceedings (2011)
15. Lamport, L., Shostak, R., Pease, M.: The Byzantine generals problem. ACM Trans. Progr. Lang. Syst. 4(3), 382–401 (Jul 1982)
16. Lee, E.K., Thekkath, C.A.: Petal: Distributed virtual disks. In: Proc. 7th Int. Conf. Architectural Support for Programming Languages and Operating Systems (1996)
17. Li, C., Porto, D., Clement, A., Gehrke, J., Preguiça, N., Rodrigues, R.: Making Geo-Replicated Systems Fast as Possible, Consistent when Necessary. In: Proc. 10th Symp. Operating Systems Design and Implementation (OSDI) (2012)
18. Mao, Y., Junqueira, F.P., Marzullo, K.: Mencius: Building Efficient Replicated State Machines for WANs. In: Proc. 8th Symp. Operating Systems Design and Implementation (OSDI) (2008)
19. Martin, J.P., Alvisi, L.: Fast byzantine consensus. IEEE Trans. Dependable Secur. Comput. 3(3), 202–215 (Jul 2006)
20. Moraru, I., Andersen, D.G., Kaminsky, M.: There is more consensus in Egalitarian parliaments. In: Proc. Symposium on Operating Systems Principles (SOSP) (2013)
21. Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system (2008)
22. Pires, M., Ravi, S., Rodrigues, R.: Generalized Paxos Made Byzantine (and Less Complex). Tech. rep. (2017)
23. van Renesse, R.: Paxos Made Moderately Complex. ACM Computing Surveys 47(3), 1–36 (2011)
24. Singh, A., Fonseca, P., Kuznetsov, P.: Zeno: Eventually Consistent Byzantine-Fault Tolerance. In: Proc. 6th Symp. Networked Systems Design and Implementation (NSDI) (2009)
25. Vukolic, M.: Quorum Systems: With Applications to Storage and Consensus. Synthesis Lectures on Distributed Computing Theory, Morgan & Claypool (2012)