

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 Miragaia Rodrigues
Dr. Full Name 2

Examination Committee

Chairperson: Prof. Full Name
Supervisor: Prof. Full Name 1 (or 2)
Member of the Committee: Prof. Full Name 3

Month Year

"Don't sweat the petty things and don't pet the sweaty things"

by George Carlin

Acknowledgments

A few words about the university, financial support, research advisor, dissertation readers, faculty or other professors, lab mates, other friends and family...

Resumo

Inserir o resumo em Português aqui com o máximo de 250 palavras e acompanhado de 4 a 6 palavras-chave...

Palavras-chave: palavra-chave1, palavra-chave2,...

Abstract

Insert your abstract here with a maximum of 250 words, followed by 4 to 6 keywords...

Keywords: keyword1, keyword2,...

Contents

Acknowledgments	v
Resumo	vii
Abstract	ix
List of Tables	xiii
List of Figures	xv
Nomenclature	1
Glossary	1
1 Introduction	1
1.1 Motivation	1
1.2 Contributions	3
1.3 Document Outline	4
2 Related Work	5
2.1 Paxos and its Variants	5
2.2 Non-crash Fault Models	8
2.3 Partial Synchrony	12
2.4 Variable Consistency Models	14
2.5 Coordination Systems	16
3 Architecture	19
3.1 Crash Fault Model	19
3.2 Byzantine Fault Model	22
3.2.1 Overview	22
3.2.2 View change	22
3.2.3 Agreement protocol	23
3.2.4 Classic ballots	24
3.2.5 Fast ballots	24
3.2.6 Checkpointing	26
3.2.7 Byzantine Generalized Paxos Pseudocode	26
3.3 Visigoth Fault Model	32
3.4 Discussion	38

3.5 Conclusion	41
Bibliography	43
A Byzantine Generalized Paxos Correctness Proofs	47
B Visigoth Generalized Paxos Correctness Proofs	51

List of Tables

3.1	Notation for the pseudocode	19
A.1	Proof notation	47
B.1	Proof notation	51

List of Figures

Chapter 1

Introduction

1.1 Motivation

The consensus problem requires that processes agree on proposed values in the presence of f faults and its solution has been essential to solve many distributed computing problems. This problem can be specified through two safety conditions and a liveness one, respectively [1]:

Agreement No two processes learn different values

Validity If a process learned a value v , then some process proposed v

Termination Every process learns a value in a finite number of steps

One of the most important contributions to this field is the Fischer, Lynch and Patterson (FLP) impossibility result that states that the *wait-free* consensus problem is unsolvable in an asynchronous system even if only one process can fail [2]. 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 [3]. 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 (i.e., committed). This problem has gone from a theoretical exercise to a fundamental part of many large-scale, modern distributed systems such as Google's Chubby [4], Spanner [5] and Apache's ZooKeeper [6]. These systems power services like the Google Search Engine, using techniques like state machine replication (SMR) 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.

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* [7]. These structures allow for the formulation of different consensus problems, including consensus problems where commutative operations are allowed to be ordered differently at different replicas (i.e., command histories). The advantage of such a problem can be seen in Fast Paxos, where fast ballots are executed by having proposers propose directly to acceptors [8]. 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 reordering 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.

The approach of allowing the system to reorder operations may feel familiar to the reader, since it resembles how weak consistency models relax consistency guarantees to allow replicas to reorder operations [9]. By relaxing consistency and allowing operations to be reordered, these models reduce coordination requirements which results in decreased latency and better operation concurrency. However, relaxing consistency guarantees also introduces the chance of state divergence which can be tolerable or not depending on the application. These approaches are critical to geo-replicated scenarios where it's important to reduce round trips between data centers and maintaining strong consistency incurs in an unacceptable latency cost.

The Paxos protocol family has been a subject of study for some time, due to its importance in the area of consensus and state machine replication. Recent work in this area includes Mencius [10] and EPaxos [11], which try to increase throughput by eliminating the bottleneck caused by having a single leader that sends proposals to acceptors. The Generalized Paxos protocol is similar to these approaches in the sense that it also decreases the load put on the leader by allowing proposers to bypass it. However, 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. A symptom of this 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. The goal of Generalized Paxos is to reduce communication steps without requiring a leader to solve conflicts that emerge from concurrently executed operations but few works have tried to align this algorithm with a multi-datacenter scenario. 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 general and complex description of generalized consensus. Therefore, as our first contribution, we simplify the problem specification from the full generalized consensus to a simpler specification that preserves the motivating scenario. This consensus problem is known as the command history problem and consists in agreeing on sequences of commands where sequences are considered equivalent if all non-commutative operations are totally ordered.

In order to perform this study, Generalized Paxos was adapted to several models while still preserving the advantageous properties which motivated its creation. The first version of the protocol starts by adapting the original protocol to the more specific consensus problem of command histories in the crash fault model. The second version of the protocol, Byzantine Generalized Paxos (BGP), solves the same problem in the Byzantine fault model. Lastly, the third version of the protocol, adapts Generalized Paxos to the Visigoth fault model [12], where the 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, allowing the system to support any combination of faults within the spectrum between crash and Byzantine faults. With this model we can study command history

consensus from different perspectives and propose a solution that allows it to be solved across a broad spectrum of system models. This is an important aspect because, although it adds complexity, it adds the ability to develop a generic protocol that can be used in different environments. As such, this work removes some generality of the problem specification while retaining the original motivating scenario but, in turn, generalizes the model, allowing the system administrator to tune the protocol. One particularly interesting application is similar to that of weakly consistent systems and geo-replicated datacenters. By specifying a consensus problem akin to command histories, our protocol could take advantage of operation commutativity to minimize the number of ballots that incur in a higher coordination cost. In this scenario, the parameterizability of Visigoth model also plays an important role since we can specify a number of slow processes that allows us to reduce the number of replicas that are required to safely commit values. Another potentially interesting scenario, related to the fault model, could be one where we tolerate arbitrary, uncorrelated faults. This is an interesting scenario since it's enough to deal with arbitrary state corruption faults, which are common within datacenters [13], but avoids the cost of dealing with coordinated Byzantine behavior.

In addition to these contributions, a checkpointing sub-protocol is also proposed to manage the accumulation of data at the acceptor processes and an optimization is proposed that reduces quorums requirements for certain sequences of commands at no additional cost. **An alternative approach is to give replicas weights according to their connectivity and performance. Since replicas with better hardware and network connectivity have better performance and reliability, they can be given a higher weight to allow for quorums to be formed with a lower number of higher quality replicas. This approach is used in WHEAT (WeigHt-Enabled Active replicaTion) to reduce latency [14].**

lysson men-
EAT, should I
?

1.3 Document Outline

The remainder of this document is structured as follows: Section 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. Section 3.1 describes what we have accomplished so far regarding the solution to generalized consensus in the Visigoth model. Section 3.2 proposes how these preliminary results should be extended to arrive at the final solution. Section 3.3 describes how the developed solution will be validated. Section 3.4 proposes a schedule for the development, implementation, validation and documentation of future work.

Chapter 2

Related Work

Due to the specific characteristics of both the problem that we wish to solve and the system model in which it will be solved, there are several areas that contain works of interest to us. Subsection 2.1 describes previous work regarding consensus and several Paxos variants. Subsection 2.2 discusses non-crash fault models as well as protocols that solve consensus in these environments. Subsection 2.3 describes different partial synchrony models and how they fit in the synchrony spectrum. Subsection 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 high-availability, large scale systems [4, 15].** Subsection 2.5 studies a few systems where these algorithms are used to enable fault-tolerance services.

2.1 Paxos and its Variants

The Paxos protocol family solves consensus by circumventing the well-known FLP impossibility result [2]. 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 of 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 [3]. 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 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 in a synchronous environment. 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.

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 [16]. 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 observes that it's possible to improve on previous results by allowing proposers to propose values directly to acceptors. To this end, the protocol distinguishes between fast and classic ballots, where fast ballots bypass the leader by sending proposals directly to acceptors and classic ballots work as in Basic Paxos. The 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 logical 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 messages broadcasts are necessary: *phase 2a* messages between a proposer and the acceptors, and *phase 2b* messages between acceptors and learners. This creates the possibility of two proposers concurrently proposing values to the acceptors and generating a conflict. 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 cost is a *phase 2* of round $i + 1$ (i.e., two additional message delays). If uncoordinated recovery is chosen then the cost is a single *phase 2b* message in round $i + 1$ (i.e., one message delay) [8]. These numbers assume that either the leader (coordinated recovery) or the acceptors (uncoordinated recovery) are learners, which is a reasonable assumption.

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 [7]. 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 increasing sequence abstraction 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 clients propose A and C , 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 clients 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.

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 [17]. 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 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 $k = n \bmod i$. 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 [10]. 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 $v \neq \text{no-op}$ 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.

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 [11]. 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 [7]. 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 is significantly lower.

2.2 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 [18]. 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 is a protocol that solves consensus for state machine replication while tolerating up to f Byzantine faults [19]. 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 message's digest, the view and the actual message. 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 Fischer, Lynch and Paterson's impossibility result [2]. A Byzantine leader may try to prevent progress by omitting pre-prepare messages when it receives operation requests from clients, but backups can trigger new views after waiting for a certain period of time.

The 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 [20]. 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 $f_{nc}(s)$ and the sum of crash faults $f_c(s)$, non-crash faults $f_{nc}(s)$ and partitioned replicas $f_p(s)$ is larger than the fault threshold. That is, the system is in anarchy 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 [9]. 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.

The creators of the Visigoth Fault Tolerance (VFT) model also make the observation that the Byzantine model is unrealistic, especially for environments such as data centers. 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 [21][13]. 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 [12]. 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 $n \geq u + s + o + 1$ (i.e., $f + s + o + 1$), 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 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 state machine replication. 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, then $x - s$ 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.

Dijkstra proposed self-stabilizing systems as a solution to the problem of keeping cyclic sequences of processes in a legitimate state even after the occurrence of transient faults [22]. In a self-stabilizing system, each process may contain several privileges and may only exchange information with 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.

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 [23]. 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 [24], 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 [22] 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 is a Byzantine fault tolerance state machine replication protocol that favors availability instead of consistency [25]. By trading strong consistency for eventual consistency, Zeno provides higher availability when networks 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 in at least one correct replica with any other quorum. 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.

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

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 [3]. 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 [12].

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 [26]. 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:

Completeness Eventually every crashed process is permanently suspected by some correct process.

Accuracy Eventually 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 $f < \lceil n/2 \rceil$

[26]. Chandra et al. also define a new class of failure detectors, Ω -accurate failure detectors [27]. 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.

The accuracy properties of the failure detectors proposed by Chandra and Toueg in [26] 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 [28] to deal with this limitation in the specification of Ω -accurate failure detectors. 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:

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 [28], $\diamond\mathcal{W}(\Gamma) \prec \diamond\mathcal{W}$, and $\diamond\mathcal{W}$ failure detectors have been shown to be the weakest class of failure detectors for which consensus is solvable [27], then it follows that $\diamond\mathcal{W}(\Gamma)$ failure detectors do not allow for consensus to be solved.

Wide-area systems are often considered to be asynchronous since they cannot 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* [29]. 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 [2] and [30] 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 (GST) 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. 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.

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 [31]. This model generalizes the concept of *process timeliness* (i.e., an upper bound Φ on the relative speeds of processes) [29] 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 another 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 [32]. 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 [33]. 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 also 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 $winner_{set}_p$ and sets $\Pi_n - winner_{set}_p$ 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

Replicating state across geographically separate datacenters requires us to reason about consistency. Traditionally, consistency models could be grouped into two categories: weak and strong. Strong consistency models, such as serializability [34], 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 [35], 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 [36]. 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 [37]. 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 [7].

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 [38]. 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.

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* [39]. 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 $P_1 \vee P_2 \vee \dots \vee P_n$, 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.

2.5 Coordination Systems

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

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 [15]. ZooKeeper uses a primary-backup scheme and an atomic broadcast algorithm called Zab that is tuned specifically for ZooKeeper's setting [6]. This protocol functions similarly to Paxos but guarantees that multiple outstanding operations are committed in 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 that could possibly violate 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 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 [4]. 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 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.

Chapter 3

Architecture

One of the main contributions of this work is an adaptation of the Generalized Paxos protocol for the Visigoth model that allows us to solve the command history problem with different fault assumptions. In the interest of clarity, the protocol will be presented in the same way it was built, that is, incrementally. First, we will start by presenting a slightly modified version of Byzantine Paxos for the crash fault model. Even though the original protocol was developed for this fault model, it solves the generalized consensus problem. However, since the generality of this problem greatly complicates the protocol itself, we specialize the problem to its original motivating form, the command history problem. Therefore, subsection 3.1 presents a protocol that solves this simplified problem in the crash fault model. In subsection 3.2, we present the Byzantine Generalized Paxos (BGP) protocol. This protocol solves the command history consensus problem for in an environment where replicas can behave arbitrarily. In addition to a description of the protocol, this section also contains both pseudocode algorithms, detailing the protocol's execution, and correctness proofs for each property. In subsection ??, the Generalized Paxos protocol is broadened from the Byzantine to the Visigoth fault model. As with the Byzantine version, the description of the protocol is accompanied by pseudocode algorithms and correctness proofs. Subsection ?? presents optimizations that can be applied to any of the versions in order to gain performance at no additional cost.

3.1 Crash Fault Model

Notation	
Symbol	Definition
\sqcup	least upper bound
\sqcap	greatest lower bound
\bullet	append operator

Table 3.1: Notation for the pseudocode

Algorithm 1 Generalized Paxos - Proposer p

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

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

Algorithm 2 Visigoth Generalized Paxos - Process p

```
1: function MERGE_SEQUENCES(old_seq, new_seq)
2:   for c in new_seq do
3:     if !CONTAINS(old_seq, c) then
4:       old_seq = old_seq • c;
5:   end for
6:   return old_seq;
7: end function
```

Algorithm 3 Visigoth Generalized Paxos - Leader l

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

```
1: upon trigger_next_ballot(type) do
2:    $ballot_l += 1$ ;
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:    $proposals = proposals \bullet prop$ ;
12:
13: upon receive(P1B,  $bal_a$ ,  $vals_a$ ) from acceptor a do
14:   if  $bal_a = ballot_l$  then
15:      $accepted[ballot_l][a] = vals_a$ ;
16:     if  $\#(accepted[ballot_l]) \geq N - f$  then
17:       PHASE_2A();
18:
19: function PHASE_2A()
20:    $maxTried_l = PROVED\_SAFE(ballot_l)$ ;
21:    $maxTried_l = maxTried_l \bullet proposals$ ;
22:   SEND(P2A_CLASSIC, view,  $ballot_l$ ,  $maxTried_l$ ) to acceptors;
23:    $proposals = \perp$ ;
24: end function
25:
26: function PROVED_SAFE(ballot)
27:    $safe\_seq = \perp$ ;
28:   for seq in  $accepted[ballot]$  do
29:      $safe\_seq = MERGE\_SEQUENCES(safe\_seq, seq)$ ;
30:   end for
31:   return  $safe\_seq$ ;
32: end function
```

Algorithm 4 Visigoth 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, proof) 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,  $val_a$ ) to leader;
16:      $bal_a = ballot$ ;
17:      $val_a[bal_a] = \perp$ ;
18:   end if
19: end function
20:
21: function PHASE_2B_CLASSIC(ballot, value)
22:   if  $ballot \geq bal_a$  and  $val_a = \perp$  then
23:      $bal_a = ballot$ ;
24:      $val_a[ballot] = value$ ;
25:     SEND(P2B, ballot, value) to learners;
26:   end if
27: end function
28:
29: function PHASE_2B_FAST(ballot, value)
30:   if  $ballot = bal_a$  and  $fast\_bal[bal_a]$  then
31:      $val_a[bal_a] = \text{MERGE\_SEQUENCES}(val_a[bal_a], value)$ ;
32:     SEND(P2B,  $bal_a$ ,  $val_a[bal_a]$ ) to learners;
33:   end if
34: end function
```

Algorithm 5 Generalized Paxos - Learner l

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

```
1: upon receive(p2b, bal, val) from acceptor  $a_i$  do
2:    $messages[bal][val][a_i] = true$ ;
3:   if  $\#(messages[bal][val]) \geq N - f$  then
4:      $learned = \text{MERGE\_SEQUENCES}(learned, val)$ ;
```

3.2 Byzantine Fault Model

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

3.2.1 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 [?] and Generalized Paxos [7], 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.

3.2.2 View change

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

In this subprotocol, the system moves through sequentially numbered views, and the leader for each view is chosen in a rotating fashion using the simple equation $leader(view) = view \bmod N$. The basic idea of this subprotocol is that acceptor processes monitor whether progress is being made on adding commands to the current sequence, and, if not, they send a message to other acceptors suspecting the current leader. If enough suspicions are collected, processes can move to the subsequent view.

In more detail, whenever an acceptor process individually perceives that liveness is not being upheld (i.e., there are pending commands that have not gathered enough support to conclude their execution, as detailed next), it starts suspecting the leader and multicasts a signed SUSPICION message for the current view to all acceptors.

To ensure that f Byzantine processes cannot trigger view changes by producing false suspicions, acceptor processes only send a view change message indicating their commitment to starting a new

view after hearing that $f + 1$ processes suspect the leader to be faulty. In particular, at this point, acceptor processes multicast a view-change message containing the new view number, the $f + 1$ signed suspicions, and signed proof that the acceptor sent the view-change message for the new view number. 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 broadcasts the view-change message, then all correct processes, upon receiving this message, will be able to validate the proof of $f + 1$ suspicions and also multicast the view-change message.

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

3.2.3 Agreement protocol

The consensus protocol allows learner processes to agree on equivalent sequences of commands (according to our previous definition of equivalence). An important conceptual distinction between the original Paxos protocol and BGP is that, in the original Paxos, each instance of consensus is called a ballot, whereas in BGP, instead of being a separate instance of consensus, ballots correspond to an extension to the sequence of learned commands of a single ongoing consensus instance. In both protocols, ballots can either be *classic* or *fast*.

In classic ballots, a leader proposes a single sequence of commands, such that it can be appended to the commands learned by the learners. A classic ballot in Generalized Paxos follows a protocol that is very similar to the one used by classic Paxos [?]. This protocol comprises 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). This is followed by a second phase where the leader picks an extension to the sequence of previously proposed commands 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.

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 the fast ballots 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.

3.2.4 Classic ballots

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

In this part of the protocol, the leader continuously collect proposals by assembling commands that 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. In classic Paxos, acceptors also promise not to participate in lower-numbered ballots, in order to prevent safety violations [?]. However, in BGP this promise is already implicit, given (1) there is only one leader per view and it's the only process allowed to propose in a classic ballot and (2) acceptors replying to that message must be in the same view as that leader.

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

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

3.2.5 Fast ballots

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

The basic idea of fast ballots is that proposers contact the acceptors directly, bypassing the leader, and then the acceptors send directly to the learners their vote for the current sequence, where this sequence now incorporates the proposed value. Learners then analyze whether conflicts might exist, in the sense of commands being voted for in a different order at different acceptors. If so, then it is necessary to fall back to a classic ballot. This is where generalized consensus allows for avoiding falling

back to this slow path, namely in the case that the commands that are sequenced in a different order at different acceptors commute.

Next, we explain each of the steps behind fast ballots 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, the fast ballot equivalent of the phase $2b$ message, which is sent from acceptors to learners, contains the current ballot number and the command sequence. However, since commands (or sequences of commands) are concurrently proposed, acceptors can receive and vote for non-commutative proposals in different orders. To ensure safety, correct learners must learn non-commutative commands in a total order. To this end, a learner must gather $N - f$ votes for equivalent sequences. That is, sequences do not necessarily have to be equal in order to be learned since commutative commands may be reordered. Recall that a sequence is equivalent to another if it can be transformed into the second one by reordering its elements without changing the order of any pair of non-commutative commands. (Note that, in the pseudocode, equivalent sequences are being treated as belonging to the same index of the *messages* variable, to simplify the presentation.) By requiring $N - f$ votes for a sequence of commands, we ensure that, given two sequences where non-commutative commands are differently ordered, only one sequence will receive enough votes even if f Byzantine acceptors vote for both sequences. Outside the set of (up to) f Byzantine acceptors, the remaining $2f + 1$ correct acceptors will only vote for a single sequence, which means there are only enough correct processes to commit one of them. Note that the fact that proposals are sent as extensions 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, 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 assemble 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.

3.2.6 Checkpointing

The pseudocode includes a checkpointing feature that allows the leader to propose a special command C^* that causes both the acceptors and learners to discard previously stored commands. This feature can be used to prevent commands from being stored indefinitely. However, since commands are kept at the acceptors to ensure that they will eventually be committed, special care has to be taken before discarding them. To prevent unlearned commands from being discarded, the checkpointing command must be sent within a sequence in a classic ballot. In phase 1 of that classic ballot, acceptors send every command they have to the leader, who waits for $N - f$ phase 1b messages. The leader can be sure that the commands were originated from proposers by verifying the signatures they contain. Since, when proposing to acceptors in fast ballots, proposers wait for acknowledgments from $N - f$ proposers, there's at least one correct acceptor in the intersection of a quorum that received a proposal and a quorum that sends commands to the leader. This means that any proposed value will be sent by some acceptor to the leader and included in the leader's sequence, along with the checkpointing command. Since acceptors must be certain that it's safe to discard previously stored commands, before sending phase 2b messages to learners, they first broadcast these messages among themselves. This round between acceptors is necessary because a Byzantine leader could send a checkpointing command to some acceptors but not others. After waiting for $N - f$ such messages, acceptors send phase 2b messages to the learners along with the cryptographic proofs exchanged in the acceptor-to-acceptor broadcast. After receiving just one message, the leader may simply validate the $N - f$ acceptor proofs contained in it and learn the commands. The learners discard previously stored state when they execute the checkpointing command.

3.2.7 Byzantine Generalized Paxos Pseudocode

This section presents the pseudocode for the Byzantine Generalized Paxos protocol.

Algorithm 6 Byzantine Generalized Paxos - Proposer p

Local variables: $ballot_type = \perp$

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

Algorithm 7 Byzantine Generalized Paxos - Process p

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

Algorithm 8 Byzantine Generalized Paxos - Leader I

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

```
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:        $valid\_proofs += 1$ ;
7:
8:   if  $valid\_proofs > f$  then
9:      $view = view_a$ ;
10:
11: upon trigger_next_ballot( $type$ ) do
12:    $ballot_l += 1$ ;
13:   SEND(BALLOT,  $type$ ) to proposers;
14:
15:   if  $type = fast$  then
16:     SEND(FAST,  $ballot_l, view$ ) to acceptors;
17:   else
18:     SEND(P1A,  $ballot_l, view$ ) to acceptors;
19:
20: upon receive(PROPOSE,  $prop$ ) from proposer  $p_i$  do
21:    $proposals = proposals \bullet prop$ ;
22:
23: upon receive(P1B,  $bal_a, view\_vals_a$ ) from acceptor  $a$  do
24:   if  $bal_a = ballot_l$  then
25:      $accepted[ballot_l][a] = SIGNED\_COMMANDS(view\_vals_a)$ ;
26:
27:   if  $\#(accepted[ballot_l]) \geq N - f$  then
28:     PHASE_2A();
29:
30: function PHASE_2A()
31:    $maxTried_l = PROVED\_SAFE(ballot_l)$ ;
32:    $maxTried_l = maxTried_l \bullet proposals$ ;
33:   if CLEAN_STATE?() then
34:      $maxTried_l = maxTried_l \bullet C^*$ ;
35:   SEND(P2A\_CLASSIC,  $ballot_l, view, maxTried_l$ ) to acceptors;
36:    $proposals = \perp$ ;
37: end function
38:
39: function PROVED_SAFE( $ballot$ )
40:    $safe\_seq = \perp$ ;
41:   for  $seq$  in  $accepted[ballot]$  do
42:      $safe\_seq = MERGE\_SEQUENCES(safe\_seq, seq)$ ;
43:   end for
44:   return  $safe\_seq$ ;
45: end function
```

Algorithm 9 Byzantine Generalized Paxos - Acceptor a (view-change)

Local variables: $suspensions = \perp$, $new_view = \perp$, $leader = \perp$, $view = 0$, $bal_a = 0$, $val_a = \perp$, $fast_bal = \perp$, $checkpoint = \perp$

```
1: upon suspect_leader do
2:   if  $suspensions[p] \neq true$  then
3:      $suspensions[p] = true$ ;
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:
11:   if  $proof_{pub_i} = \langle suspicion, view \rangle$  then
12:      $suspensions[i] = proof$ ;
13:
14:   if  $\#(suspensions) > f$  and  $new\_view[view + 1][p] = \perp$  then
15:      $change\_proof = \langle view\_change, view + 1 \rangle_{priv_a}$ ;
16:      $new\_view[view + 1][p] = change\_proof$ ;
17:      $SEND(VIEW\_CHANGE, view + 1, suspensions, change\_proof)$ ;
18:
19: upon  $receive(VIEW\_CHANGE, new\_view_i, suspensions, change\_proof_i)$  from acceptor  $i$  do
20:   if  $new\_view_i \leq view$  then
21:     return;
22:
23:    $valid\_proofs = 0$ ;
24:   for  $p$  in acceptors do
25:      $proof = suspensions[p]$ ;
26:      $last\_view = new\_view_i - 1$ ;
27:     if  $proof_{pub_p} = \langle suspicion, last\_view \rangle$  then
28:        $valid\_proofs += 1$ ;
29:
30:   if  $valid\_proofs \leq f$  then
31:     return;
32:
33:    $new\_view[new\_view_i][i] = change\_proof_i$ ;
34:   if  $new\_view[view_i][a] = \perp$  then
35:      $change\_proof = \langle view\_change, new\_view_i \rangle_{priv_a}$ ;
36:      $new\_view[view_i][a] = change\_proof$ ;
37:      $SEND(VIEW\_CHANGE, view_i, suspensions, change\_proof)$ ;
38:
39:   if  $\#(new\_view[new\_view_i]) \geq N - f$  then
40:      $view = new\_view_i$ ;
41:      $leader = view \bmod N$ ;
42:      $suspensions = \perp$ ;
43:      $SEND(LEADER, view, new\_view[view_i])$  to leader;
```

Algorithm 10 Byzantine Generalized Paxos - Acceptor a (agreement)

Local variables: $suspensions = \perp$, $new_view = \perp$, $leader = \perp$, $view = 0$, $bal_a = 0$, $val_a = \perp$, $fast_bal = \perp$, $checkpoint = \perp$

```
1: upon receive( $P1A, ballot, view_l$ ) from leader  $l$  do
2:   if  $view_l = view$  then
3:      $PHASE\_1B(ballot)$ ;
4:
5: upon receive( $FAST, ballot, view_l$ ) from leader do
6:   if  $view_l = view$  then
7:      $fast\_bal[ballot] = true$ ;
8:
9: upon receive( $P2B, ballot, value, proof$ ) from acceptor  $i$  do
10:  if  $proof_{pub_i} \neq \langle ballot, value \rangle$  then
11:    return;
12:   $checkpoint[ballot][i] = proof$ ;
13:  if  $\#(checkpoint[ballot]) \geq N - f$  then
14:     $SEND(P2B, ballot, value, checkpoint[ballot])$  to learners;
15:     $val_a = \perp$ ;
16:
17: upon receive( $P2A\_CLASSIC, ballot, view, value$ ) from leader do
18:   if  $view_l = view$  then
19:      $PHASE\_2B\_CLASSIC(ballot, value)$ ;
20:
21: upon receive( $P2A\_FAST, value$ ) from proposer do
22:    $PHASE\_2B\_FAST(value)$ ;
23:
24: function  $PHASE\_1B(ballot)$ 
25:   if  $bal_a < ballot$  then
26:      $SEND(P1B, ballot, val_a)$  to leader;
27:      $bal_a = ballot$ ;
28:      $val_a[bal_a] = \perp$ ;
29:   end if
30: end function
31:
32: function  $PHASE\_2B\_CLASSIC(ballot, value)$ 
33:   if  $ballot \geq bal_a$  and  $val_a = \perp$  then
34:      $bal_a = ballot$ ;
35:      $val_a[ballot] = value$ ;
36:     if  $CONTAINS(value, C^*)$  then
37:        $proof = \langle suspicion, view \rangle_{priv_a}$ ;
38:        $SEND(P2B, ballot, value, proof)$  to acceptors;
39:     else
40:        $SEND(P2B, ballot, value)$  to learners;
41:     end if
42: end function
43:
44: function  $PHASE\_2B\_FAST(value)$ 
45:   if  $fast\_bal[bal_a]$  then
46:      $val_a[bal_a] = MERGE\_SEQUENCES(val_a[bal_a], value)$ ;
47:      $SEND(P2B, bal_a, val_a[bal_a])$  to learners;
48:   end if
49: end function
```

Algorithm 11 Byzantine Generalized Paxos - Learner I

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  $(\#(messages[ballot][value]) > f$  and
4:      $ISUNIVERSALLYCOMMUTATIVE(value))$  then
5:      $learned = MERGE\_SEQUENCES(learned, value)$ ;
6: upon receive( $P2B, ballot, value, proofs$ ) from acceptor  $a$  do
7:    $valid\_proofs = 0$ ;
8:   for  $i$  in  $acceptors$  do
9:      $proof = proofs[i]$ ;
10:    if  $proof_{pub_i} = \langle ballot, value \rangle$  then
11:       $valid\_proofs += 1$ ;
12:
13:   if  $valid\_proofs > f$  then
14:      $learned = MERGE\_SEQUENCES(learned, value)$ ;
```

3.3 Visigoth Fault Model

The Visigoth model differs from its Byzantine counterpart by allowing s processes to be slow but correct []. 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 a message. In VFT's Quorum Gathering Primitive (QGP), a process gathers a quorum by waiting for messages from $N - s$ distinct processes. After T time units have passed, of the x processes that are unresponsive, only s of them may be slow, which means that $x - s$ processes must be faulty. 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, and phase 2b, where acceptors send their votes to the learners.

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 $N - f$ votes require to be learned. However, since in the Visigoth model we want to take advantage of the additional assumptions to reduce the quorum to $N - u$, its possible for the leader to employ a split vote to send two different values to two sets of acceptors and ignore 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 be Byzantine and vote for both values and the leader can employ 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, v_2 to other two and ignores the last two, then the timeout is reached and the required quorum size is reduced from 5 to 4. Each value has two votes from both of the Byzantine acceptors and one vote from one of the two 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 sending phase 2b messages to learners, acceptors also resend the leader's phase 2a message to other acceptors. This prevents the leader from maliciously ignoring acceptors to force the quorum to decrease. To ensure that this replayed message originated from the leader and not from a Byzantine acceptor, it includes the leader's signature. This modification implies that, when gathering a quorum, the timeout must be increased from $2T$ to $3T$ since there are up to three message delays until a message is received by a learner.

Algorithm 12 Visigoth 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:     proof =  $\langle ballot, c \rangle_{priv_p}$ ;
8:     SEND(P2A_FAST, ballot, c, proof) to acceptors;
9:   else
10:    SEND(PROPOSE, c) to leader;
```

Algorithm 13 Visigoth Generalized Paxos - Process p

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

Algorithm 14 Visigoth Generalized Paxos - Leader I

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

```
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:        $valid\_proofs += 1$ ;
7:
8:   if  $valid\_proofs > f$  then
9:      $view = view_a$ ;
10:
11: upon trigger_next_ballot( $type$ ) do
12:    $ballot_l += 1$ ;
13:   SEND(BALLOT,  $ballot_l$ ,  $type$ ) to proposers;
14:
15:   if  $type = fast$  then
16:     SEND(FAST,  $ballot_l$ ,  $view$ ) to acceptors;
17:   else
18:     SEND(P1A,  $ballot_l$ ,  $view$ ) to acceptors;
19:
20: upon receive(PROPOSE,  $prop$ ) from proposer  $p_i$  do
21:    $proposals = proposals \bullet prop$ ;
22:
23: upon receive(P1B,  $bal_a$ ,  $view\_vals_a$ ) from acceptor  $a$  do
24:   if  $bal_a = ballot_l$  then
25:      $accepted[ballot_l][a] = SIGNED\_COMMANDS(view\_vals_a)$ ;
26:     if  $\#(accepted[ballot_l]) \geq N - u$  then
27:       PHASE_2A();
28:
29: function PHASE_2A()
30:    $maxTried_l = PROVED\_SAFE(ballot_l)$ ;
31:    $maxTried_l = maxTried_l \bullet proposals$ ;
32:    $proof = \langle view, ballot_l, maxTried_l \rangle_{priv\_leader}$ ;
33:   SEND(P2A\_CLASSIC,  $view$ ,  $ballot_l$ ,  $maxTried_l$ ,  $proof$ ) to acceptors;
34:    $proposals = \perp$ ;
35: end function
36:
37: function PROVED_SAFE( $ballot$ )
38:    $safe\_seq = \perp$ ;
39:   for  $seq$  in  $accepted[ballot]$  do
40:      $safe\_seq = MERGE\_SEQUENCES(safe\_seq, seq)$ ;
41:   end for
42:   return  $safe\_seq$ ;
43: end function
```

Algorithm 15 Visigoth Generalized Paxos - Acceptor a (view-change)

Local variables: $suspensions = \perp$, $new_view = \perp$, $leader = \perp$, $view = 0$, $bal_a = 0$, $val_a = \perp$, $fast_bal = \perp$, $checkpoint = \perp$

```
1: upon suspect_leader do
2:   if  $suspensions[p] \neq true$  then
3:      $suspensions[p] = true$ ;
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:
11:   if  $proof_{pub_i} = \langle suspicion, view \rangle$  then
12:      $suspensions[i] = proof$ ;
13:
14:   if  $\#(suspensions) > f$  and  $new\_view[view + 1][p] = \perp$  then
15:      $change\_proof = \langle view\_change, view + 1 \rangle_{priv_a}$ ;
16:      $new\_view[view + 1][p] = change\_proof$ ;
17:      $SEND(VIEW\_CHANGE, view + 1, suspensions, change\_proof)$ ;
18:
19: upon  $receive(VIEW\_CHANGE, new\_view_i, suspensions, change\_proof_i)$  from acceptor  $i$  do
20:   if  $new\_view_i \leq view$  then
21:     return;
22:
23:    $valid\_proofs = 0$ ;
24:   for  $p$  in acceptors do
25:      $proof = suspensions[p]$ ;
26:      $last\_view = new\_view_i - 1$ ;
27:     if  $proof_{pub_p} = \langle suspicion, last\_view \rangle$  then
28:        $valid\_proofs += 1$ ;
29:
30:   if  $valid\_proofs \leq f$  then
31:     return;
32:
33:    $new\_view[new\_view_i][i] = change\_proof_i$ ;
34:   if  $new\_view[view_i][a] = \perp$  then
35:      $change\_proof = \langle view\_change, new\_view_i \rangle_{priv_a}$ ;
36:      $new\_view[view_i][a] = change\_proof$ ;
37:      $SEND(VIEW\_CHANGE, view_i, suspensions, change\_proof)$ ;
38:
39:   if  $\#(new\_view[new\_view_i]) \geq N - f$  then
40:      $view = new\_view_i$ ;
41:      $leader = view \bmod N$ ;
42:      $suspensions = \perp$ ;
43:      $SEND(LEADER, view, new\_view[view_i])$  to leader;
```

Algorithm 16 Visigoth Generalized Paxos - Acceptor *a* (agreement)

Local variables: $suspicions = \perp$, $new_view = \perp$, $leader = \perp$, $view = 0$, $bal_a = 0$, $val_a = \perp$, $fast_bal = \perp$, $checkpoint = \perp$

```
1: upon receive(P1A, ballot, viewl) from leader l do
2:   if viewl = view then
3:     PHASE_1B(ballot);
4:
5: upon receive(FAST, ballot, viewl) from leader do
6:   if viewl = view then
7:     fast_bal[ballot] = true;
8:
9: upon receive(P2A_CLASSIC, view, ballot, value, proof) from leader do
10:  if viewl = view and  $\langle ballot, value \rangle = proof_{pub_{leader}}$  then
11:    PHASE_2B_CLASSIC(ballot, value, proof);
12:
13: upon receive(P2A_REPLAY, ballot, view, value, proof) from acc. do
14:  if viewl = view and ( $\langle view, ballot, value \rangle = proof_{pub_{leader}}$  or
     $\langle ballot, value \rangle = proof_{pub_{prop}}$ ) then
15:    PHASE_2B_CLASSIC(ballot, value, proof);
16:
17: upon receive(P2A_FAST, ballot, value, proof) from proposer p do
18:  if  $\langle ballot, value \rangle = proof_{pub_p}$ 
19:    PHASE_2B_FAST(ballot, value, proof);
20:
21: upon receive(VERIFICATION, ballot, value) from learner l do
22:  SEND(VERIFICATION, ballot, value);
23:
24: function PHASE_1B(ballot)
25:   if  $bal_a < ballot$  then
26:     SEND(P1B, ballot, vala) to leader;
27:     bala = ballot;
28:     vala[bala] =  $\perp$ ;
29:   end if
30: end function
31:
32: function PHASE_2B_CLASSIC(ballot, value, proof)
33:   if  $ballot \geq bal_a$  and  $val_a = \perp$  then
34:     bala = ballot;
35:     vala[ballot] = value;
36:     SEND(P2B, ballot, value) to learners;
37:     SEND(P2A_REPLAY, ballot, value, proof) to acceptors;
38:   end if
39: end function
40:
41: function PHASE_2B_FAST(ballot, value, proof)
42:   if  $ballot = bal_a$  and fast_bal[bala] then
43:     vala[bala] = MERGE_SEQUENCES(vala[bala], value);
44:     SEND(P2B, bala, vala[bala]) to learners;
45:     SEND(P2A_REPLAY, ballot, value, proof) to acceptors;
46:   end if
47: end function
```

Algorithm 17 Visigoth Generalized Paxos - Learner I

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

```
1: upon receive( $P2B, ballot, value$ ) from acceptor  $a$  do
2:    $messages[ballot][value][a] = true$ ;
3:   if  $quorumSize[ballot][value] = \perp$  then
4:      $quorumSize[ballot][value] = N - s$ ;
5:     STARTTIMER( $3T, TIMEREENDED, ballot, value$ );
6:
7:   if ( $\#(messages[ballot][value]) \geq quorumSize[ballot][value]$ 
and ( $quorumSize[ballot][value] = N - s$  or
 $verificationMessages[ballot][value] \geq N - u$ ) or
 $(\#(messages[ballot][value]) > u$  and ISUNIVERSALLYCOMMUTA-
TIVE( $value$ )) then
8:      $learned = \text{MERGE\_SEQUENCES}(learned, value)$ ;
9:
10: upon receive( $VERIFICATION, ballot, value$ ) from acceptor  $a$  do
11:    $verificationMessages[ballot][value][a] = true$ ;
12:
13: function TIMEREENDED( $ballot, value$ )
14:    $quorumSize[ballot][value] = N - u$ ;
15:   SEND( $VERIFICATION, ballot$ ) to acceptors;
16:
17:   if ( $\#(messages[ballot][value]) \geq quorumSize[ballot][value]$ 
and ( $quorumSize[ballot][value] = N - s$  or
 $verificationMessages[ballot][value] \geq N - u$ ) or
 $(\#(messages[ballot][value]) > u$  and ISUNIVERSALLYCOMMUTA-
TIVE( $value$ )) then
18:      $learned = \text{MERGE\_SEQUENCES}(learned, value)$ ;
```

3.4 Discussion

Extending the protocol to universally commutative commands. The Generalized Paxos specification 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 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 proposed by a correct proposer. (Furthermore, for “read-only” commands, it is possible to run these against one single learner in the crash model, since there is no requirement to withstand f faults for these commands.) This optimization is particularly useful in the context of geo-replicated systems, since it can be significantly faster to wait for the $f + 1$ st message instead of the $N - f$ th one.

Generalized Paxos and weak consistency. The key distinguishing feature of the specification of Generalized Paxos [7] 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 [35], or consistency models that mix strong and weak consistency levels like RedBlue [38], 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. E.g., checking the balance of a bank account and making a deposit do not commute since the output of the former depends on their relative order. However, some systems prefer to run both as weakly consistent operations, even though it may cause executions that are not explained by a sequential execution, since the semantics are still acceptable given that the final state that is reached is the same and no invariants of the application are violated [38].

Handling faults in the fast case. A result that was stated in the original Generalized Paxos paper [7] 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 are of size $N - e$ and $N - f$, respectively, which means there is a dependence relationship between these three concepts: number of tolerated faults, system size, and quorum size. As such, if we don't mind keeping e small, then the fast ballot quorum will be a larger fraction of the total system size, which itself will be smaller. Conversely, if we wish to allow fast ballots to progress in the presence of the maximum number of faults, we can set $e = f$, which means that $N > 3f$ and $N - e = N - f$.

An interesting observation stemming from our results is that, since Byzantine fault tolerance also requires a total system size of $N > 3f$ 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 while not increasing the overall replication factor.

3.5 Conclusion

To Do

Bibliography

- [1] M. Vukolić. Quorum systems: With applications to storage and consensus. *Synthesis Lectures on Distributed Computing Theory*, 3(1):1–146, 2012.
- [2] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM*, 32(2):374–382, 1985. ISSN 00045411. doi: 10.1145/3149.214121. URL <http://portal.acm.org/citation.cfm?doid=3149.214121>.
- [3] 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>.
- [4] M. Burrows. The Chubby lock service for loosely-coupled distributed systems. *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation SE - OSDI '06*, pages 335–350, 2006. ISSN <null>. URL <http://dl.acm.org/citation.cfm?id=1298455.1298487>.
- [5] 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. *Proceedings of OSDI'12: Tenth Symposium on Operating System Design and Implementation*, pages 251–264, 2012. ISSN 07342071. doi: 10.1145/2491245. URL papers3://publication/uuid/5F75593B-DCBB-466E-A964-B13CC0875E9D.
- [6] F. P. Junqueira, B. C. Reed, and M. Serafini. Zab: High-performance broadcast for primary-backup systems. *Proceedings of the International Conference on Dependable Systems and Networks*, pages 245–256, 2011. ISSN 1530-0889. doi: 10.1109/DSN.2011.5958223.
- [7] L. Lamport. Generalized Consensus and Paxos. (April):60, 2005. ISSN <null>. doi: MSR-TR-2005-33. URL <http://research.microsoft.com/apps/pubs/default.aspx?id=64631>.
- [8] L. Lamport. Fast Paxos. *Distributed Computing*, 19(2):79–103, 2006. ISSN 01782770. doi: 10.1007/s00446-006-0005-x.
- [9] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat. Providing high availability using lazy replication. *ACM Transactions on Computer Systems*, 10(4):360–391, 1992. ISSN 07342071. doi: 10.1145/138873.138877.

- [10] Y. Mao, F. P. Junqueira, and K. Marzullo. Mencius: Building Efficient Replicated State Machines for WANs. *Proceedings of the Symposium on Operating System Design and Implementation*, pages 369–384, 2008. URL https://www.usenix.org/legacy/events/osdi08/tech/full_papers/mao/mao.pdf.
- [11] I. Moraru, D. G. Andersen, and M. Kaminsky. There is more consensus in Egalitarian parliaments. *Sosp '13*, pages 358–372, 2013. doi: 10.1145/2517349.2517350. URL <http://dl.acm.org/citation.cfm?doid=2517349.2517350>.
- [12] 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>.
- [13] S3 data corruption?: June 22, 2008. <https://forums.aws.amazon.com/thread.jspa?threadID=22709>, . Accessed: 2016-12-22.
- [14] J. Sousa and A. Bessani. Separating the WHEAT from the Chaff: An Empirical Design for Geo-Replicated State Machines. *Proceedings of the IEEE Symposium on Reliable Distributed Systems*, 2016-Janua:146–155, 2016. ISSN 10609857. doi: 10.1109/SRDS.2015.40.
- [15] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. ZooKeeper: Wait-free Coordination for Internet-scale Systems. *USENIX Annual Technical Conference*, 8:11–11, 2010. ISSN 0210945x. URL http://portal.acm.org/citation.cfm?id=1855851{%}%5Cnhttps://www.usenix.org/event/usenix10/tech/full_{_}papers/Hunt.pdf.
- [16] R. van Renesse. Paxos Made Moderately Complex. *ACM Computing Surveys*, 47(3):1–36, 2011. ISSN 15577341. doi: 10.1145/2673577.
- [17] T. Kraska, G. Pang, and M. Franklin. Mdcc: Multi-data center consistency. *Proceedings of the 8th ...*, pages 113–126, 2013. ISSN 9781450319942. doi: 10.1145/2465351.2465363. URL <http://dl.acm.org/citation.cfm?id=2465363>.
- [18] L. Lamport, R. Shostak, and M. Pease. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, 1982. ISSN 01640925. doi: 10.1145/357172.357176.
- [19] M. Castro and B. Liskov. Practical Byzantine Fault Tolerance. *Proceedings of the Symposium on Operating System Design and Implementation*, (February):1–14, 1999. ISSN 07342071. doi: 10.1145/571637.571640. URL <http://www.itu.dk/stud/speciale/bepjea/xwebtex/litt/practical-byzantine-fault-tolerant.pdf>.
- [20] S. Liu, C. Cachin, V. Quema, and M. Vukolic. XFT: Practical fault tolerance beyond crashes. pages 1–32, 2015. URL <http://arxiv.org/pdf/1502.05831v1.pdf>.
- [21] Amazon s3 availability event: July 20, 2008. <http://status.aws.amazon.com/s3-20080720.html>, . Accessed: 2016-12-22.

- [22] E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644, 1974. ISSN 00010782. doi: 10.1145/361179.361202.
- [23] 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>.
- [24] Z. Kalbarczyk and K. Iyer. Characterization of linux kernel behavior under errors. *2003 International Conference on Dependable Systems and Networks, 2003. Proceedings.*, 00(ii):459–468, 2003. doi: 10.1109/DSN.2003.1209956. URL <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1209956>.
- [25] A. Singh, P. Fonseca, and P. Kuznetsov. Zeno: Eventually Consistent Byzantine-Fault Tolerance. *Nsdi*, pages 169–184, 2009. URL https://www.usenix.org/legacy/event/nsdi09/tech/full_papers/singh/singh.pdf.
- [26] T. Deepak Chandra, t. J. Thomas, S. Toueg, T. D. Chandra, and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, 1996. ISSN 00045411. doi: 10.1145/226643.226647.
- [27] 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>.
- [28] R. Guerraoui and A. Schiper. Gamma accurate failure detectors, 1996.
- [29] 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>.
- [30] D. Dolev, C. Dwork, and L. Stockmeyer. On the minimal synchronism needed for distributed consensus. *24th Annual Symposium on Foundations of Computer Science (sfcs 1983)*, 34(1):77–97, 1983. ISSN 0272-5428. doi: 10.1109/SFCS.1983.41.
- [31] M. K. Aguilera, C. Delporte-Gallet, H. Fauconnier, and S. Toueg. Partial synchrony based on set timeliness. *Distributed Computing*, 25(3):249–260, 2012. ISSN 01782770. doi: 10.1007/s00446-012-0158-8.
- [32] S. Chaudhuri. More Choices Allow More Faults: Set Consensus Problems in Totally Asynchronous Systems, 1993. ISSN 08905401. URL <http://linkinghub.elsevier.com/retrieve/pii/S0890540183710436>.
- [33] P. Zielinski. AntiwO: The weakest failure detector for set agreement. *Distributed Computing*, 22(5-6):335–348, 2010. ISSN 01782770. doi: 10.1007/s00446-010-0101-9.

- [34] 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>.
- [35] 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, 1995. ISSN 01782770. doi: 10.1007/BF01784241.
- [36] 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>.
- [37] A. Silberstein, A. Silberstein, B. F. Cooper, B. F. Cooper, U. Srivastava, U. Srivastava, E. Vee, E. Vee, R. Yerneni, R. Yerneni, R. Ramakrishnan, and R. Ramakrishnan. PNUTS: Yahoo!’s Hosted Data Serving Platform. *Proceedings of PVLDB 2008*, 1(2):765, 2008. ISSN 21508097. doi: 10.1145/1376616.1376693. URL <http://portal.acm.org/citation.cfm?doid=1376616.1376693>.
- [38] C. Li, D. Porto, A. Clement, J. Gehrke, R. Rodrigues, and N. Preguiça. Making Geo-Replicated Systems Fast as Possible, Consistent when Necessary. *OSDI’12 Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, pages 265–278, 2012. URL <http://dl.acm.org/citation.cfm?id=2387880.2387906>.
- [39] V. Balesgas, 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>.

Appendix A

Byzantine Generalized Paxos

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
$X \xRightarrow{e} Y$	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)
$learned_{l_i}$	Learner l_i 's <i>learned</i> sequence of commands
$learned(l_i, s)$	$learned_{l_i}$ contains the sequence s
$maj_accepted(s)$	$N - f$ acceptors sent phase 2b messages to the learners for sequence s
$min_accepted(s)$	$f + 1$ acceptors sent phase 2b messages to the learners for sequence s
$proposed(s)$	A proposer proposed s

Table A.1: 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, $\forall s, s' \in \mathcal{P}, \forall l_i, l_j \in \mathcal{L}, learned(l_j, s) \wedge learned(l_i, s') \implies \exists \sigma_1, \sigma_2 \in \mathcal{P}, s \sim s' \bullet \sigma_1 \vee s' \sim s \bullet \sigma_2$

Proof:

1.1. At any given instant, $\forall s, s' \in \mathcal{P}, \forall l_i, l_j \in \mathcal{L}, \text{learned}(l_i, s) \wedge \text{learned}(l_j, s') \implies (\text{maj_accepted}(s) \vee (\text{min_accepted}(s) \wedge (s \sim x \bullet \sigma_1 \vee x \sim s \bullet \sigma_2))) \wedge (\text{maj_accepted}(s') \vee (\text{min_accepted}(s') \wedge (s' \sim x \bullet \sigma_1 \vee x \sim s' \bullet \sigma_2)))$, $\exists \sigma_1, \sigma_2 \in \mathcal{P}, \forall x \in \mathcal{P}$

Proof: A sequence can only be learned if the learner gathers $N - f$ votes (i.e., $\text{maj_accepted}(s)$) or if it is universally commutative (i.e., $s \sim x \bullet \sigma_1 \vee x \sim s \bullet \sigma_2$, $\exists \sigma_1, \sigma_2 \in \mathcal{P}, \forall x \in \mathcal{P}$) and the learner gathers $f + 1$ votes (i.e., $\text{min_accepted}(s)$). The first case includes both gathering $N - f$ votes directly from each acceptor (Algorithm 11 lines {1-4}) and gathering $N - f$ proofs of vote from only one acceptor, as is the case when the sequence contains a special checkpointing command (Algorithm 11 {6-11}). The second case requires that the sequence must be commutative with any other (Algorithm 11 {1-4}). This is encoded in the logical expression $s \sim x \bullet \sigma_1 \vee x \sim s \bullet \sigma_2$ which is true if either the learned sequence s can be extended with a suffix σ_1 to any other sequence x or if any sequence x can be extended with some prefix σ_2 to be equivalent to s .

1.2. At any given instant, $\forall s, s' \in \mathcal{P}, \text{maj_accepted}(s) \wedge \text{maj_accepted}(s') \implies \exists \sigma_1, \sigma_2 \in \mathcal{P}, s \sim s' \bullet \sigma_1 \vee s' \sim s \bullet \sigma_2$

Proof: Proved by contradiction.

1.2.1. At any given instant, $\exists s, s' \in \mathcal{P}, \forall \sigma_1, \sigma_2 \in \mathcal{P}, \text{maj_accepted}(s) \wedge \text{maj_accepted}(s') \wedge s \not\sim s' \bullet \sigma_1 \wedge s' \not\sim s \bullet \sigma_2$

Proof: Contradiction assumption.

1.2.2. Take s and s' which are certain to exist by 1.2.1, s and s' are non-commutative

Proof: If $\forall \sigma_1, \sigma_2 \in \mathcal{P}, s \not\sim s' \bullet \sigma_1 \wedge s' \not\sim s \bullet \sigma_2$ then s and s' must contain non-commutative commands differently ordered. Otherwise, they would be possible to extend to equivalent sequences.

1.2.3. At any given instant, $\neg(\text{maj_accepted}(s) \wedge \text{maj_accepted}(s'))$

Proof: Since s and s' are non-commutative, therefore not equivalent, and each correct acceptor only votes once for a new proposal (Algorithm 10, lines {28-43}), any learner will only obtain $N - f$ votes for one of the sequences (Algorithm 11, lines {1-4}).

1.2.4. Q.E.D.

1.3. Take s and s' , at any given instant, $\forall x \in \mathcal{P}, \exists \sigma_1, \sigma_2 \in \mathcal{P}, (\text{maj_accepted}(s) \vee (\text{min_accepted}(s) \wedge (s \sim x \bullet \sigma_1 \vee x \sim s \bullet \sigma_2))) \wedge (\text{maj_accepted}(s') \vee (\text{min_accepted}(s') \wedge (s' \sim x \bullet \sigma_1 \vee x \sim s' \bullet \sigma_2))) \implies s \sim x \bullet \sigma$

Proof: By 1.2 and by definition of $(s \sim x \bullet \sigma \vee x \sim s \bullet \sigma)$.

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

Proof: By 1.1 and 1.3.

1.5. Q.E.D.

2. At any given instant, $\forall l_i, l_j \in \mathcal{L}, \text{learned}(l_j, \text{learned}_j) \wedge \text{learned}(l_i, \text{learned}_i) \implies \exists \sigma_1, \sigma_2 \in \mathcal{P}, \text{learned}_i \sim \text{learned}_j \bullet \sigma_1 \vee \text{learned}_j \sim \text{learned}_i \bullet \sigma_2$

Proof: By 1.

3. Q.E.D.

Non-Triviality

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

Proof:

1. At any given instant, $\forall l_i \in \mathcal{L}, \forall s \in \mathcal{P}, learned(l_i, s) \implies \forall x \in \mathcal{P}, \exists \sigma \in \mathcal{P}, maj_accepted(s) \vee (min_accepted(s) \wedge (s \sim x \bullet \sigma \vee x \sim s \bullet \sigma))$

Proof: By Algorithm 10 lines {28-43} and Algorithm 11 lines {1-4,6-11}, 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, $\forall s \in \mathcal{P}, maj_accepted(s) \vee min_accepted(s) \implies proposed(s)$

Proof: By Algorithm 10 lines {15-19, 28-43}, for either $N - f$ or $f + 1$ acceptors to accept a proposal it must have been proposed by a proposer.

3. At any given instant, $\forall s \in \mathcal{P}, learned(l_i, s) \implies proposed(s), \forall l_i \in \mathcal{L}$

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 11 lines {1-4,6-11}, 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. $\forall l_i \in \mathcal{L}, \forall s, x \in \mathcal{P}, \exists \sigma \in \mathcal{P}, maj_accepted(s) \vee (min_accepted(s) \wedge (s \sim x \bullet \sigma \vee x \sim s \bullet \sigma)) \xRightarrow{e} learned(l_i, s)$

Proof: By Algorithm 10 lines {28-43} and Algorithm 11 lines {1-4,6-11}, when either $N - f$ or $f + 1$ (if s is universally commutative) acceptors accepts a sequence s (or some equivalent sequence), eventually s will be learned by any correct learner.

2. $\forall s \in \mathcal{P}, proposed(s) \xRightarrow{e} \forall x \in \mathcal{P}, \exists \sigma \in \mathcal{P}, maj_accepted(s) \vee (min_accepted(s) \wedge (s \sim x \bullet \sigma \vee x \sim s \bullet \sigma))$

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 10 lines {38-43}) and, in the second case, it's sent in phase 1b messages to the in leader in the next ballot (Algorithm 10 lines {20-26}) and incorporated in the next proposal (Algorithm 8 lines {13-18,20-32}).

3. $\forall l_i \in \mathcal{L}, \forall s \in \mathcal{P}, \text{proposed}(s) \xRightarrow{e} \text{learned}(l_i, s)$

Proof: By 1 and 2.

4. Q.E.D.

Appendix B

Visigoth Generalized Paxos

Correctness Proofs

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

Invariant/Symbol	Definition
\sim	Equivalence relation between sequences
$X \xRightarrow{e} Y$	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)
$learned_{l_i}$	Learner l_i 's <i>learned</i> sequence of commands
$learned(l_i, s)$	$learned_{l_i}$ contains the sequence s
$min_accepted(s)$	$f + 1$ acceptors sent phase 2b messages to the learners for sequence s
$proposed(s)$	A proposer proposed s
$acc_send(x, s)$	At least x acceptors have sent votes for the sequence s
$wait_learn(t, l, s)$	At least t time units have passed since the learner l received the first vote for sequence s and $N - s$ votes haven't been gathered

Table B.1: Proof notation

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, $\forall s, s' \in \mathcal{P}, \forall l_i, l_j \in \mathcal{L}, learned(l_j, s) \wedge learned(l_i, s') \implies \exists \sigma \in \mathcal{P}, s \sim s' \bullet \sigma \vee s' \sim s \bullet \sigma$

Proof:

1.1. At any given instant, $\forall seq, seq' \in \mathcal{P}, \forall l_i, l_j \in \mathcal{L}, learned(l_i, seq) \wedge learned(l_j, seq') \implies \exists \sigma_1, \sigma_2 \in \mathcal{P}, \forall x \in \mathcal{P}, (acc_send(N - s, seq) \vee (acc_send(N - u, seq) \wedge wait_learn(3T, l_i, seq)) \vee (min_accepted(seq) \wedge (seq \sim x \bullet \sigma_1 \vee x \sim seq \bullet \sigma_1))) \wedge (acc_send(N - s, seq') \vee (acc_send(N - u, seq') \wedge wait_learn(3T, l_j, seq')) \vee (min_accepted(seq') \wedge (seq' \sim x \bullet \sigma_2 \vee x \sim seq' \bullet \sigma_2)))$

Proof: A sequence can be learned if: (1) the learner gathers either $N - s$ votes (Algorithm 17 lines {1-4}), (2) the learner gathers $N - u$ votes after a timeout of $3T$ occurs (Algorithm 17 {5, 10-13}) or (3) if the sequence is universally commutative and the learner gathers $f+1$ votes (Algorithm 17 {7-8}). This requires that the sequence must be commutative with any other possible sequence, which is encoded in the logical expression $s \sim x \bullet \sigma \vee x \sim \bullet \sigma$. This expression is true if either the learned sequence s can be extended with some suffix σ_1 to any other sequence x or if any sequence x can be extended with some prefix σ_2 to be equivalent to s .

1.2. At any given instant, $\forall seq, seq', x \in \mathcal{P}, \exists \sigma_1, \sigma_2 \in \mathcal{P}, (acc_send(N - s, seq) \vee (acc_send(N - u, seq) \wedge wait_learn(3T, l_i, seq)) \vee (min_accepted(seq) \wedge (seq \sim x \bullet \sigma_1 \vee x \sim seq \bullet \sigma_1))) \wedge (acc_send(N - s, seq') \vee (acc_send(N - u, seq') \wedge wait_learn(3T, l_j, seq')) \vee (min_accepted(seq') \wedge (seq' \sim x \bullet \sigma_2 \vee x \sim seq' \bullet \sigma_2))) \implies \exists \sigma_3 \in \mathcal{P}, seq \sim seq' \bullet \sigma_3 \vee seq' \sim seq \bullet \sigma_3$

Proof:

1.2.1. At any given instant, $\forall seq, seq' \in \mathcal{P}, (acc_send(N - s, seq) \vee (acc_send(N - u, seq) \wedge wait_learn(3T, l_i, seq))) \wedge (acc_send(N - s, seq') \vee (acc_send(N - u, seq') \wedge wait_learn(3T, l_j, seq')))) \implies \exists \sigma \in \mathcal{P}, seq \sim seq' \bullet \sigma \vee seq' \sim seq \bullet \sigma$

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 votes. Since a correct acceptor will not vote for two non-commutative sequences, we must prove that in, any two gathered quorums, the intersection between them contains at least one correct quorum.

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 $N = u + o + s + 1$:

$$N - s + N - s > N + o$$

$$N - 2s > o$$

$$u + o + s + 1 - 2s > o$$

$$u + 1 - s > 0$$

$$u + 1 > s$$

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

Given two quorums of size Q_2 , when the first quorum is gathered the sizes are $q_1 = N - u - a$ and $q_2 = N - u - b$. This means that there are $u - a - s$ crashed processes and the system size is now $N' = N - u + a + s$. 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 have crashed, which means that the system is now of size $N' = N - u + a + s$. To guarantee that the intersection between quorums contains at least one correct process, the following must hold $Q_1 + Q_2 - N' > o$:

$$Q_1 + Q_2 - N' > o$$

$$N - s + N - u + a - N + u - a - s > o$$

$$N - 2s > o$$

$$53 \quad u + o + s + 1 - 2s > o$$

$$u - s + 1 > 0$$

1.2.2. At any given instant, $\forall seq, seq', \sigma \in \mathcal{P}, (min_accepted(seq) \wedge (seq \sim x \bullet \sigma_1 \vee x \sim seq \bullet \sigma_1)) \implies \exists \sigma_2 \in \mathcal{P}, seq \sim seq' \bullet \sigma_2 \vee seq' \sim seq \bullet \sigma_2$

Proof: If seq is universally commutative then, trivially, seq is commutative with any other sequence seq' .

2. At any given instant, $\forall l_i, l_j \in \mathcal{L}, learned(l_j, learned_j) \wedge learned(l_i, learned_i) \implies \exists \sigma \in \mathcal{P}, learned_i \sim learned_j \bullet \sigma \vee learned_j \sim learned_i \bullet \sigma$

Proof: By 1.

3. Q.E.D.

Non-Triviality

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

Proof:

1. At any given instant, $\forall l_i \in \mathcal{L}, \forall seq \in \mathcal{P}, learned(l_i, seq) \implies \forall x \in \mathcal{P}, \exists \sigma \in \mathcal{P}, acc_send(N - s, seq) \vee (acc_send(N - u, seq) \wedge wait_learn(3T, l_i, seq)) \vee (min_accepted(seq) \wedge (seq \sim x \bullet \sigma \vee x \sim seq \bullet \sigma))$

Proof: By Algorithm 16 lines {31-44} and Algorithm 17 lines {1-8}, if a correct learner learned a sequence s 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, $\forall seq, x \in \mathcal{P}, \exists \sigma \in \mathcal{P}, acc_send(N - s, seq) \vee (acc_send(N - u, seq) \wedge wait_learn(3T, l_i, seq)) \vee (min_accepted(seq) \wedge (seq \sim x \bullet \sigma \vee x \sim seq \bullet \sigma)) \implies proposed(seq)$

Proof: By Algorithm 16 lines {9-11, 17-19}, for either a quorum to be gathered or $f + 1$ acceptors to accept a proposal, it must have been proposed by a proposer.

3. At any given instant, $\forall seq \in \mathcal{P}, learned(l_i, seq) \implies proposed(seq), \forall l_i \in \mathcal{L}$

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 17 lines {8,18}, 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. $\forall l_i \in \mathcal{L}, \forall seq, x \in \mathcal{P}, \exists \sigma \in \mathcal{P}, acc_send(N - s, seq) \vee (acc_send(N - u, seq) \wedge wait_learn(3T, l_i, seq)) \vee (min_accepted(seq) \wedge (seq \sim x \bullet \sigma \vee x \sim seq \bullet \sigma)) \xRightarrow{e} learned(l_i, seq)$

Proof: By Algorithm 16 lines {31-44} and Algorithm 17 lines {7-8,17-18}, 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. $\forall seq \in \mathcal{P}, proposed(seq) \xRightarrow{e} \forall x \in \mathcal{P}, \exists \sigma \in \mathcal{P}, acc_send(N - s, seq) \vee (acc_send(N - u, seq) \wedge wait_learn(3T, l_i, seq)) \vee (min_accepted(seq) \wedge (seq \sim x \bullet \sigma \vee x \sim seq \bullet \sigma))$

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 16 lines {27-31}) 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 16 lines {24-29}). After being sent to the leader, the sequence is incorporated in the next proposal and sent to the acceptors along with others proposed commands (Algorithm 14 lines {23-27,29-35}).

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

Proof: By 1 and 2.

4. Q.E.D.