algorithms

MDPI

Article

Generalized Paxos Made Byzantine (and Less Complex) †

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

[1] INESC-ID and Instituto Superior Técnico (Universidade de Lisboa), R. Alves Redol 9, 1000-029 Lisbon, Portugal; rodrigo.miragaia.rodrigues@tecnico.ulisboa.pt

[2] Department of Computer Science and Information Sciences Insititute, University of Southern California, Los Angeles, CA 90007, USA; srivatsan@srivatsan.in

* Correspondence: miguel.pires@tecnico.ulisboa.pt; Tel.: +351-213100300

† This paper is an extended version of our paper published in SSS 2017: Stabilization, Safety, and Security of Distributed Systems, Boston, MA, USA, 5–8 November 2017.

Received: 26 February 2018; Accepted: 9 August 2018; Published: 17 August 2018

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 a 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 in the context of blockchain 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 attempt to provide a simpler description of Generalized Paxos, 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, and provide the respective correctness proof.

Keywords: Byzantine fault tolerance; consensus; Paxos

1. Introduction

The evolution of the Paxos [1] protocol is a unique chapter in the history of Computer Science. It was first described in 1989 through a technical report [2], and was only published a decade later [1]. 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 [3] and the distributed systems [4] research communities. In addition, 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 [5], or the open source ZooKeeper project [6], 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 [7,8].

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 [9,10]. 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 [11]. Second, many proposals target improving the Paxos protocol by eliminating communication costs [12], including an important evolution of the protocol called Generalized Paxos [13], which has the noteworthy aspect

of having lower communication costs by leveraging a specification that is weaker than traditional consensus. In particular, instead of forcing all processes to agree on the same value, 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 a significant research activity on the corresponding weaker consistency models for replicated systems [14,15].

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.

As such, the paper makes several contributions, which are listed next.

- 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 present a simplified version of the Generalized Paxos protocol, complete with pseudocode;
- we extend the Generalized Paxos protocol to the Byzantine fault model;
- we present a description of the Byzantine Generalized Paxos protocol 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: Section 2 is a detailed overview of Paxos and related protocols that inspired the algorithm in this paper. Section 3 introduces the model and specification of Generalized Paxos. Section 4 presents a simplified version of the Generalized Paxos protocol in the crash fault model. Section 5 presents the Generalized Paxos protocol that is resilient against Byzantine failures. Section 6 presents correctness proofs, organized according to the properties defined in the problem statement of Section 3. Section 7 discusses some optimizations and concludes the paper. This paper is an extended version of [16], with permission from Springer Nature.

2. Background and Overview

2.1. Paxos and Its Variants

2.1.1. Classic Paxos

The Paxos protocol family solves consensus by finding an equilibrium in face of the well-known FLP impossibility result [17]. It does this by always guaranteeing safety in an asynchronous system, but at the same time making the observation that most of the time systems have periods during which they can be considered synchronous, since long delays are often sporadic and temporary. Therefore, Paxos only foregoes progress during the temporary periods of asynchrony, or if more than f faults occur for a system of $n = 2f + 1$ replicas [7]. The classic form of Paxos employs 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, acceptors to proposer (and, in phase 2, 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 [8]. 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 proposal to learning). Since there are no implications on the quorum size or guarantees provided by Paxos, the reduced latency comes at no additional cost.

2.1.2. Fast Paxos

Fast Paxos observes that it is possible to improve on the previous latency (in terms of common case message steps) by allowing proposers to propose values directly to acceptors [12]. 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 Basic Paxos. The reduced latency of fast ballots comes at the additional 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 total number of replicas is $3f + 1$ instead of the usual $2f + 1$ and the quorum size for fast and classic ballots is the same. The optimized commit scenario occurs during fast ballots, in which only two messages broadcasts are necessary: *phase 2a* messages between a proposer and the acceptors, and *phase 2b* messages between acceptors and learners. This creates the possibility of two proposers concurrently proposing values to the acceptors and generating a conflict, which must be resolved by falling back to a recovery protocol.

2.1.3. Generalized Paxos

Generalized Paxos addresses Fast Paxos' shortcomings regarding collisions. More precisely, it allows acceptors to accept different sequences of commands as long as non-commutative operations are totally ordered [13]. Non-commutativity between operations is generically represented as an interference relation. 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 abstraction of an increasing set of values and allow us to define 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 consensus can be defined as:

- **Consistency**—$learned[l_1]$ and $learned[l_2]$ are always compatible, for all learners l_1 and l_2.

For two *c-structs* to be compatible, they must have a *common upper bound*. This means that, for any two learned *c-structs* such as $learned[l_1]$ and $learned[l_2]$, there must exist some *c-struct* to which they are both prefixes. This prohibits non-commutative commands from being concurrently accepted because no subsequent *c-struct* would extend them both since it would not have a total order of non-commutative operations. For instance, consider a set of commands $\{A, B, C\}$ and an interference relation between commands A and B (i.e., they are non-commutative with respect to each other). If proposers propose A and C concurrently, some learners may learn one command before the other and the resulting *c-structs* would be either $C \bullet A$ or $A \bullet C$. These are compatible because there are *c-structs* that extend them, namely $A \bullet C \bullet B$ and $C \bullet A \bullet B$. These *c-structs* that extend them are valid because the interfering commands are totally ordered. However, if two proposers propose A and B, learners could learn either one in the first ballot and these *c-structs* would not be compatible because no *c-struct* extends them. Any *c-struct* would start either by $A \bullet B$ or $B \bullet A$, which means that an interference relation would be violated. In the Generalized Paxos protocol, when such a collision occurs, no value is chosen and the leader intervenes by starting a new ballot and proposing a *c-struct*. Defining *c-structs* as command histories enables acceptors to agree on different sequences of commands and still preserve consistency as long as dependence relationships are not violated. This means that commutative commands can be ordered differently regarding each other but interfering commands must preserve the same order across each sequence at any learner. This guarantees that solving the consensus problem for histories is enough to implement a state-machine replicated system.

2.1.4. Mencius

Mencius is also a variant of Paxos that tries to address the bottleneck 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 the process p_k, such that $k = n \ mod \ i$. Leaders with nothing to propose can skip their turn by proposing a *no-op*. If a leader is slow or faulty, the other replicas can execute *phase 1* to revoke the leader's right to propose a value, but they can only propose a *no-op* instead [18]. Considering that non-leader replicas can only propose *no-ops*, a *no-op* command from the leader can be accepted in a single message delay since there is no chance of another value being accepted. If some non-leader server revokes the leader's right to propose and suggests a *no-op*, then the leader can still suggest a value $v \neq$ *no-op*, which will eventually be accepted as long as l is not permanently suspected. Mencius also takes advantage of commutativity by allowing out-of-order commits, where values x and y can be learned in different orders by different learners if there does not exist a dependence relationship between them.

2.1.5. Egalitarian Paxos

Egalitarian Paxos (EPaxos) extends the goal of Mencius of achieving a better throughput than Paxos by removing the bottleneck caused by having a leader [19]. 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]. If two replicas unknowingly propose commands concurrently, one will commit its proposal in one round trip after getting replies from a quorum of replicas. However, some replica will see that another command was concurrently proposed and may interfere with the already committed command. If the commands are non-commutative then the replica must reply with a dependency between the commands, committing its command in two rounds trips. This commit latency is achieved by using a *fast-path quorum* of $f + \lfloor \frac{f+1}{2} \rfloor$ replicas. Similarly to Mencius, EPaxos achieves a substantially higher throughput than Multi-Paxos.

2.2. Byzantine Fault Tolerant Replication

The Byzantine Generals Problem is defined as a set of Byzantine generals that are camped in the outskirts of an enemy city and have to coordinate an attack. Each general can either decide to attack or retreat and there may be f traitors among the generals that try to prevent the loyal generals from agreeing on the same action. The problem is solved if every loyal general agrees on what action to take [20]. Like the traitorous generals, a process that suffers a Byzantine fault may display an arbitrary behaviour and, in case of multiple Byzantine faults, an adversary may even coordinate multiple faulty replicas in an attack.

Practical Byzantine Fault Tolerance (PBFT)

PBFT is a protocol that solves consensus while tolerating up to f Byzantine faults [9]. The system moves through configurations called *views* in which one replica is the primary and the remaining replicas are the backups. The safety property of the algorithm requires that operations be totally ordered. The protocol starts when a client sends a request for an operation to the primary, which in turn assigns a sequence number to the request and multicasts a *pre-prepare* message to the backups. This message contains the timestamp, the digest of the client's message, the view and the actual request. If a backup replica accepts the pre-prepare message, after verifying that the view number and timestamp are correct, it multicasts a *prepare* message and adds both messages to its log. The prepare message is similar to the pre-prepare message except that it does not contain the client's request message. Both of these phases ensure that the requested operation is totally ordered at every correct replica (note that the two phases described informally are not necessary for safety as demonstrated by [21]). The protocol's safety property requires that the replicated service must satisfy linearizability and, therefore, operations must be totally ordered. After receiving $2f$ prepare messages, a replica

multicasts a *commit* message and commits the message to its log when it has received $2f$ commit messages from other replicas. The liveness property requires that clients must eventually receive replies to their requests, provided that there are at most $\lfloor \frac{N-1}{3} \rfloor$ faults and the transmission time does not increase continuously. This property represents a weak liveness condition but one that is enough to circumvent the FLP impossibility result [17]. A Byzantine leader may try to prevent progress by omitting pre-prepare messages when it receives operation requests from clients, but backups can trigger new views after waiting for a certain period of time. The description of the Byzantine Generalized Paxos protocol presented in this paper largely follows from the PBFT protocol.

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 stop executing it by *crashing*. If a process does not follow the algorithm assigned to it, then it is *byzantine*. This paper considers the *authenticated* Byzantine model: every process can produce cryptographic digital signatures [22]. Furthermore, for clarity of exposition, we assume authenticated perfect links [23], 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 [23].) A process may be a *learner*, *proposer* or *acceptor*. Informally, proposers provide input values that must be agreed upon by learners and the acceptors help the learners *agree* on a value.

Problem Statement. In Generalized Paxos, each learner l maintains a monotonically increasing sequence of commands $learned_l$. We define these 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 a *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 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:

Nontriviality. $learned_l$ can only contain proposed commands.

Stability. If $learned_l = v$ then, at all later times, $v \sqsubseteq learned_l$, for any l and v.

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.

Liveness. For any proposal s and correct learner l, eventually $learned_l$ contains s.

4. Crash Fault Tolerant Protocol

This section describes the crash fault tolerant version of the Generalized Paxos protocol for our simplified problem. The only modifications applied to the protocol were made to make it simpler while still ensuring its correctness. The protocol should still be recognizable as Generalized Paxos since its message pattern and control flow remain the same. However, we chose to describe it in detail, both in the interest of clarity and also to showcase how the specialization to the command history problem affects the protocol.

4.1. 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 Fast Paxos protocol and our simplified Generalized Paxos is that, in Fast Paxos [12], each instance of consensus is called a ballot and agrees upon a single value, whereas in our protocol, much like the original Generalized Paxos, instead of being separate instances of consensus, ballots correspond to an extension to the sequence of learned commands of a single ongoing consensus instance. In both protocols, ballots can either be *classic* or *fast*.

In classic ballots, a leader proposes a single sequence of commands, such that it can be appended to the commands learned by the learners. A classic ballot in Generalized Paxos follows a protocol that is very similar to the one used by classic Paxos [2] (cf. Algorithms 1 and 2). This protocol comprises a first phase where each acceptor conveys to the leader the sequences it has voted for. This allows the protocol to preserve safety and also allows leader to resend unlearned commands. This is followed by a second phase where the leader picks an extension to the sequence of commands relayed in *phase 1b* messages and broadcasts it to the acceptors. The acceptors send their votes to the learners, who then, after gathering enough support for a given extension to the current sequence, append the new commands to their own sequences of learned commands and discard the already learned ones.

Algorithm 1 Generalized Paxos—Proposer p

Local variables: $ballot\_type = \perp, ballot = 0$

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

Algorithm 2 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:     return old_seq
6: end function
```

In fast ballots, multiple proposers can concurrently propose either single commands or sequences of commands by sending them directly to the acceptors (we use the term *proposal* to denote either the command or sequence of commands that was proposed). In this case, concurrency implies that acceptors may receive proposals in a different order. If the resulting sequences are equivalent, then they are successfully learned in two message delays. If not, the protocol must fall back to using a classic ballot.

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

4.1.1. Classic Ballots

As previously mentioned, classic ballots work in a similar way to previous Paxos protocols. Therefore, we will highlight the points where Generalized Paxos departs from the Classic Paxos protocol, in particular where it is due to behaviors caused by our simplified specification of Generalized Paxos.

In this part of the protocol, the leader continuously collects proposals by assembling commands received from the proposers in a sequence (cf. Algorithm 3). This sequence is built by appending arriving proposals to a sequence containing every proposal received since the previous ballot (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. This message also

implicitly conveys a promise not to participate in lower-numbered ballots, in order to prevent safety violations [2].

After gathering a quorum of $N - f$ *phase 1b* messages, the leader initiates *phase 2a* by sending a message with a proposal to the acceptors. The procedure followed by the leader to construct this proposal is critical to prevent conflicts between sequences proposed in different ballots as well as to ensure liveness even when conflicts occur during fast ballots. There are two possible scenarios when observing the quorum Q of gathered *phase 1b* messages: either there is one reported sequence s that was voted for at least $f + 1$ acceptors in the latest ballot or there is none. If such a sequence exists then it is guaranteed to be the only one that may have been learned. Since $2f + 1$ votes are necessary for any sequence to be learned and at least $f + 1$ acceptors voted for s then any other non-commutative sequence gathered at most $2f$ votes, which is insufficient for it to be learned. If no sequence in the quorum gathered $f + 1$ votes then the leader can be sure that no value was or will be learned in that ballot. Since any sequence present in the quorum gathered at most f votes and there are only f acceptors outside of it, any sequence gathered at most $2f$ votes, which is also not enough for it to be learned. However, even if the latest ballot didn't result in the learning of a value, the leader still has to pick the most up-to-date sequence in order to extend it with his proposals. Notice that, even though the latest ballot may not have reached consensus on a sequence, some previous ballot did and the *phase 2b* quorum of that ballot intersects in the current quorum of *phase 1b* messages in $f + 1$ acceptors. Therefore, we arrive at a well-defined value picking rule: given a quorum Q of *phase 1b* messages, if some sequence s has more than f votes at the highest ballot in which some acceptor voted for, then that sequence is chosen as the prefix of the leader's proposal. If no such sequence exists, then the leader picks the longest prefix that is present in $f + 1$ sequences. It's possible to further simplify this rule by noting that the second case encases the first, since the longest possible prefix ($\sqsubseteq$) of a sequence is the sequence itself. More formally:

> **Leader rule**. For a quorum Q of *phase 1b* messages, pick the longest prefix present in the sequences of at least $f + 1$ messages in Q.

After picking the most up-to-date sequence accepted by a quorum, the leader appends the commands present in *phase 1b* messages that are not in the chosen sequence. This ensures liveness since any proposer's command that reaches more than f acceptors before the next ballot begins will eventually be included in an accepted proposal. After executing this rule, the leader simply appends the proposers' commands to the sequence and sends it to the acceptors in *phase 2a* messages.

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. Please note that for a sequence to be learned, a learner does not have to receive $N - f$ votes for the exact same sequence but for equivalence sequences (in accordance to our previous definition of equivalence).

Algorithm 3 Generalized Paxos—Leader l

Local variables: $ballot_l = 0, proposals = \perp, accepted = \perp$

upon trigger_next_ballot($type$) **do**
    $ballot_l \mathrel{+}= 1$
    SEND($BALLOT, ballot_l, type$) to proposers

    if $type = fast$ **then**
        SEND($FAST, ballot_l, view$) to acceptors
    else
        SEND($P1A, ballot_l, view$) to acceptors

upon receive($PROPOSE, prop$) from proposer p_i **do**
    if ISUNIVERSALLYCOMMUTATIVE($prop$) **then**
        SEND($P1A, ballot, prop$) to acceptors
    else
        $proposals = proposals \bullet prop$

upon receive($P1B, ballot, bal_a, vals_a$) from acceptor a **do**
    if $ballot_a = ballot_l$ **then**
        $accepted[ballot_l][a] = \langle bal_a, vals_a \rangle$
        if $\#(accepted[ballot_l]) \geq N - f$ **then**
            PHASE_2A()

function PHASE_2A()
    $votes = \perp$
    $k = -1$
    for a **in** $acceptors$ **do**
        $bal_a = accepted[ballot_l][a][0]$
        $val_a = accepted[ballot_l][a][1]$
        if $bal_a > k$ **then**
            $k = bal_a$
            $votes = \perp$
        else if $bal_a = k$ **then**
            $votes[val_a] \mathrel{+}= 1$
            if $votes[val_a] > f$ **then**
                $maxTried_l = val_a$
                break

    for a **in** $acceptors$ **do**
        $maxTried_l =$ MERGE_SEQUENCES($maxTried_l, accepted[ballot_l][a]$)

    $maxTried_l = maxTried_l \bullet proposals$
    SEND($P2A\_CLASSIC, ballot_l, maxTried_l$) to acceptors
    $proposals = \perp$
    $maxTried_l = \perp$
end function

4.1.2. Fast Ballots

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

The basic idea of fast ballots is that proposers contact the acceptors directly (code for acceptors in Algorithm 4), bypassing the leader (code for leaders in Algorithm 5), and then the acceptors send their votes on proposals to the learners. 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 [12].

Algorithm 4 Generalized Paxos—Acceptor a

Local variables: $leader = \perp, bal_a = 0, val_a = \perp, fast\_bal = \perp$

```
1:  upon receive(P1A, ballot) from leader do
2:      PHASE_1B(ballot)
3:
4:  upon receive(FAST, ballot) from leader do
5:      fast_bal[ballot] = true
6:
7:  upon receive(P2A_CLASSIC, ballot, value) from leader do
8:      PHASE_2B_CLASSIC(ballot, value)
9:
10: upon receive(P2A_FAST, ballot, value) from proposer p do
11:     PHASE_2B_FAST(ballot, value)
12:
13: function PHASE_1B(ballot)
14:     if bal_a < ballot then
15:         SEND(P1B, ballot, bal_a, val_a) to leader
16:         bal_a = ballot
17:         val_a = ⊥
18: end function
19:
20: function PHASE_2B_CLASSIC(ballot, value)
21:     if ballot ≥ bal_a and val_a = ⊥ then
22:         bal_a = ballot
23:         if ISUNIVERSALLYCOMMUTATIVE(value) then
24:             SEND(P2B, ballot, value) to learners
25:         else
26:             val_a[ballot] = value
27:             SEND(P2B, ballot, value) to learners
28: end function
29:
30: function PHASE_2B_FAST(ballot, value)
31:     if ballot = bal_a and fast_bal[bal_a] then
32:         if ISUNIVERSALLYCOMMUTATIVE(value) then
33:             SEND(P2B, ballot, value) to learners
34:         else
35:             val_a[bal_a] = MERGE_SEQUENCES(val_a[bal_a], value)
36:             SEND(P2B, bal_a, val_a[bal_a]) to learners
37: end function
```

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. These proposals are sent directly from the proposers to the acceptors.

Algorithm 5 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** $\#(messages[ballot][value]) \geq N - f$ or (isUniversallyCommutative($value$) and $\#(messages[ballot][value]) > f$) **then**
4: $learned =$ merge_sequences($learned, value$)

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, a *phase 2b* message is sent from acceptors to learners, containing 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. Please note that, in Algorithm 3 lines {32–33} and Algorithm 5 lines {2–3}, equivalent sequences are being treated as belonging to the same index of the *votes* or *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. Since each acceptor will only vote for a single sequence, there are only enough correct processes to commit one of them. Please note that the fact that proposals are sent as extensions of 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 at 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. In that case, the sequences sent by the acceptors in *phase 2b* messages will not be equivalent and will not be learned. In order to preserve liveness, the leader subsequently runs a classic ballot and gathers the acceptors' previous votes in *phase 1b*. After reaching a quorum of *phase 1b* messages, it assembles a single serialization for every previously proposed command, which it will then send to the acceptors along with new proposals. Therefore, if non-commutative commands fail to be learned 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 and liveness.

The assembling of previous commands in a single serialization is done through a deterministic procedure. In the first part of this procedure, the leader guarantees safety by picking the most recent previously learned sequence. In the second part of the procedure, the leader extracts commands not included in the previous chosen sequence and appends them to it. This guarantees that any proposed command will eventually be learned, ensuring liveness. The last component of the leader's proposal is a sequence with new sequences sent by proposers.

5. Byzantine Fault Tolerant Protocol

This section presents our Byzantine fault tolerant Generalized Paxos Protocol (or BGP, for short) (code for proposer in Algorithm 6). In BGP, the number of acceptor processes is a function of the maximum number of tolerated Byzantine faults f, specifically, $\geq 3f + 1$, and quorums are any set of $N - f$ processes.

Algorithm 6 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
```

5.1. 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 [12] 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.

5.2. View Change

The goal of the view change subprotocol is to elect a distinguished proposer process, called the leader (code for leader in Algorithm 7), 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 [9].

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 $leader(view) = view \bmod N$. The protocol works continuously by having acceptor processes monitor whether progress is being made on adding commands to the current sequence, and, if not, by multicasting 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. 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 $f + 1$ suspicions and also multicast the view-change message.

Algorithm 7 Byzantine Generalized Paxos—Leader l

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

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** $view\_proof_{pub_p} == \langle view\_change, view_a \rangle$ **then**
6: $valid\_proofs \mathrel{+}= 1$
7: **if** $valid\_proofs > f$ **then**
8: $view = view_a$
9:
10: **upon** *trigger_next_ballot(type)* **do**
11: $ballot_l \mathrel{+}= 1$
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 **do**
19: **if** ISUNIVERSALLYCOMMUTATIVE$(prop)$ **then**
20: SEND$(P2A_CLASSIC, ballot_l, view, prop)$
21: **else**
22: $proposals = proposals \bullet prop$
23:
24: **upon** *receive(*$P1B, ballot, bal_a, proven, val_a, proofs$*)* from acceptor a **do**
25: **if** $ballot \neq ballot_l$ **then**
26: **return**
27:
28: $valid\_proofs = 0$
29: **for** i **in** $acceptors$ **do**
30: $proof = proofs[proven][i]$
31: **if** $proof_{pub_i} == \langle bal_a, proven \rangle$ **then**
32: $valid\_proofs \mathrel{+}= 1$
33:
34: **if** $valid\_proofs > N - f$ **then**
35: $accepted[ballot_l][a] = proven$
36: $notAccepted[ballot_l] = notAccepted[ballot_l] \bullet (val_a \setminus proven)$
37:
38: **if** $\#(accepted[ballot_l]) \geq N - f$ **then**
39: PHASE_2A()
40:
41: **function** PHASE_2A()
42: $maxTried =$ LARGEST_SEQ$(accepted[ballot_l])$
43: $previousProposals =$ REMOVE_DUPLICATES$(notAccepted[ballot_l])$
44: $maxTried = maxTried \bullet previousProposals \bullet proposals$
45: SEND$(P2A_CLASSIC, ballot_l, view, maxTried)$ to acceptors
46: $proposals = \bot$
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 (cf. Algorithm 8). 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.

5.3. Agreement Protocol

The consensus protocol allows learner processes to agree on equivalent sequences of commands (according to the definition of equivalence presented in Section 3). An important conceptual distinction between Fast Paxos [12] and our protocol is that ballots correspond to an extension of the sequence of learned commands of a single ongoing consensus instance, instead of being a separate instance of consensus. 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.

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 [1], 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 contact all acceptors directly in order to extend the current sequence (in case there are no conflicts between concurrent proposals). However, both types of ballots contain an additional round, called the verification phase, in which acceptors broadcast proofs among each other indicating their committal to a sequence. This additional round comes after the acceptors receive a proposal and before they send their votes to the learners.

Next, we present the protocol for each type of ballot in detail. We start by describing fast ballots since their structure has consequences that influence classic ballots. Figures 1 and 2 illustrate the message pattern for fast and classic ballots, respectively. In these illustrations, arrows that are composed of solid lines represent messages that can be sent multiple times per ballot (once per proposal) while arrows composed of dotted lines represent messages that are sent only once per ballot.

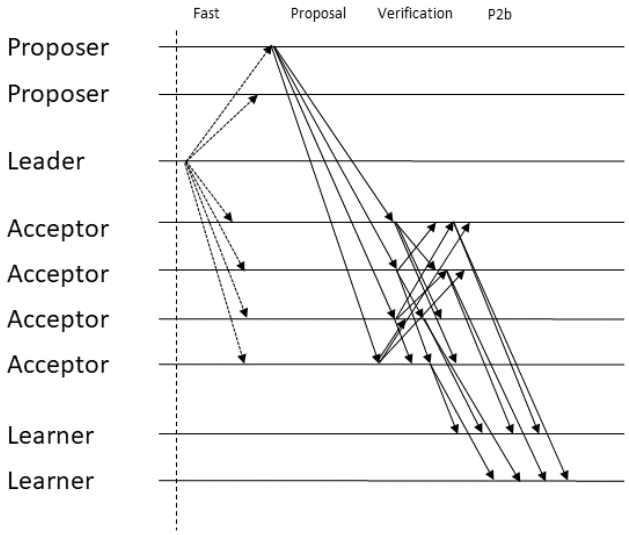

Figure 1. BGP's fast ballot message pattern.

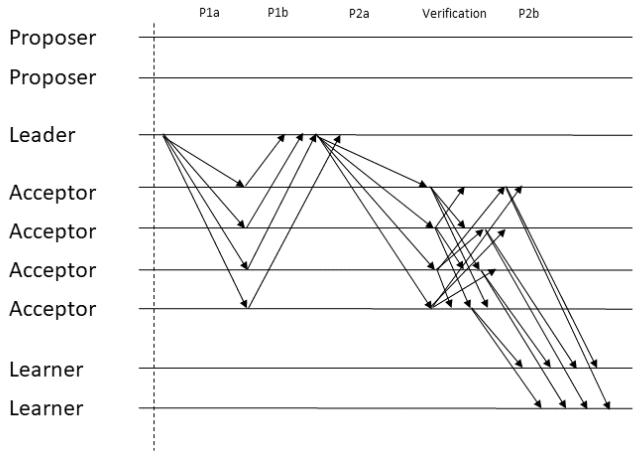

Figure 2. BGP's classic ballot message pattern.

Algorithm 8 Byzantine Generalized Paxos—Acceptor a (view change)

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

upon *suspect_leader* **do**
  if $suspicions[p] \neq true$ **then**
    $suspicions[p] = true$
    $proof = \langle suspicion, view \rangle_{priv_a}$
    SEND$(SUSPICION, view, proof)$

upon *receive*$(SUSPICION, view_i, proof)$ from acceptor i **do**
  if $view_i \neq view$ **then**
    return
  if $proof_{pub_i} == \langle suspicion, view \rangle$ **then**
    $suspicions[i] = proof$
  if $\#(suspicions) > f$ and $new\_view[view + 1][p] == \perp$ **then**
    $change\_proof = \langle view\_change, view + 1 \rangle_{priv_a}$
    $new\_view[view + 1][p] = change\_proof$
    SEND$(VIEW_CHANGE, view + 1, suspicions, change_proof)$

upon *receive*$(VIEW\_CHANGE, new\_view_i, suspicions, change\_proof_i)$ from acceptor i **do**
  if $new\_view_i \leq view$ **then**
    return

  $valid\_proofs = 0$
  for p **in** *acceptors* **do**
    $proof = suspicions[p]$
    $last\_view = new\_view_i - 1$
    if $proof_{pub_p} == \langle suspicion, last\_view \rangle$ **then**
      $valid\_proofs \mathrel{+}= 1$

  if $valid\_proofs \leq f$ **then**
    return

  $new\_view[new\_view_i][i] = change\_proof_i$
  if $new\_view[view_i][a] == \perp$ **then**
    $change\_proof = \langle view\_change, new\_view_i \rangle_{priv_a}$
    $new\_view[view_i][a] = change\_proof$
    SEND$(VIEW_CHANGE, view_i, suspicions, change_proof)$

  if $\#(new\_view[new\_view_i]) \geq N - f$ **then**
    $view = view_i$
    $leader = view \bmod N$
    $suspicions = \perp$
    SEND$(LEADER, view, new_view[view_i])$ to leader

5.3.1. Fast 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 to the learners. If a conflict exists and progress is not being made, the protocol reverts to using a classic ballot. This is where generalized consensus allows us to avoid falling back to this slow path, namely in the case where commands that ordered differently at different acceptors commute (code for acceptors in Algorithm 9).

However, this concurrency introduces safety problems even when a quorum is reached for some sequence. If we keep the original Fast Paxos message pattern [12], it is possible for one sequence s to be learned at one learner l_1 while another non-commutative sequence s' is learned before s at another learner l_2. Suppose s obtains a quorum of votes and is learned by l_1 but the same votes are delayed indefinitely before reaching l_2. In the next classic ballot, when the leader gathers a quorum of *phase 1b* messages it must arbitrate an order for the commands that it received from the acceptors and it does not know the order in which they were learned. This is because, of the $N - f$ messages it received, f may not have participated in the quorum and another f may be Byzantine and lie about their vote, which only leaves one correct acceptor that participated in the quorum and a single vote is not enough to determine if the sequence was learned or not. If the leader picks the wrong sequence, it would be proposing a sequence s' that is non-commutative to a learned sequence s. Since the learning of s was delayed before reaching l_2, l_2 could learn s' and be in a conflicting state with respect to l_1, violating consistency. In order to prevent this, sequences accepted by a quorum of acceptors must be monotonic extensions of previous accepted sequences. Regardless of the order in which a learner learns a set of monotonically increasing sequences, the resulting state will be the same. The additional verification phase is what allows acceptors to prove to the leader that some sequence was accepted by a quorum. By gathering $N - f$ proofs for some sequence, an acceptor can prove that at least $f + 1$ correct acceptors voted for that sequence. Since there are only another $2f$ acceptors in the system, no other non-commutative value may have been voted for by a quorum.

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 resulting sequence and the current ballot to the other acceptors, along with a signed tuple of these two values. Intuitively, this broadcast corresponds to a verification phase where acceptors gather proofs that a sequence gathered enough support to be committed. These proofs will be sent to the leader in the subsequent classic ballot in order for it to pick a 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 (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. As in the non-Byzantine protocol, the fact that proposals are sent as extensions to previous sequences makes the protocol robust against the network reordering of non-commutative commands.

Algorithm 9 Byzantine Generalized Paxos—Acceptor a (agreement)

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

upon *receive*$(P1A, ballot, view_l)$ from leader l **do**
  if $view_l == view$ and $bal_a < ballot$ **then**
    SEND$(P1B, ballot, bal_a, proven, val_a, proofs[bal_a])$ to leader
    $bal_a = ballot$
    $val_a = \perp$

upon *receive*$(FAST, ballot, view_l)$ from leader **do**
  if $view_l == view$ **then**
    $fast\_bal[ballot] = true$

upon *receive*$(VERIFY, view_i, ballot_i, val_i, proof)$ from acceptor i **do**
  if $proof_{pub_i} == \langle ballot_i, val_i \rangle$ and $view == view_i$ **then**
    $proofs[ballot_i][val_i][i] = proof$
    if $\#(proofs[ballot_i][val_i]) \geq N - f$ **then**
      $proven = val_i$
      SEND$(P2B, ballot_i, val_i, proofs[ballot_i][value_i])$ to learners

upon *receive*$(P2A\_CLASSIC, ballot, view, value)$ from leader **do**
  if $view_l == view$ **then**
    PHASE_2B_CLASSIC$(ballot, value)$

upon *receive*$(P2A\_FAST, value)$ from proposer **do**
  PHASE_2B_FAST$(value)$

function PHASE_2B_CLASSIC$(ballot, value)$
  $univ\_commut =$ ISUNIVERSALLYCOMMUTATIVE(val_a)
  if $ballot \geq bal_a$ and $val_a == \perp$ and $!fast\_bal[bal_a]$ and ($univ\_commut$ or $proven == \perp$ or $proven ==$ SUBSEQUENCE$(value, 0, \#(proven))$) **then**
    $bal_a = ballot$
    if $univ\_commut$ **then**
      SEND$(P2B, bal_a, value)$ to learners
    else
      $val_a = value$
      $proof = \langle ballot, val_a \rangle_{priv_a}$
      $proofs[ballot][val_a][a] = proof$
      SEND$(VERIFY, view, ballot, val_a, proof)$ to acceptors
end function

function PHASE_2B_FAST$(ballot, value)$
  if $ballot == bal_a$ and $fast\_bal[bal_a]$ **then**
    if ISUNIVERSALLYCOMMUTATIVE$(value)$ **then**
      SEND$(P2B, bal_a, value)$ to learners
    else
      $val_a = val_a \bullet value$
      $proof = \langle ballot, val_a \rangle_{priv_a}$
      $proofs[ballot][val_a][a] = proof$
      SEND$(VERIFY, view, ballot, val_a, proof)$ to acceptors
end function

Step 3: Acceptors to learners. Similarly to what happens in classic ballots, the fast ballot equivalent of the *phase 2b* message, which is sent from acceptors to learners, contains 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.

5.3.2. Classic Ballots

Classic ballots work in a way that is very close to the original Paxos protocol [1]. 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 our specification of the consensus problem.

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 [1]. 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 $N - f$ 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. Please note that waiting for $N - f$ *phase 1b* messages is not what makes the leader be sure that a certain sequence was learned in a previous ballot. The leader can be sure that some sequence was learned because each *phase 1b* message contains cryptographic proofs from $2f + 1$ acceptors stating that they would vote for that sequence. Since there are only $3f + 1$ acceptors in the system, no other non-commutative sequence could have been learned. Even though each *phase 1b* message relays enough proofs to ensure the leader that some sequence was learned, the leader still needs to wait for $N - f$ such messages to be sure that he is aware of any sequence that was previously learned. Please note that, since each command is signed by the proposer (this signature and its check are not explicit in the pseudocode), a Byzantine acceptor cannot relay made-up commands. However, it can omit commands from its *phase 1b* message, which is why it is 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 in order to ensure all of the intended properties. In particular, the proposal cannot contain already learned non-commutative commands in different relative orders than the one in which they were learned, in order to preserve consistency, and it must contain unlearned proposals from both the current and the previous ballots, in order to preserve liveness (this differs from sending a single value with the highest ballot number as in the classic specification). Due to the importance and required detail of the leader's value picking rule, it will be described next in its own subsection.

The acceptors reply to *phase 2a* messages by broadcasting their verification messages containing the current ballot, the proposed sequence and proof of their committal to that sequence. After receiving $N - f$ verification messages, an acceptor sends its *phase 2b* messages to the learners, containing the ballot, the proposal from the leader and the $N - f$ proofs gathered in the verification phase. As is the case in the fast ballot, when a learner receives a *phase 2b* vote, it validates the $N - f$ proofs contained in it. Waiting for a quorum of $N - f$ messages for a sequence ensures the learners that at least one of those messages was sent by a correct acceptor that will relay the sequence to the leader in the next classic ballot (the learning of sequences also differs from the original protocol in the quorum size, due to the fault model, and in that the learners would wait for a quorum of matching values instead of equivalent sequences, due to the consensus specification).

5.3.3. 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 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 of commands in an arbitrary order. Since these commands are guaranteed to not have been 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.

5.3.4. Byzantine Leader

The correctness of the protocol is heavily dependent on the guarantee that the sequence accepted by a quorum of acceptors is an extension of previous proven sequences. Otherwise, if the network rearranges *phase 2b* messages such that they're seen by different learners (cf. Algorithm 10) in different orders, they will result in a state divergence. If, however, every vote is a prefix of all subsequent votes then, regardless of the order in which the sequences are learned, the final state will be the same.

Algorithm 10 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
```

This state equivalence between learners is ensured by the correct execution of the protocol since every vote in a fast ballot is equal to the previous vote with a sequence appended at the end (Algorithm 9 lines {43–46}) and every vote in a classic ballot is equal to all the learned votes concatenated with unlearned votes and new proposals (Algorithm 7 lines {42–45}) which means that new votes will be extensions of previous proven sequences. However, this begs the question of how the protocol fares when Byzantine faults occur. In particular, the worst case scenario occurs when both f acceptors and the leader are Byzantine (remember that a process can have multiple roles, such as leader and acceptor). In this scenario, the leader can purposely send *phase 2a* messages for a sequence that is not prefixed by the previously accepted values. Coupled with an asynchronous network, this malicious message can be delivered before the correct votes of the previous ballot, resulting in different learners learning sequences that may not be extensible to equivalent sequences.

To prevent this scenario, the acceptors must ensure that the proposals they receive from the leader are prefixed by the values they have previously voted for. Since an acceptor votes for its val_a sequence after receiving $N - f$ verification votes for an equivalent sequence and stores it in its *proven* variable, the acceptor can verify that it is a prefix of the leader's proposed value (i.e., $proven \sqsubseteq value$). A practical implementation of this condition is simply to verify that the subsequence of $value$ starting at the index 0 up to index $length(proven) - 1$ is equivalent to the acceptor's *proven* sequence.

5.4. Checkpointing

BGP includes an additional feature that deals with the indefinite accumulation of state at the acceptors and learners. This is of great practical importance since it can be used to prevent the storage of commands sequences from depleting the system's resources. This feature is implemented by a special command C^*, proposed by the leader, which causes both acceptors and learners to safely discard previously stored commands. However, the reason acceptors accumulate state continuously is because each new proven sequence must contain any previous proven sequence. This ensures that an asynchronous network cannot reorder messages and cause learners to learn in different orders. In order to safely discard state, we must implement a mechanism that allows us to deal with reordered messages that do not contain the entire history of learned commands.

To this end, when a learner learns a sequence that contains a checkpointing command C^* at the end, it discards every command in its *learned* sequence except C^* and sends a message to the acceptors notifying them that it executed the checkpoint for some command C^*. Acceptors stop participating in the protocol after sending *phase 2b* messages with checkpointing commands and wait for $N - f$

notifications from learners. After gathering a quorum of notifications, the acceptors discard their state, except for the command C^*, and resume their participation in the protocol. Please note that since the acceptors also leave the checkpointing command in their sequence of proven commands, every valid subsequent sequence will begin with C^*. The purpose of this command is to allow a learner to detect when an incoming message was reordered. The learner can check the first position of an incoming sequence against the first position of its *learned* and, if a mismatch is detected, it knows that either a pre and post-checkpoint message has been reordered.

When performing this check, two possible anomalies that can occur: either (1) the first position of the incoming sequence contains a C^* command and the learner's *learned* sequence does not, in which case the incoming sequence was sent post-checkpoint and the learner is missing a sequence containing the respective checkpoint command; or (2) the first position of the *learned* sequence contains a checkpoint command and the incoming sequence does not, in which case the incoming sequence was assembled pre-checkpoint and the learner has already executed the checkpoint.

In the first case, the learner can simply store the post-checkpoint sequences until it receives the sequence containing the appropriate C^* command at which point it can learn the stored sequences. Please note that the order in which the post-checkpoint sequences are executed is irrelevant since they're extensions of each other. In the second case, the learner receives sequences sent before the checkpoint sequence that it has already executed. In this scenario, the learner can simply discard these sequences since it knows that it executed a subsequent sequence (i.e., the one containing the checkpoint command) and proven sequences are guaranteed to be extensions of previous proven sequences.

To simplify the algorithm presentation, this extension to the protocol is not included in the pseudocode description.

6. Correctness Proofs

This section argues for the correctness of the Byzantine Generalized Paxos protocol in terms of the specified consensus properties (Table 1 summarizes the BGP proof notation).

Table 1. BGP proof notation.

Invariant/Symbol	Definition
$\sim$	Equivalence relation between sequences
$X \overset{e}{\Longrightarrow} Y$	X implies that Y is eventually true
$X \sqsubseteq Y$	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)
$\mathcal{B}$	Set of ballots
$\bot$	Empty command
$learned_{l_i}$	Learner l_i's *learned* sequence of commands
$learned(l_i, s)$	$learned_{l_i}$ contains the sequence s
$maj\_accepted(s, b)$	$N - f$ acceptors sent phase 2b messages to the learners for sequence s in ballot b
$min\_accepted(s, b)$	$f + 1$ acceptors sent phase 2b messages to the learners for sequence s in ballot b
$proposed(s)$	A correct proposer proposed s

6.1. Consistency

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, $\forall s, s' \in \mathcal{P}, \forall l_i, l_j \in \mathcal{L}, learned(l_i, s) \wedge learned(l_j, s') \implies (maj\_accepted(s, b) \vee (min\_accepted(s, b) \wedge s \bullet \sigma_1 \sim x \bullet \sigma_2)) \wedge (maj\_accepted(s', b') \vee (min\_accepted(s', b') \wedge s' \bullet \sigma_1 \sim x \bullet \sigma_2)), \exists \sigma_1, \sigma_2 \in \mathcal{P} \cup \{\bot\}, \forall x \in \mathcal{P}, \forall b, b' \in \mathcal{B}$

Proof: A sequence can only be learned in some ballot b if the learner gathers $N - f$ votes (i.e., $maj\_accepted(s, b)$), each containing $N - f$ valid proofs, 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., $min\_accepted(s, b)$). The first case requires gathering $N - f$ votes from each acceptor and validating that each proof corresponds to the correct ballot and value (Algorithm 10, lines {1–12}). The second case requires that the sequence must be commutative with any other and at least $f + 1$ matching values are gathered (Algorithm 10, {14–18}). This is encoded in the logical expression $s \bullet \sigma_1 \sim x \bullet \sigma_2$ which is true if the accepted sequence s and any other sequence x can be extended to an equivalent sequence, therefore making it impossible to result in a conflict.

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

Proof: We divide the following proof in two main cases: (1.2.1.) sequences s and s' are accepted in the same ballot b and (1.2.2.) sequences s and s' are accepted in different ballots b and b'.

1.2.1. At any given instant, $\forall s, s' \in \mathcal{P}, \forall b \in \mathcal{B}, maj\_accepted(s, b) \wedge maj\_accepted(s', b) \implies \exists \sigma_1, \sigma_2 \in \mathcal{P} \cup \{\bot\}, s \bullet \sigma_1 \sim s' \bullet \sigma_2$

Proof: Proved by contradiction.

1.2.1.1. At any given instant, $\forall s, s' \in \mathcal{P}, \forall \sigma_1, \sigma_2 \in \mathcal{P} \cup \{\bot\}, \forall b \in \mathcal{B}, maj\_accepted(s, b) \wedge maj\_accepted(s', b) \wedge s \bullet \sigma_1 \not\sim s' \bullet \sigma_2$

Proof: Contradiction assumption.

1.2.1.2. 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' contain non-commutative commands.

Proof: The statement $\forall s, s' \in \mathcal{P}, \forall \sigma_1, \sigma_2 \in \mathcal{P} \cup \{\bot\}, s \bullet \sigma_1 \not\sim s' \bullet \sigma_2$ is trivially false because it implies that, for any combination of sequences and suffixes, the extended sequences would never be equivalent. Since there must be some s, s', σ_1 and σ_2 for which the extensions are equivalent (e.g., $s = s'$ and $\sigma_1 = \sigma_2$), then the statement is false.

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

1.2.2. At any given instant, $\forall s, s' \in \mathcal{P}, \forall b, b' \in \mathcal{B}, maj\_accepted(s, b) \wedge maj\_accepted(s', b') \wedge b \neq b' \implies \exists \sigma_1, \sigma_2 \in \mathcal{P} \cup \{\bot\}, s \bullet \sigma_1 \sim s' \bullet \sigma_2$

Proof: To prove that values accepted in different ballots are extensible to equivalent sequences, it suffices to prove that for any sequences s and s' accepted at ballots b and b', respectively, such that $b < b'$ then $s \sqsubseteq s'$. By Algorithm 9 lines {11–16, 35, 46}, any correct acceptor only votes for a value in variable val_a when it receives $2f + 1$ proofs for a matching value. Therefore, we prove that a value val_a that receives $2f + 1$ verification messages is always an extension of a previous val_a that received $2f + 1$ verification messages. By Algorithm 9 lines {32, 43}, val_a only changes when a leader sends a proposal in a classic ballot or when a proposer sends a sequence in a fast ballot.

In the first case, val_a is substituted by the leader's proposal which means we must prove that this proposal is an extension of any val_a that previously obtained $2f + 1$ verification votes. By Algorithm 7 lines {24–39, 41–47}, the leader's proposal is prefixed by the largest of the proven sequences (i.e., val_a sequences that received $2f + 1$ votes in the verification phase) relayed by a quorum of acceptors in *phase 1b* messages. Please note that, the verification in Algorithm 9 line {27} prevents a Byzantine leader from sending a sequence that is not an extension of previous proved sequences. Since the verification phase prevents non-commutative sequences from being accepted by a quorum, every proven sequence in a ballot is extensible to equivalent sequences which means that the largest proven sequence is simply the most up-to-date sequence of the previous ballot.

To prove that the leader can only propose extensions to previous values by picking the largest proven sequence as its proposal's prefix, we need to assert that a proven sequence is an extension of any previous sequence. However, since that is the same result that we are trying to prove, we must use induction to do so:

1. Base Case: In the first ballot, any proven sequence will be an extension of the empty command $\perp$ and, therefore, an extension of the previous sequence.

2. Induction Hypothesis: Assume that, for some ballot b, any sequence that gathers $2f + 1$ verification votes from acceptors is an extension of previous proven sequences.

3. Inductive Step: By the quorum intersection property, in a classic ballot $b + 1$, the *phase 1b* quorum will contain ballot b's proven sequences. Given the largest proven sequence s in the *phase 1b* quorum (which, by our hypothesis, is an extension of any previous proven sequences), by picking s as the prefix of its *phase 2a* proposal (Algorithm 7, lines {41–47}), the leader will assemble a proposal that is an extension of any previous proven sequence.

In the second case, a proposer's proposal c is appended to an acceptor's val_a variable. By definition of the append operation, $val_a \sqsubseteq val_a \bullet c$ which means that the acceptor's new value $val_a \bullet c$ is an extension of previous ones.

1.3. For any pair of proposals s and s', at any given instant, $\forall x \in \mathcal{P}, \exists \sigma_1, \sigma_2, \sigma_3, \sigma_4 \in \mathcal{P} \cup \{\perp\}, \forall b, b' \in \mathcal{B}, (maj\_accepted(s, b) \vee (min\_accepted(s, b) \wedge s \bullet \sigma_1 \sim x \bullet \sigma_2)) \wedge (maj\_accepted(s', b') \vee (min\_accepted(s', b') \wedge s \bullet \sigma_1 \sim x \bullet \sigma_2)) \implies s \bullet \sigma_3 \sim s' \bullet \sigma_4$

Proof: By 1.2 and by definition of $s \bullet \sigma_1 \sim x \bullet \sigma_2$.

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

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.

6.2. Nontriviality

Theorem 2. *If all proposers are correct, $learned_l$ can only contain proposed commands.*

Proof:

1. At any given instant, $\forall l_i \in \mathcal{L}, \forall s \in \mathcal{P}, learned(l_i, s) \implies \forall x \in \mathcal{P}, \exists \sigma \in \mathcal{P}, \forall b \in \mathcal{B}, maj\_accepted(s, b) \vee (min\_accepted(s, b) \wedge (s \sim x \bullet \sigma \vee x \sim s \bullet \sigma))$
   Proof: By Algorithm 9 lines {16, 30, 41} and Algorithm 10 lines {1–18}, if a correct learner learned a sequence s at any given instant then either $N - f$ or $f + 1$ (if s is universally commutative) acceptors must have executed *phase 2b* for s.
2. At any given instant, $\forall s \in \mathcal{P}, \forall b \in \mathcal{B}, maj\_accepted(s, b) \vee min\_accepted(s, b) \implies proposed(s)$
   Proof: By Algorithm 9 lines {18–23}, for either $N - f$ or $f + 1$ acceptors to accept a proposal it must have been proposed by a proposer (note that the leader is considered a distinguished proposer).
3. At any given instant, $\forall s \in \mathcal{P}, \forall l_i \in \mathcal{L}, learned(l_i, s) \implies proposed(s)$
   Proof: By 1 and 2.
4. Q.E.D.

6.3. Stability

Theorem 3. *If $learned_l = s$ then, at all later times, $s \sqsubseteq learned_l$, for any sequence s and correct learner l.*

Proof. By Algorithm 10 lines {12, 18}, a correct learner can only append new commands to its *learned* command sequence. □

6.4. Liveness

Theorem 4. *For any proposal s from a correct proposer, and correct learner l, eventually $learned_l$ contains s.*

Proof:

1. $\forall\, l_i \in \mathcal{L}, \forall s, x \in \mathcal{P}, \exists \sigma \in \mathcal{P}, \forall b \in \mathcal{B}, maj\_accepted(s, b) \vee (min\_accepted(s, b) \wedge (s \sim x \bullet \sigma \vee x \sim s \bullet \sigma)) \stackrel{e}{\implies} learned(l_i, s)$
   Proof: By Algorithm 9 lines {10–15, 28–29, 41–42} and Algorithm 10 lines {1–18}, when either $N - f$ or $f + 1$ (if s is universally commutative) acceptors accept a sequence s (or some equivalent sequence), eventually s will be learned by any correct learner.
2. $\forall s \in \mathcal{P}, proposed(s) \stackrel{e}{\implies} \forall x \in \mathcal{P}, \exists \sigma \in \mathcal{P}, \forall b \in \mathcal{B}, maj\_accepted(s, b) \vee (min\_accepted(s, b) \wedge (s \sim x \bullet \sigma \vee x \sim s \bullet \sigma))$
   Proof: A proposed sequence is either conflict-free when its incorporated into every acceptor's current sequence or it creates conflicting sequences at different acceptors. In the first case, it is accepted by a quorum (Algorithm 9, lines {10–15, 28–29, 41–42}) and, in the second case, it is sent in *phase 1b* messages to the in leader in the next ballot (Algorithm 9, lines {1–4}) and incorporated in the next proposal (Algorithm 7, lines {24–47}).
3. $\forall l_i \in \mathcal{L}, \forall s \in \mathcal{P}, proposed(s) \stackrel{e}{\implies} learned(l_i, s)$
   Proof: By 1 and 2.
4. Q.E.D.

7. Conclusions 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.

7.1. 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 from our work 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.

7.2. 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 2*b* 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.

The usefulness of this optimization is severely reduced if these sequences are processed like any other, by being appended to previous sequences at the leader and acceptors. New proposals are appended to previous proven sequences to maintain the invariant that subsequent proven sequences are extensions of previous ones. Since the previous proven sequences to which a proposal will be appended to are probably not universally commutative, the resulting sequence will not be as well. We can increase this optimization's applicability by sending these sequences immediately to the learners, without appending them to previously accepted ones. This special handling has the added benefit of bypassing the verification phase, resulting in reduced latency for the requests and less traffic generated per sequence. This extension can also be easily implemented by adding a single check in Algorithm 7 lines {19–20}, Algorithm 9 lines {29–30, 40–41} and Algorithm 10 lines {14–18}.

7.3. 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 [24], or consistency models that mix strong and weak consistency levels like RedBlue [25], 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 [25].

Author Contributions: All the authors contribute equally to this work.

Funding: The research of R. Rodrigues is funded by the European Research Council (ERC-2012-StG-307732) and by FCT (UID/CEC/50021/2013).

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Lamport, L. The Part-Time Parliament. *ACM Trans. Comput. Syst.* **1998**, *16*, 133–169. [CrossRef]
2. Lamport, L. *The Part-Time Parliament*; Technical Report; DEC SRC: Maynard, MA, USA, 1989.
3. De Prisco, R.; Lampson, B.; Lynch, N.A. Revisiting the Paxos Algorithm. In *Distributed Algorithms, Proceedings of the 11th Workshop on Distributed Algorithms, Saarbrücken, Germany, 24–26 September 1997*; Springer: Berlin/Heidelberg, Germany, 1997; LNCS 1320.
4. Lee, E.K.; Thekkath, C.A. Petal: Distributed Virtual Disks. In Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems, Cambridge, MA, USA, 1–4 October 1996.
5. Burrows, M. The Chubby Lock Service for Loosely-coupled Distributed Systems. In Proceedings of the 7th Symposium on Operating Systems Design and Implementation, Seattle, WA, USA, 6–8 November 2006.
6. Junqueira, F.; Reed, B.; Serafini, M. Zab: High-performance Broadcast for Primary-backup Systems. In Proceedings of the 41st International Conference on Dependable Systems and Networks, Hong Kong, China, 27–30 June 2011.
7. Lamport, L. Paxos Made Simple. *SIGACT News* **2001**, *32*, 18–25.
8. van Renesse, R. Paxos Made Moderately Complex. *ACM Comput. Surv.* **2011**, *47*, 1–36. [CrossRef]
9. Castro, M.; Liskov, B. Practical Byzantine Fault Tolerance. In Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI), New Orleans, LA, USA, 22–25 February 1999.
10. Lamport, L. Byzantizing Paxos by Refinement. In *Distributed Computing, Proceedings of the 25th International Symposium (DISC 2011), Rome, Italy, 20–22 September 2011*; Peleg, D., Ed.; Springer: Berlin/Heidelberg, Germany, 2011; pp. 211–224.
11. Nakamoto, S. Bitcoin: A Peer-to-Peer Electronic Cash System. 2008. Available online: https://bitcoin.org/en/bitcoin-paper (accessed on 17 September 2018).
12. Lamport, L. Fast paxos. *Distrib. Comput.* **2006**, *19*, 79–103. [CrossRef]
13. Lamport, L. *Generalized Consensus and Paxos*; Microsoft ResearchTechnical Report MSR-TR-2005-33; 2005. Available online: https://www.microsoft.com/en-us/research/publication/generalized-consensus-and-paxos/ (accessed on 17 September 2018).
14. Ladin, R.; Liskov, B.; Shrira, L. Lazy replication: Exploiting the semantics of Distributed Services. In Proceedings of the 9th Symposium on Principles Distributed Computing, Quebec City, QC, Canada, 22–24 August 1990.
15. DeCandia, G.; Hastorun, D.; Jampani, M.; Kakulapati, M.; Lakshman, A.; Pilchin, A.; Sivasubramanian, S.; Vosshall, P.; Vogels, W. Dynamo: Amazon's Highly Available Key-value Store. In Proceedings of the 21st ACM SIGOPS Symposium on Operating Systems Principles (SOSP '07), Stevenson, WA, USA, 14–17 October 2007.
16. Pires, M.; Ravi, S.; Rodrigues, R. Generalized Paxos Made Byzantine (and Less Complex). In *Stabilization, Safety, and Security of Distributed Systems*; Spirakis, P., Tsigas, P., Eds.; Springer International Publishing: Cham, Switzerland, 2017; pp. 203–218.
17. Fischer, M.J.; Lynch, N.A.; Paterson, M.S. Impossibility of Distributed Consensus with one Faulty Process. *J. ACM* **1985**, *32*, 374–382. [CrossRef]
18. Mao, Y.; Junqueira, F.P.; Marzullo, K. Mencius: Building Efficient Replicated State Machines for WANs. In Proceedings of the Symposium on Operating System Design and Implementation, San Diego, CA, USA, 8–10 December 2008; pp. 369–384.
19. Moraru, I.; Andersen, D.G.; Kaminsky, M. There is more consensus in Egalitarian parliaments. In Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (Sosp '13), Farminton, PA, USA, 3–6 November 2013; pp. 358–372.
20. Lamport, L.; Shostak, R.; Pease, M. The Byzantine Generals Problem. *ACM Trans. Progr. Lang. Syst.* **1982**, *4*, 382–401. [CrossRef]

21. Dolev, D.; Dwork, C.; Stockmeyer, L. On the Minimal Synchronism Needed for Distributed Consensus. *J. ACM* **1987**, *34*, 77–97. [CrossRef]
22. Vukolic, M. *Quorum Systems: With Applications to Storage and Consensus*; Synthesis Lectures on Distributed Computing Theory, Morgan & Claypool; 2012. Available online: https://www.morganclaypool.com/doi/abs/10.2200/S00402ED1V01Y201202DCT009 (accessed on 17 September 2018).
23. Cachin, C.; Guerraoui, R.; Rodrigues, L.E.T. *Introduction to Reliable and Secure Distributed Programming*, 2nd ed.; Springer: Berlin/Heidelberg, Germany, 2011.
24. Ahamad, M.; Neiger, G.; Burns, J.E.; Kohli, P.; Hutto, P.W. Causal memory: Definitions, implementation, and programming. In *Distributed Computing*; Springer: Berlin/Heidelberg, Germany, 1995; Volume 9, pp. 37–49.
25. 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 the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI '12), Hollywood, CA, USA, 8–10 October 2012.