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), Lisbon, Portugal
miguel.pires@tecnico.ulisboa.pt, rodrigo.rodrigues@inesc-id.pt
[2] University of Southern California, Los Angeles, USA
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 [7]. Paxos [11], 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 [10], and was only published a decade later [11]. 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 [5] and the distributed systems [17] 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 [2], or the open source ZooKeeper project [8], 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 [12,24].

P. Spirakis and P. Tsigas (Eds.): SSS 2017, LNCS 10616, pp. 203–218, 2017.
https://doi.org/10.1007/978-3-319-69084-1_14

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 [15,20]. 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 [22]. Second, many proposals target improving the Paxos protocol by eliminating communication costs [14], including an important evolution of the protocol called Generalized Paxos [13], which has the noteworthy aspect of having lower communication costs by leveraging a more general specificationan 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 [6,9].

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 makes 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 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: Sect. 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. This section also presents a proof that the Byzantine Generalized Paxos protocol guarantees *consistency*, while the correctness proofs for the remaining properties are included in a tech report. Section 5 concludes the paper with a discussion of several optimizations and practical considerations. The complete tech report with the formal proofs is available on the ArXiv repository.

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 [7]. 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 [12]. 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 [24]. 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 [14]. 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 [13]. 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 [19] 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 [21]. 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 [13]. 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. [16]. 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 [4]. 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 [25]. 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 [20]. 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 [26]. Furthermore, for clarity of exposition, we assume authenticated perfect links [3], 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 [3].) 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 Sect. 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 [14] and Generalized Paxos [13], 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 [4].

Algorithm 2. Byzantine Generalized Paxos - Process p

```
1:  function MERGE_SEQUENCES(old_seq, new_seq)
2:      for c in new_seq do
3:          if !CONTAINS(old_seq, c) then
4:              old_seq = old_seq • c;
5:      end for
6:      return old_seq;
7:  end function
8:
9:  function SIGNED_COMMANDS(full_seq)
10:     signed_seq = ⊥;
11:     for c in full_seq do
12:         if verify_command(c) then
13:             signed_seq = signed_seq • c;
14:     end for
15:     return signed_seq;
16: end function
```

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.

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.

Algorithm 3. Byzantine Generalized Paxos - Leader l

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

```
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:     proposals = proposals • prop;
20: upon receive(P1B, bal_a, view_vals_a) from acceptor a do
21:     if bal_a = ballot_l then
22:     accepted[ballot_l][a] = SIGNED_COMMANDS(view_vals_a);
23:         if #(accepted[ballot_l]) ≥ N − f then
24:             PHASE_2A();
25:
26: function PHASE_2A()
27:     maxTried_l = PROVED_SAFE(ballot_l);
28:     maxTried_l = maxTried_l • proposals;
29:     if CLEAN_STATE?() then
30:         maxTried_l = maxTried_l • C*;
31:     SEND(P2A_CLASSIC,ballot_l,view, maxTried_l) to acceptors;
32:     proposals = ⊥;
33: end function
34:
35: function PROVED_SAFE(ballot)
36:     safe_seq = ⊥;
37:     for seq in accepted[ballot] do
38:         safe_seq = MERGE_SEQUENCES(safe_seq, seq);
39:     end for
40:     return safe_seq;
41: end function
```

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 [11], 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).

Next, we present the protocol for each type of ballot in detail.

4.4 Classic Ballots

Classic ballots work in a way that is very close to the original Paxos protocol [11]. 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.

Algorithm 4. 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:   valid_proofs = 0;
21:   for p in acceptors do
22:     proof = suspicions[p];
23:     last_view = new_view_i − 1;
24:     if proof_pub_p = ⟨suspicion, last_view⟩ then
25:       valid_proofs += 1;
26:   if valid_proofs ≤ f then
27:     return;
28:   new_view[new_view_i][i] = change_proof_i;
29:   if new_view[view_i][a] = ⊥ then
30:     change_proof=⟨view_change, new_view_i⟩_priv_a;
31:     new_view[view_i][a] = change_proof;
32:     SEND(VIEW_CHANGE, view_i, suspicions, change_proof);
33:   if #(new_view[new_view_i]) ≥ N − f then
34:     view = view_i;
35:     leader = view mod N;
36:     suspicions = ⊥;
37:     SEND(LEADER, view, new_view[view_i]) to leader;
```

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 [11]. 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.

Upon receiving phase 1b messages, the leader checks that the commands are authentic by validating command signatures. (This is needed due to the Byzantine model.) After gathering a quorum of $N - f$ responses, the leader initiates phase 2a by sending a message with a proposal to the acceptors (as in the original protocol, but with a quorum size adjusted for the Byzantine model). This proposal is constructed by appending the proposals received from the proposers to a sequence that contains every command in the sequences that were previously accepted by the acceptors in the quorum (instead of sending a single value with the highest ballot number in the classic specification).

The acceptors reply to phase 2a messages by sending phase 2b messages to the learners, containing the ballot and the proposal from the leader. After receiving $N - f$ votes for a sequence, a learner learns it by extracting the commands that are not contained in his *learned* sequence and appending them in order. (This differs from the original protocol in the quorum size, due to the fault model, and by the fact that learners would wait for a quorum of matching values, due to the consensus specification.)

Algorithm 5. Byzantine Generalized Paxos - Acceptor a (agreement)

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

```
1:  upon receive(P1A, ballot, view_l) from leader l do
2:      if view_l = view then
3:          PHASE_1B(ballot);
4:
5:  upon receive(FAST, ballot, view_l) from leader do
6:      if view_l = view then
7:          fast_bal[ballot] = true;
8:
9:  upon receive(P2B,ballot,value,proof) from acceptor i
    do
10:     if proof_pub_i ≠ ⟨ballot, value⟩ then
11:         return;
12:     checkpoint[ballot][i] = proof;
13:     if #(checkpoint[ballot]) ≥ N − f then
14:         SEND(P2B, ballot, value, checkpoint[ballot])
    to learners;
15:         val_a = ⊥;
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_1B(ballot)
25:     if bal_a < ballot then
26:         SEND(P1B, ballot, val_a) to leader;
27:         bal_a = ballot;
28:         val_a[bal_a] = ⊥;
29: end function
30:
31: function PHASE_2B_CLASSIC(ballot, value)
32:     if ballot ≥ bal_a and val_a = ⊥ then
33:         bal_a = ballot;
34:         val_a[ballot] = value;
35:         if CONTAINS(value, C*) then
36:             proof = ⟨suspicion, view⟩_priv_a;
37:             SEND(P2B, ballot, value, proof) to acceptors;
38:         else
39:             SEND(P2B, ballot, value) to learners;
40: end function
41:
42: function PHASE_2B_FAST(value)
43:     if fast_bal[bal_a] then
44:         val_a[bal_a] =MERGE_SEQUENCES(val_a[bal_a], value);
45:         SEND(P2B, bal_a, val_a[bal_a]) to learners;
46: end function
```

4.5 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 directly to the learners their vote for the current sequence, where this sequence now incorporates the proposed value. If a learner can gather $N - f$ votes for a sequence (or an equivalent one), then it is learned. If, however, a conflict exists between sequences then they will not be considered equivalent and at most one of them will gather enough votes to be learned. Conflicts are dealt with by maintaining the proposals at the acceptors so they can be sent to the leader and learned in the next classic ballot. This differs from Fast Paxos where recovery is performed through an additional round-trip.

Next, we explain each of these steps in more 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.

Step 2: Acceptors to Learners. Acceptors append the proposals they receive to the proposals they have previously accepted in the current ballot and broadcast the result to the learners. Similarly to what happens in classic ballots, the fast ballot equivalent of the phase 2*b* message, which is sent from acceptors to

Algorithm 6. Byzantine Generalized Paxos - Learner l

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

```
1:  upon receive(P2B, ballot, value) from acceptor a
    do
2:      messages[ballot][value][a] = true;
3:      if #(messages[ballot][value]) ≥ N-f or (#(mes-
        sages[ballot][value]) > f and isUniversallyCommuta-
        tive(value)) then
4:          learned = merge_sequences(learned, value);
5:
6:  upon receive(P2B, ballot, value, proofs) from
    acceptor a do
7:      valid_proofs = 0;
8:      for i in acceptors do
9:          proof = proofs[i];
10:         if proof_pub_i = ⟨ballot, value⟩ then
11:             valid_proofs += 1;
12:     if valid_proofs > f then
13:         learned = merge_sequences(learned, value);
```

learners, contains the current ballot number and the command sequence. However, since commands (or sequences of commands) are concurrently proposed, acceptors can receive and vote for non-commutative proposals in different orders. To ensure safety, correct learners must learn non-commutative commands in a total order. To this end, a learner must gather $N - f$ votes for equivalent sequences. 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, equivalent sequences are being treated as belonging to the same index of the *messages* 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.

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 2*b* 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 1*b*. 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.

Checkpointing. A checkpointing feature allows the leader to propose a special command C^* that causes processes to discard stored commands. However, since commands are kept at the acceptors to ensure that they will eventually be com-

mitted, the checkpointing command must be sent within a sequence in a classic ballot along with the commands stored by $N - f$ acceptors. Since, when proposing to acceptors in fast ballots, proposers wait for acknowledgments from $N - f$ acceptors, all proposed sequences will be sent to the leader and included in the leader's sequence, along with the checkpointing command. Since acceptors must be certain that it's safe to discard previously stored commands, before sending phase 2b messages to learners, they first broadcast these messages among themselves to ensure that a Byzantine leader can't make a subset of acceptors discard state. After waiting for $N - f$ such messages, acceptors send phase 2b messages to the learners along with the cryptographic proofs exchanged in the acceptor-to-acceptor broadcast. After receiving just one message, a learner may simply validate the $N - f$ proofs and learn the commands. The learners discard previously stored state when they execute the checkpointing command.

4.6 Correctness

We now prove the correctness of the presented Byzantine Generalized Paxos protocol. Invariants and symbols specific to our proof are defined in Table 1. Due to space constraints, we only discuss the proof of consistency, but the remaining proofs and an extended version of the protocol to address cross-ballot consistency are available in a technical report [23].

Table 1. Proof notation

Invariant/Symbol	Definition
$\sim$	Equivalence relation between sequences
$X \sqsubseteq Y$	The sequence X is a prefix of sequence Y
$\mathcal{L}$	Set of learner processes
$\mathcal{P}$	Set of proposals (commands or sequences of commands)
$\bot$	Empty command
$learned_{l_i}$	Learner l_i's *learned* sequence of commands
$learned(l_i, s)$	$learned_{l_i}$ contains the sequence s
$maj\_accepted(s)$	$N - f$ acceptors sent phase 2b messages to the learners for sequence s
$min\_accepted(s)$	$f + 1$ acceptors sent phase 2b messages to the learners for sequence s

Theorem 1. *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.*

Proof:

1. At any given instant, $\forall s, s' \in \mathcal{P}, \forall l_i, l_j \in \mathcal{L}, learned(l_j, s) \wedge learned(l_i, s') \implies \exists \sigma_1, \sigma_2 \in \mathcal{P} \cup \{\bot\}, s \bullet \sigma_1 \sim s' \bullet \sigma_2$
   Proof:

1.1. At any given instant, $\forall s, s' \in \mathcal{P}, \forall l_i, l_j \in \mathcal{L}, learned(l_i, s) \wedge learned(l_j, s') \implies (maj\_accepted(s) \vee (min\_accepted(s) \wedge s \bullet \sigma_1 \sim x \bullet \sigma_2)) \wedge (maj\_accepted(s') \vee (min\_accepted(s') \wedge s' \bullet \sigma_1 \sim x \bullet \sigma_2)), \exists \sigma_1, \sigma_2 \in \mathcal{P} \cup \{\bot\}, \forall x \in \mathcal{P}$
Proof: A sequence can only be learned if the learner gathers $N - f$ votes (i.e., $maj\_accepted(s)$) or if it is universally commutative (i.e., $s \bullet \sigma_1 \sim x \bullet \sigma_2$, $\exists \sigma_1, \sigma_2 \in \mathcal{P} \cup \{\bot\}, \forall x \in \mathcal{P}$) and the learner gathers $f + 1$ votes (i.e., $min\_accepted(s)$).The first case includes both gathering $N - f$ votes directly from each acceptor (Algorithm 6 lines {1–4}) and gathering $N - f$ proofs of vote from only one acceptor, as is the case when the sequence contains a special checkpointing command (Algorithm 6 {6–11}). The second case requires that the sequence must be commutative with any other (Algorithm 6 {1–4}). This is encoded in the logical expression $s \bullet \sigma_1 \sim x \bullet \sigma_2$ which is true the if learned sequence can be extended with σ_1 to the same that any other sequence x can be extended to with a possibly different sequence σ_2, therefore making it impossible to result in a conflict.

1.2. At any given instant, $\forall s, s' \in \mathcal{P}, maj\_accepted(s) \wedge maj\_accepted(s') \implies \exists \sigma_1, \sigma_2 \in \mathcal{P} \cup \{\bot\}, s \bullet \sigma_1 \sim s' \bullet \sigma_2$
Proof: Proved by contradiction.

1.2.1. At any given instant, $\exists s, s' \in \mathcal{P}, \forall \sigma_1, \sigma_2 \in \mathcal{P} \cup \{\bot\}, maj\_accepted(s) \wedge maj\_accepted(s') \wedge s \bullet \sigma_1 \not\sim s' \bullet \sigma_2$
Proof: Contradiction assumption.

1.2.2. Take a pair proposals s and s' that meet the conditions of 1.2.1 (and are certain to exist by the previous point), then s and s' are non-commutative
Proof: If $\forall \sigma_1, \sigma_2 \in \mathcal{P} \cup \{\bot\}, s \bullet \sigma_1 \not\sim s' \bullet \sigma_2$, then s and s' must contain non-commutative commands differently ordered. Otherwise, some combination of σ_1 and σ_2 would be commutative. If $s \bullet \sigma_1 \not\sim s' \bullet \sigma_2$ even for commutative σ_1 and σ_2 then s and s' must contain non-commutative commands in different relative orders.

1.2.3. At any given instant, $\neg(maj\_accepted(s) \wedge maj\_accepted(s'))$
Proof: Since s and s' are non-commutative, therefore not equivalent, and each correct acceptor only votes once for a new proposal (Algorithm 5, lines {31–46}), any learner will only obtain $N - f$ votes for one of the sequences (Algorithm 6, lines {1–4}).

1.2.4. A contradiction is found, Q.E.D.

1.3. For any pair of proposals s and s', at any given instant, $\forall x \in \mathcal{P}, \exists \sigma_1, \sigma_2, \sigma_3, \sigma_4 \in \mathcal{P} \cup \{\bot\}, (maj\_accepted(s) \vee (min\_accepted(s) \wedge s \bullet \sigma_1 \sim x \bullet \sigma_2)) \wedge (maj\_accepted(s') \vee (min\_accepted(s') \wedge s \bullet \sigma_1 \sim x \bullet \sigma_2)) \implies s \bullet \sigma_3 \sim s' \bullet \sigma_4$
Proof: By 1.2 and by definition of $s \bullet \sigma_1 \sim x \bullet \sigma_2$.

1.4. At any given instant, $\forall s, s' \in \mathcal{P}, \forall l_i, l_j \in \mathcal{L}, learned(l_i, s) \wedge learned(l_j, s') \implies \exists \sigma_1, \sigma_2 \in \mathcal{P} \cup \{\bot\}, s \bullet \sigma_1 \sim s' \bullet \sigma_2$
Proof: By 1.1 and 1.3.

1.5. Q.E.D.

2. At any given instant, $\forall l_i, l_j \in \mathcal{L}, learned(l_j, learned_j) \wedge learned(l_i, learned_i) \implies \exists \sigma_1, \sigma_2 \in \mathcal{P} \cup \{\perp\}, learned_i \bullet \sigma_1 \sim learned_j \bullet \sigma_2$
Proof: By 1.
3. Q.E.D.

5 Conclusion and Discussion

We presented a simplified description of the Generalized Paxos specification and protocol, and an implementation of Generalized Paxos that is resilient against Byzantine faults. We now draw some lessons and outline some extensions to our protocol that present interesting directions for future work and hopefully a better understanding of its practical applicability.

Handling Faults in the Fast Case. A result that was stated in the original Generalized Paxos paper [13] 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.

Extending the Protocol to 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 decreasing the number of phase $2b$ messages required to learn to a smaller $f + 1$ quorum. Since, by definition, these sequences are guaranteed to never produce conflicts, the $N - f$ quorum is not required to prevent learners from learning conflicting sequences. Instead, a quorum of $f + 1$ is sufficient to ensure that a correct acceptor saw the command and will eventually propagate it 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.

Generalized Paxos and Weak Consistency. The key distinguishing feature of the specification of Generalized Paxos [13] is allowing learners to learn concurrent proposals in a different order, when the proposals commute. This idea is closely related to the work on weaker consistency models like eventual or causal

consistency [1], or consistency models that mix strong and weak consistency levels like RedBlue [18], which attempt to decrease the cost of executing operations by reducing coordination requirements between replicas. The link between the two models becomes clearer with the introduction of universally commutative commands in the previous paragraph. 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, even though it may cause executions that are not explained by a sequential execution, since the semantics are still acceptable given that the final state that is reached is the same and no invariants of the application are violated [18].

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

References

1. Ahamad, M., Neiger, G., Burns, J.E., Kohli, P., Hutto, P.W.: Causal memory: definitions, implementation, and programming. Distrib. Comput. **9**(1), 37–49 (1995)
2. Burrows, M.: The chubby lock service for loosely-coupled distributed systems. In: Proceedings of 7th Symposium on Operating Systems Design and Implementation (2006)
3. Cachin, C., Guerraoui, R., Rodrigues, L.: Introduction to Reliable and Secure Distributed Programming, 2nd edn. Springer, Heidelberg (2011). doi:10.1007/978-3-642-15260-3
4. Castro, M., Liskov, B.: Practical byzantine fault tolerance. In: Proceedings of 3rd Symposium on Operating Systems Design and Implementation (OSDI) (1999)
5. De Prisco, R., Lampson, B., Lynch, N.A.: Revisiting the paxos algorithm. In: Mavronicolas, M., Tsigas, P. (eds.) WDAG 1997. LNCS, vol. 1320, pp. 111–125. Springer, Heidelberg (1997). doi:10.1007/BFb0030679
6. DeCandia, G., et al.: Dynamo: Amazon's highly available key-value store. In: Proceedings of 21st Symposium on Operating Systems Principles (SOSP) (2007)
7. Fischer, M.J., Lynch, N.A., Paterson, M.S.: Impossibility of distributed consensus with one faulty process. J. ACM **32**(2), 374–382 (1985)
8. Junqueira, F., Reed, B., Serafini, M.: Zab: high-performance broadcast for primary-backup systems. In: 41st International Conference on Dependable Systems and Networks (2011)
9. Ladin, R., Liskov, B., Shrira, L.: Lazy replication: exploiting the semantics of distributed services. In: Proceedings of 9th Symposium on Principles Distributed Computing (1990)
10. Lamport, L.: The part-time parliament. Technical report, DEC SRC (1989)
11. Lamport, L.: The part-time parliament. ACM Trans. Comput. Syst. **16**(2), 133–169 (1998)
12. Lamport, L.: Paxos made simple. SIGACT News **32**(4), 18–25 (2001)
13. Lamport, L.: Generalized consensus and paxos. Technical report, Technical Report MSR-TR-2005-33, Microsoft Research (2005)

14. Lamport, L.: Fast paxos. Distrib. Comput. **19**(2), 79–103 (2006)
15. Lamport, L.: Byzantizing paxos by refinement. In: Peleg, D. (ed.) DISC 2011. LNCS, vol. 6950, pp. 211–224. Springer, Heidelberg (2011). doi:10.1007/978-3-642-24100-0_22
16. Lamport, L., Shostak, R., Pease, M.: The byzantine generals problem. ACM Trans. Progr. Lang. Syst. **4**(3), 382–401 (1982)
17. Lee, E.K., Thekkath, C.A.: Petal: distributed virtual disks. In: Proceedings of 7th International Conference on Architectural Support for Programming Languages and Operating Systems (1996)
18. Li, C., Porto, D., Clement, A., Gehrke, J., Preguiça, N., Rodrigues, R.: Making geo-replicated systems fast as possible, consistent when necessary. In: Proceedings of 10th Symposium on Operating Systems Design and Implementation (OSDI) (2012)
19. Mao, Y., Junqueira, F.P., Marzullo, K.: Mencius: building efficient replicated state machines for WANs. In: Proceedings of 8th Symposium on Operating Systems Design and Implementation (OSDI) (2008)
20. Martin, J.P., Alvisi, L.: Fast byzantine consensus. IEEE Trans. Dependable Secur. Comput. **3**(3), 202–215 (2006)
21. Moraru, I., Andersen, D.G., Kaminsky, M.: There is more consensus in Egalitarian parliaments. In: Proceedings of Symposium on Operating Systems Principles (SOSP) (2013)
22. Nakamoto, S.: Bitcoin: a peer-to-peer electronic cash system (2008)
23. Pires, M., Ravi, S., Rodrigues, R.: Generalized Paxos Made Byzantine (and Less Complex). Tech. rep. (2017)
24. van Renesse, R.: Paxos made moderately complex. ACM Comput. Surv. **47**(3), 1–36 (2011)
25. Singh, A., Fonseca, P., Kuznetsov, P.: Zeno: eventually consistent byzantine-fault tolerance. In: Proceedings of 6th Symposium on Networked Systems Design and Implementation (NSDI) (2009)
26. Vukolic, M.: Quorum systems: with applications to storage and consensus. In: Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool (2012)