Generalized Paxos made Byzantine, Visigoth and Less Complex

Miguel Elias Bastos Pires

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisor(s): Prof. Rodrigo Seromenho Miragaia Rodrigues

Examination Committee

Chairperson: Prof. Mário Jorge Costa Gaspar da Silva
Supervisor: Prof. Rodrigo Seromenho Miragaia Rodrigues
Member of the Committee: Prof. Alysson Neves Bessani

November 2017

Acknowledgments

I would like to thank my supervisor Prof. Rodrigo Rodrigues and our colleague Srivatsan Ravi for their guidance and support throughout the course of this thesis. I would also like to thank my friends and family whose support helped me through my academic life.

This research is supported by the European Research Council through ERC-2012-StG-307732.

Resumo

Um dos membros mais recentes da família de protocolos *Paxos* é o protocolo *Generalized Paxos*. Esta variante tem a característica de se separar da especificação original de consenso, o que lhe permite ter uma condição de consistência mais fraca onde diferentes processos podem ter noções diferentes da sequência de comandos que está a ser concordada. No entanto, tal como o *Paxos* original, o protocolo *Generalized Paxos* não tem uma implementação simples. Para além disso, com a recente adoção prática de protocolos de tolerância a faltas Bizantinas, é relevante entender como é que o *Generalized Paxos* pode ser implementado no modelo Bizantino. O mesmo pode ser dito relativamente ao modelo *Visigoth* que tem como alvos ambientes semelhantes a *datacenters* ao permitir assunções de faltas e sincronia parametrizáveis. Esta dissertação faz várias contribuições. Em primeiro lugar, fornecemos uma descrição do protocolo *Generalized Paxos* mais fácil de entender, baseada numa especificação de consenso simplificada, juntamente com uma especificação em pseudocódigo que pode ser prontamente implementada. Em segundo lugar, estendemos o protocolo para o modelo de faltas Bizantino, fornecendo também uma descrição em pseudocódigo para facilitar o seu mapeamento numa linguagem de programação. Esta contribuição é suplementada com provas de correção e uma discussão de extensões e otimizações relevantes. Em terceiro lugar, expandimos esta implementação para o modelo de faltas *Visigoth*, fornecendo também uma descrição acessível em pseudocódigo e provas de correção.

Palavras-chave: *Paxos*, Consenso, Tolerância a faltas, Modelo de Faltas Bizantinas

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 of a sequence being agreed upon. However, much like its 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. The same point can be made with respect to the Visigoth model which targets datacenter-like environments by allowing parameterizable fault and synchrony assumptions. This dissertation makes several contributions. First, we provide a description of Generalized Paxos that is easier to understand, based on a simpler specification of consensus, along with a pseudocode specification that can be readily implemented. Second, we extend the protocol to the Byzantine fault model, providing also a pseudocode description to ease its mapping into a code implementation. This contribution is supplemented with correctness proofs and a discussion of relevant extensions and optimizations. Third, we extend this implementation to the Visigoth fault model, providing with it an accessible pseudocode description and correctness proofs.

Keywords: Paxos, Consensus, Fault Tolerance, Byzantine Fault Model

Contents

List of Tables

List of Figures

Acronyms

ACID Atomicity, Consistency, Isolation, Durability. 22

ASC Arbitrary State Corruption. 14, 15

BFT Byzantine Fault Tolerance. 13–16, 39, 41

BGP Byzantine Generalized Paxos. 4, 5, 12, 40, 45, 48, 49, 52, 53, 60, 61, 63, 66, 69, 70, 78, 79

CFT Crash Fault Tolerance. 14, 16, 29, 30, 39, 40, 49, 52, 78, 79

DNS Domain Name System. 22

EPaxos Egalitarian Paxos. 10, 11

FaB Fast Byzantine. 5, 16, 53

FIFO First In, First Out. 21

FLP Fischer, Lynch and Patterson. 1, 7, 13

MDCC Multi-Data Center Consistency. 10, 79

PBFT Practical Byzantine Fault Tolerance. 12, 14–16

QGP Quorum Gathering Primitive. 64, 66, 67, 69

RDBMS Relational Database Management System. 22

SMR State Machine Replication. 1, 11, 12, 15, 16, 21

VFT Visigoth Fault Tolerance. 13, 14, 16, 59, 60, 64, 66, 67, 69, 70, 78, 79

VGP Visigoth Generalized Paxos. 4, 5, 60, 61, 64, 69, 70, 79

WAN Wide Area Network. 16

WHEAT Weight-Enabled Active Replication. 16

XFT Cross Fault Tolerance. 14, 16, 78

Zab ZooKeeper Atomic Broadcast. 21

Chapter 1

Introduction

1.1 Motivation

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 [1]. Despite being a theoretical problem, its solution is critical to many modern large-scale systems since it allows data to be replicated among distributed processes. Systems such as Google's Chubby [2], Spanner [3] and Apache's ZooKeeper [4] use techniques like State Machine Replication (SMR) [5, 6] to allow them to remain highly available even in the presence of faults and asynchronous communication channels. SMR implements a fault tolerant system by modeling its processes as state machines that must receive the same inputs, execute the same state transitions and output the same results. Usually, to ensure that every state machine transitions to the same states, consensus is used to guarantee that inputs are processed in the same order. This technique is widely used to implement fault tolerant services and it's one of the reasons why consensus is so important.

One of the most important contributions to this field is the Fischer, Lynch and Patterson (FLP) impossibility result that states that consensus is unsolvable in an asynchronous system even if only one process can fail [7]. Lamport's Paxos algorithm is able to solve consensus, circumventing the FLP impossibility result, by ensuring that values are always safely decided while progress is only guaranteed when the system is synchronous for a sufficient amount of time [8]. In other words, Paxos overcomes the FLP result by weakening the liveness condition since it can't guarantee that correct processes will decide a value in a finite number of steps. In Paxos, there are 3 types of processes: *proposers*, which propose values to be committed; *acceptors*, which vote on proposed values; and *learners*, that learn values voted on by a quorum of acceptors.

Paxos [9], 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 [9]. 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 [11] and the distributed systems [12]

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 a key component of Google's Chubby lock service [2] and Megastore storage system [13], and also of the open source ZooKeeper project [4], 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 [8, 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, 16]. 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 the blockchain [17]. Second, many proposals target improving the Paxos protocol by eliminating communication costs [18], including an important evolution of the protocol called Generalized Paxos [19], which has the noteworthy aspect of having lower communication costs by leveraging a more general specification than traditional consensus that can lead to a weaker requirement in terms of ordering of commands across replicas. In particular, instead of forcing all processes to agree on the same value (as with traditional consensus), it allows processes to pick an increasing sequence of commands that differs from process to process in that commutative commands may appear in a different order. The practical importance of such weaker specifications is underlined by significant research activity on the corresponding weaker consistency models for replicated systems [20, 21].

Generalized consensus is a generalization of traditional consensus that abstracts the problem of agreeing on a single value to a problem of agreeing on a monotonically increasing set of values. This problem is defined in terms of a set of values called command structures, *c-structs* [19]. These structures allow for the formulation of different consensus problems, including specifications where commutative operations are allowed to be ordered differently at different replicas (i.e., command histories). The advantage of such a problem becomes clear when considering the optimization proposed in a protocol called Fast Paxos, where fast ballots are executed by having proposers propose directly to acceptors [18]. By avoiding sending the proposal to the leader, values can be learned in the optimal number of two message delays. However, if two proposers concurrently propose different values to acceptors, a conflict arises and at least an additional message delay is required for the leader to solve it. This is the cost that Fast Paxos pays in order to commit values in a single round trip. Generalized consensus allows us to reduce this cost, if we define the problem as one of agreeing on command histories. Since histories are considered equivalent if non-commutative operations are totally ordered, the only operations that force the leader to intervene are non-commutative ones. An additional advantage of the generalized consensus formulation stems from its generality and from the fact that we can use the Generalized Paxos protocol to solve it despite its high level of abstraction. This protocol can be used to solve any consensus problem that can be defined in terms of generalized consensus, not only the command history problem. The reason why Generalized Paxos can take advantage of the possibility of reorder-

ing commutative commands is that it allows acceptors to accept different but compatible *c-structs*. Two *c-structs* are considered to be compatible if they can later be extended to equivalent *c-structs*. In command histories, if all non-commutative commands are totally ordered, then two *c-structs* are considered equivalent.

One application of the generalized consensus specification can be to implement SMR using command histories to agree on equivalent sequences of operations. For instance, consider a system with four operations $\{A, B, C, D\}$ where C and D are non-commutative. If two proposers concurrently propose the operations A and B, some acceptors could accept A first and then B and other acceptors could accept the operations in the inverse order. However, this would not be considered a conflict and the leader wouldn't have to intervene since the operations commute. If two proposers tried to commit C and D, acceptors could accept them in different orders which would be considered a conflict since these operations are non-commutative. In this situation, no *c-struct* would be chosen and the leader would be forced to intervene by initiating a higher-numbered ballot to commit either $w \bullet C \bullet D$ or $w \bullet D \bullet C$. It's important to note that this is only one possible application of the Generalized Paxos protocol and that this protocol solves any problem that can be defined by the generalized consensus specification.

Despite generalized consensus' potential, it's still an understudied problem and its formulation is rather complex and abstract which makes it hard to understand and reason about. This complexity also makes the algorithm hard to implement and adapt to different scenarios. There are several symptoms of this complexity. One of them is that only the original Generalized Paxos protocol exists for this problem and few works make use of it. Another consequence of the lack of knowledge about generalized consensus is that there are several potentially interesting research questions that researchers haven't answered. For instance, despite the connection between the commutativity observation that motivated generalized consensus and the reduced coordination requirements made possible by weak consistency, it is unclear how Generalized Paxos would function in geo-replicated scenarios where the goal is to minimize cross-datacenter round trips. Similarly, there is not much research on what effects the Byzantine assumption would have on the solution of the generalized consensus problem.

1.2 Contributions

The goal of this work is to perform a thorough study of the generalized consensus problem to gain a deeper knowledge about the applicable protocols, such as Generalized Paxos, and how they behave in different scenarios. One of the greatest barriers in the comprehension and adoption of Generalized Paxos is the complexity of its description which, in turn, is caused by a very generic specification of consensus. Much in the same way that the clarification of the Paxos protocol contributed to its practical adoption, it's also important to simplify the description of Generalized Paxos.

Furthermore, we believe it's also relevant to extend this protocol to non-crash fault models, such as the Byzantine and the Visigoth fault models, since it will open the possibility of adopting Generalized Paxos in different scenarios. In particular, the Byzantine fault model has recently gained traction in the blockchain community given the rise in popularity of cryptocurrencies like Bitcoin [17]. The Visigoth fault

model targets environments like datacenters where a large number of servers are connected through a network with high security barriers, which makes it both unlikely that multiple processes will act maliciously in a coordinated way and also likely that arbitrary behavior will stem from state corruption faults due to the sheer number of components in the datacenter [22]. In the Visigoth model, fault and synchrony assumptions are parameterizable in order to allow for any amount of synchrony, ranging from full asynchrony to full synchrony, and any amount of Byzantine or crash faults, ranging from strictly crash to fully Byzantine. This allows the system administrator to parameterize the model to fit the network's characteristics which are more likely to be predictable in a datacenter. This model allows us to study command history consensus from different perspectives and propose a solution that can solve this problem across a broad spectrum of system models. This is an important aspect because, although it adds complexity, it also adds the ability to develop a generic protocol that can be used in different environments. This can be seen as removing some generality in the problem specification while retaining the original motivating scenario while, at the same time, generalizing the fault model.

Concretely, this work makes the following contributions:

- A simplified version of generalized consensus, which preserves its motivating scenario of agreeing on command histories;
- a protocol derived from Generalized Paxos that solves the aforementioned consensus problem while improving the original protocol's understandability and ease of mapping into a code implementation;
- the Byzantine Generalized Paxos (BGP) protocol, which is an extension of Generalized Paxos to the Byzantine model, with a description that is more accessible than the original, including pseudocode and correctness proofs;
- the Visigoth Generalized Paxos (VGP) protocol, which is an extension of Byzantine Generalized Paxos to the Visigoth model, with an accessible description, complete with pseudocode and correctness proofs;
- several extensions and optimizations to the previous protocols.

1.3 Document Outline

The remainder of this document is structured as follows: Chapter 2 is divided in five subsections and surveys works that are related to our own and may provide relevant insights into the problem we're trying to solve. Each subsection describes scientific works in a specific area of interest to us. Chapter 3 has a mainly pedagogical purpose. It describes the original generalized consensus problem as well as components of Generalized Paxos that are vital for the functioning of the protocol but whose reasoning can be opaque to the reader. Chapter 4 describes a simplified version of generalized consensus and proposes a protocol to implement its solution. Extensions to the protocol are also discussed along with possible scenarios in which they may be helpful. Chapter 5 adapts the simplified consensus problem

to the Byzantine fault model and presents its solution, Byzantine Generalized Paxos. Similarly to its counterpart in the crash fault model, we discuss extensions to the protocol as well as how it differs from the most similar protocol in the literature, Fast Byzantine (FaB) Paxos [15]. Chapter 6 presents a solution, Visigoth Generalized Paxos, for the consensus problem defined for the Byzantine fault model but taking into account the Visigoth model's parameterizable fault and synchrony assumptions. Both Chapter 5 and Chapter 6 also present correctness proofs for their respective protocols with respect to our proposed Byzantine command history consensus problem. Chapter 7 concludes this work by discussing what was learned and what unexplored avenues of research are left for future work.

Chapter 2

Related Work

Due to the specific characteristics of both the problem that we wish to solve and the models in which we propose to solve it, there are several areas that contain works of interest to us. Section 2.1 describes previous work regarding consensus and several Paxos variants. Section 2.2 discusses non-crash fault models as well as protocols that solve consensus in these environments. Section 2.3 describes different partial synchrony models and how they fit in the synchrony spectrum. Section 2.4 discusses variable consistency models that attempt to strike a balance between strong and weak consistency in order to obtain advantages from both. Despite the fact that the Paxos protocol family has been a subject of study for some time, this topic is now more relevant than ever since these algorithms have become widely used by the industry to implement large-scale, highly available services [2, 4]. Section 2.5 studies a few systems where these algorithms are used to enable fault-tolerance services.

2.1 Paxos and its Variants

Classic Paxos. The Paxos protocol family solves consensus by circumventing the well-known FLP impossibility result [7]. It does this by making the observation that, while consensus is unsolvable in asynchronous systems, most of the time systems can be considered synchronous since delays are sporadic and temporary. Therefore, as long as consistency is guaranteed regardless of synchrony, Paxos can forgo progress during the temporary periods of asynchrony. Using this insight, Paxos ensures consistency even when the system is asynchronous but can only guarantee progress while it is synchronous and no more than f faults occur for a system of $2f + 1$ replicas [8]. The classic form of Paxos employs a set of proposers, acceptors and learners and runs in a sequence of ballots. To ensure progress during synchronous periods, proposals are serialized by a distinguished proposer, the leader. It is the leader that gathers proposals from other proposers and requests, in a *phase 1a* message, that acceptors promise to not accept any ballots smaller than its own. Each acceptor responds to the leader's request, in a *phase 1b* message, with either a promise or a message stating that it has already sent a promise to a proposer with a higher-numbered ballot. If the leader manages to obtain a promise from a quorum of acceptors, then it sends a *phase 2a* message containing its proposal. The acceptors

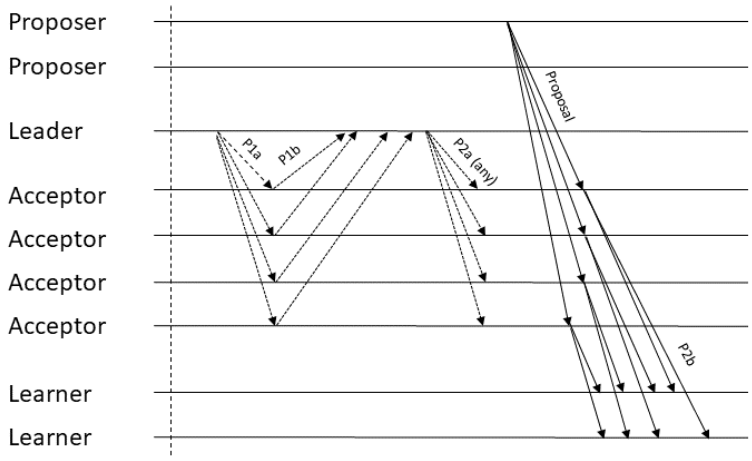

Figure 2.1: Fast Paxos' fast ballot message pattern

may accept the proposal and inform the learners with a *phase 2b* message, if another proposer hasn't requested a promise for a higher-numbered ballot in the meanwhile. The fact that any proposer can request a promise and prevent lower ballots from being completed is the reason why Paxos can only ensure progress when an environment is in a synchronous period. Some proposer must be able to establish itself as the only leader in order to prevent proposers from continuously requesting promises for ballots with higher numbers.

Multi-Paxos. We now describe the form in which Paxos is most commonly deployed, Multi(Decree)-Paxos. Multi-Paxos provides an optimization of the basic Paxos' message pattern by omitting the first phase of messages from all but the first ballot for each leader [14]. 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 without requesting a promise from the acceptors. 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.

Fast Paxos. Fast Paxos observes that it's possible to improve on previous results by allowing proposers to propose values directly to acceptors [18]. 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 fast ballots' reduced latency 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 to 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. However, we can also choose to maximize f instead of e and the classic quorum becomes a majority quorum while a fast quorum must be larger than $\lceil \frac{3N}{4} \rceil$. This configuration is interesting since we can tolerate the largest possible number of faults but fast ballots are only possible if a larger number of replicas are correct. The optimized commit scenario occurs during fast ballots, in

which only two message broadcasts are necessary: *proposal* messages between a proposer and the acceptors, and *phase 2b* messages between acceptors and learners. This is shown in Figure 2.1, where dotted lines represent messages that only need to be exchanged once per ballot and solid lines represent messages that can be exchanged more than once. This message pattern creates the possibility of two proposers concurrently proposing values to the acceptors and generating a conflict. The cost of dealing with this conflict depends on the chosen method of recovery, coordinated or uncoordinated, and if the leader or acceptors are also learners. The leader may choose the recovery method in the beginning of the ballot. If coordinated recovery is chosen in a ballot i, then the acceptors send *phase 2b* messages both to the learners and to the leader, who performs the conflict detection. After a conflict is detected, the leader picks a value and sends it in a *phase 2a* message to the acceptors. The cost of recovery with this method is equivalent to a *phase 2* of round $i+1$ (i.e., two additional message delays). If uncoordinated recovery is chosen, the acceptors also broadcast *phase 2b* messages between each other and, if necessary, pick a value is some deterministic way and send it in a *phase 2b* message to the learners. The cost of this approach is a single message in round $i+1$ (i.e., one message delay) but more messages have to be exchanged than in coordinated recovery.

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 [19]. 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 wouldn't 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.

MDCC. Multi-Data Center Consistency (MDCC) is an optimistic commit protocol that uses Generalized Paxos for transaction processing. MDCC uses fast commutative ballots to commit several commutative updates in the same ballot with potentially different orders across storage nodes [23]. To preserve global constraints (e.g., the stock should never go below zero), a demarcation technique is used to limit the amount of transactions that can be committed in a ballot. MDCC is one of the few works that takes advantage of Generalized Paxos to concurrently commit commutative operations.

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 [24]. 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* that will eventually be accepted as long as l isn't permanently suspected. The usage of an unreliable failure detector is enough to guarantee that eventually all faulty processes, and only those, will be 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 isn't a dependence relationship between them. Experimental evaluation confirms that Mencius is able of scaling to a higher throughput than Paxos.

EPaxos. Egalitarian Paxos (EPaxos) extends Mencius' goal of achieving a better throughput than Paxos by removing the bottleneck caused by having a leader. Additionally, EPaxos also achieves optimal commit latency in certain conditions and graceful performance degradation if replicas fail or become slow [25]. To avoid choosing a leader, the proposal of commands for a command slot is done in a decentralized manner. Each replica can propose a command with ordering constraints attached so that interfering commands can be totally ordered. In this way, EPaxos takes advantage of the commutativity observations made by Generalized Paxos that state that commutative commands may be ordered differently at different replicas without losing state convergence [19]. 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 commutative then there is no need to impose a particular order but, if they're not commutative, then the replica must reply with a dependency between the commands. If a replica commits a command A in a single round trip, then the replica that tries to concurrently propose a non-commutative command B would receive a reply with the ordering $B \rightarrow \{A\}$, to indicate that B should be committed after A. In this scenario, committing

command B would require two round trips since the proposer would need to send another message to inform the other replicas of the new ordering constraint. This commit latency is achieved by using a *fast-path quorum* of $f + \lfloor \frac{f+1}{2} \rfloor$ replicas which is the same as the *slow-path quorum* of $f + 1$ replicas, for $f \leq 2$. Similarly to Mencius, EPaxos achieves a substantially higher throughput than Multi-Paxos, especially when 2% of commands interfere with each other. However, since, unlike Mencius, replicas don't have to wait for previous replicas to propose their commands, commit latency can be significantly lower.

Raft. Raft is a consensus protocol that manages a replicated log which is used to implement SMR among replicas [26]. This protocol is similar to Paxos in terms of the performance and assurances that it provides but has the additional goal of striving for better understandability. To this end, unlike Paxos, Raft separates leader election, log replication and safety into distinct components. The leader uses a heartbeat mechanism to inform the *followers* that it hasn't crashed. If a follower doesn't receive a message from the leader for a period greater than the *election timeout*, it increments the *term*, begins an election and transitions to the *candidate* state. In this state, the replica votes for itself and requests votes from other processes. A candidate wins the election and becomes the leader if it receives votes from a majority of replicas in the cluster. To ensure that eventually some *candidate* is elected as the leader, election timeouts are randomized to reduce the probability that multiple servers will timeout at the same time and divide the votes. Log replication is performed by having the leader service client requests, which contain commands to be executed by the state machines. The leader appends each command to its log and requests that followers do the same. When the leader sends a new entry to the followers, it also sends the corresponding index and term so that a follower can check if it's missing commands before adding the new one to its log. This consistency check ensures that the logs match during normal operation. However, a leader can fail before fully replicating new entries to the followers, which can cause divergence between the followers' and the new leader's logs. To reintroduce consistency between the two logs, the leader finds the latest entry in which both agree, deletes later entries in the follower's log and sends the entries that it has after that point. However, the restrictions and checks described so far are not enough to guarantee safety, in particular because a follower might not write a leader's entries, then be elected as the leader and overwrite the followers' logs. To prevent this from happening, Raft restricts the candidates that can be elected as the leader by ensuring that the new leader's log contains every committed entry. This is done by including information about the candidate's log when requesting votes. If the voter's log is more up-to-date than the candidate's log, the voter refuses to vote for it. Since the leader must obtain a majority of votes and an entry is only committed after also gathering a majority of votes, the leader is sure to contact at least one process that has committed the latest value.

Viewstamped Replication. Viewstamped Replication is a replication technique based on a primary-backup scheme where the system is composed of multiple groups that contain several *cohorts* [27]. In each group, one cohort is the primary and the remaining are the backups. If the primary crashes, a view-change protocol is executed and one of the backup replicas becomes the new primary. The system makes progress by having client groups create transactions composed of several procedure calls and making those calls to server groups. When a transaction terminates, a two-phase commit protocol

is executed to replicate the corresponding changes. The client's primary acts as the coordinator for the commit protocol and the servers' primaries act as participants. Events such as the processing of procedure calls or prepare messages are written to the communication buffer as records (and possibly flushed to the backups), to ensure that the committed transactions are not lost. Viewstamped Replication also introduces *viewstamps* which are the concatenation of *timestamps*, which are unique within a view, and *view identifiers*. Each replica in the system maintains viewstamps in a sequence, the *history*, for each event that is reflected in its state. These viewstamps are used to determine if, after the failure of a primary, the effects of an on-going transaction survived and it can commit or if it must be aborted.

Even though some Paxos variants attempt to improve on the original protocols by removing the bottleneck caused by having a single leader or by shortening the number of message steps in the common case, few works align these improvements with non-crash fault models. Our Byzantine Generalized Paxos protocol attempts to bridge the gap between these two types of extensions to consensus-solving protocols in order to make their usage more viable within datacenters where arbitrary faults occur.

2.2 Protocols for Non-Crash Fault Models

Non-crash fault models emerged to cope with the effect of malicious attacks and software errors. These models (e.g., the arbitrary fault model) assume a stronger adversary than previous crash fault models. 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 [28]. Like the traitorous generals, the Byzantine adversary is one that may force a faulty replica to display arbitrary behaviour and even coordinate multiple faulty replicas in an attack. It's interesting to survey non-crash fault models since we intend to explore the effects that different models have on the solutions for the generalized consensus problem.

PBFT. Practical Byzantine Fault Tolerance (PBFT) is a protocol that solves consensus for SMR while tolerating up to f Byzantine faults [29]. 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 doesn't contain the client's request message. Both of these phases are needed to ensure that the requested operation is totally ordered at every correct replica. 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 doesn't increase continuously. This property represents a weak liveness condition but one that is enough to circumvent the FLP impossibility result [7]. 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.

Several other Byzantine Fault Tolerance (BFT) protocols have been proposed which extend the initial protocol in different directions, such as the use of trusted components in order to reduce the replication factors [30, 31], finding new building blocks to easily construct new Byzantine protocols [32], or weakening the fault model while still supporting arbitrary faults in order to reduce the replication factor and quorum size. From these extensions the ones that are more directly related to this thesis are those that target data center environments and geo-replicated systems, where heightened security makes coordinated Byzantine behavior unlikely but the large amounts of equipment increase the occurrence of arbitrary faults due to hardware failures [22, 33]. This is due to our initial motivation of developing protocols that target geo-replicated systems hosted across multiple datacenters, in which not only faults are a common occurrence but there is also a need for reducing hardware and power costs.

VFT. The creators of Visigoth Fault Tolerance (VFT) also make the observation that the Byzantine model is too pessimistic, especially for environments such as data centers [22]. The Byzantine adversary's strength forces protocols to incur in an unnecessary cost of $3f + 1$ replicas. VFT also notes that while data centers rarely observe arbitrary coordinated faults, they commonly experience data corruption faults (e.g., bit flips) and these also represent a type of arbitrary behaviour [34, 35]. To reconcile the need to tolerate different types of arbitrary faults, VFT observes that it is very unlikely for data corruption to affect the same data across multiple replicas. Therefore, the key distinguishing factor between these types of arbitrary faults is *correlation*. VFT proposes a customizable model that defines a limit of u faults, that bounds the number of allowed omission (i.e., crash) and comission (i.e., arbitrary) faults, a limit of o correlated comission faults, a limit of r arbitrary faults and, for every process p_i, s correct processes p_j that are slow with respect to it. These additional assumptions allow the Visigoth model to decrease the replication factor from $N \geq 2u + r + 1$ (i.e., $3f + 1$ in the traditional notation) to $N \geq u + s + o + 1$, when $u > s$. These parameters allow the system administrator to configure the system to tolerate any combination of faults. The authors also propose a VFT state machine replication protocol, which is adapted from the BFT-SMaRt protocol [36] and contains two main message patterns: the first consists in two message steps that are executed during epoch changes to collect information about previous epochs and disseminate this information to replicas; and the second consists in two multicast phases that are executed for each client request to implement the replicated state machine. To gather a quorum, the gatherer tries to collect a larger quorum of $N - s$ replies, which can be seen as collecting replies from the fast but potentially faulty replicas. If a timeout occurs, the quorum is reduced to $N - u$ and this can be seen as collecting replies from correct but possibly slow replicas. This reduced quorum size is enough to ensure an intersection because, since x replicas didn't reply by the first timeout and only s may be slow, $x - s$ replicas must have crashed and won't participate in future quorums. The experimental evaluation

confirms that VFT can tolerate uncorrelated arbitrary faults with resource-efficiency and performance that resembles that of CFT systems instead of BFT.

XFT. The cost of $3f + 1$ replicas in a PBFT system stems from the immense power given to the adversary. The Cross Fault Tolerance (XFT) model claims that this adversary is too strong when compared to the behavior of deployed systems [33]. XFT assumes that machine and network faults are uncorrelated and this assumption allows it to have a cost equal to crash fault tolerant systems, $2f + 1$, while still tolerating f non-crash faults. The safety condition for this model is that correct replicas commit requests in a total order, as long as the system is not in anarchy. A system is in anarchy if there are any non-crash faults f_{nc} and the sum of crash faults f_c, non-crash faults f_{nc} and partitioned replicas f_p is larger than the fault threshold. That is, the system is in anarchy at a given moment s if and only if $f_{nc}(s) > 0 \ \wedge \ f_{nc}(s) + f_c(s) + f_p(s) > f$. Liveness is also guaranteed as long as the system is outside of anarchy. XFT replicates client requests to $f + 1$ replicas synchronously while the remaining f may learn about requests through lazy replication [37]. If an active replica fails, a view change is triggered. A view change requires all the replicas in the new view to collect the recent state, including the commit log, which may be substantial. This overhead can be decreased through the use of checkpointing. The protocol includes the exchange of four rounds of messages: SUSPECT, VIEW-CHANGE, VC-FINAL and NEW-VIEW. The experimental evaluation performed by the authors shows that, after each crash, the system triggers a view change that last less than 10 seconds (note that this value is dependent on network quality and other details).

Self-stabilizing Systems. Dijkstra proposed self-stabilizing systems as a solution to the problem of keeping cyclic sequences of processes (i.e., processes connected to two neighbors, forming a ring) in a legitimate state even after the occurrence of transient faults [38]. In this self-stabilizing system, each process may contain several privileges and may only exchange information with its neighbors. A privilege is a boolean function on the state of the machine and the state of its neighbors. The requirement of keeping the system in a legitimate state is met if: (1) one or more privileges are present; (2) in each state, the possible moves transfer the system to a legitimate state again (3) each privilege exists in at least one legitimate state; and (4) for any pair of legitimate states, there is a sequence of moves that transfers the system from one to the other. These systems define a set of rules for transferring privileges in a way such that a system that starts from an arbitrary state is guaranteed to converge to a legitimate state and remain in a set of legitimate states. Self-stabilizing systems are an example of a non-crash model that isn't strictly stronger than crash fault models since, due to the system's ring-like structure, a crash fault would render the system unusable. A self-stabilizing system tolerates any arbitrary corruption of the state but, unlike other non-crash models, it assumes that a process always executes correct code.

ASC. The Arbitrary State Corruption (ASC) model defines ASC faults as faults that occur at a process p and either make it crash or assign an arbitrary value to a variable of the process's state. The assignment of an arbitrary value to a variable of p's state may also alter its control flow in a way such that the process may execute any of its instructions [39]. This modeling stems from three main observations: (1) most tolerable failures are caused by state corruptions instead of malicious behavior, (2) control-flow faults are very likely to cause incorrect outputs [40], and (3) the relevant failures are caused by transient

faults instead of permanent faults being repeatedly activated. In the context of this model, the ASC hardening technique is proposed to deal with ASC faults. This technique guarantees that, if a correct hardened process p_c receives a message m from a faulty hardened process p_{f1}, then p_c discards m without modifying its state. If a faulty process p_{f2} receives m and modifies its local state, then it crashes before sending an output message. ASC hardening uses message integrity checks to prevent network corruption faults and state integrity checks to replicate process state. Every time a value is read from a variable, it's checked against the replicated value. If a mismatch occurs, the process crashes itself to preserve integrity. Control-flow faults are prevented through the use of gates, which are sequences of instructions that ensure that a portion of code is executed the appropriate number of times. For instance, we may want to ensure that an event handler executes exactly once per event. For each control-flow gate, two variables and a label are created, respectively, c, c' and L. A gate executes a sequence of checks and assignments to ensure the required semantics. A useful type of control-flow gate is an *at-most-once* gate, which ensures that code isn't executed twice even in the presence of faults. An *at-most-once* gate implements these semantics by crashing the process if $c \neq c' \vee c = L$. In the first execution of the code, the condition fails, both c and c' are set to L and the execution continues. In a second execution, the condition would evaluate to true since c has already been set to L and the process would crash. The $c \neq c'$ condition prevents an arbitrary state fault from corrupting the variable's value. Another relevant type of control-flow gate is an *at-least-once* gate, which ensures that a gate has been executed before. For an *at-least-once* gate, if $c = c' = L$ the gate has been reached before. Otherwise, the process crashes itself. ASC is similar to self-stabilizing systems [38] in the sense that both assume that only state can become corrupted, while the executed code is always correct. An ASC-hardened implementation of Paxos has a throughput up to 70% higher than PBFT.

Zeno. Zeno is a Byzantine fault tolerance SMR protocol that favors availability instead of consistency [41]. By trading strong consistency for eventual consistency, Zeno provides higher availability when network partitions occur and better performance when replica latency is heterogeneous. A client request is said to be *weakly complete* if the client gathers matching replies from a *weak quorum* of $f+1$ replicas. This ensures the client that at least one correct replica will execute the request and commit it to the linear history. For a request to be *strongly complete*, the client must wait for matching replies from a *strong quorum* of $2f + 1$ replicas. In the case of a partition, two clients concurrently issuing requests may cause divergent histories that can later be merged to restore consistency between replicas. When a replica receives a client request, it performs the appropriate security checks (e.g., message authenticity, correct view, etc) and executes the requested operation. If the request was for a weak operation, the replica replies to the client immediately after the operation terminates. However, if the request was for a strong operation, the replica multicasts a *COMMIT* message to the other replicas and waits for $2f$ matching messages. After receiving $2f$ *COMMIT* messages, the replica forms a commit certificate for the messages it received, stores it and replies to the client. To further increase availability, Zeno's view change protocol is also guaranteed to proceed with only a weak quorum of replicas. Strongly consistent BFT systems require view change protocols to gather a strong quorum to ensure that a view change quorum intersects with any other quorum in at least one correct replica. If this property isn't ensured,

another quorum could commit a request that would be unseen by the view change. In Zeno, a request can go unnoticed by view change quorum, resulting in divergent histories and a subsequent merge.

WHEAT. Weight-Enabled Active Replication (WHEAT) is a WAN-optimized SMR protocol implemented on top of BFT-SMaRt, a BFT consensus protocol with a message pattern similar to PBFT [42, 36]. WHEAT employs a number of optimizations that were empirically shown to produce the most results in improving the latency of SMR in wide-area networks. The first two optimizations, *read-only* and *tentative executions*, were introduced in PBFT with the goal of reducing the number of communication steps [29]. Since read-only operations don't modify the service's state, replicas can execute a request immediately after checking its authenticity and verifying that it is indeed read-only. The reply to the client can be sent after other requests reflected in the tentative state have committed. Tentative executions allow a replica to execute a request tentatively after it has been prepared by a Byzantine quorum, instead of locally committed. This means that the request can be executed after the first all-to-all round instead of the second. However, both optimizations require the client to wait for $2f + 1$ replies instead of $f + 1$. The authors also determine that it's possible to reduce latency by adding replicas to the system while maintaining the quorum size. However, this isn't possible with traditional quorum gathering due to safety violations. To take advantage of this optimization without sacrificing safety, WHEAT uses weighted replication where, instead of requiring a quorum to access a majority of replicas, a quorum must obtain a majority of votes. The optimization is enabled by assigning a higher number of votes to replicas with better connectivity or performance. For instance, in a CFT system of four replicas, three could have a weight of 1 and the fourth could have a weight of 2. To achieve a majority vote, a quorum would have to obtain at least 3 votes, which can be done by gathering votes from three replicas with a weight of 1 or from two replicas, one with a weight of 1 and another one with a weight of 2. The same reasoning applies in BFT mode with the difference that a quorum would have to reach a Byzantine majority of votes. The value of these optimizations is validated by the experimental evaluation which reports a 37% and 58% global median improvement in latency in the BFT and CFT models, respectively.

FaB Paxos. Perhaps the most closely related work is Fast Byzantine (FaB) Paxos, 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 [15]. A variant that is proposed in the same paper is the Parameterized FaB Paxos protocol, which generalizes FaB Paxos 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 Paxos protocol is tight since it requires $5f + 1$ acceptors to guarantee 2-step execution while preserving both safety and liveness.

Protocols like VFT and XFT note that the Byzantine model is overtly pessimistic when compared to the actual characteristics of datacenter environments and use that observation to reduce the cost imposed by fault tolerance [22, 33]. Similarly, WHEAT uses optimizations like weighted voting schemes to take advantage of replicas with better performance or connectivity to reduce the cost of committing [42].

However, not many Byzantine fault tolerant protocols have tried to align themselves with these goals while, at the same time, using Generalized Paxos' contributions to achieve lower synchronization requirements. Since the contributions from the first set of protocols are more aligned with the modification of the fault model and the contributions presented by Generalized Paxos stem from increased assumptions at the problem specification, there is the potential to merge them in a protocol that takes advantage of both.

2.3 Partial Synchrony

The design of the Paxos protocol is based on the observation that, although it's possible to guarantee safety in a fully asynchronous environment, a system can only ensure liveness when the network behaves synchronously for a sufficient amount of time [8]. It's common to encapsulate this reasoning about timeliness in an abstraction called a failure detector. Paxos relies on an *eventual* failure detector Ω that abstracts the notion that, for Paxos to progress, eventually every correct process must agree on which correct process is the leader [1]. Therefore, it makes sense for us to study the effects that different synchrony assumptions encapsulated in these failure detectors have on the consensus problem and on the algorithms that solve it. Additionally, the Visigoth model allows for a configurable amount of synchrony to be tolerated by the system, which makes it even more relevant for us to study its effects [22].

Ω-accurate Failure Detectors. As previously mentioned, a failure detector is a formalism that allows us to abstract the assumptions a system is allowed to make about its environment. Unreliable failure detectors were proposed by Chandra and Toueg as a way to circumvent impossibility results due to asynchrony. These failure detectors would output information about which processes have crashed, therefore encapsulating the necessity of determining whether a process is crashed or slow [43]. The guarantees provided by the failure detector are specified in terms of *completeness*, which requires that a failure detector must eventually suspect every crashed process, and *accuracy*, which restricts the mistakes a failure detector can make. A class of failure detectors of particular interest is Eventually Weak failure detectors $\Diamond\mathcal{W}$, since it is the weakest of all classes defined by Chandra and Toueg. $\Diamond\mathcal{W}$ failure detectors provide the following guarantees:

- **Eventual Weak Completeness** – there is a time after which every process that crashes is permanently suspected by some correct process.
- **Eventual Weak Accuracy** – there is a time after which some correct process is never suspected by any correct process.

Chandra and Toueg prove that consensus is solvable using a $\Diamond\mathcal{W}$ failure detector with $f < \lceil n/2 \rceil$ [43]. Chandra et al. also define a new class of failure detectors, Ω-accurate failure detectors [44]. The Ω class is shown to be at least as strong as $\Diamond\mathcal{W}$ and any failure detector that allows consensus to be solved must be at least as strong as Ω. Therefore, any failure detector that allows consensus to be solved is at least as strong as $\Diamond\mathcal{W}$ which is then the weakest possible class of failure detector with which consensus can be solved.

Γ-accurate Failure Detectors. The accuracy properties of the failure detectors proposed by Chandra and Toueg in [43] must apply to all processes in the system. If a network partition occurs, the accuracy property can be violated since correct processes in one partition may be suspected to have failed by processes in a different partition. Γ-accurate failure detectors were proposed by Guerraoui and Schiper to deal with this limitation in the specification of Ω-accurate failure detectors [45]. Γ-accurate failure detectors apply the completeness and accuracy properties to a subset of the system's processes and allow consensus to be solved when partitions can occur. The accuracy property for a Γ-accurate Eventually Weak failure detector $\Diamond\mathcal{W}(\Gamma)$, becomes:

- **Eventual weak Γ-accuracy** – eventually some correct process (not necessarily in Γ) is never suspected by any process in Γ.

Since Γ-accurate Eventually Weak failure detectors are strictly weaker than Ω-accurate Eventually Weak failure detectors [45], $\Diamond\mathcal{W}(\Gamma) \prec \Diamond\mathcal{W}$, and $\Diamond\mathcal{W}$ failure detectors have been shown to be the weakest class of failure detectors for which consensus is solvable [44], then it follows that $\Diamond\mathcal{W}(\Gamma)$ failure detectors do not allow for consensus to be solved.

Partial Synchrony. Wide-area systems are often considered to be asynchronous since it's quite difficult to provide a fixed upper bound Δ on the time messages take to be delivered (communication synchrony) or a fixed upper bound Φ on the drift rate between processes' clocks (process synchrony). However, Dwork, Lynch and Stockmeyer note that one reasonable situation is when there exists an upper bound Δ on message delivery latency but it isn't known *a priori* [46]. In a realistic deployment, a system could consider the network to be synchronous for most of the time since connectivity and network problems are sporadic. In this situation, the impossibility results shown in [7] and [47] don't apply since communication can be considered synchronous. To take advantage of an existent but unknown upper bound Δ on communication time, a consensus-solving protocol can't explicitly use Δ as a timeout value because it's unknown. Additionally, for this protocol to progress, there should be a time after which this upper bound holds. Therefore, an additional constraint is applied to the model: there is a global stabilization time after which communication latency is bounded by Δ. An algorithm that solves consensus is required to ensure *safety* regardless of how asynchronous the system behaves, but it only needs to ensure *termination* if Δ eventually holds. In order to extend the model to allow partially synchronous processes, it's possible to modify the additional constraint to: there is a global stabilization time after which communication latency and relative processor speed are bounded by Δ and Φ, respectively. The authors also show that f-resilient consensus is possible in the partially synchronous model if $N \geq 2f+1$, for crash faults, and $N \geq 3f+1$, for Byzantine faults. These lower bounds are the same regardless of whether processors are considered synchronous or partially synchronous.

Set Timeliness. Aguilera et al. propose a partial synchrony model based on the notion of *set timeliness* and show that it can be used to study problems weaker than consensus [48]. This model generalizes the concept of *process timeliness* (i.e., an upper bound Φ on the relative speeds of processes) [46] to sets of processes by considering a set of processes P as a single virtual process that executes an action whenever a process in P executes an action and applying the definition of process timeliness to virtual processes. A set of processes P is considered to be timely with respect to an-

other set of processes Q if, for some integer i, every interval that contains i steps in Q also contains at least one step of a process in P. This model is used to prove possibility and impossibility results for the f-resilient k-set agreement problem. This problem is a generalization of the consensus problem where, instead of having N processes that have to agree on a single value, they need to decide on at most k different values [49]. To solve this problem, Aguilera et al. propose a f-resilient k-anti-Ω failure detector that, for every process p, outputs a set $fdOutput_p$ such that there exists a correct process c that eventually doesn't belong to $fdOutput_p$. Anti-Ω failure detectors were shown to be the weakest failure detectors that allow set agreement to be solved [50]. The algorithm works for two sets P and Q of sizes k and $f + 1$, respectively, such that P is timely with respect to Q. In this f-resilient k-anti-Ω failure detector, each process has a periodically incremented heartbeat and timeout timers for every set A, such that A is a subset of all processes Π_n of size k. Whenever a process detects that *any* process in some set A incremented its heartbeat, it resets A's timer. If A's timer expires for a process p, p increments its timeout value for A and also increments a shared register $Counter[A, p]$ that represents A's untimeliness with respect to p. An accusation counter of a set A is the $(f + 1)$-st smallest value of $Counter[A, *]$. Each process p picks the set that has the smallest accusation counter as its $winnerset_p$ and sets $\Pi_n - winnerset_p$ as the output of k-anti-Ω. Since P is timely with respect to Q, eventually the accusation counter of P stops changing and one of the sets whose accusation counter stops changing is the smallest and is picked as the winner.

2.4 Variable Consistency Models

One of the possible applications of a protocol that attempts to reduce coordination requirements while solving consensus in a non-crash fault model would be in the context of datacenters and possibly geo-replicated systems. Replicating state across geographically separate datacenters requires us to reason about consistency since geo-replication usually implies that only two of the following three properties are available at the same time: strong consistency semantics, availability or partition-tolerance [51]. Traditionally, consistency models could be grouped into two categories: weak and strong. Strong consistency models, such as serializability [52], ensure a total order of operations across all replicas and provide familiar one-copy semantics but incur in an unacceptable latency cost for many applications. Weak consistency models, such as causal consistency [53], address this flaw by relaxing the order guarantees that are ensured and allowing operations to be concurrently executed without coordination between replicas. A smaller subset of this category is eventual consistency where ordering guarantees are relaxed and state divergence is allowed as long as it is eventually reconciled [54]. Other models attempt to strike a balance between these two extremes. Timeline consistency ensures that all replicas of a record apply updates in the same order by forwarding updates to a master. However, reads are executed at any replica and return consistent but possibly stale data [55]. Variable consistency models emerged to address the tension between weak and strong consistency by relaxing ordering guarantees for commutative operations while requiring non-commutative operations to be totally ordered across all replicas. These models are particularly relevant to this work since their approach is based on the same

commutativity observation that generalized consensus uses to define a relaxed version of the consensus problem [19].

RedBlue. RedBlue is a variable consistency model that allows two consistency levels to coexist, red and blue. Blue operations may be executed in different orders at different replicas and are considered to be fast while red operations must be totally ordered across all replicas and are considered slow [56]. For a RedBlue system to remain in a valid state, pairs of non-commutative operations and operations that may violate invariants should be totally ordered. Taking the example of a banking system, since the *withdraw* operation may violate the invariant that the balance should not be less than zero, it must be totally ordered to ensure state convergence. Therefore, an operation u must be labeled red if it may result in an invariant being violated or if there is another operation v that is non-commutative with u. All other operations may be labeled as blue. To maximize the amount of operations that can be labeled as blue, RedBlue notes that while operations themselves may not be commutative, it's often possible to make their updates to the system's data commute. To do this, operations are divided into two parts: a generator operation and a shadow operation. Generator operations compute the changes an operation makes to the state without any side-effect and are only executed in the site in which they are submitted. Shadow operations apply the changes computed by the corresponding generator operation and are executed at all sites. This allows some operations like *deposit* and *accrueInterest* to be treated as commutative by first calculating the amount of interest to be deposited and then treating that amount as a deposit. However, it's important to note that the side-effects are computed at the site where the operation was submitted and, therefore, other replicas might see unexpected changes in their local view of the system's state. Continuing the previous example with a banking account replicated in two sites, if an *accrueInterest(5%)* operation is submitted while one site has a balance of $100 and the other has a balance of $200, due to a concurrent *deposit(100)* operation, then depending on which site the operation is submitted the interest deposited would be either $5 or $10 at both sites. This makes the system convergent but seems anomalous in terms of user experience.

Indigo. The Explicit Consistency model also strives to preserve as much concurrency as possible while preserving consistency when needed. This model guarantees that application-specific invariants are maintained by detecting and handling sets of operations that may cause invariants to be violated, *I-offender sets* [57]. Indigo is a middleware layer that implements this model and its approach consists of three steps: (1) detecting *I-offender sets*, (2) choosing either an invariant repair or violation avoidance technique for handling these sets and (3) instrumenting the application code to use the chosen mechanism. The discovery of *I-offender sets* is performed through static analysis of operation post-conditions and application invariants, which are both specified by the application developer in first-order logic. This approach allows the programmer to define fine-grained consistency semantics that reduce ordering constraints between operations. As previously mentioned, to deal with conflicting sets of operations, the developer may select either invariant repair or violation avoidance techniques. Indigo provides a library of objects (e.g., sets, maps, trees) that repair invariants according to different conflict resolution policies which the programmer can extend to support additional invariants. Indigo also provides reservation techniques that avoid the concurrent execution of possibly conflicting operations. The base mechanism

of these techniques are multi-level lock reservations which provide replicas with *shared forbid*, *shared allow* or *exclusive allow* rights to execute an action. Whenever a replica holds a certain type of right, no other replica can hold rights of a different kind. Multi-level locks allow for the enforcement of any type invariant but Indigo supports additional techniques that provide increased concurrency. For instance, multi-level mask reservations allow two conflicting operations to be concurrently executed if some other predicate remains true. This is useful for invariants like $P_1 \vee P_2 \vee ... \vee P_n$, where the concurrent execution of only two operations is not enough for the invariant to be executed. This type of reservation can be seen as a vector of multi-level locks where, if a replica obtains a shared allow lock in one entry, it must obtain a shared forbid lock in some other entry.

Variable consistency models like the ones presented in this section are comparable to the generalized consensus problem since both are able to lower latency by reducing the need for replicas to exchange information. However, few works use these consistency models in tandem with non-crash fault models. This would be a potentially interesting combination since, as previously mentioned, some environments, like datacenters and geo-replicated systems, are appropriate for critical systems that can benefit from lower coordination requirements and also from tolerance to arbitrary faults.

2.5 Coordination Systems

Due to the importance of Paxos and SMR techniques in the implementation of large-scale Internet services, it's useful to survey their application in modern systems.

ZooKeeper. ZooKeeper is a replicated coordination service for distributed applications. It maintains a consistent image of state across all replicas and totally orders updates such that subsequent operations can implement high-level abstractions such as leader election, synchronization and configuration maintenance [4]. ZooKeeper uses a primary-backup scheme and an algorithm called ZooKeeper Atomic Broadcast (Zab) that is tuned specifically for ZooKeeper's setting [58]. This protocol functions similarly to Paxos but guarantees that multiple outstanding operations are committed in First In, First Out (FIFO) order. The reason why this additional guarantee is necessary is because state changes are incremental with respect to the previous state, so there is an implicit dependence relation on the order of operations. Paxos doesn't guarantee that operations are learned in FIFO order since, even if multiple client operations are batched in a single proposal, a primary failure can cause the multiple outstanding requests to be reordered. If two primaries failed after sending *phase 2a* messages to a subset of a quorum of acceptors, the next primary would execute Paxos' *phase 1*, learn about the partially decided values and pick a new order for the operations which would possibly differ from FIFO order. Zab was built specifically to guarantee that outstanding ZooKeeper operations are committed in FIFO order despite being submitted concurrently. The protocol functions in three phases: (1) discovery, (2) synchronization and (3) broadcast. In the first two phases, the leader obtains the longest history of the latest epoch and sends the new history to the followers. These two phases guarantee that outstanding proposals are committed in FIFO order. In the broadcast phase, the leader sends proposals to the followers, who write the proposals to stable storage and reply with an acknowledgment. After receiving a quorum of acknowledgments,

the leader sends a commit message $\langle v, z \rangle$ to indicate that the followers should deliver all undelivered proposals that are causally related to z, $z' \prec z$.

Chubby. Chubby is a lock service that provides reliable, advisory locking functionality for loosely-coupled distributed systems. Chubby exposes both a namespace and an interface that resemble a distributed file system, through which clients acquire and release locks [2]. A Chubby cell consists of a small set of replicas where one of them functions as a master. The master is elected through a consensus protocol by obtaining a quorum of votes. The master must also obtain promises that the replicas won't elect a different master for a time period known as the *master lease*. Clients send master location requests to the replicas listed in the Domain Name System (DNS) and non-master replicas respond with the master's identity. Once a client knows the master's location it will direct all requests to that location. Write requests are propagated to the replicas through Paxos and the leader sends an acknowledgment to the client when the write reaches a quorum. Read requests can be serviced by the master alone because, as long as the master lease has not expired, no other master can exist and issue write requests.

Megastore. Megastore is a storage system that strives to provide the advantages of both traditional Relational Database Management Systems (RDBMS) and NoSQL datastores [13]. In particular, it tries to combine the strong consistency guarantees and rich set of features of relational databases with the high scalability and performance provided by NoSQL systems, while at the same time implementing the fault-tolerance necessary to ensure high availability. It does this by providing Atomicity, Consistency, Isolation, Durability (ACID) semantics only within fine-grained data partitions and weaker consistency guarantees for transactions that manipulate data across several partitions. Paxos is used to replicate a write-ahead log over a group of peers, with each consensus instance deciding the value of a log's index. Some optimizations are implemented to achieve fast reads and single round-trip writes. To provide the latter, besides deciding the value of a log's position, each consensus instance also picks a leader for the next consensus instance. This optimization resembles Multi-Paxos, except that it can pick different leaders for subsequent instances whereas Multi-Paxos provides single round-trip commits as long as the leader remains the same.

Chapter 3

Generalized Consensus

One of our most important contributions is the revised consensus problem. In Lamport's original specification, the consensus problem is generalized from agreeing on a single value to agreeing on an increasing set of values [19]. The consensus problem is defined in terms of *c-structs* and includes four properties, where $learned[l]$ is the sequence of commands learned by a learner l, $propCmd$ is the set of all proposed commands and $Str(propCmd)$ is the set of all *c-structs* that can be constructed from the elements in $propCmd$:

- **Nontriviality** $learned[l] \in Str(propCmd)$ always holds, for every learner l.
- **Stability** It is always the case that $learned[l] = v$ implies $v \sqsubseteq learned[l]$ at all later times, for any learner l and *c-struct* v.
- **Consistency** $learned[l_1]$ and $learned[l_2]$ are always compatible, for all learners l_1 and l_2.
- **Liveness** If $c \in propCmd$ then eventually $learned[l]$ contains c.

Two of the reasons why the generalized consensus problem and its solution, the Generalized Paxos protocol, are so hard to understand, is due to the amount of mathematical operations defined and how they're used to compose a consensus-solving algorithm. In the following sections, we will step through these mathematical formalisms and provide the intuition as to how they're used in the Generalized Paxos protocol. We will also provide some insights on other nontrivial components of Generalized Paxos that require careful examination of the original description in order to be understood. Since these components are profoundly affected by our simplification of generalized consensus, it makes sense for us to describe their original form. Later chapters will introduce the simplified consensus problem appropriate for each fault model and also describe how some important issues are solved in light of the new problem. This chapter allows us to draw a clear parallelism between those descriptions and the original generalized consensus problem.

3.1 Operations on *C-structs*

The consistency property stated above defines what a consensus-solving protocol must guarantee to be considered *safe*. In generalized consensus, consistency relies on a notion of *compatibility* between *c-structs*. As Lamport states, two *c-structs*, v and w are compatible if and only if they have a common upper bound, that is, if there exists a *c-struct* z to which they are both prefixes, $v \sqsubseteq z \wedge w \sqsubseteq z$. Intuitively, this means that if two *c-structs* can be extended to the same *c-struct*, then each can be learned by a different learner without violating consistency. However, if two *c-structs* contain conflicting commands in different relative orders, then there is no *c-struct* that extends both and the *c-structs* cannot be learned safely. This notion is crucial to the understanding of generalized consensus and, even though our simplified problem forgoes *c-structs* in favor of sequences of commands, our safety condition also relies on the idea of two sequences being able to be extended to equivalent sequences.

As previously mentioned, generalized consensus' consistency specification relies on the notion of compatibility which, in turn, relies on the notion of a *common upper bound*. An upper bound of a set T is a *c-struct* v such that $w \sqsubseteq v$ for any w in T. Intuitively, an upper bound is an extension of every *c-struct* in a set. A lower bound of a set T is defined analogously as a *c-struct* v such that $v \sqsubseteq w$ for any w in T. Upper and lower bounds are most commonly used in Generalized Paxos as components in two operations, least upper bounds $\sqcup$ and greatest lower bounds $\sqcap$. A least upper bound $\sqcup$ of a set of *c-structs* T is an upper bound v of T such that $v \sqsubseteq w$ for any upper bound w of T. One could define this operation $\sqcup T$ as the upper bound of T that is a lower bound of every upper bound in T. Note that if $v = w$ then $v \sqsubseteq w$ and $w \sqsubseteq v$. Analogously, a greatest lower bound $\sqcap$ of a set of *c-structs* T is a lower bound v of T such that $w \sqsubseteq v$ for any lower bound w of T.

The operations we just defined are used in critical components of the Generalized Paxos protocol, namely in the leader's value picking procedure. However, the usage of these operations combined with the already complex functioning of some of Generalized Paxos' components makes it difficult for the reader to understand the semantics of what is being computed. In order to ease the later examination of components that rely on these operations, it's helpful to illustrate their execution with an example. Let T be the set of *c-structs* $\{A; A \bullet B; A \bullet B \bullet C\}$ and let the set of all *c-structs* contain any combination of the commands A, B, C, D and E, as long as they appear in alphabetical order (recall that Generalized Paxos allows for the definition of any consensus problem specified in terms of *c-structs*). If an upper bound of T is a *c-struct* v such that $w \sqsubseteq v$ for any w in T, then there are three possible upper bounds, namely $A \bullet B \bullet C$, $A \bullet B \bullet C \bullet D$ and $A \bullet B \bullet C \bullet D \bullet E$. Of these three upper bounds, we can compute $\sqcup T$ by noting that only the upper bound $A \bullet B \bullet C$ is a lower bound of all three upper bounds. As previously mentioned, the consistency property requires learned *c-structs* to be compatible (i.e., to have a common upper bound). If the previously defined set T was the set of learned sequences in a Generalized Paxos execution, then the system would be functioning correctly because any two *c-structs* in T are compatible (i.e., can be extended to some possible *c-struct*). Any set of compatible *c-structs* is guaranteed to have a least upper bound.

Let's redefine the set T to contain the following *c-structs* $\{A; A \bullet C; A \bullet B \bullet C\}$. This may seem like an

Algorithm 1 Original Generalized Paxos - Excerpt from the leader's code

Local variables: $ballot_l = 0,\ maxTried_l = \bot$

1: **function** PHASE_2A(bal, Q)
2: $maxTried_l$ = **run** PROVED_SAFE(Q, bal);
3: $maxTried_l = maxTried_l \bullet new\_proposals$;
4: **run** SEND($P2A, ballot_l, maxTried_l$) to acceptors;
5: **end function**
6:
7: **function** PROVED_SAFE(Q, m)
8: $k = max(i \mid (i < m) \wedge (\exists a \in Q :\ val_a[i]\ \neq null))$;
9: $RS = \{R \in k\text{-}quorum \mid \forall a \in R \cap Q : val_a[k] \neq null\}$;
10: $\gamma(R) = \sqcap\{v_a[k] \mid a \in Q \cap R\}$;
11: $\Gamma = \{\gamma(R) \mid R \in RS\}$;
12:
13: **if** $RS = \varnothing$ **then**
14: **return** $\{val_a[k] \mid (a \in Q) \wedge (val_a[k] \neq null)\}$;
15: **else**
16: **return run** $\sqcup\Gamma$;
17: **end function**

innocuous modification but it's enough to make the computation of $\sqcup T$ impossible. Taking into account the *c-structs* in T, we can see that T doesn't have an upper bound because there is no *c-struct* to which every *c-struct* in T is a prefix. No upper bound of $A \bullet B \bullet C$ will also be an upper bound of $A \bullet C$. In other words, $A \bullet B \bullet C$ and $A \bullet C$ are not compatible. If T was the set of *c-structs* learned in a Generalized Paxos execution, the system wouldn't be functioning correctly because it's impossible to compute $\sqcup T$. Intuitively, we can see that this is because the *c-structs* $A \bullet B \bullet C$ and $A \bullet C$ conflict with each other and, therefore, could never be learned by different learners.

3.2 *C-structs* in Generalized Paxos

In section 3.1, we mentioned how learned commands must be compatible in order for safety to be upheld. However, in order for this compatibility invariant to be preserved, the leader and the acceptors must act in a way that only causes compatible *c-structs* to be learned. For this reason, the leader can't pick any value as his proposal before sending it to the acceptors. The previously defined operations are mostly employed in the procedure executed by the leader when choosing a value to which he can append the proposers' commands to. In *phase 2a*, before sending new proposed commands to the acceptors, the leader must first pick a value to which its proposals will be appended. To ensure that the consistency requirements are met, acceptors must only vote for *safe* values. In this context, a *c-struct* is safe if it's the extension of a choosable value at a lower ballot. In other words, if it was possible for a *c-struct* w to be chosen by a majority in a previous ballot and $w \sqsubseteq v$, then v is safe. Therefore, to guarantee correctness, the leader must append new proposals to a safe *c-struct*. The procedure that Generalized Paxos' leader follows in order to pick a safe *c-struct* is depicted in Algorithm 1 in its original notation.

Algorithm 2 provides a translation of each line in the $Proved\_Safe$ procedure to an informal description. This is included to illustrate two things. First, how the complex notation of this procedure makes it

Algorithm 2 Informal explanation of the Proved_Safe procedure

```
1: function PROVED_SAFE(Q, m)
2:     Q is the quorum gathered to vote for the current ballot m
3:     k is the largest ballot number (but smaller than m) in which some acceptor in Q voted for
4:     RS is the set of k-quorums (quorums possibly gathered at k) in which every acceptor also in Q
       voted at k
5:     γ(R) is the set of greatest lower bounds of the votes sent by acceptors in both R and Q
6:     Γ is the set of greatest lower bounds of the votes sent by acceptors in both Q and a set where
       every acceptor also Q voted at k
7:     if RS is empty (there is no k-quorum R for which every acceptor in R and Q have voted) then
8:         return Any value reported in phase 1b messages
9:     else
10:        return The least upper bound of Γ (greatest lower bounds of votes)
11: end function
```

difficult to understand or even translate into a practical implementation. Second, how, even in an informal format, the composition of the previously defined operations results in a procedure that is semantically difficult to follow. We believe this supports our argument that despite generalized consensus' great practical utility, its current state hampers its adaptation into a real implementation. Therefore, it's relevant and timely for the generalized consensus problem and its solution to be simplified as well as extended, wherever possible.

The following sections will discuss some of Generalized Paxos' components that are of particular importance to the protocol's correct functioning and whose original description can be difficult to parse.

3.3 Leader Value Picking Rule

One of Generalized Paxos' most crucial components is the rule employed by the leader in order to choose its *phase 2a* proposal in a given classic ballot. This rule is carefully constructed in order to not only allow the protocol to deal with conflicts between non-commutative proposals but also prevent a far more subtle situation in which consistency could be jeopardized. Consider a system of $N = 3f + 1$ acceptors where, in some fast ballot, two non-commutative commands, A and B, are proposed. Consider also that a majority of acceptors votes for a sequence where A precedes B and the remaining f acceptors vote for a non-commutative sequence. In this situation, the first sequence is chosen and learned at one learner but consider that the majority of votes is delayed before reaching a second learner. In the subsequent classic ballot, the leader would request *phase 1b* messages from acceptors which would contain their votes on the previous fast ballot. Some of the values reported in the quorum of *phase 1b* messages will contain A preceding B and others will contain non-commutative sequences. If the non-commutative sequence is chosen and sent to the acceptors in a *phase 2a*, the learner in which the previous vote was delayed could receive a majority of votes for the leader's proposal and learn a sequence that conflicts with another learner's learned sequence. Therefore, the rule that governs the sequence picked by the leader must take into account the possibility that learners may have different views of the learned commands. Since it is unsafe for the leader to assume that the learners share the same view of learned commands at the beginning of a ballot, the leader must consider safety even when

sequences are proposed in different ballots.

Generalized Paxos deals with this problem by having the leader of some ballot m propose a sequence $w \bullet \sigma$, where w belongs to the set of values returned by $Proved\_Safe(Q, m)$ and σ is one possible serialization of the proposals sent by the proposers to the leader. In the original protocol, if a majority was reached in the previous fast ballot, the value w is the value chosen by the acceptors present in the intersection between the quorum Q of gathered *phase 1b* messages and the quorum R of *phase 2b* messages sent to the learners in the previous ballot. Since the value sent by the leader in *phase 2a* contains commands that were possibly learned in the previous ballot, a learner will also have to take that into account when it's learning. It can learn the leader's proposal safely by merging it with its $learned$ sequence in a way such that already learned commands are not duplicated in the $learned$ sequence.

Despite its importance to the correctness of Generalized Paxos, as we established in the previous section, the leader's value picking rule suffers in terms of understandability. Our simplified specification of consensus as agreement on commands histories allows us to describe this rule in clearer terms that could easily be transformed into a practical implementation. The protocols proposed throughout this work will feature different rules according both to the underlying fault model and to our simplified specification of consensus.

3.4 Quorum Sizes in Generalized Paxos

One of the main drawbacks of both Fast and Generalized Paxos is that they require both the total system size and the quorum size to increase with the number of faults one wishes to tolerate while still allowing for fast executions [18, 19]. This is widely considered a direct consequence of allowing for learning proposals in two message steps. However, a reader unfamiliar with the Fast Paxos protocol may think that simple majority quorums suffice, since a majority is enough for any two quorums to intersect which would prevent non-commutative values from being learned at different relative orders. In order to clarify this aspect of fast learning, we go through the original description step by step and gradually derive Fast Paxos' restrictions on quorum and system size. Although this section refers mostly to contributions stated in Fast Paxos, these insights are applicable to Generalized Paxos as well.

The increased quorum requirements arise from the necessity of defining a rule for the leader to pick a value to be sent in its *phase 2a* messages [18]. After executing *phase 1*, the leader has gathered a quorum Q of *phase 1b* messages from the acceptors relaying their votes regarding the latest ballot. In the common case, no value has been picked by the acceptors for the current ballot, but this is not necessarily the case since, in the case of a leader failure, new leaders are not guaranteed to be aware of the latest round. To further complicate the problem, since fast ballots allow multiple values to be concurrently proposed, the acceptors can relay different values voted for in the previous ballot. If for a ballot k some value v has already been sent by a previous leader to the acceptors such that a majority voted for it, no other non-commutative value will be accepted. Lamport states that [18]:

> **Observation 4.** (...) a value v has been or might yet be chosen in round k only if there exists a k-quorum R such that $vr(a) = k$ and $vv(a) = v$ for every acceptor in $R \cap Q$.

To determine what the leader's behavior needs to be when picking a value, it's useful to think about what actions are available when it has gathered Q:

- If no value satisfies Obs. 4, then no value has been or might yet be picked. In this case, the leader can choose any value present in the *phase 1b* messages.
- If one value v satisfies Obs. 4, then that value is the only value that has been or might yet be picked in round k and the leader can only pick v in *phase 2a*.
- If there is more than one value that satisfies Obs. 4, then the leader has no way of choosing a value.

Lamport solves this problem by making the third case impossible. He does this by noticing that this case requires Obs. 4 to be true for two distinct values v and w, which implies that there are k-quorums R_v and R_w such that every acceptor in $R_v \cap Q$ has voted for v and every acceptor in $R_w \cap Q$ has voted for w. This becomes impossible if we force $R_v \cap R_w \cap Q \neq \emptyset$. This rule is formalized as:

> **Quorum Requirement** For any round numbers j and i:
>
> (a) Any i-quorum and any j-quorum have a non-empty intersection.
>
> (b) If j is a fast round number, then any i-quorum and any two j-quorums have non-empty intersection.

The system and quorum sizes can then be defined with respect to parameters f and e, respectively, the maximum number of faults the system can tolerate while still being able to function correctly and the maximum number of faults the system can tolerate while still being able to support two step executions. The system size N must take into account the following restrictions: $N > 2f$ and $N > 2e + f$. If we wish to always allow fast executions then $e = f$ and the minimum value of N becomes $3f + 1$. Similarly, the fast and classic quorums both become $N - e = N - f = 2f + 1$. This is the cost imposed on the system in order to allow the safe learning of values in two message steps at any moment. Another alternative would be to minimize e such that two step executions are only possible while the system doesn't experience faults (i.e., $e = 0$). In this case, the strictest restriction on system size is $N > 2f$. However, even though the size of a classic quorum would be only $f + 1$, a fast quorum would have to contact every process in the system.

Chapter 4

Crash Fault Model

4.1 Model

Before describing our consensus-solving protocol for the crash fault tolerant model (CFT), we must first specify precisely our simplified consensus problem and in which conditions the system will operate. We start by describing the fault and network models in which the protocol will function before defining the consensus problem's requirements.

4.1.1 Network Model

For the crash fault tolerant protocol, we consider an *asynchronous* system in which a set of N processes communicate by *sending* and *receiving* messages. For simplicity, we assume perfect links [59], where a message that is sent by a non-faulty sender is eventually received exactly once and any message delivered by a process q with sender p was previously sent to q by process p (such links can be implemented trivially using retransmission and elimination of duplicates [59]). There are three types of roles in the system: proposers, acceptors and learners, and each executes a specific algorithm. Informally, proposers provide input values that must be agreed upon by learners, the acceptors help the learners *agree* on a value by issuing votes, and learners learn commands by appending them to a local sequence of commands to be executed, $learned_l$. Proposers can also distinguish themselves as the leader in order to help the system make progress. Each process can execute one or more roles without any interference between them. Our protocol requires a minimum number N of acceptor processes and we assume that they have identifiers in the set $\{0, ..., N-1\}$. In contrast, the number of proposer and learner processes can be set arbitrarily.

4.1.2 Fault Model

The aforementioned minimum number N of acceptor processes is a function of the maximum number of tolerated crash faults f, namely $N > 3f$. This condition implies that the system is able to function correctly despite up to f crash faults. A *correct* process is one that eventually executes the actions in

the algorithms assigned to it, such as sending and receiving messages. Quorums are defined as a set of $N - f$ processes which, in the usual case where N is tight (i.e., $N = 3f + 1$), is equal to a quorum of $2f + 1$. Note that in the traditional description of both Fast and Generalized Paxos, N is constrained by two conditions, namely $N > 2f$ and $N > 2e + f$, where e is the number of faults tolerated by the system while allowing for fast ballots to take place [18, 19]. Similarly, Generalized Paxos' quorums are defined as $N - e$ and $N - f$ for fast and classic ballots, respectively. This implies that if we wish to always allow for fast executions then $e = f$ which means that $N > 3f$ and both quorums are $2f + 1$. The same reasoning applies to our CFT version of Generalized Paxos, but, since our goal is to simplify its description, we specialize the algorithm to the case where we always allow for fast executions (i.e., $e = f$). Therefore, in our protocol both fast and classic ballots can always be executed, as long as the system experiences only up f faults, and the quorums are the same size in both cases.

4.1.3 Problem Statement

In our 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 a *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. In this case, a prefix is not a strict prefix because a sequence is considered to be a prefix of itself. 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 chapter 2), we specialize to agreeing on equivalent sequences of commands:

1. **Nontriviality.** $learned_l$ can only contain proposed commands.
2. **Stability.** If $learned_l = v$ then, at all later times, $v \sqsubseteq learned_l$, for any l and v.
3. **Consistency.** At any time and for any two 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 proposer, and learner l, eventually $learned_l$ contains s.

4.2 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.

Algorithm 3 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;

4.2.1 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 [18], 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*.

Algorithm 4 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: $old\_seq = old\_seq \bullet c$;
5: **return** old_seq;
6: **end function**

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 [9]. 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, as explained in section 3.3, 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.

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.

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's 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. 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 [9].

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. As was described in section 3.3, 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's 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 com-

Algorithm 5 Generalized Paxos - Leader l

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

1: **upon** trigger_next_ballot($type$) **do**
2: $ballot_l \mathrel{+}= 1$;
3: SEND$(BALLOT, ballot_l, type)$ to proposers;
4:
5: **if** $type = fast$ **then**
6: SEND$(FAST, ballot_l, view)$ to acceptors;
7: **else**
8: SEND$(P1A, ballot_l, view)$ to acceptors;
9:
10: **upon** receive$(PROPOSE, prop)$ from proposer p_i **do**
11: **if** ISUNIVERSALLYCOMMUTATIVE$(prop)$ **then**
12: SEND$(P1A, ballot, prop)$ to acceptors;
13: **else**
14: $proposals = proposals \bullet prop$;
15:
16: **upon** receive$(P1B, ballot, bal_a, vals_a)$ from acceptor a **do**
17: **if** $ballot_a = ballot_l$ **then**
18: $accepted[ballot_l][a] = \langle bal_a, vals_a \rangle$;
19: **if** $\#(accepted[ballot_l]) \geq N - f$ **then**
20: PHASE_2A();
21:
22: **function** PHASE_2A()
23: $votes = \bot$;
24: $k = -1$;
25: **for** a **in** $acceptors$ **do**
26: $bal_a = accepted[ballot_l][a][0]$;
27: $val_a = accepted[ballot_l][a][1]$;
28: **if** $bal_a > k$ **then**
29: $k = bal_a$;
30: $votes = \bot$;
31: **else if** $bal_a = k$ **then**
32: $votes[val_a] \mathrel{+}= 1$;
33: **if** $votes[val_a] > f$ **then**
34: $maxTried_l = val_a$;
35: **break**;
36:
37: **for** a **in** $acceptors$ **do**
38: $maxTried_l =$ MERGE_SEQUENCES$(maxTried_l, accepted[ballot_l][a])$;
39:
40: $maxTried_l = maxTried_l \bullet proposals$;
41: SEND$(P2A_CLASSIC, ballot_l, maxTried_l)$ to acceptors;
42: $proposals = \bot$;
43: $maxTried_l = \bot$;
44: **end function**

mands 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. Note that for a sequence to be learned, a learner doesn't have to receive $N - f$ votes for the exact same sequence but for equivalence sequences (in accordance to our previous definition of equivalence).

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, bypassing the leader, 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 [18].

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.

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. Note that, in Algorithm 5 lines

Algorithm 6 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: $val_a = \perp$;
18: **end function**
19:
20: **function** PHASE_2B_CLASSIC($ballot, value$)
21: **if** $ballot \geq bal_a$ and $val_a = \perp$ **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**

Algorithm 7 Generalized Paxos - Learner l

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

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($learned, value$);

{32-33} and Algorithm 7 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. 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.

4.2.2 Discussion

Universally Commutative Commands

The specification of command history consensus dictates that the only commands whose relative order can be reversed at different learners are those that commute with each other. This restriction has the advantage of allowing for running a replicated state machine on top of the Generalized Paxos protocol, while still ensuring that the state machine behaves in a way that is indistinguishable from running it on top of the original Paxos protocol. However, the downside of this use of commutative operations in the context of Generalized Paxos is that this commutativity check is done at runtime, and, if non-commutative operations are issued concurrently, then we must fall back to the slower classic Paxos protocol.

This downside 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 for the learner to be sure that the proposed sequence was witnessed by at least one correct acceptor that can propagate them to others. Furthermore, for "read-only" commands, it is possible to run these against a single learner, since there is no requirement to withstand f faults for these commands.

Even though proposers can propose universally commutative sequences that are guaranteed to never cause conflicts with any other sequence, in both classic and fast ballots, proposals are appended to a prefix of sequences. However, the resulting sequence is unlikely to be universally commutative since it contains every previous sequence. This greatly diminishes the applicability of the overall optimization since it's only possible when every command in the history is universally commutative. Proposals are appended to proven sequences because, if they were sent individually, the network could reorder them and cause a conflict between multiple learners. However, since universally commutative commands are guaranteed to never cause conflicts, it's safe to send them individually and allow for them to be learned in arbitrary orders. When the leader or the acceptors receive universally commutative sequences, they immediately trigger the next phase without appending them to previous sequences. The reason why this is possible is because, since the proposal is a universally commutative sequence, it can arrive at the learner in a different order relative to other sequences at different learners while the final state will still converge across the system.

This optimization is particularly useful in the context of geo-replicated systems, since it can be significantly faster to wait for the $f+1$th message instead of the $N-f$th one. It can also be easily implemented with an additional check in some of the protocol's algorithms. This can be seen in Algorithm 5 lines $\{$11-12$\}$, Algorithm 6 lines $\{$23-24,32-33$\}$ and Algorithm 7 lines $\{$3-4$\}$.

Generalized Paxos and Weak Consistency

The key distinguishing feature of the specification of Generalized Paxos [19] 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 [53], or consistency models that mix strong and weak consistency levels like RedBlue [56], 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 subsection. In the case of weakly consistent replication (or multi-level replication), weakly consistent requests can be executed as if they were universally commutative, even if in practice that may not be the case. For instance, 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 [56].

Chapter 5

Byzantine Fault Model

5.1 Model

In this section, we define our simplified consensus problem for the Byzantine fault tolerant (BFT) model and describe how the possibility of Byzantine faults has an effect on both the network assumptions and the consensus problem. We start by describing the fault and network assumptions in which the protocol will function and then we move on the consensus specification itself.

5.1.1 Network Model

For the Byzantine tolerant protocol, we consider an *asynchronous* system in which a set of $n \in \mathbb{N}$ processes communicate by *sending* and *receiving* messages. Furthermore, for clarity of exposition, we assume authenticated perfect links [59], where a message that is sent by a non-faulty sender is eventually received exactly once and the receiver is guaranteed that the message originated from its stated sender (such links can be implemented trivially using retransmission, elimination of duplicates, and point-to-point message authentication codes [59]). As is the case in the CFT protocol, each process executes an algorithm depending on its role, which can be of proposer, acceptor or learner (a process can have multiple roles simultaneously). Our protocol requires a minimum number N of acceptor processes and we assume that they have identifiers in the set $\{0, ..., N-1\}$. In contrast, the number of proposer and learner processes can be set arbitrarily.

5.1.2 Fault Model

Each process executes one or more algorithms assigned to it according to its roles, but may fail in two different ways. First, it may stop executing an algorithm by *crashing*. Second, it may stop following an algorithm assigned to it, in which case it is considered *Byzantine*. We say that a non-Byzantine running process is *correct*. Our protocol functions under the *authenticated* Byzantine model where every process can produce cryptographic digital signatures [1]. The number of acceptor processes N is a function of the maximum number of tolerated Byzantine faults f, namely $N \geq 3f + 1$, and quorums are any set of

$N - f$ processes.

5.1.3 Problem Statement

As in the CFT version of our simplified specification of Generalized Paxos, each learner l maintains a monotonically increasing sequence of commands $learned_l$. Similarly, our definition of equivalence remains the same as it was defined in Section 4.1.3, where 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 *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, $learned_l$. Since we wish to not only strengthen the fault model but also preserve the understandability and ease of implementation of the consensus problem defined for the CFT model, our Byzantine version of the command history problem will be based our previous specification of consensus. The requirements of the consensus problem for the Byzantine model are:

1. **Nontriviality.** If all proposers are correct, $learned_l$ can only contain proposed commands, for any correct learner l.
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.

5.2 Protocol

This section presents our Byzantine fault tolerant Generalized Paxos Protocol (or BGP, for short).

Algorithm 8 Byzantine Generalized Paxos - Proposer p

Local variables: $ballot\_type = \perp$

1: **upon** *receive*$(BALLOT, type)$ **do**
2: $\quad ballot\_type = type$;
3:
4: **upon** *command_request*(c) **do**
5: $\quad$ **if** $ballot\_type == fast\_ballot$ **then**
6: $\quad\quad$ SEND$(P2A_FAST, c)$ to acceptors;
7: $\quad$ **else**
8: $\quad\quad$ SEND$(PROPOSE, c)$ to leader;

5.2.1 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 [18] and Generalized Paxos [19], 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.

5.2.2 View Change

The goal of the view change subprotocol is to elect a distinguished proposer 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 [29].

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, 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 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

Algorithm 9 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**

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.

5.2.3 Agreement Protocol

The consensus protocol allows learner processes to agree on equivalent sequences of commands (according to the definition of equivalence present in Section 5.1). Much like in our previous simplified version of Generalized Paxos, 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 previously 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 [9], 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 implicate classic ballots. Figures 5.1 and 5.2 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.

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 isn't 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.

However, this concurrency introduces safety problems even when a quorum is reached for some sequence. If we keep the original Fast Paxos message pattern [18], it's possible for one sequence s to

Algorithm 10 Byzantine Generalized Paxos - Acceptor a (view change)

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

1: **upon** *suspect_leader* **do**
2: **if** $suspicions[p] \neq true$ **then**
3: $suspicions[p] = true$;
4: $proof = \langle suspicion, view \rangle_{priv_a}$;
5: SEND$(SUSPICION, view, proof)$;
6:
7: **upon** *receive*$(SUSPICION, view_i, proof)$ from acceptor i **do**
8: **if** $view_i \neq view$ **then**
9: **return**;
10: **if** $proof_{pub_i} == \langle suspicion, view \rangle$ **then**
11: $suspicions[i] = proof$;
12: **if** $\#(suspicions) > f$ and $new\_view[view + 1][p] == \perp$ **then**
13: $change\_proof = \langle view\_change, view + 1 \rangle_{priv_a}$;
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** $new\_view_i \leq view$ **then**
19: **return**;
20:
21: $valid\_proofs = 0$;
22: **for** p **in** $acceptors$ **do**
23: $proof = suspicions[p]$;
24: $last\_view = new\_view_i - 1$;
25: **if** $proof_{pub_p} == \langle suspicion, last\_view \rangle$ **then**
26: $valid\_proofs \mathrel{+}= 1$;
27:
28: **if** $valid\_proofs \leq f$ **then**
29: **return**;
30:
31: $new\_view[new\_view_i][i] = change\_proof_i$;
32: **if** $new\_view[view_i][a] == \perp$ **then**
33: $change\_proof = \langle view\_change, new\_view_i \rangle_{priv_a}$;
34: $new\_view[view_i][a] = change\_proof$;
35: SEND$(VIEW_CHANGE, view_i, suspicions, change_proof)$;
36:
37: **if** $\#(new\_view[new\_view_i]) \geq N - f$ **then**
38: $view = view_i$;
39: $leader = view \bmod N$;
40: $suspicions = \perp$;
41: SEND$(LEADER, view, new_view[view_i])$ to leader;

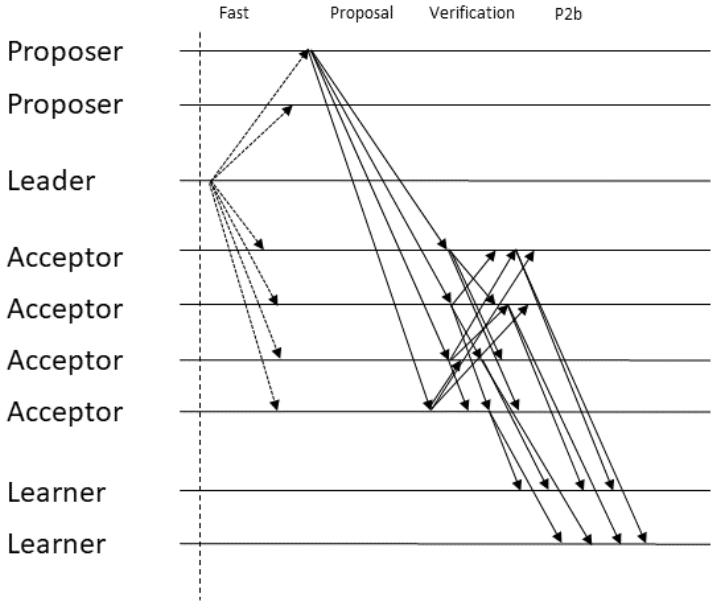

Figure 5.1: BGP's fast ballot message pattern

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 doesn't 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 isn't 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.

An interesting alternative to requiring $N - f$ proofs from each acceptor, would be for the leader to wait for $2f + 1$ matching *phase 1b* messages. Since at least $f + 1$ of those would be correct, only that sequence could've been learned since any other non-commutative sequence would obtain at most $2f$ votes. Zyzzyva uses a similar approach of waiting for $3f + 1$ to commit requests in single round-trip in executions where no faults occur [60]. However, this approach is unsuitable for BGP since it's possible for a sequence to be chosen by a quorum without the leader being aware of more than $f + 1$ votes in

its quorum. Since $f + 1$ votes aren't enough to ensure the leader that the sequence was chosen by a quorum, the leader wouldn't be able to pick a learned sequence.

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. This 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. Note that the fact that proposals are sent as extensions to previous sequences is critical to the safety of the protocol. In particular, since the votes from acceptors can be reordered by the network before being delivered to the learners, if these values were single commands, it would be impossible to guarantee that non-commutative commands would be learned in a total order.

Step 3: Acceptors to learners. 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.

Algorithm 11 Byzantine Generalized Paxos - Acceptor a (agreement)

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

1: **upon** *receive*$(P1A, ballot, view_l)$ from leader l **do**
2: **if** $view_l == view$ and $bal_a < ballot$ **then**
3: SEND$(P1B, ballot, bal_a, proven, val_a, proofs[bal_a])$ to leader;
4: $bal_a = ballot$;
5: $val_a = \perp$;
6:
7: **upon** *receive*$(FAST, ballot, view_l)$ from leader **do**
8: **if** $view_l == view$ **then**
9: $fast\_bal[ballot] = true$;
10:
11: **upon** *receive*$(VERIFY, view_i, ballot_i, val_i, proof)$ from acceptor i **do**
12: **if** $proof_{pub_i} == \langle ballot_i, val_i \rangle$ and $view == view_i$ **then**
13: $proofs[ballot_i][val_i][i] = proof$;
14: **if** $\#(proofs[ballot_i][val_i]) \geq N - f$ **then**
15: $proven = val_i$;
16: SEND$(P2B, ballot_i, val_i, proofs[ballot_i][value_i])$ to learners;
17:
18: **upon** *receive*$(P2A\_CLASSIC, ballot, view, value)$ from leader **do**
19: **if** $view_l == view$ **then**
20: PHASE_2B_CLASSIC$(ballot, value)$;
21:
22: **upon** *receive*$(P2A\_FAST, value)$ from proposer **do**
23: PHASE_2B_FAST$(value)$;
24:
25: **function** PHASE_2B_CLASSIC$(ballot, value)$
26: $univ\_commut =$ ISUNIVERSALLYCOMMUTATIVE(val_a);
27: **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**
28: $bal_a = ballot$;
29: **if** $univ\_commut$ **then**
30: SEND$(P2B, bal_a, value)$ to learners;
31: **else**
32: $val_a = value$;
33: $proof = \langle ballot, val_a \rangle_{priv_a}$;
34: $proofs[ballot][val_a][a] = proof$;
35: SEND$(VERIFY, view, ballot, val_a, proof)$ to acceptors;
36: **end function**
37:
38: **function** PHASE_2B_FAST$(ballot, value)$
39: **if** $ballot == bal_a$ and $fast\_bal[bal_a]$ **then**
40: **if** ISUNIVERSALLYCOMMUTATIVE$(value)$ **then**
41: SEND$(P2B, bal_a, value)$ to learners;
42: **else**
43: $val_a = val_a \bullet value$;
44: $proof = \langle ballot, val_a \rangle_{priv_a}$;
45: $proofs[ballot][val_a][a] = proof$;
46: SEND$(VERIFY, view, ballot, val_a, proof)$ to acceptors;
47: **end function**

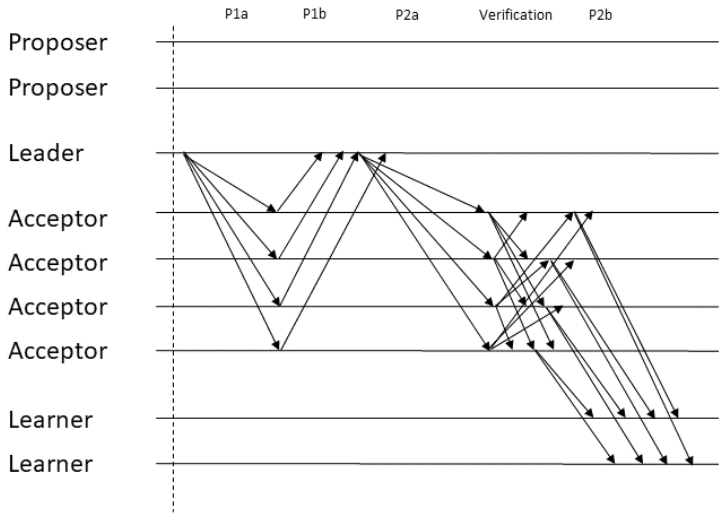

Figure 5.2: BGP's classic ballot message pattern

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.

Classic Ballots

Classic ballots work in a way that is very close to the original Paxos protocol [9]. 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 [9]. 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. In the CFT version of the protocol, the leader could determine if some value had been chosen by a majority of acceptors by looking for $f + 1$ equivalent prefixes in the quorum of *phase 1b* messages. If some sequence had $f + 1$ votes, no other sequence could have been chosen since there are only other $2f$ acceptors in the system. However, the same isn't true for BGP because $f + 1$ identical votes for some value only attest that at least one correct process voted for that value. It's impossible to determine if some value was chosen by a majority unless a correct process witnesses $2f + 1$ identical votes. Note that, since each command is signed by the proposer (this signature and its check are not explicit in the pseudocode), a Byzantine acceptor can't relay made-up commands. However, it can omit commands from its *phase 1b* message, which is why it's necessary for the leader to be sure that at least one correct acceptor in its quorum took part in the verification quorum of any learned sequence.

After gathering a quorum of $N - f$ *phase 1b* messages, the leader initiates *phase 2a* where it assembles a proposal and sends it to the acceptors. This proposal sequence must be carefully constructed 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.)

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.

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 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.

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 11 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 9 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.

Algorithm 12 Byzantine Generalized Paxos - Learner I

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** $proof_{pub_i} == \langle ballot, value \rangle$ **then**
6: $valid\_proofs \mathrel{+}= 1$;
7:
8: **if** $valid\_proofs \geq N - f$ **then**
9: $messages[ballot][value][a] = proofs$;
10:
11: **if** $\#(messages[ballot][value]) \geq N - f$ **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: $learned = learned \bullet value$;
19:
20: **function** MERGE_SEQUENCES(old_seq, new_seq)
21: **for** c **in** new_seq **do**
22: **if** !CONTAINS(old_seq, c) **then**
23: $old\_seq = old\_seq \bullet c$;
24: **return** old_seq;
25: **end function**

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.

5.2.4 Discussion

Handling Faults in the Fast Case

A result that was stated in the original Generalized Paxos paper [19] is that to tolerate f crash faults and allow for fast ballots whenever there are up to e crash faults, the total system size N must uphold two conditions: $N > 2f$ and $N > 2e + f$. Additionally, the fast and classic quorums must be of size $N - e$ and $N - f$, respectively. This implies that there is a price to pay in terms of number of replicas and quorum size for being able to run fast operations during faulty periods. An interesting observation is that since Byzantine fault tolerance already requires a total system size of $3f + 1$ and a quorum size of $2f + 1$, we are able to amortize the cost of both features (i.e., we are able to tolerate the maximum number of faults for fast execution without paying a price in terms of the replication factor and quorum size).

Universally Commutative Commands

In the CFT version of the protocol, we mentioned that a downside of taking advantage of commutative commands to reduce synchronization requirements is that the commutativity check must be done at runtime. However, a possible extension to the protocol considers sequences that don't require the check to be performed because they are certain to be commutative with any other sequence. This extension allows us to optimize the protocol both by bypassing the verification round and by allowing learners to learn these universally commutative sequences after witnessing a reduced quorum of $f + 1$ learners since there are no possible conflicts with other sequences. Furthermore, this reduced quorum is enough for a learner be sure that the proposal was proposed by a valid proposer. This extension would also optimize the protocol's performance in a geo-replicated setting because there could be a significant latency reduction between waiting for the fastest $f + 1$ acceptors and the fastest $N - f$.

Much like the equivalent extension to the CFT protocol, 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. As in the previous protocol, we can increase this optimization's applicability by sending these sequences immediately to the learners, without appending them to previously accepted ones. However, in BGP 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 9 lines $\{$19-20$\}$, Algorithm 11 lines $\{$29-30,40-41$\}$ and Algorithm 12 lines $\{$14-18$\}$.

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 why acceptors accumulate state continuously is because each new proven sequence must contain any previous proven sequence. This ensures that an asynchronous network can't 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 don't 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. 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 doesn't, 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 doesn't, 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. 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.

For brevity, this extension to the protocol isn't included in the pseudocode description.

Fast Byzantine Paxos Comparison

In comparison to the FaB Paxos protocol, our Byzantine Generalized Paxos protocol requires a lower number of acceptors than what is stipulated by FaB Paxos' lower bound [15]. However, this does not constitute a violation of the result since BGP does not guarantee a two step execution in the Byzantine scenario. Instead, BGP only provides a two communication step latency when proposed sequences are universally commutative with any other sequence. In the common case, BGP requires three messages steps for a sequence to be learned. In other words, Byzantine Generalized Paxos introduces an additional broadcast phase to decrease the requirements regarding the minimum number of acceptor processes. This may be a sensible trade-off in systems that target datacenter environments where communication between machines is fast and a high percentage of costs is directly related to equipment. The fast communication links would mitigate the latency cost of having an additional phase between the acceptors and the high cost of equipment and power consumption makes the reduced number of acceptor processes attractive.

5.3 Correctness Proofs

This section argues for the correctness of the Byzantine Generalized Paxos protocol in terms of the specified consensus properties.

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
$\perp$	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

Table 5.1: BGP proof notation

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

   Proof:

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

   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., $s \bullet \sigma_1 \sim x \bullet \sigma_2,\ \exists \sigma_1, \sigma_2 \in \mathcal{P} \cup \{\perp\}, \forall x \in \mathcal{P}$) 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 12, 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 12, {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, $\forall s, s' \in \mathcal{P}, \forall b, b' \in \mathcal{B}, maj\_accepted(s, b) \wedge maj\_accepted(s', b') \implies \exists \sigma_1, \sigma_2 \in \mathcal{P} \cup \{\perp\}, s \bullet \sigma_1 \sim s' \bullet \sigma_2$

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 11 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 11 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 9 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. Note that, the verification in Algorithm 11 line $\{$27$\}$ prevents a Byzantine leader from sending a sequence that isn't 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 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 9, 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.

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 11 lines $\{$16,30,41$\}$ and Algorithm 12 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 11 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.

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 12 lines $\{$12,18,20-26$\}$, a correct learner can only append new commands to its $learned$ command sequence.

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 11 lines $\{$10-15,28-29,41-42$\}$ and Algorithm 12 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's accepted by a quorum (Algorithm 11, lines $\{$10-15,28-29,41-42$\}$) and, in the second case, it's sent in *phase 1b* messages to the in leader in the next ballot (Algorithm 11, lines $\{$1-4$\}$) and incorporated in the next proposal (Algorithm 9, 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.

Chapter 6

Visigoth Fault Model

6.1 Model

In this chapter, we evolve the protocol to work in the Visigoth fault model. This model brings the dual advantage, compared to the asynchronous Byzantine model, of reducing replication requirements due to its assumptions that arbitrary faults to not originate from coordinated malice and that the network is not as unpredictable as in an asynchronous environment. To take advantage of the Visigoth model's lower replication requirements, we will adopt its fault and synchrony assumptions. However, throughout this section, we will point out some modifications that enable our protocol to preserve its solution of the command history consensus problem as well as retain its Paxos-like structure.

6.1.1 Network Model

Similarly to the Byzantine model, processes communicate by sending and receiving messages through authenticated perfect links where messages sent by non-faulty senders are eventually received exactly once and faulty processes cannot forge messages. However, unlike the two previously defined models, there is a bound on how asynchronous the system can be. In particular, the number of processes that can be slow with respect to some process is bounded by s, where a process p is slow with respect to another process q if one or more messages from p to q (or vice-versa) than longer than T time units to be transmitted. We preserve the original Paxos structure where each process executes an algorithm for each role. The system must also have a minimum number of acceptor processes, each with an identifier in the set $\{0, ..., N-1\}$. As in the previous models, each process can embody multiple roles and the number of proposer and learner processes can be set arbitrarily.

6.1.2 Fault Model

As in VFT, our fault model allows two types of faults: *commission* faults where a process sends a message that doesn't follow the protocol; and *omission* faults where a process crashes or fails to send a message it should have sent. The total number of faults is bounded by u, out of which o correlated

commission faults can occur. Commission faults are correlated if they can result in the sending of messages that can be used to trigger the same message collection step in the protocol (e.g., if two faulty messages both count as *phase 2b* votes for an invalid sequence, then those faults would be considered correlated). A process that hasn't suffered either a commission or omission fault is considered correct. In the VFT paper, it was shown that consensus is not solvable in a system of less than $u + min(u, s) + o + 1$ processes [22]. Therefore, we consider a total number of acceptors $N = u + min(u, s) + o + 1$. As in the original VFT, we focus only on the advantageous case when $u > s$. VFT's additional assumptions allow us to infer that a certain minimum number of processes must have crashed after a specific time period has passed. However, if it's possible for the system to have more slow processes than crashed processes, it becomes impossible to determine if the non-responsive processes are crashed or simply slow. We also consider the Byzantine model where processes can produce digital signatures, since, like in BGP, processes will have to broadcast and gather cryptographic proofs.

6.1.3 Problem Statement

The consensus problem that our Visigoth fault tolerant Generalized Paxos protocol solves is identical to the one solved by BGP. Even though the solution itself differs due the parameterizability of the VFT model, the consensus problem corresponds to the aforementioned command history problem adapted to the possibility of Byzantine behavior. For clarity, we repeat the description of the consensus problem's requirements:

1. **Nontriviality.** If all proposers are correct, $learned_l$ can only contain proposed commands.
2. **Stability.** If $learned_l = seq$ then, at all later times, $seq \sqsubseteq learned_l$, for any sequence seq 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 seq from a correct proposer, and correct learner l, eventually $learned_l$ contains seq.

6.2 Protocol

This section presents our Visigoth fault tolerant Generalized Paxos protocol (VGP). This protocol shares many similarities with BGP. In the following sections, we will describe the protocol while pointing differences introduced by VFT's assumptions.

6.2.1 Overview

Much like the Visigoth model shares many similarities with the Byzantine model, our Visigoth Generalized Paxos protocol shares most of its structure and message pattern with BGP, namely the following two components:

Algorithm 13 Visigoth Generalized Paxos - Proposer p

Local variables: $ballot\_type = \perp$

1: **upon** *receive*$(BALLOT, type)$ **do**
2: $\quad ballot\_type = type$;
3:
4: **upon** *command_request*(c) **do**
5: $\quad$ **if** $ballot\_type == fast\_ballot$ **then**
6: $\quad\quad$ SEND$(P2A_FAST, c)$ to acceptors;
7: $\quad$ **else**
8: $\quad\quad$ SEND$(PROPOSE, c)$ to leader;

- **View Change** – The goal of the view change subprotocol is to ensure that one of the proposers is elected as the leader, who helps the agreement protocol to make progress. If the current leader is perceived to be preventing progress, the acceptors share their suspicions with each other and, if enough suspicions are gathered, they elect a new leader.
- **Agreement** – The goal of the agreement subprotocol is to extend the learners' learned sequence with commands proposed by the proposers. In order to do this, acceptors cast their votes in either *classic* or *fast* ballots, where fast ballots incur in fewer message steps but may run into contention among concurrent requests, requiring a subsequent classic ballot to fix the conflict.

Next, we describe solely the agreement subprotocol since it's the only one affected by the Visigoth model. The view change subprotocol is identical to the one used in Byzantine Generalized Paxos.

6.2.2 Agreement Protocol

Much like its Byzantine counterpart, VGP preserves the original structure of the Fast Paxos protocol where ballots can be either classic or fast. Proposers send their proposals to either the leader or the acceptors, depending on the type of ballot being currently executed. For clarity, we describe the protocol in its entirety albeit more succinctly than in the previous chapter. We start by explaining how classic and fast ballots interact with each other, noting how this interaction is critical to safety, and then we briefly describe the protocol's behavior in both of these types of ballots.

The purpose of fast ballots is to allow proposers to bypass the leader by sending their proposals directly to the acceptors. Conflicts between non-commutative sequences may split the acceptors' votes in a way such that neither value can achieve the necessary quorum and progress isn't being made. In this situation, the protocol falls back to a classic ballot in order for the leader to propose a single serialization that includes the conflicting sequences. However, even though two non-commutative sequences are prevented from both being accepted in the same ballot, it's still possible for safety to be threatened when considering sequences that are accepted in different ballots. If any sequence could be proposed in each ballot, it would be possible for consistency to be violated because two non-commutative sequences could be accepted in different ballots while their respective *phase 2b* votes could also be delayed in a way such that they reach different learners in different orders.

Both BGP and VGP solve this problem by noting that if each sequence is an extension of previous sequences then the order in which they are incorporated in the learner's $learned$ sequence is irrelevant.

Algorithm 14 Visigoth Generalized Paxos - Leader I

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** $valid\_proofs > u$ **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 p_i **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** $valid\_proofs > N - u$ **then**
35: $accepted[ballot_l][a] = proven$;
36: $notAccepted[ballot_l] = notAccepted[ballot_l] \bullet (val_a \setminus proven)$;
37:
38: **if** $\#(accepted[ballot_l]) \geq N - u$ **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: $proof = \langle ballot_l, view, maxTried_l \rangle_{priv_l}$;
46: SEND$(P2A_CLASSIC, ballot_l, view, maxTried_l, proof)$ to acceptors;
47: $proposals = \bot$;
48: **end function**

For a learner to be able to learn a sequence that is prefixed with possibly previously learned commands, it must accumulate learned commands in its $learned$ sequence, in order to detect duplicates in incoming messages. To maintain the invariant that every sequence that obtains a quorum of votes must be an extension of previous proven sequences, we must consider separately when and how sequences are assembled and voted for in each type of ballot.

In fast ballots, it's trivial to guarantee this invariant since we just need to accumulate each proposed command at the acceptors such that each new proposal is appended to a sequence of previous proposals (e.g., variable val_a in Algorithm 15). Whenever a command or sequence of commands is appended to the sequence of accumulated proposals, the acceptor executes the verification phase and, after gathering a quorum of valid proofs, it sends a *phase 2b* message voting for the entire sequence, not just the new proposal sent by the proposer. Classic ballots, however, require special care, since proposals are assembled solely by the leader. To maintain our previously defined invariant, the leader must be aware of which sequences gathered a quorum of votes. Much like in BGP, this is ensured by having the acceptors gather a quorum of $N - f$ proofs from themselves and other acceptors such that they can send them to the leader in their *phase 1b* messages. After waiting for a quorum of these messages, the leader can not only know if some reported sequence was voted for by at least $f+1$ acceptors, precluding a non-commutative sequence from being accepted by a quorum, but it also knows this for any sequence which may have been learned.

Next, we will present the protocol in detail. We start our description with fast ballots since their execution affects how proposals are assembled in classic ballots.

Fast Ballots

As previously mentioned, in fast ballots proposals are sent by the proposers directly to the acceptors in order to bypass the leader and allow a faster execution. This ballot can be described in the following steps:

Step 1: Proposers to acceptors. The leader starts by informing proposers and acceptors that a fast ballot is starting by sending a message at the beginning of the ballot. For the remainder of the ballot the proposers send their proposed commands or sequences of commands to acceptors. In order to maintain the invariant that any sequence accepted by a quorum is an extension of previous ones, the acceptors append these sequences to the sequence that they maintain locally.

Step 2: Acceptors to acceptors. In order for the acceptors to be able to prove that, with respect with some sequence that they voted for, no other non-commutative sequence may have been learned, they broadcast a signed tuple with the current ballot and the sequence. After gathering $N - f$ such proofs and validating their signatures, the acceptors can prove to the leader that at least $f+1$ correct acceptors voted for the same sequence which means that no non-commutative sequence may have been learned since there are only another $2f$ acceptors in the system.

Step 3: Acceptors to learners The acceptors issue their votes to the learners containing not only the current ballot and sequence but also the $N - f$ proofs. The learners validate the proofs contained in the votes and count the vote if at least $N - f$ of the signatures are valid. After gathering $N - f$ such

votes, a learner is sure that, due to the quorum intersection property, at least one correct acceptor in its *phase 2b* quorum will also participate in the next classic ballot's *phase 1b* quorum and report the current sequence to the leader.

Classic Ballots

Classic ballots in VGP work in a similar way to the original Generalized Paxos protocol [19]. In *phase 1a*, the leader sends a message prompting the acceptors to report back with their accumulated sequence of values and the proofs for the sequences they have voted for. In *phase 1b*, the acceptors respond to the leader's request by sending the sequence of commands they accumulated in the previous ballot and the proofs they gathered for the values they have voted for. Note that their sequence of accumulated commands may contain a sequence for which they don't have proofs. This means that the verification phase didn't complete for that sequence and it wasn't learned. This unproven is sequence is included so that it can be committed in the classic ballot, ensuring liveness. When the leader receives $N - f$ *phase 1b* messages from the acceptors, it's guaranteed that any value accepted by a quorum in the previous ballot will be present in the *phase 1b* quorum and any value in the quorum for which there are $N - f$ proofs was or will be learned. With this knowledge the leader can assemble its proposal sequence by concatenating the proven sequences (i.e., the sequences reported in *phase 1b* messages for which the leader has $N - f$ proofs), the unproven sequences (i.e., the sequences report in *phase 1b* messages for which the leader doesn't have $N - f$ proofs) and the proposals that the leader has received directly from proposers.

After assembling its proposal, the leader sends it to the acceptors in *phase 2a* messages. The acceptors then broadcast their verification messages among each other. As in the fast ballots, these messages contain the current ballot, the sequence being verified and a signed tuple of the same ballot and sequence. After receiving $N-f$ verification messages with valid proofs, the acceptors send a *phase 2b* message to the learners expressing their vote for the sequence. This vote contains the current ballot, the sequence being voted for and the $N - f$ proofs gathered in the previous phase. Even though each vote contains $N - f$ proofs, the learner waits for $N - f$ such votes before learning the sequence. This is because, for the learner to safely learn a sequence it must be sure that, not only $f + 1$ correct acceptors agreed to vote for the sequence but also, of the $N - f$ acceptors whose vote is in the *phase 2b* quorum, at least one of them will be a correct acceptor that also participates in the next classic ballot's *phase 1b* in order to inform the leader that the sequence was learned.

Quorum Gathering in VFT

Note that until this point, the VGP protocol seems entirely indistinguishable from its Byzantine counterpart. However, the improved latency showcased in VFT's state machine replication protocol (VFT-SMaRt) is made possible by the Visigoth model's increased assumptions regarding process synchrony. These assumptions are neatly encapsulated in the Quorum Gathering Primitive (QGP), which implements the actions of gathering a quorum of messages for a given step of the protocol. Since these

Algorithm 15 Visigoth Generalized Paxos - Acceptor a (agreement)

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

1: **upon** *receive*$(P1A, ballot, view_l)$ from leader l **do**
2: **if** $view_l == view$ and $bal_a < ballot$ **then**
3: SEND$(P1B, ballot, bal_a, proven, val_a, proofs[bal_a])$ to leader;
4: $bal_a = ballot$;
5:
6: **upon** *receive*$(FAST, ballot, view_l)$ from leader **do**
7: **if** $view_l == view$ **then**
8: $fast\_bal[ballot] = true$;
9:
10: **upon** *receive*$(P2A\_CLASSIC, ballot, view_l, value, proof)$ **do**
11: **if** $view_l == view$ and $proof_{pub_{leader}} == \langle ballot, view_l, value\rangle$ **then**
12: SEND$(P2A_CLASSIC, ballot, view_l, value, proof)$ to acceptors;
13: PHASE_2B_CLASSIC$(ballot, value)$;
14:
15: **upon** *receive*$(P2A\_FAST, value)$ from proposer **do**
16: PHASE_2B_FAST$(value)$;
17:
18: **upon** *receive*$(VERIFY, view_i, ballot_i, val_i, proof)$ from acceptor i **do**
19: **if** $proof_{pub_i} == \langle ballot_i, val_i\rangle$ or $view == view_i$ **then**
20: $proofs[ballot_i][val_i][i] = proof$;
21: **if** $\#(proofs[ballot_i][val_i]) \geq N - u$ **then**
22: $proven = val_i$;
23: SEND$(P2B, ballot_i, val_i, proofs[ballot_i][value_i])$ to learners;
24:
25: **upon** *receive*$(VERIFY\_REQ, ballot_l, value_l)$ from learner l **do**
26: **if** $\#(proofs[ballot_l][value_l]) \geq N - u$ **then**
27: SEND$(VERIFY_RESP, ballot, value, proofs[ballot_l][value_l])$;
28:
29: **function** PHASE_2B_CLASSIC$(ballot, value)$
30: $univ\_commut =$ ISUNIVERSALLYCOMMUTATIVE(val_a);
31: **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**
32: $bal_a = ballot$;
33: **if** $univ\_commut$ **then**
34: SEND$(P2B, bal_a, value)$ to learners;
35: **else**
36: $val_a = value$;
37: $proof = \langle ballot, val_a\rangle_{priv_a}$;
38: $proofs[ballot][val_a][a] = proof$;
39: SEND$(VERIFY, view, ballot, val_a, proof)$ to acceptors;
40: **end function**
41:
42: **function** PHASE_2B_FAST$(ballot, value)$
43: **if** $ballot == bal_a$ and $fast\_bal[bal_a]$ **then**
44: **if** ISUNIVERSALLYCOMMUTATIVE$(value)$ **then**
45: SEND$(P2B, bal_a, value)$ to learners;
46: **else**
47: $val_a = val_a \bullet value$;
48: $proof = \langle ballot, val_a\rangle_{priv_a}$;
49: $proofs[ballot][val_a][a] = proof$;
50: SEND$(VERIFY, view, ballot, val_a, proof)$ to acceptors;
51: **end function**

assumptions are encapsulated in a single primitive, it seems logical to simply substitute BGP's quorum gathering procedure with the QGP. However, unlike the Fast and Generalized Paxos protocols, the VFT-SMaRt protocol doesn't specialize its algorithms by role, considering instead a system of N identical processes [22]. Since we wish to preserve the role specialization present in Generalized Paxos, we must adapt VFT-SMaRt's quorum gathering procedure to a Paxos-like structure.

The Visigoth model differs from its Byzantine counterpart by allowing s processes to be slow but correct [22]. A process i is defined to be slow with respect to j if messages from i to j (or vice-versa) take more than T time units to be transmitted. This assumption allows us to gather more efficient quorums by leveraging the knowledge that only s processes can take more than T time units to send and process a message. In VFT's Quorum Gathering Primitive, a process gathers a quorum by waiting for messages from $N - s$ distinct processes. Since the Visigoth model provides a bound on the time messages from correct processes take to be transmitted, a gatherer process can set a timer that allows all correct and non-slow messages to be delivered. After a timeout occurs, of the x processes that are unresponsive, only s of them may be slow, which means that $x - s$ processes must be faulty (recall that we consider only the case when $u > s$). This allows us to leverage the assumption of s slow but correct processes to decrease the quorum size to $N - u$ while still guaranteeing the intersection properties necessary for safety. We make use of this mechanism to adapt our Byzantine Generalized Paxos protocol to the Visigoth fault model. This includes changing the quorum gathering in *phase 1b*, where acceptors relay their previous votes to the leader, in the verification phase, where acceptors exchange cryptographic proofs, and *phase 2b*, where acceptors send their votes to the learners.

In our previous implementation of the Byzantine Generalized Paxos protocol, the nature of the consensus problem combined with the possibility of faults required altering the message pattern to contain an additional broadcast round. In the Visigoth model, this broadcast round serves a new purpose in addition to the gathering of proofs. In Byzantine Generalized Paxos, a Byzantine leader can at most prevent progress until a new leader is elected. Even if a leader causes a split vote by sending different values to some acceptors in its *phase 2a* messages, at most one of those values can obtain the $N - u$ votes required to be learned. However, since in the Visigoth model we want to take advantage of the additional assumptions to reduce the quorum to $N - u$, it becomes possible for the leader to attempt a split vote by sending two different values to two sets of acceptors and ignoring others such that the ignored acceptors are more than s and the timeout is reached, causing the required quorum size to be reduced. Since o of the remaining acceptors can vote for both values and the leader could attempt a split vote between the remaining $u + s + 1$ (including the acceptors that were previously ignored), there are enough votes for both values to be committed, violating the safety property. One configuration where this would happen would be with $u = o = 2,\ s = 1,\ N = u + o + \min(u, s) + 1 = 6$. In this system the initial quorum is $N - s = 5$ and the reduced quorum is $N - u = 4$. If the leader sends v_1 to two acceptors (one correct and one faulty), v_2 to other two (also one correct and one faulty) and ignores the last two, then the timeout is reached and the required quorum size is reduced from 5 to 4. In this case, each value has two votes from both of the Byzantine acceptors and one vote from one of the two correct acceptors that received the split vote. Since the previously ignored acceptors are correct, the leader can

employ another split vote to divide them between v_1 and v_2 to achieve four votes for both values.

To prevent this situation, when acceptors receive a *phase 2a* message from the leader, they must replay it to all other acceptors. This prevents the leader from purposely ignoring acceptors to force the quorum to decrease. Since the leader can employ a split vote in a way such that at most o acceptors receive a *phase 2a* for each sequence, the acceptors can't rely on receiving $f + 1$ replays in order to be sure that they are correct. To ensure that the replay message originated from the leader and not from a Byzantine acceptor, it must include the leader's signature. After receiving this message the acceptor executes the algorithm just as it would if the message had come from the leader (Algorithm 15 lines $\{$10-12$\}$).

Timings

One important aspect of the original Visigoth description is related to the setting of timeouts. In VFT, all processes are expected to initiate the quorum gathering so they may commit a value. The timeouts used by the Quorum Gathering Primitive to trigger the quorum reduction must be set according to both the upper bound on message latency between processes and the maximum relative delay that can be expected between two processes initiating the primitive. This is encoded in the following property guaranteed by Quorum Gathering Primitive:

> *Safety-Intersection*: if there are two instances of QGP, such that all correct processes that are not crashed and not slow towards the respective gatherer processes initiate the protocol within δ time window such that $T + \delta < T_{QGP}$, then, if the two correct gatherers p and p' gather M and M' respectively, M and M' intersect in at least one correct replica.

This specification is directly related to the fact that all correct processes are expected to gather messages from a quorum, which is why the timeout value T_{QGP} must be larger than the synchrony bound T plus the maximum time δ that can elapse between two processes initiating the gathering procedure. However, since in our system model processes are differentiated by roles as in traditional Paxos, the gathering logic will be reflected at multiple roles. For brevity, the entire procedure is only explicit at the learners' pseudocode (Algorithm 16). The aforementioned difference in system modeling requires us to reason about the maximum delay that can be observed between two votes sent in any given message step. Take the worst case scenario where two message paths observe the biggest possible latency difference at the learners after being sent by the leader: in the first case, the *phase 2a* message reaches an acceptor almost immediately and triggers the sending of a *phase 2b* message that also reaches the learner almost immediately; the second message takes the maximum amount of time T that a non-faulty and non-slow message can take to reach some acceptors. Consider that the leader is also Byzantine and ignores a subset of the acceptors. Due to this behavior, some acceptors will have to wait for the replay broadcast sent by other acceptors to be aware of the *phase 2a* message. Suppose that, for both the replay round and the corresponding *phase 2b* message it triggers, it takes T time units for the message to reach the intended recipients. In this case, a total of $3T$ message delays will separate

Algorithm 16 Visigoth Generalized Paxos - Learner I

Local variables: $learned = \bot$, $storedProofs = \bot$, $quorumSize = \bot$, $verificationMessages = \bot$, $validVotes = \bot$

1: **upon** *receive(*$P2B, ballot, value, proofs$*)* from acceptor a **do**
2: $storedProofs[ballot][value][a] = \langle a, proofs \rangle$;
3: $valid\_proofs = 0$;
4: **for** i **in** $acceptors$ **do**
5: $proof = proofs[i]$;
6: **if** $proof_{pub_i} == \langle ballot, value \rangle$ **then**
7: $valid\_proofs \mathrel{+}= 1$;
8:
9: **if** $valid\_proofs \geq N - s$ **then**
10: $validVotes[ballot][value][a] = proofs$;
11:
12: **if** $quorumSize[ballot][value] = \bot$ **then**
13: $quorumSize[ballot][value] = N - s$;
14: STARTTIMER$(3T,$ TIMERENDED$, ballot, value)$;
15:
16: **if** $(\#(validVotes[ballot][value]) \geq quorumSize[ballot][value]$ and $(quorumSize[ballot][value] = N - s$ or $verificationMessages[ballot][value] \geq N - u)$ **then**
17: $learned =$ MERGE_SEQUENCES$(learned, value)$;
18:
19: **upon** *receive(*$P2B, ballot, value$*)* from acceptor a **do**
20: **if** ISUNIVERSALLYCOMMUTATIVE$(value)$ **then**
21: $validVotes[ballot][value][a] = true$;
22: **if** $\#(validVotes[ballot][value]) > u$ **then**
23: $learned = learned \bullet value$;
24:
25: **upon** *receive(*$VERIFY\_RESP, ballot, value$*)* from acceptor a **do**
26: $verificationMessages[ballot][value][a] = true$;
27: **if** $(\#(validVotes[ballot][value]) \geq quorumSize[ballot][value]$ and $(quorumSize[ballot][value] = N - s$ or $verificationMessages[ballot][value] \geq N - u)$ **then**
28: $learned =$ MERGE_SEQUENCES$(learned, value)$;
29:
30: **function** TIMERENDED$(ballot, value)$
31: $quorumSize[ballot][value] = N - u$;
32:
33: **for** a **in** $acceptors$ **do**
34: $acc\_proofs = storedProofs[a]$;
35: $valid\_proofs = 0$;
36: **for** i **in** $acceptors$ **do**
37: $proof = acc\_proofs[i]$;
38: **if** $proof_{pub_i} == \langle ballot, value \rangle$ **then**
39: $valid\_proofs \mathrel{+}= 1$;
40:
41: **if** $valid\_proofs \geq N - u$ **then**
42: $validVotes[ballot][value][a] = proofs$;
43:
44: SEND$(VERIFY_REQ, ballot)$ to acceptors;
45: **end function**
46:
47: **function** MERGE_SEQUENCES(old_seq, new_seq)
48: **for** c **in** new_seq **do**
49: **if** !CONTAINS(old_seq, c) **then**
50: $old\_seq = old\_seq \bullet c$;
51: **return** old_seq;
52: **end function**

the two message paths without either of them being faulty or slow. Therefore, the timeout value T_{QGP} at the learners must be set to $3T$.

Quorums

There are several aspects of VGP that are dependent on possible quorum sizes. As is the case with BGP, we must ensure that the leader receives a *phase 1b* message from at least one correct acceptor relaying the most recently learned sequence along with enough proofs to ensure that no other non-commutative sequence may have been learned. Since the only aspect of the VGP protocol that differs significantly from BGP is the procedure through which the quorums are gathered, the same reasoning for correctness applies. As long as quorums are guaranteed to intersect in more than o acceptors, we are sure that two non-commutative sequences cannot both receive a quorum of verification messages. In order to demonstrate the protocol's correctness, section 6.3 shows that this condition is upheld for any two gathered quorums.

However, even though we are able of extending BGP to the Visigoth model using VFT-SMaRt's QGP to collect messages without introducing additional complexity to the protocol's message pattern, there exists some potential for conflict between the two protocols when we consider that they are designed to expect different quorum sizes. In particular, BGP requires acceptors to gather $N - f$ proofs in order to ensure the leader of the following classic ballot that enough acceptors committed to some sequence. Since learners only learn after witnessing $N - f$ collections of such proofs, we also guarantee that, when a sequence is learned, not only no other non-commutative sequence is learned in the same ballot but also the leader of the next classic ballot will be aware of every learned sequence. However, as we previously mentioned, VFT uses either a quorum of $N - s$ acceptors or a smaller quorum of $N - u$ during its message gathering procedure. Although the original VFT protocol proves that any two quorums intersect, it's unclear what effect the variable quorums have on the collection of proofs in the verification phase. In particular, when a learner receives a vote, he has no way of knowing whether to expect $N - s$ or $N - u$ proofs.

In the interest of preserving safety, VGP initially assumes that only the s slow processes may be silent for more than $3T$ time units and requires at least $N - s$ proofs in order to accept a *phase 2b* message. If a phase $2b$ message carries with it less than $N - s$ proofs, it's not counted as a valid vote in *phase 2b*. However, since we want the protocol to progress in the presence of faults, *phase 2b* messages that don't contain at least $N - s$ proofs are stored and, if the timeout lowers the minimum quorum size to $N - u$, the *phase 2b* messages are rechecked in order to see if they contain at least $N - u$ proofs (Algorithm 16 lines $\{$30-42$\}$). This rechecking can be done simply by recounting the valid proofs in each vote and considering not only those with at least $N - s$ votes but also those with at least $N - u$ votes. If more than $N - u$ such votes exist, the learner moves on to gathering a verification quorum that must be assembled whenever the required quorum is reduced from $N - s$ to $N - u$. The reason why this verification quorum is needed is due to the possibility of multiple quorums being assembled concurrently, as was described in the original VFT description [22]. As previously mentioned, it's important to note that, for conciseness, the QGP is only explicit at the learner. However, both the leader and the acceptors also implement the

exact same behavior when gathering a quorum of messages. Therefore, when an acceptor gathers verification messages, it first waits for $N - s$ such messages but, after a timeout of $3T$, it can send its *phase 2b* vote with just $N - u$ votes.

6.2.3 Discussion

Universally Commutative Commands

As is the case with Byzantine Generalized Paxos, there is the possibility of extending VGP by handling universally commutative commands differently than commands that may not commute with others. Since the impact and requirements of this optimization have been discussed at length in other chapters, we will refrain from repeating the same points here. However, it's interesting to note how the command history problem (and its solution) are well suited for an adaptation to the VFT model, since the advantages of both are aligned. The Visigoth model's additional assumptions make it specially useful in a datacenter environment where not only coordinated malicious behavior is unlikely, due to the high security barriers in place, but also the majority of costs scale inversely with the system's throughput capability (i.e., higher traffic requires more servers, switches, cooling equipment). Using the commutativity assumption to reduce coordination requirements, a protocol that solves the command history problem can improve both latency and the total number of messages exchanged. The extension of handling universally commutative commands increases the protocol's throughput and latency gains even further, such that the fault model, the consensus problem and the protocol's implementation all contribute to a system well suited to be used in a datacenter-like environment.

6.3 Correctness Proofs

This section argues for the correctness of the Visigoth Generalized Paxos protocol in terms of the specified consensus properties.

Consistency

Theorem 5. *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, $\forall seq, seq' \in \mathcal{P}, \forall l_i, l_j \in \mathcal{L}, learned(l_j, seq) \wedge learned(l_i, seq') \implies \exists \sigma_1, \sigma_2 \in \mathcal{P} \cup \{\bot\}, seq \bullet \sigma_1 \sim seq' \bullet \sigma_2$

Proof:

1.1. At any given instant, $\forall seq, seq' \in \mathcal{P}, \forall l_i, l_j \in \mathcal{L}, \forall b, b' \in \mathcal{B}, learned(l_i, seq) \wedge learned(l_j, seq') \implies \exists \sigma_1, \sigma_2 \in \mathcal{P}, \forall x \in \mathcal{P}, (acc\_send(N-s, seq, b) \vee (acc\_send(N-u, seq, b) \wedge wait\_learn(3T, l_i, seq, b)) \vee (min\_accepted(seq, b) \wedge seq \bullet \sigma_1 \sim x \bullet \sigma_2)) \wedge (acc\_send(N-s, seq', b') \vee (acc\_send(N-u, seq', b') \wedge wait\_learn(3T, l_j, seq', b')) \vee (min\_accepted(seq', b') \wedge seq' \bullet \sigma_1 \sim x \bullet \sigma_2))$

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
$\perp$	Empty command
$learned_{l_i}$	Learner l_i's $learned$ sequence of commands
$learned(l_i, s)$	$learned_{l_i}$ contains the sequence s
$min\_accepted(s, b)$	$u+1$ acceptors sent *phase 2b* messages to the learners for sequence s in ballot b
$proposed(s)$	A correct proposer sent a message with proposal s
$acc\_send(x, s, b)$	At least x acceptors have sent verification messages for the sequence s in ballot b
$wait\_learn(t, l, s, b)$	After t time units have passed since the learner l received the first *phase 2b* message with a sequence s in ballot b, $N - s$ *phase 2b* messages haven't been gathered

Table 6.1: VGP proof notation

Proof: A sequence can be learned if: (1) the learner gathers either $N - s$ votes (Algorithm 16, lines $\{$1-17$\}$), (2) the learner gathers $N - u$ votes after a timeout of $3T$ occurs and a verification quorum is gathered (Algorithm 16, lines $\{$25-28,30-45$\}$) or (3) if the sequence is universally commutative and the learner gathers $f + 1$ votes (Algorithm 16, lines $\{$19-23$\}$). The latter is encoded in the logical expression $s \bullet \sigma_1 \sim x \bullet \sigma_2$ which is true if the 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 since the definition of conflicting sequences are sequences which cannot be extended to equivalent sequences.

1.2. At any given instant, $\forall seq, seq', x \in \mathcal{P}, \exists \sigma_1, \sigma_2, \sigma_3, \sigma_4 \in \mathcal{P} \cup \{\perp\}, \forall b, b' \in \mathcal{B}, (acc\_send(N - s, seq, b) \vee (acc\_send(N - u, seq, b) \wedge wait\_learn(3T, l_i, seq, b))) \wedge (acc\_send(N - s, seq', b') \vee (acc\_send(N - u, seq', b') \wedge wait\_learn(3T, l_j, seq', b'))) \implies \exists \sigma_5, \sigma_6 \in \mathcal{P} \cup \{\perp\}, seq \bullet \sigma_5 \sim seq' \bullet \sigma_6$

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

1.2.1. At any given instant, $\forall seq, seq' \in \mathcal{P}, \forall b \in \mathcal{B}, (acc\_send(N - s, seq, b) \vee (acc\_send(N - u, seq, b) \wedge wait\_learn(3T, l_i, seq, b))) \wedge (acc\_send(N - s, seq', b) \vee (acc\_send(N - u, seq', b) \wedge wait\_learn(3T, l_j, seq', b))) \implies \exists \sigma_1, \sigma_2 \in \mathcal{P} \cup \{\perp\}, seq \bullet \sigma_1 \sim seq' \bullet \sigma_2$

Proof: To prove that if any two sequences are accepted then they must be extensible to equivalent sequences, we must prove that two non-commutative sequences can't both gather a quorum of *phase 2b* messages to be learned. In order to gather for correct acceptors to send a *phase 2b* message for a sequence, they must gather $N - f$ verification messages (Algorithm 15, lines $\{$18-23$\}$). A correct acceptor will not send phase verification messages for two non-commutative sequences because it will only receive the commands in them once and therefore, only assemble them in one possible serialization (Algorithm 15, lines $\{$10,13$\}$). Since we know that a correct acceptor will only send one phase verification message for one possibly non-commutative sequence, we must prove that, in any two gathered quorums, the intersection between them contains at least one correct process.

Gathered quorums can be of size $Q_1 = N - s$ or $N - u \leq Q_2 < N - s$, which means there are three possible combinations: (1) two quorums of size Q_1, (2) two quorums of size Q_2 and (3) one quorum of size Q_1 and one quorum of size Q_2.

To guarantee that two quorums of size Q_1 intersect in at least one correct process, it must hold that $N - s + N - s - N > o$. Assuming the worst case scenario where $N = u + o + s + 1$:

$$\begin{aligned} N - s + N - s - N &> o \\ N - 2s &> o \\ u + o + s + 1 - 2s &> o \\ u + 1 - s &> 0 \\ u + 1 &> s \end{aligned}$$

Since we already assume that $u > s$, the condition holds.

Given two quorums with a size that is somewhere in the interval $N-u \leq Q_2 < N-s$, when the first quorum is gathered the sizes are $q_1 = N-u+a$ and $q_2 = N-u+b$. The quantities a and b are such that q_1 and q_2 may be in some arbitrary position within the aforementioned interval. This means that there are $u-a-s$ crashed processes since, out of the maximum number of processes u that can fail to participate in the quorum, a did participate and s did not but may be slow instead of faulty. Therefore, when these two quorums are assembled, we know that the system size is $N' = N - u + a + s$. To guarantee that the intersection between the quorums contains at least one correct process, the following must hold $q_1 + q_2 - N' > o$.

$$q_1 + q_2 - N' > o$$
$$N - u + a + N - u + b - N + u - a - s > o$$
$$N - u + b - s > o$$
$$u + o + s + 1 - u + b - s > o$$
$$b + 1 > 0$$

Given a quorum of size $Q_1 = N - s$ and a quorum of size $Q_2 = N - u + a$, at least $u - a - s$ processes are faulty since, out of the maximum number of processes u that can fail to participate in the quorum, a did participate and s did not but may be slow instead of faulty. This means that, when the quorums are gathered, the system's size is $N' = N - u + a + s$. To guarantee that the intersection between quorums contains at least one correct process, the following must hold $Q_1 + Q_2 - N' > o$:

$$Q_1 + Q_2 - N' > o$$
$$N - s + N - u + a - N + u - a - s > o$$
$$N - 2s > o$$
$$u + o + s + 1 - 2s > o$$
$$u - s + 1 > 0$$

Since we already assume that $u > s$, the condition holds.

1.2.2. At any given instant, $\forall s, s' \in \mathcal{P}, \forall b, b' \in \mathcal{B}, (acc\_send(N-s, seq, b) \vee (acc\_send(N-u, seq, b) \wedge wait\_learn(3T, l_i, seq, b))) \wedge (acc\_send(N - s, seq', b') \vee (acc\_send(N - u, seq', b') \wedge wait\_learn(3T, l_j, seq', b'))) \wedge b \neq b' \implies \exists \sigma_1, \sigma_2 \in \mathcal{P} \cup \{\perp\}, seq \bullet \sigma_1 \sim seq' \bullet \sigma_2$

Proof: To prove that values accepted in different ballots are extensible to equivalent sequences, it suffices to prove that for any sequences seq and seq' accepted at ballots b and b', respectively, such that $b < b'$ then $seq \sqsubseteq seq'$. By Algorithm 15 lines $\{$18-23$\}$, 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 15 lines $\{$36,47$\}$, 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 14 lines $\{$24-39,41-48$\}$, the leader's proposal is prefixed by the largest of the proven sequences (i.e., val_a sequences that receives $2f + 1$ votes) relayed by a quorum of acceptors in *phase 1b* messages. Note that, the verification in Algorithm 15 line $\{31\}$ prevents a Byzantine leader from sending a sequence that isn't 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 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 seq in the *phase 1b* quorum (which, by our hypothesis, is an extension of any previous proven sequences), by picking seq as the prefix of its *phase 2a* proposal (Algorithm 14, lines $\{$41-48$\}$), 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 seq and seq', at any given instant, $\forall x \in \mathcal{P}, \exists \sigma_1, \sigma_2, \sigma_3, \sigma_4 \in \mathcal{P} \cup \{\perp\}, \forall b, b' \in \mathcal{B}, (acc\_send(N-s, seq, b) \vee (acc\_send(N-u, seq, b) \wedge wait\_learn(3T, l_i, seq, b)) \vee (min\_accepted(seq, b) \wedge seq \bullet \sigma_1 \sim x \bullet \sigma_2)) \wedge (acc\_send(N-s, seq', b') \vee (acc\_send(N-u, seq', b') \wedge wait\_learn(3T, l_j, seq', b') \vee (min\_accepted(seq', b') \wedge seq \bullet \sigma_1 \sim x \bullet \sigma_2))) \implies seq \bullet \sigma_3 \sim seq' \bullet \sigma_4$

Proof: By 1.2 and by definition of $seq \bullet \sigma_i \sim x \bullet \sigma_j$

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

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.

Nontriviality

Theorem 6. *If all proposers are correct, $learned_l$ can only contain proposed commands*

Proof:

1. At any given instant, $\forall l_i \in \mathcal{L}, \forall seq \in \mathcal{P}, learned(l_i, seq) \implies \forall x \in \mathcal{P}, \exists \sigma_1, \sigma_2 \in \mathcal{P} \cup \{\perp\}, \forall b \in \mathcal{B}, acc\_send(N-s, seq, b) \vee (acc\_send(N-u, seq, b) \wedge wait\_learn(3T, l_i, seq, b)) \vee (min\_accepted(seq, b) \wedge seq \bullet \sigma_1 \sim x \bullet \sigma_2)$

   Proof: By Algorithm 15 lines {18-23,33-34,44-45} and Algorithm 16 lines {1-17,19-23}, if a correct learner learned a sequence seq at any given instant then either a quorum was gathered or $f+1$ (if seq is universally commutative) acceptors must have executed *phase 2b* for seq.

2. At any given instant, $\forall seq, x \in \mathcal{P}, \exists \sigma_1, \sigma_2 \in \mathcal{P}, \forall b \in \mathcal{B}, acc\_send(N-s, seq, b) \vee (acc\_send(N-u, seq, b) \wedge wait\_learn(3T, l_i, seq, b)) \vee (min\_accepted(seq, b) \wedge seq \bullet \sigma_1 \sim x \bullet \sigma_2) \implies proposed(seq)$

   Proof: By Algorithm 15 lines {10-13, 15-16}, for either a quorum to be gathered 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, $\forall l_i \in \mathcal{L}, \forall seq \in \mathcal{P}, learned(l_i, seq) \implies proposed(seq)$

   Proof: By 1 and 2.

4. Q.E.D.

Stability

Theorem 7. *If $learned_l = seq$ then, at all later times, $seq \sqsubseteq learned_l$, for any sequence seq and correct learner l*

Proof: By Algorithm 16 lines {17,23,28}, a correct learner can only append new commands to its $learned$ command sequence.

Liveness

Theorem 8. *For any correct learner l and any proposal seq from a correct proposer, eventually $learned_l$ contains seq*

Proof:

1. $\forall\, l_i \in \mathcal{L}, \forall seq, x \in \mathcal{P}, \exists \sigma_1, \sigma_2 \in \mathcal{P} \cup \{\perp\}, \forall b \in \mathcal{B}, acc\_send(N-s, seq, b) \vee (acc\_send(N-u, seq, b) \wedge wait\_learn(3T, l_i, seq, b)) \vee (min\_accepted(seq, b) \wedge seq \bullet \sigma_1 \sim x \bullet \sigma_2) \overset{e}{\implies} learned(l_i, seq)$

   Proof: By Algorithm 15 lines {17-22,32-33,45-46} and Algorithm 16 lines {1-17,19-23}, if either a quorum is gathered or $f+1$ (if seq is universally commutative) acceptors accept a sequence seq (or some equivalent sequence), eventually seq will be learned by any correct learner.

2. $\forall seq \in \mathcal{P}, proposed(seq) \stackrel{e}{\Longrightarrow} \forall x \in \mathcal{P}, \exists \sigma_1, \sigma_2 \in \mathcal{P} \cup \{\bot\}, \forall b \in \mathcal{B}, acc\_send(N-s, seq, b) \vee (acc\_send(N-u, seq, b) \wedge wait\_learn(3T, l_i, seq, b)) \vee (min\_accepted(seq, b) \wedge seq \bullet \sigma_1 \sim x \bullet \sigma_2)$

   Proof: A proposed sequence is either conflict-free when it's incorporated into every acceptor's current sequence or it creates conflicting sequences at different acceptors. In the first case, it's accepted by a quorum (Algorithm 15, lines $\{$18-23$\}$) and learned after the votes reach the learners. In the second case, eventually the next ballot is initiated and the sequence is sent in *phase 1b* messages to the leader (Algorithm 15, lines $\{$1-4$\}$). After being sent to the leader, the sequence is incorporated in the next proposal and sent to the acceptors a long with others proposed commands (Algorithm 14, lines $\{$24-39,41-48$\}$).

3. $\forall l_i \in \mathcal{L}, \forall seq \in \mathcal{P}, proposed(seq) \stackrel{e}{\Longrightarrow} learned(l_i, seq)$

   Proof: By 1 and 2.

4. Q.E.D.

Chapter 7

Conclusion

This dissertation focused on the problem of achieving consensus in a distributed system and its solutions. In particular, this work builds upon one of the protocols of the widely known Paxos family of consensus-solving protocols, Generalized Paxos [19]. This protocol makes use of a generic consensus problem that uses special structures, called *c-structs*, to allow for the definition of any consensus problem. By solving this generalized consensus problem, Generalized Paxos is able to present a solution that covers a broad spectrum of problems. One of these problems is the command history problem which makes the observation that commands don't necessarily conflict with each other and may result in the same state being produced regardless of the order in which they are executed. This problem allows coordination requirements to be reduced since not every command is required to be committed in a total order. However, the universality of the generalized consensus specification comes at the cost of a very complex description of both generalized consensus and Generalized Paxos. With this in mind, one of the initial goals of this work was to produce a specification that still took advantage of the command history problem but didn't carry with it all the generality that made the original problem and protocol so complex.

The Paxos protocol family also includes variants that reduce the number of message steps required to learn a value. The aforementioned Generalized Paxos protocol is one of these protocols that enable the system to learn commands in just two message steps in the common case instead of the usual four. This fastness relative to the complexity of the protocol's message pattern comes at the cost of increasing the system's quorum sizes and total number of processes in function of the number of faults that we wish to tolerate while still allowing two-step executions. This work also intended to retain this property whenever doing so didn't imply an unreasonable cost in another aspect of the protocol.

Another facet of the distributed systems literature focuses on the Byzantine fault model, in which protocols are designed to withstand not only crash faults but also arbitrary and possibly malicious behavior by some of the system's participants. Several works make their contributions in this field by extending consensus-solving algorithms to the Byzantine model. However, few works have aligned this goal with the previously stated ones, namely allowing fast executions and taking advantage of the commutativity assumption to reduce coordination requirements. This is a sparse area in the distributed protocols

literature to which we wanted to contribute.

Finally, newer non-crash fault models make the observation that the Byzantine model is too pessimistic in that it allows for up to f processes, in a system of at least $3f + 1$, to display malicious behavior in a coordinated way. In order to reduce the stringent system requirements imposed by the strong faulty behavior permitted to the processes, models like VFT [22] and XFT [33] have introduced specifications that allow the system to tolerate some arbitrary behavior with a lesser cost. VFT also allows the network administrator to parameterize the allowed behavior both in terms of synchronism and fault tolerance. These models are specially suited for datacenter environments where not only is the network highly secured and monitored, making coordinated malicious behavior unlikely, but also the high amount of hardware makes protocols with less requirements more attractive. Protocols can take advantage of the more realistic assumptions provided by these models to become more viable options in scenarios like the one that was just mentioned. Therefore, we also wanted to build a version of the protocol that took advantage of these newer models to be better suited for datacenter environments and perhaps scenarios in which systems are geo-replicated.

7.1 Achievements

In line with the previously mentioned goals, this thesis proposes three protocols, each targeting a different fault model. The contributions are added incrementally such that each protocol contains contributions introduced in previous ones, when applicable.

The first protocol targets the CFT model and its major contribution is the simplification of generalized consensus to a problem that still takes advantage of the commutativity observation without generalized consensus' increased complexity. By specializing generalized consensus into the problem of agreeing on commands histories, the specification of the protocol also becomes easier to understand and less reliant on complicated mathematical formalisms. This is an important contribution since it increases the understandability of both the problem and its solution, making it easier to translate both to an actual implementation. This protocol is described extensively both textually and through pseudocode. We also describe an optimization that allows certain commands to be committed with a reduced quorum. This extension further extends the protocol's applicability to scenarios such as datacenter environments.

Our second major contribution is the adaptation of Generalized Paxos to the Byzantine fault model. In order to build this protocol, we propose a specification of consensus that builds upon our previous simplified consensus problem by taking into account the possibility of Byzantine behavior. With this problem in mind, we describe Byzantine Generalized Paxos, providing both a textual description and also pseudocode that can help the mapping of the protocol into a practical implementation. Two extensions are proposed to this protocol. The first describes how the protocol can take advantage of universally commutative commands to both commit a sequence using a reduced quorum and also bypass the additional verification phase imposed by the possibility of Byzantine faults. The second extension is a checkpointing feature that allows the protocol to deal with the accumulation of state at both the learners and the acceptors. This extension increases the protocol's applicability in practical scenarios since it

allows the system to free up memory space by executing the checkpoint subprotocol when resources demand it. To further convince the reader of this protocol's validity, we also present proofs that argue that the protocol correctly ensures each of the properties described in the consensus specification.

The third protocol, Visigoth Generalized Paxos, and the final contribution of this thesis is the adaptation of BGP to the Visigoth fault model. We believe VFT to be a good fit for Generalized Paxos since VFT's synchrony assumptions and Generalized Paxos commutativity observation have the potential to greatly reduce coordination requirements within a controlled system such as a datacenter. The overall message pattern of VGP is similar to that of its Byzantine counterpart. However, VFT's synchrony assumptions and their effects on quorum sizes have non-obvious consequences on the resolution of the command history problem. For this contribution, the description focuses mostly on the problems introduced by VFT's assumptions and how the protocol is modified to deal with them. The protocol is accompanied by a pseudocode description and correctness proofs, both structured similarly to their analogous versions in BGP.

A final minor contribution that precedes the previously mentioned protocols and consensus problems can be found in chapter 3. This chapter attempts to describe some important but complicated aspects of the original generalized consensus and Generalized Paxos specifications. This is intended to help familiarize the reader with the complexities of the works to which our contributions try to add to and may help shed some light on some of Generalized Paxos' more opaque components.

7.2 Future Work

The contributions proposed in this dissertation can be augmented through several extensions which are mostly aligned with the goal of making the described protocols more attractive in real-world systems. We propose that this can be done either by showcasing its execution in tandem with proven implementations or by reducing our protocols' costs and assumptions.

One of the directions in which this work could be extended would be that of integrating either BGP or our CFT version of Generalized Paxos with a proven implementation such as Zookeeper [4]. This would demonstrate BGP's validity in a realistic environment and also point out any shortcomings that might constrain its ability to handle real workloads in terms of either throughput or latency.

One particularly interesting application of our CFT consensus protocol would be in the context of distributed transactions. The reason why this is a noteworthy domain that might benefit from an integration with our protocol is because it would allow clients to safely replicate transactions throughout a distributed system while taking advantage of the amount of commutativity allowed by each transaction to commit some transactions in a single round-trip. Multi-Data Center Consistency already adapts the original Generalized Paxos protocol to this setting, taking advantage of its assumptions on transaction commutativity [23]. However, our simplified specification of consensus, paired with our description of Generalized Paxos, could present some additional advantages, namely in the learning of transactions that are universally commutative and in the readability and maintainability of the code base. Since universally commutative commands only require a quorum of $f + 1$ acceptors, this extension could allow

transactions to commit with a lower latency, specially in the multi-datacenter scenario.

Some variable consistency models like RedBlue and Explicit Consistency also take advantage of the fact that some operations aren't required to be totally ordered with respect to every other operation [56, 57]. In this aspect, these models share similar assumptions and advantages with Generalized Paxos. It would be interesting to study whether the two approaches are orthogonal in a way such that they can be merged to obtain systems with useful properties.

Bibliography

[1] M. Vukolic. *Quorum Systems: With Applications to Storage and Consensus*. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2012.

[2] M. Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 335–350, Berkeley, CA, USA, 2006. USENIX Association. ISBN 1-931971-47-1. URL http://dl.acm.org/citation.cfm?id=1298455.1298487.

[3] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's globally-distributed database. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 251–264, Berkeley, CA, USA, 2012. USENIX Association. ISBN 978-1-931971-96-6. URL http://dl.acm.org/citation.cfm?id=2387880.2387905.

[4] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, pages 11–11, Berkeley, CA, USA, 2010. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1855840.1855851.

[5] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21 (7):558–565, July 1978. ISSN 0001-0782. doi: 10.1145/359545.359563. URL http://doi.acm.org/10.1145/359545.359563.

[6] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4):299–319, Dec. 1990. ISSN 0360-0300. doi: 10.1145/98163.98167. URL http://doi.acm.org/10.1145/98163.98167.

[7] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, Apr. 1985. ISSN 0004-5411. doi: 10.1145/3149.214121. URL http://doi.acm.org/10.1145/3149.214121.

[8] L. Lamport. Paxos Made Simple. *ACM SIGACT news distributed computing column 5*, 32(4):51–58,

2001. ISSN 01635700. doi: 10.1145/568425.568433. URL http://portal.acm.org/citation.cfm?doid=568425.568433.

[9] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998. ISSN 0734-2071. doi: 10.1145/279227.279229. URL http://doi.acm.org/10.1145/279227.279229.

[10] L. Lamport. The part-time parliament. Technical report, DEC SRC, 1989.

[11] R. D. Prisco, B. W. Lampson, and N. A. Lynch. Revisiting the paxos algorithm. In *Proceedings of the 11th International Workshop on Distributed Algorithms*, WDAG '97, pages 111–125, London, UK, UK, 1997. Springer-Verlag. ISBN 3-540-63575-0. URL http://dl.acm.org/citation.cfm?id=645954.675657.

[12] E. K. Lee and C. A. Thekkath. Petal: Distributed virtual disks. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VII, pages 84–92, New York, NY, USA, 1996. ACM. ISBN 0-89791-767-7. doi: 10.1145/237090.237157. URL http://doi.acm.org/10.1145/237090.237157.

[13] J. Baker, C. Bond, J. C. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *Proceedings of the Conference on Innovative Data system Research (CIDR)*, pages 223–234, 2011. URL http://www.cidrdb.org/cidr2011/Papers/CIDR11_Paper32.pdf.

[14] R. Van Renesse and D. Altinbuken. Paxos made moderately complex. *ACM Comput. Surv.*, 47(3):42:1–42:36, Feb. 2015. ISSN 0360-0300. doi: 10.1145/2673577. URL http://doi.acm.org/10.1145/2673577.

[15] J.-P. Martin and L. Alvisi. Fast byzantine consensus. *IEEE Trans. Dependable Secur. Comput.*, 3 (3):202–215, July 2006. ISSN 1545-5971. doi: 10.1109/TDSC.2006.35. URL http://dx.doi.org/10.1109/TDSC.2006.35.

[16] L. Lamport. Byzantizing paxos by refinement. In *Proceedings of the 25th International Conference on Distributed Computing*, DISC'11, pages 211–224, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-24099-7. URL http://dl.acm.org/citation.cfm?id=2075029.2075058.

[17] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008.

[18] L. Lamport. Fast paxos. *Distributed Computing*, 19(2):79–103, Oct 2006. ISSN 1432-0452. doi: 10.1007/s00446-006-0005-x. URL https://doi.org/10.1007/s00446-006-0005-x.

[19] L. Lamport. Generalized consensus and paxos. Technical report, Technical Report MSR-TR-2005-33, Microsoft Research, 2005.

[20] R. Ladin, B. Liskov, and L. Shrira. Lazy replication: Exploiting the semantics of distributed services. In *Proceedings of the Ninth Annual ACM Symposium on Principles of Distributed Computing*, PODC '90, pages 43–57, New York, NY, USA, 1990. ACM. ISBN 0-89791-404-X. doi: 10.1145/93385.93399. URL http://doi.acm.org/10.1145/93385.93399.

[21] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, Oct. 2007. ISSN 0163-5980. doi: 10.1145/1323293.1294281. URL http://doi.acm.org/10.1145/1323293.1294281.

[22] D. Porto, J. Leitão, C. Li, A. Clement, A. Kate, F. Junqueira, and R. Rodrigues. Visigoth Fault Tolerance. *Proceedings of the Tenth European Conference on Computer Systems*, pages 8:1—-8:14, 2015. doi: 10.1145/2741948.2741979. URL http://doi.acm.org/10.1145/2741948.2741979.

[23] T. Kraska, G. Pang, M. J. Franklin, S. Madden, and A. Fekete. Mdcc: Multi-data center consistency. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 113–126, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1994-2. doi: 10.1145/2465351.2465363. URL http://doi.acm.org/10.1145/2465351.2465363.

[24] Y. Mao, F. P. Junqueira, and K. Marzullo. Mencius: Building efficient replicated state machines for wans. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 369–384, Berkeley, CA, USA, 2008. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1855741.1855767.

[25] I. Moraru, D. G. Andersen, and M. Kaminsky. There is more consensus in egalitarian parliaments. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 358–372, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2388-8. doi: 10.1145/2517349.2517350. URL http://doi.acm.org/10.1145/2517349.2517350.

[26] D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 305–319, Philadelphia, PA, 2014. USENIX Association. ISBN 978-1-931971-10-2. URL https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro.

[27] B. M. Oki and B. H. Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, PODC '88, pages 8–17, New York, NY, USA, 1988. ACM. ISBN 0-89791-277-2. doi: 10.1145/62546.62549. URL http://doi.acm.org/10.1145/62546.62549.

[28] L. Lamport, R. Shostak, and M. Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, July 1982. ISSN 0164-0925. doi: 10.1145/357172.357176. URL http://doi.acm.org/10.1145/357172.357176.

[29] M. Castro and B. Liskov. Practical byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, OSDI '99, pages 173–186, Berkeley, CA, USA, 1999. USENIX Association. ISBN 1-880446-39-1. URL http://dl.acm.org/citation.cfm?id=296806.296824.

[30] G. S. Veronese, M. Correia, A. N. Bessani, L. C. Lung, and P. Verissimo. Efficient byzantine fault-tolerance. *IEEE Trans. Comput.*, 62(1):16–30, Jan. 2013. ISSN 0018-9340. doi: 10.1109/TC.2011.221. URL http://dx.doi.org/10.1109/TC.2011.221.

[31] R. Kapitza, J. Behl, C. Cachin, T. Distler, S. Kuhnle, S. V. Mohammadi, W. Schröder-Preikschat, and K. Stengel. Cheapbft: Resource-efficient byzantine fault tolerance. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 295–308, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1223-3. doi: 10.1145/2168836.2168866. URL http://doi.acm.org/10.1145/2168836.2168866.

[32] P.-L. Aublin, R. Guerraoui, N. Knežević, V. Quéma, and M. Vukolić. The next 700 bft protocols. *ACM Trans. Comput. Syst.*, 32(4):12:1–12:45, Jan. 2015. ISSN 0734-2071. doi: 10.1145/2658994. URL http://doi.acm.org/10.1145/2658994.

[33] S. Liu, P. Viotti, C. Cachin, V. Quéma, and M. Vukolic. Xft: Practical fault tolerance beyond crashes. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 485–500, Berkeley, CA, USA, 2016. USENIX Association. ISBN 978-1-931971-33-1. URL http://dl.acm.org/citation.cfm?id=3026877.3026915.

[34] Amazon s3 availability event: July 20, 2008. http://status.aws.amazon.com/s3-20080720.html, 2008. Accessed: 2016-12-22.

[35] S3 data corruption?: June 22, 2008. https://forums.aws.amazon.com/thread.jspa?threadID=22709, 2008. Accessed: 2016-12-22.

[36] A. Bessani, J. Sousa, and E. E. P. Alchieri. State machine replication for the masses with bft-smart. In *Proceedings of the 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '14, pages 355–362, Washington, DC, USA, 2014. IEEE Computer Society. ISBN 978-1-4799-2233-8. doi: 10.1109/DSN.2014.43. URL http://dx.doi.org/10.1109/DSN.2014.43.

[37] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat. Providing high availability using lazy replication. *ACM Trans. Comput. Syst.*, 10(4):360–391, Nov. 1992. ISSN 0734-2071. doi: 10.1145/138873.138877. URL http://doi.acm.org/10.1145/138873.138877.

[38] E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Commun. ACM*, 17(11):643–644, Nov. 1974. ISSN 0001-0782. doi: 10.1145/361179.361202. URL http://doi.acm.org/10.1145/361179.361202.

[39] M. Correia, D. G. Ferro, F. P. Junqueira, and M. Serafini. Practical hardening of crash-tolerant systems. *Proceedings of the 2012 USENIX conference on Annual Technical Conference*, page 41, 2012. URL http://dl.acm.org/citation.cfm?id=2342821.2342862.

[40] W. Gu, Z. Kalbarczyk, R. K. Iyer, and Z.-Y. Yang. Characterization of linux kernel behavior under errors. In *DSN*, pages 459–468. IEEE Computer Society, 2003. ISBN 0-7695-1952-0. URL http://dblp.uni-trier.de/db/conf/dsn/dsn2003.html#GuKIY03.

[41] A. Singh, P. Fonseca, P. Kuznetsov, R. Rodrigues, and P. Maniatis. Zeno: Eventually consistent byzantine-fault tolerance. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'09, pages 169–184, Berkeley, CA, USA, 2009. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1558977.1558989.

[42] J. Sousa and A. Bessani. Separating the wheat from the chaff: An empirical design for geo-replicated state machines. In *2015 IEEE 34th Symposium on Reliable Distributed Systems (SRDS)*, pages 146–155, Sept 2015. doi: 10.1109/SRDS.2015.40.

[43] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, Mar. 1996. ISSN 0004-5411. doi: 10.1145/226643.226647. URL http://doi.acm.org/10.1145/226643.226647.

[44] T. D. Chandra, V. Hadzilacos, and S. Toueg. The Weakest Failure Detector for Solving Consensus. *Journal of the ACM*, 43(4):685—-722, 1996. ISSN 00045411. doi: 10.1145/234533.234549. URL http://doi.acm.org/10.1145/234533.234549.

[45] R. Guerraoui and A. Schiper. "gamma-accurate" failure detectors. In *Proceedings of the 10th International Workshop on Distributed Algorithms*, WDAG '96, pages 269–286, London, UK, UK, 1996. Springer-Verlag. ISBN 3-540-61769-8. URL http://dl.acm.org/citation.cfm?id=645953.675643.

[46] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the Presence of Partial Synchrony. *Journal of the ACM*, 35(2):288—-323, 1988. ISSN 00045411. doi: 10.1145/42282.42283. URL http://doi.acm.org/10.1145/42282.42283.

[47] D. Dolev, C. Dwork, and L. Stockmeyer. On the minimal synchronism needed for distributed consensus. *Journal of the ACM*, 34(1):77–97, Jan. 1987. ISSN 0004-5411. doi: 10.1145/7531.7533. URL http://doi.acm.org/10.1145/7531.7533.

[48] M. K. Aguilera, C. Delporte-Gallet, H. Fauconnier, and S. Toueg. Partial synchrony based on set timeliness. In *Proceedings of the 28th ACM Symposium on Principles of Distributed Computing*, PODC '09, pages 102–110, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-396-9. doi: 10.1145/1582716.1582737. URL http://doi.acm.org/10.1145/1582716.1582737.

[49] S. Chaudhuri. More choices allow more faults: Set consensus problems in totally asynchronous systems. *Inf. Comput.*, 105(1):132–158, July 1993. ISSN 0890-5401. doi: 10.1006/inco.1993.1043. URL http://dx.doi.org/10.1006/inco.1993.1043.

[50] P. Zielinski. Anti-Ω: The weakest failure detector for set agreement. In *Proceedings of the Twenty-seventh ACM Symposium on Principles of Distributed Computing*, PODC '08, pages 55–64, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-989-0. doi: 10.1145/1400751.1400761. URL http://doi.acm.org/10.1145/1400751.1400761.

[51] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, June 2002. ISSN 0163-5700. doi: 10.1145/564585.564601. URL http://doi.acm.org/10.1145/564585.564601.

[52] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990. ISSN 0164-0925. doi: 10.1145/78969.78972. URL http://doi.acm.org/10.1145/78969.78972.

[53] M. Ahamad, G. Neiger, J. E. Burns, P. Kohli, and P. W. Hutto. Causal memory: definitions, implementation, and programming. *Distributed Computing*, 9(1):37–49, Mar 1995. ISSN 1432-0452. doi: 10.1007/BF01784241. URL https://doi.org/10.1007/BF01784241.

[54] Y. Saito and M. Shapiro. Optimistic replication. *ACM Computing Surveys*, 37(1):42–81, 2005. ISSN 03600300. doi: 10.1145/1057977.1057980. URL http://doi.acm.org/10.1145/1057977.1057980.

[55] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow.*, 1(2):1277–1288, Aug. 2008. ISSN 2150-8097. doi: 10.14778/1454159.1454167. URL http://dx.doi.org/10.14778/1454159.1454167.

[56] C. Li, D. Porto, A. Clement, J. Gehrke, N. Preguiça, and R. Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 265–278, Berkeley, CA, USA, 2012. USENIX Association. ISBN 978-1-931971-96-6. URL http://dl.acm.org/citation.cfm?id=2387880.2387906.

[57] V. Balegas, S. Duarte, C. Ferreira, R. Rodrigues, N. Preguiça, M. Najafzadeh, and M. Shapiro. Putting consistency back into eventual consistency. *Proceedings of the Tenth European Conference on Computer Systems - EuroSys '15*, pages 1–16, 2015. doi: 10.1145/2741948.2741972. URL http://doi.acm.org/10.1145/2741948.2741972.

[58] F. P. Junqueira, B. C. Reed, and M. Serafini. Zab: High-performance broadcast for primary-backup systems. In *Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems&Networks*, DSN '11, pages 245–256, Washington, DC, USA, 2011. IEEE Computer Society. ISBN 978-1-4244-9232-9. doi: 10.1109/DSN.2011.5958223. URL http://dx.doi.org/10.1109/DSN.2011.5958223.

[59] C. Cachin, R. Guerraoui, and L. Rodrigues. *Introduction to Reliable and Secure Distributed Programming*. Springer Publishing Company, Incorporated, 2nd edition, 2011. ISBN 3642152597.

[60] R. Kotla, A. Clement, E. Wong, L. Alvisi, and M. Dahlin. Zyzzyva: Speculative byzantine fault tolerance. *Commun. ACM*, 51(11):86–95, Nov. 2008. ISSN 0001-0782. doi: 10.1145/1400214.1400236. URL http://doi.acm.org/10.1145/1400214.1400236.