Generalized Consensus for Practical Fault Tolerance

Mohit Garg

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Engineering

August 24, 2018
Blacksburg, Virginia

Keywords: Distributed Systems, Fault Tolerance, Byzantine Fault Tolerance, State Machine Replication, Multi-Leader Consensus, Blockchain

Generalized Consensus for Practical Fault Tolerance

Mohit Garg

(ABSTRACT)

Despite extensive research on Byzantine Fault Tolerant (BFT) systems, overheads associated with such solutions preclude widespread adoption. Past efforts such as the Cross Fault Tolerance (XFT) model address this problem by making a weaker assumption that a majority of replicates are correct and communicate synchronously. Although XPaxos of Liu et al. (applying the XFT model) achieves similar performance as Paxos, it does not scale with the number of faults. Also, its reliance on a single leader introduces considerable downtime in case of failures. This thesis presents Elpis, the first multi-leader XFT consensus protocol. By adopting the Generalized Consensus specification from the Crash Fault Tolerance model, we were able to devise a multi-leader protocol that exploits the commutativity property inherent in the commands ordered by the system. Elpis maps accessed objects to non-faulty processes during periods of synchrony. Subsequently, these processes order all commands which access these objects. Experimental evaluation confirms the effectiveness of this approach: Elpis achieves up to 2x speedup over XPaxos and up to 3.5x speedup over state-of-the-art Byzantine Fault-Tolerant Consensus Protocols

This work is supported in part by Air Force Office of Scientific Research (AFOSR) under grant FA9550-15-1-0098 and by National Science Foundation (NSF) under grant CNS-1523558

Generalized Consensus for Practical Fault Tolerance

Mohit Garg

(GENERAL AUDIENCE ABSTRACT)

Online services like Facebook, Twitter, Netflix and Spotify to cloud services like Google and Amazon serve millions of users which include individuals as well as organizations. They use many distributed technologies to deliver a rich experience. The distributed nature of these technologies has removed geographical barriers to accessing data, services, software, and hardware. An essential aspect of these technologies is the concept of the shared state. Distributed databases with multiple replicated data nodes are an example of this shared state. Maintaining replicated data nodes provides several advantages such as (1) availability so that in case one node goes down the data can still be accessed from other nodes, (2) quick response times, by placing data nodes closer to the user, the data can be obtained quickly, (3) scalability by enabling multiple users to access different nodes so that a single node does not cause bottlenecks. To maintain this shared state some mechanism is required to maintain consistency, that is the copies of these shared state must be identical on all the data nodes. This mechanism is called Consensus, and several such mechanisms exist in practice today which use the Crash Fault Tolerance (CFT). The CFT model implies that these mechanisms provide consistency in the presence of nodes crashing. While the state-of-the-art for security has moved from assuming a trusted environment inside a firewall to a perimeter-less and semi-trusted environment with every service living on the internet, only the application layer is required to be secured while the core is built just with an idea of crashes in mind. While there exists comprehensive research on secure Consensus mechanisms which utilize what is called the Byzantine Fault Tolerance (BFT) model, the extra costs required to implement these mechanisms and comparatively lower performance in a geographically distributed setting has impeded widespread adoption. A new model recently proposed tries to find a cross between these models that is achieving security while paying no extra costs called the Cross Fault Tolerance (XFT). This thesis presents Elpis, a consensus mechanism which uses precisely this model that will secure the shared state from its core without modifications to the existing setups while delivering high performance and lower response times. We perform a comprehensive evaluation on AWS and demonstrate that Elpis achieves 3.5x over the state-of-the-art while improving response times by as much as 50%.

Dedication

...to my parents and Meenu.

Acknowledgments

It was towards the end of my second consecutive semester taking a course by Professor Binoy Ravindran that I finally walked up to him and asked him if he had some research projects related to Distributed Systems. From that day onwards, he has zealously and selflessly offered his guidance and his time. His advice has helped me grow professionally and personally. For this, I would like to thank him immensely. I could not have asked for a better mentor.

Working on research problems in distributed algorithms is challenging, and you can get easily blindsided by incorrect approaches. However, the ability to discuss ideas with Dr. Sebastiano Peluso who has an encyclopedic knowledge of the field has been an absolute luxury. His unparalleled professionalism and his methodology for breaking down complex problems did not only simplify my work but has made me a better researcher.

There is a reason why research groups exist. Having the ability to ask for help from experienced graduate students whether it be when your code does not work or when you need guidance to traverse the graduate school maze is a privilege. I would like to thank Balaji and Anthony for their never-ending support whether it was when I got stuck on anything in the lab or outside of it. I would also like to thank Daniel for our conversations from our shared frame of reference of being Masters students from India.

My mom and dad have been a constant source of advice and encouragement. I knew that whatever kind of day I have, in the evening I can get on a call with them and everything would just be fine. I thank them for their sacrifices, and I dedicate this thesis to them.

My partner Meenu serendipitously met me when I started working towards this thesis. Throughout the ups and the downs, she has offered me endless support, be it in the form of hour-long motivational speeches or at times keeping me focused on the task at hand. It is an understatement when I say that I could not have done it without her.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Any significant online service today is supposed to be globally accessible. From social media services like Facebook and Twitter to multimedia streaming services like Netflix and Spotify to online marketplaces like Amazon, the ability to scale to serve hundreds of millions of active users is vital for their continued growth. Consequently, these companies have spent a lot of capital and effort in setting up their cloud infrastructures internally or have relied on public cloud services like Amazon Web Services and Google Cloud. These cloud services work by *replicating* data across tens of thousands of data servers across the globe which interact with each other through internal networks. Replication aids scalability by not only reducing response times by enabling users to access local servers but also helps distribute processing and storage loads. Maintaining large clusters brings additional challenges as at any given time a sizable number of servers and network components fail. Any downtime for these online services is costly. For Fortune 1000, the average total cost of application downtime is between \$1.25 billion to \$2.5 billion [1]. High availability by building services that tolerate faults is hence no longer the exception but the norm.

Therefore, the developers are using *stateless techniques* like microservices [2], and *stateful replication techniques* like distributed databases so that the overall system continues to work transparently to the end user even in the presence of components failures. Stateful replication is especially challenging as it requires coordination amongst processes to maintain a strongly consistent state. The costs become significant in the geographically replicated (geo-replicated) services due to considerably high network latencies. As an alternative, the geo-replicated systems available today provide weaker consistency guarantees which complicate the application development due to the constant reconciliation between transaction timestamps.

Additionally, while applications using these replicated services are expected to emphasize security, the services only consider the failures that arise due to components crashing and stop working altogether. Any arbitrary faults due to data corruptions, faulty hardware, misconfigurations, software bugs or even malicious behavior are ignored entirely and left to checksum mechanisms at the application layer. The existing techniques categorized as Byzantine Fault-Tolerance have seen low adoption in large-scale systems outside of cryptocurrencies and safety-critical systems in particular due to their need for extra components as well as relatively poor performance, particularly in the geo-replicated setting. This thesis addresses the challenges of security, performance, and availability of geographical replication while demanding no extra resources by presenting a new Consensus algorithm which provides stronger consistency and reliability.

1.1 Motivation

Consensus algorithms are designed using two prominent fault models the Crash Fault-Tolerant (CFT) model and the Byzantine Fault-Tolerant (BFT) model. The BFT model has gained traction recently with the proliferation of the Cryptocurrency technologies. However, the BFT algorithms are expensive as they require more resources and use complex messaging patterns that require higher network bandwidths. Compared to CFT protocols, BFT protocols need bigger *quorums* (Ref. Section 2.3.4), more communication steps and use cryptographic signatures during message exchanges. Notably, in geo-scale deployments where round-trip timings (RTT) are higher, the requirement of more nodes in the quorum, as well as more communication steps, tremendously increases the user-perceived latencies, discouraging wide-spread adoption. Various approaches [3, 4] have tried to improve the performance of BFT protocols, but due to the lower bounds of the BFT model itself [5] none of them were able to reduce both the number of communication steps as well as the quorum size.

If we study the BFT model, the need of bigger quorums is a direct result of the assumption that there exist *slow* processes due to an asynchronous network along with some Byzantine processes. In practice, this scenario can take place when an adversary is able to affect the network on a wide scale (affecting multiple processes) as well as attack multiple servers in another location, all in a coordinated fashion. To address this limitation, in [6], the authors propose the Cross Fault Tolerance (XFT) model, which is a trade-off between Crash Fault Tolerance and Byzantine Fault Tolerance models. Mainly, the XFT model relaxes the assumption that the adversary can launch coordinated attacks, which is unlikely in geo-scale deployments. The XFT model provides the same quorum size of t+1 and uses the same number of communication steps as the CFT model under normal conditions.

The proposed consensus solution under the XFT model, XPaxos, is a leader-based protocol whose performance is similar to CFT-based Raft/Multi-Paxos [7]. While this algorithm is built with an assumption that befits the geo-replicated setting, the accompanying algorithm does not exactly solve the requirements of scalability and performance. The XPaxos leader only changes when there is a fault and as such XPaxos inherits the shortcomings of the leader-based approach in particular the imbalanced load distribution in which the leader does more work than other nodes in the system, high latency for requests originating from non-leader processes due to the requirement of forwarding, and the inability to deliver any commands whenever the current leader is slow or Byzantine until and unless a leader change takes place.

Multi-leader consensus solutions [8, 9, 10] have been proposed for the CFT model in recent years to address the aforementioned issues. Such solutions adopt the Generalized form of Consensus [11], which exploits the underlying commutativity of commands entering the system, such that the commutative commands can be ordered differently across different nodes and only non-commutative commands need consistent ordering across all the nodes. Our goal is to therefore provide a Generalized consensus algorithm in the XFT model which achieves high performance in the geo-replicated setting by using XFT, an adversary model which befits the Geo-replicated setting.

1.2 Summary of Thesis Contribution

This thesis presents Elpis, the first multi-leader Cross Fault Tolerant (XFT) consensus protocol, which exploits the underlying commutativity of commands to provide fast decisions in two communication steps from any node, in the common case. We are able to achieve this by exploiting workload locality that is very common in geo-scale deployments.

The core idea of Elpis is about having *ownership* at a finer granularity. Rather than having a single leader that is responsible for ordering all commands regardless of their commutative property, we assign ownership to individual nodes such that each node mostly proposes only commutative commands with respect to other nodes. As a result, each node is responsible for deciding the order of all its commands that commute with other nodes. We define commutativity by the object(s) that a command accesses during its execution. With this, we assign *ownership* to nodes on a per object basis. Such ownership assignment guarantees that no other node will propose conflicting commands, and thus, fast decisions in two communication steps can be achieved from the owner nodes.

Elpis also allows for dynamic ownership changes. Individual nodes can gain ownership of any object(s) using a special *ownership* acquisition phase. We recognize the conflicts due to concurrent ownership acquisition of the same object(s) by multiple nodes. We address this with a rectification mechanism that follows the ownership acquisition phase during conflicts and minimizes the number of node retries to acquire ownership.

We implemented Elpis and the competitors - XPaxos, PBFT, M^2Paxos, and Zyzzyva - in Java, using the JGroups messaging toolkit. We extensively evaluated each of the existing solutions and contrasted their performances to show the gains achieved by our solution. To summarize, Elpis achieves up to 2x speedup over XPaxos and up to 3.5x speedup over state-of-the-art Byzantine Fault-Tolerant Consensus Protocols.

In summary, the contributions of this thesis are:

- The design and implementation of Elpis, the first Generalized Cross Fault Tolerant (XFT) Consensus algorithm designed for geo-replicated systems.
- An ownership conflict resolution protocol that minimizes the ownership acquisition retries using a more cohesive algorithm.
- An extensive evaluation and comparison to existing state-of-the-art in the BFT space as well as XPaxos itself.

1.3 Thesis Organization

The rest of the thesis is organized as follows:

- Chapter 2, provides a background on Replication, Consensus, key abstractions, and the existing state-of-the-art solution that are related to the contribution, Elpis.
- Chapter 3 presents the system model, the assumptions and the definition of the problem tackled by Elpis.
- Chapter 4 introduces Elpis at a high level and then delves into the details and presents the algorithm pseudo-code.
- Chapter 5 presents the correctness proof of Elpis.
- Chapter 6 provides a comprehensive evaluation of Elpis and competitors.
- Chapter 7 provides the key insights, avenues of future work and concludes the thesis.

Chapter 2

Background and Related Work

Distributed computing is the coordination among components interconnected by network links to achieve a common goal. This coordinated system is simply referred to as a distributed system. For the discussion in this thesis, a *process* abstraction is used to refer to any component which performs computation such as a computer, a process within a computer or even a specific thread.

2.1 Fault-Tolerance

The processes which make up a distributed system can *fail*. A *failure* occurs whenever the process deviates from the execution of the computation. This deviation takes place when the process crashes and stops running subsequent steps, skips over steps, or acts arbitrarily either accidentally or due to adversaries. Whenever a process *fails*, it is assumed to fail completely. In other words, the unit of measurement of failures is a process. Fault-tolerance is the property of the system to continue operating correctly despite any failures. The *reliability* of the system indicates the probability of success of the computation in the system. The *availability* of a system is proportional to the time the system stays operational. Availability is directly related to the reliability of the system - the higher the reliability, the higher the availability.

2.2 Replication

A fundamental aspect of computing, replication implies maintaining consistency among redundant computing resources (hardware or software) to gain reliability, high availability, and fault-tolerance. On access, the replicated entity behaves like a single, non-replicated entity despite any failures. External clients can submit requests to this replicated entity. However, the process of replication is transparent to the clients. Replication is often confused with Backup however the backup resources are accessed infrequently and maintain their state for a longer duration of time.

Replication can be *passive* wherein a primary processes all the requests and copies the state synchronously or asynchronously to other processes, or it can be *active* wherein multiple primaries process every single request. Therefore, any process can be queried to attain the state of the system. Passive replication is simple and available widely in several enterprise database systems [12] [13]. However, it is ill-suited for fault-tolerance. In case the primary process fails while transferring the state, the system could become inconsistent. When the system chooses a new process as the primary, it could serve client queries from an older state.

Therefore, it is imperative for fault-tolerance that the replication is active. All processes should execute the client requests and maintain their states so that in case of a failure any non-faulty process (or a set of non-faulty processes) can be queried for the current state of the system. This approach is termed as *State Machine Replication*(SMR) wherein every process transfers states by executing requests in order. SMR is frequently employed to mask failures by replicating servers (a state machine) and coordinating clients with these replicated servers.

For replication, the state machine should be deterministic that is given an initial state, and a set of congruently ordered inputs the state machines transition to the same state and produce the same output. In case of divergence, however, different clients may receive different outputs. In the presence of faults, this deviation is particularly likely because a failed process could differ in its state and output from other machines. Consensus (or agreement) algorithms are therefore used for SMR to assure that the servers maintain consistent state even in the presence of faults. To maintain consistency, however, all processes should observe an equivalent order of inputs that is an order which results in the transition to the same state from a given initial state.

2.3 Consensus

How to reach consensus among a set of processes is a fundamental problem in Distributed Systems. Consensus algorithms are used by processes to decide on a common value to maintain a consistent state or decide the future course of action under a model in which the processes can fail.

2.3.1 Fault Models

The fault models identify the relevant properties and interaction characteristics. Broadly speaking two prominent models are utilized to design distributed systems,

Crash Fault-Tolerant (CFT): A model which encapsulates crash faults where a process stop execution. The process executes the computation correctly up to some time and subsequently stop any local computation as well as any message exchange.

Byzantine Fault-Tolerant(BFT) Model: The processes can deviate from the algorithm in any conceivable manner either due to accidental faults or due to adversarial or malicious behavior. The Byzantine Fault-Tolerant model specifies these faults as well as the ability of the adversary to control the execution of processes or affect the messages exchanged.

The failure assumptions along with deployment scenarios, system parameters, and design specifications play a huge role in system design[14]. Further assumptions can be made depending on whether the process failure can be detected by other processes or if recovery from failures is possible.

2.3.2 Communication

Generally, perfect links or *reliable* links are assumed for designing Consensus algorithms which guarantee the following properties.

Reliable Delivery If a message is sent from one correct process to another then the message is eventually delivered.

No duplication A message is delivered exactly once.

The reliability should be guaranteed by network protocols at lower communication layers. Distributed systems differ from shared memory systems in the fact that the time for function invocation is significant. The characteristics of how processes and network links interact with the passing of time and the timing assumptions for communication delays are hence pivotal for designing Consensus algorithms.

Synchronous Communication: Communication is termed to be *synchronous* if there is a fixed upper bound on message transmission delays.

Asynchronous Communication: If there are, however, no bounds on message transmission delays then the communication is termed to be *asynchronous*. This does not imply that the messages would take longer to get delivered but just that the delays can be arbitrary.

The timing assumptions are important as the FLP result [15] proves that no deterministic algorithm achieves consensus if the network is asynchronous even if a single process crashes. Consequently, several algorithms assume alternate models of *synchrony* such as the *eventually synchronous* model where the network can be asynchronous from time to time but is eventually synchronous. These algorithms guarantee *liveness* only during the *synchronous* periods. Alternatively, several *randomized* algorithms have been designed for the asynchronous model which is beyond the discussion of this thesis but are exciting to learn about.

2.3.3 Safety and Liveness

Generally, the Consensus algorithms guarantee the following properties.

Validity: If a process decides some value v then this value should be proposed by some process. In other words, the Consensus algorithm cannot invent values out of thin air.
Safety: Stated roughly, the safety property states that the algorithm should not do anything wrong. Consensus algorithms are intended to achieve agreement. Hence, the safety property states that no two processes decide on different values.
Liveness: The processes decide on some value eventually or in other words termination is guaranteed.

2.3.4 Resilience

Consensus algorithms are designed with the assumption that only a limited number t of processes fail, which could range from a minority of processes to a single process. [14]. The number is an upper bound, that is t processes may fail, however, not that these t processes exhibit failure under each and every execution. This relation between the number of total processes N and the number of potentially faulty processes t in the system is termed as the *resilience.*

The idea of *quorums* is repeatedly used to design Consensus algorithms. In the context of Crash Fault-Tolerant algorithms, the *quorum* is any majority of processes. Every two quorum of processes have a non-empty intersection, and for this reason, the quorum is the majority of processes. The core idea is the realization that if a majority of processes Q pick a value v in the instance i then no other majority can exist which chooses a value v' different from v in instance i. If we consider any other majority Q' different from Q there would be at least be one non-faulty process which is common to both Q and Q'. Since this common process either chooses v in i before v' or v' in i before v consistency is guaranteed. If there are N processes and t of them are faulty then the set of $N-t$ processes must form a majority. In other words,

$$N - t > \frac{N}{2} \iff N > 2t$$

However, in the case of Byzantine faults, two majority quorums might not intersect in a non-faulty process. A Byzantine quorum is a set of more than $\frac{N+t}{2}$ processes such that the quorums intersect at one *correct* process. For t processes, every Byzantine quorum consists of

$$\frac{N+t}{2} - t = \frac{N-t}{2}$$

Two Byzantine quorums together would have more than

$$\frac{N-t}{2} + \frac{N-t}{2} = N - t$$

correct processes. Hence, one correct process would exist in both quorums. Up to t correct processes can be slow: hence there must exist a Byzantine quorum of correct processes in the system, to guarantee progress. Hence,

$$N - t > \frac{N+t}{2} \iff N > 3t$$

2.3.5 Cryptography

Consensus algorithms which execute in untrusted and open environments where messages can be modified or corrupted rely on cryptographic methods. The most commonly used methods include,

Hash Function: A function H takes a string x of arbitrary length and returns a value h which is a short string of fixed length. A Hash function is collision free that is no process including one subject to arbitrary faults can find two strings x and x' such that $H(x) = H(x')$.

Message Authentication Codes (MACs): A shared symmetric key between two participants is used to authenticate messages transferred between the participants. MACs are used to prevent forgery by any adversary which does not possess the shared key. MACs are a combination of key-generation, signing and verifying functions. A key-generator function G outputs a key k which is dependent on the security parameters. A signing function S generates a tag l for the key k. A verifying function V outputs a value of *accept* or *reject* depending on the inputs k, x and l.

Digital Signatures: A digital signature is created with a private key, and verified with the corresponding public key of an asymmetric key-pair. Only the holder of the private key can generate this signature, and typically anyone knowing the public key can verify it. A key-generator function G outputs a private key s_k and a public key p_k. A signing function S takes p_k and x and returns a tag l. A verifying function V outputs *accept* or *reject* by considering the inputs p_k, x and l.

2.4 Consensus Algorithms

Paxos is the most famous and widely studied Consensus algorithm. It was introduced by Leslie Lamport in an allegorical form using the voting process in the parliament of a fictional Greek island of the same name[16]. Paxos was developed independently along with Oki and Liskov's View-Stamped Replication [17]. Both of these algorithms employ the two-phase commit algorithm [18] whereby first the processes express their intention (by voting) to process a transaction and when enough votes are collected these processes commit the transaction. In case of a crash , however, the two-phase commit algorithm can block and some form of crash recovery mechanism is required. View-Stamped replication and Paxos introduce the idea of *views* and view ids where each event taking place in a particular view has the same unique id. This approach helps transfer state from one view to the next in case of failures.

Subsequently, several algorithms have been proposed based on the Crash Fault Tolerance (CFT) model. For instance, Raft [7], Fast Paxos[19], EPaxos[20], M^{2}Paxos [10] help achieve consensus in the presence of crash faults albeit under varying assumptions on the number of processes required to tolerate F faults. For the faults that are beyond the purview of the CFT model, Lamport's Byzantine Fault-Tolerance (BFT) model [5] is employed which addresses arbitrary faults. This model has motivated several BFT algorithms such as Practical Byzantine Fault-Tolerance [21] and Zyzzyva [3] amongst others.

Recently, the Cross Fault-Tolerance (XFT) model has been proposed by Liu et al. [6] along with the XPaxos consensus algorithm which strives to achieve the guarantees of the BFT algorithms by incurring the cheaper costs of the CFT algorithms.

2.4.1 Crash Fault-Tolerance

2.4.1.1 Paxos

To achieve Consensus amongst a set of totally connected [1] processes each process runs the Paxos algorithm. Each process assumes one or more roles: *Proposer*, *Acceptor* or a *Learner*. A *Proposer* acts as a coordinator for client requests. All client requests are sent to the *Proposer*. The *Acceptors* respond to messages from the *Proposer* and the *Learners* discover the value chosen by the *Acceptors*.

Paxos achieves this by having *Proposers* send a *Proposal* to all *Acceptors* with a unique number and the client request they received and have *Acceptors* only accept proposals with the highest proposal number amongst all the Proposals they receive. If the Proposal received has a higher proposal number than what an Acceptor has seen before the Acceptors send an acknowledgment otherwise the Acceptors respond with a not-acknowledged message.

For each client request received a new Consensus Instance is initiated. A single Consensus Instance proceeds as follows (Figure 2.1).

[1] All pairs of processes can exchange messages with each other

Phase 1:

(a) The Proposer selects a unique proposal number *n* and sends a *prepare* request to all the Acceptors.

(b) If the Acceptors haven't received any *prepare* greater than *n* then the Acceptors respond with an acknowledgment thus promising that they won't accept any other proposal with number less than or equal to n.

Phase 2:

(a) If the Proposer receives *prepare* acknowledgments from a majority of acceptors then the Proposer sends an *accept* for value *v* with the proposal number *n*.

(b) If an acceptor receives an *accept* with a proposal number equal to *n* then the Acceptors accept the proposal unless they have received some proposal greater than *n*. If that's the case then the Acceptors will *reject* the accept.

Phase 3:

(a) In this phase after receiving acknowledgment from a majority of Acceptors, the Proposer sends a *learn* message to all the Learners with the proposal number *n* and the value *v*. If the Learners *reject* then the Proposer retries from Phase 1.

The algorithm can be optimized by having Acceptors send the *learn* message to Learners after they accept which saves one communication delay.

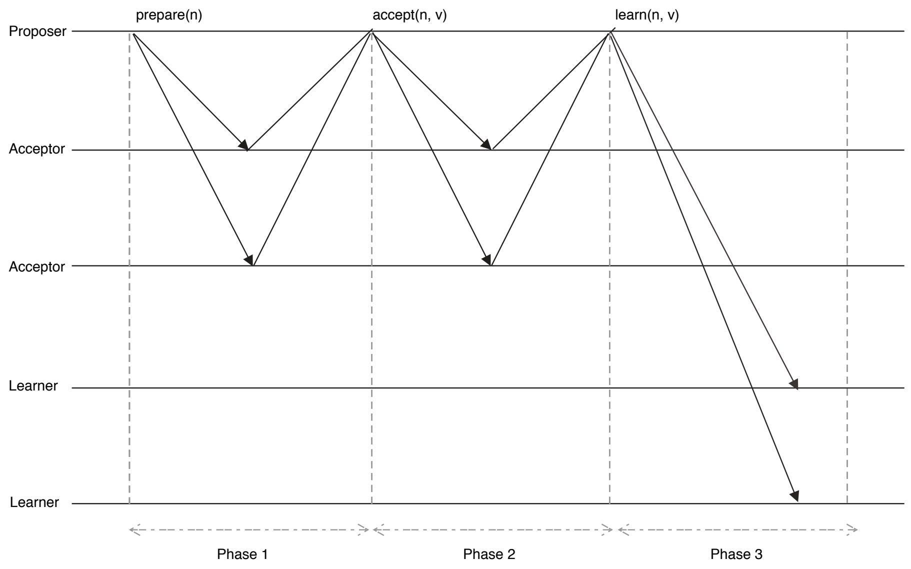

Figure 2.1: Three phases of Paxos - Prepare, Accept and Learn

2.4.1.2 Multi-Paxos and Raft

Paxos describes how to reach an agreement on a single value, self-described as a single-decree consensus algorithm. For reaching consensus on a multiple values several instances of the protocol are run. This Multi-Paxos approach brings several challenges. For instance, consider the following scenario.

Dueling Proposers Two Proposers p_i and p_j (Figure 2.2) send a prepare with proposal numbers n_i and n_j respectively such that $n_j > n_i$. Let's assume a majority of Acceptors complete Phase 1b for proposal number n_j. As a result p_i prepares for Phase 3a. Meanwhile, p_j completes it's Phase 1. At this point the highest proposal number that a majority of Acceptors has seen is n_j. Now Acceptors will reject Phase 2 for p_i because it's proposal number $n_i < n_j$. The p_i would retry Phase 1 with a higher proposal number $n_k > n_j$. Hence, p_j would not be able to complete it's Phase 2. it is straightforward to see that this can go on forever. Hence, the Liveness property cannot be guaranteed. To solve this problem, Multi-Paxos assumes a single Proposer which acts as the Leader. In case this Leader fails a new Leader is chosen by using timeouts or by randomization.

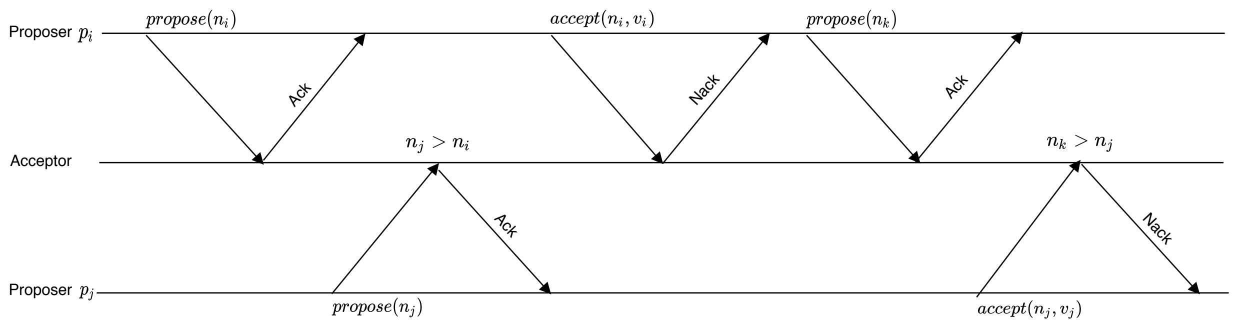

Figure 2.2: Three phases of Paxos - Prepare, Accept and Learn

Raft[7] is one of the most widely employed Multi-Paxos protocol [22, 23, 24, 25, 26]. The protocol works by replicating an indexed log across processes which can assume the role of a *Leader*, *Follower*, and/or a *Candidate*. The algorithm proceeds in a sequence of terms with a single elected Leader for every term. Under normal operation, the client requests are forwarded to the Leader.

Upon receiving the request the Leader enters the command to it's log and sends an `Append-Entries` request which includes the leader's term, the term and index of the log entry immediately preceding the new entry it just added and it's commit index. The *Followers* append the entry if they receive a request with a term higher than their current term and an index that immediately follows the previous index it appended the entry to. If this is the case then the *Followers* send an acknowledgment to the *Leader*. Once the Leader receives an acknowledgment from a majority of processes it commits the entry it just added and all the previous entries and increments its commit index. The *Followers* send heartbeat requests to the *Leader* at predetermined intervals. If the *Followers* don't receive a heartbeat, then they assume the role of a *Candidate*, increment the term and send a `RequestVote` request to all the processes. If a *Candidate* receives a vote from a majority of processes, it becomes a *Leader* and sends an empty `AppendEntries` request to all the processes. Subsequently, it can start serving client requests.

2.4.1.3 Fast Paxos

The basic Paxos algorithm requires three message delays to learn a value. Fast-Paxos [19] aims to reduce the message delays to two (which is optimum [11]) given the condition that the number of *Acceptors* $N > 3F$, where N is the number of Acceptors and t is the number of faulty processes as opposed to $N > 2t$ required for basic Paxos.

The *Quorum* size in the basic Paxos is the majority of processes wherein any two quorums intersect at not less than a single process. However, Fast Paxos specifies *Fast Quorums.* The size of the *Fast Quorums* should be such that any two *FastQuorums* Q_1 and Q_2 should intersect at the classical quorum. Hence, $Q_1 \cap Q_2 \cap Q$ is an empty set.

The Client sends an `Accept` directly to all the processes if the *Leader* has no value to propose. In this case, the *Acceptors* send the acknowledgment to the leader similar to Phase 2b and the leader follows basic Paxos from there on. Hence, the processes achieve consensus in two communication delays which is optimal.

However, if a conflict is detected, then the *Leader* can *coordinate* the recovery by sending an `Accept` message with a new round. In this case, four communication delays are required. Alternatively, the *Leader* can specify coordination rules for *Acceptors* in advance. In the case of a conflict, the *Acceptors* can follow the rule resulting in three communication delays.

2.4.1.4 Generalized Paxos

Generalized Paxos abstracts the idea of agreeing on a single value in each instance of the algorithm as is the case in *Basic Paxos* to agreeing on a set of values. In the context of State Machine Replication if a *Learner* learns a set of commands, a C-Struct then non-commutative commands can be ordered arbitrarily. The resulting C-Structs would be equivalent.

Commutative Commands: If the ordering of the command execution set is determinant to the output of the state machine, then the commands are non-commutative. However if the ordering between commands produces the same final state and output, the commands are commutative. For example, let's assume a distributed key-value store which implements State Machine Replication and implements `Put(K, V)` and `Remove(K)` requests. A `Put(K,` `V)` sets a value `V` for a key `K` and `Remove(K)` removes the key `K` and the value associated with `K`. Let's consider commands $C_1 =$ `Put(2, 3)`, $C_2 =$ `Remove(7)`, and $C_3 =$ `Remove(2)`. While C_1 and C_2 can be ordered and learned arbitrarily (they are non-commutative), C_1 and C_3 have ordered in the same manner for all learners. Hence, while the C-Struct $C_1 \bullet C_2 \bullet C_3$ and $C_2 \bullet C_1 \bullet C_3$ are equivalent $C_2 \bullet C_1 \bullet C_3$ and $C_2 \bullet C_3 \bullet C_1$ would not result in the same state and are not equivalent.

As illustrated in the case of Dueling Proposers, a single *Leader* is only required to guarantee *Liveness*. Safety is guaranteed regardless of the presence of the number of *Leaders*. Hence, Generalized Consensus allows faster operation as multiple *Leaders* can directly send non-commutative commands to *Learners* and only the ordering amongst commutative commands have to be learned. Once, the learners learn a contiguous C-Struct the C-Struct can be applied to the State Machine. EPaxos and M^{2}Paxos discussed below utilize this intuition and are designed to be employed for *State Machine Replication* in Wide Area Networks (WAN).

2.4.1.5 EPaxos

Egalitarian Paxos or EPaxos is a *multi-leader* or *leaderless* Paxos algorithm. Clients can choose to forward the request to any process which in turn acts as the request's *leader*. While, *Paxos* and *Fast-Paxos* try to order commands before committing, EPaxos enables *Proposers* to commit commands as they are proposed but collects dependencies which are used to determine the order while executing and applying commands to the *State Machine*.

Initially, the *command leader* tries to commit the command in one communication delay using a Fast Quorum, $Q = t + \lfloor \frac{t+1}{2} \rfloor$ however, if this fails due to conflicts then the *command leader* takes two additional communication delays by following committing commands with the classic $\lfloor \frac{N}{2} \rfloor + 1$ quorum. After committing, for each command, the dependencies are collected recursively in the form of a directed graph. The commands are then executed in a reverse topological order using their sequence numbers. After that, the processes execute the commands. Processes then execute all the remaining commands in the order of their sequence numbers.

Multi-Leader approaches are especially attractive for Wide Area Networks because (1) single leader approaches cause bottlenecks at the leader (2) a leader failure in a particular region can cause problems with availability while a new leader is chosen.

2.4.1.6 M^{2}Paxos

M^{2}Paxos is also a *multi-leader* CFT Consensus algorithm. However, it takes a different approach as compared to EPaxos. Rather than calculating dependencies, M^{2}Paxos tries to map objects accessed by commands to processes. For instance, in the example given in

Section 2.4.1.4 the value of the key `K` of `Put(K, V)` and `Remove(K)` is the object accessed by the corresponding command. Hence, a process acquires *exclusive ownership* of each key `K` and subsequently orders any commands that access `K`. M^{2}Paxos uses classical quorums of size $\lfloor \frac{N}{2} \rfloor + 1$ which translates to better scalability as compared to using fast quorums as in the case of EPaxos since the *Proposer* has to wait for a lesser number of acknowledgments.

The performance is further enhanced by *workload locality* that is that in real application workloads, objects access local processes with higher frequency. For example, let's assume a Banking System which implements a deposit operation `Deposit(a,` $\langle amount \rangle$`)` and a transfer operation `Transfer(a, b,` $\langle amount \rangle$`)` where a and b are account ids. In this case, it is far more likely that all operations on objects a and b would be localized to a particular region.

Hence, if a local process p_l receives a request with operation `Deposit(a,` $\langle amount \rangle$`)` it assumes the ownership of a by running a *Prepare* phase for the object a similar to the Phase 1 of Paxos and subsequently orders all commands that access these objects providing replication in two communication delays. If a different process receives a `Deposit(a,` $\langle amount \rangle$`)` operation that accesses a then, in this case, the process forwards the request to p_l which already has the ownership of a. However, if another process p_r receives a `Transfer(a, b)` request which accesses both a and b then it can steal the ownership from p_l by running it is own *Prepare* phase with objects a and b. Subsequently, all operations which access a and b will be ordered by p_r.

While M^2Paxos provides the highest performance for a sufficiently localized workload, it can livelock due to *DuelingProposers* as discussed earlier where more than one processes try to acquire ownership of the same objects.

2.4.2 Byzantine Fault-Tolerance

While the protocols discussed above guarantee safety despite any crashes, they don't tolerate Byzantine Faults [5]. Malfunctioning components or active adversaries can cause processes to equivocate and send conflicting messages to their peers. The presence of active adversaries may affect the network by denial-of-service attacks or by actively coordinating with other adversaries. Additionally, these faults can be purely accidental like software errors, disk corruption, network misconfiguration to name a few. These faults are more relevant today than ever due to the growing number of Cryptocurrencies and associated Blockchain protocols [27] where the presence of an adversary is taken into consideration.

2.4.2.1 PBFT

PBFT [21] introduced by Castro and Liskov is the first efficient Byzantine Fault-Tolerant (BFT) algorithm. It is also the most widely studied and formally verified BFT algorithm.

It requires $N = 3t + 1$ processes, and it is hardened by using digital signatures or Message Authentication Codes (MAC) and message broadcasts to verify messages sent by all processes. PBFT operates in a sequence of *views* with a single *Primary* for each *view*. All clients requests are forwarded to the *Primary*. To ensure *liveness*, the *Backup* processes set a timer as soon as they receive a valid request. If the request is not executed by the timer expiration a $view - change$ is executed which on completion results in a new *view* with a new *Leader*.

On receiving signed clients request, the *Primary* sends a `Pre-Prepare` message with the view number and the proposal number to the *Backup* processes. The *Backup* processes verify the digital signature, the message digest, match the view with the current view and if it is valid, send a signed `Prepare` message.

If a *Backup* receives $2t$ `Prepare` messages from different processes that match the view number and proposal number in the `Pre-Prepare` message, then the `Commit` message is broadcasted to all the processes. If a backup receives `Commit` message from $2t + 1$ different processes, then the command is committed, executed, and the result is returned to the client. If the client receives $t+1$ such responses, it can accept the response and consider the request to be successfully replicated.

Broadcast and Quorum Size A set of $2t+1$ messages justifies the validity of the broadcast mechanism. At most t messages could be from faulty processes, the rest of the $t+1$ messages are from non-faulty processes and would contain the same value which would indeed be the correct value.

2.4.2.2 Zyzzyva

Zyzzyva [3] is a BFT consensus protocol that achieves consensus in three communication delays including the client request and reply. PBFT, in contrast, requires five communication steps. The client forwards the request to the *primary* which then appends the sequence number and broadcasts the message to all the *backup* processes. The *backup* processes speculatively execute the request and forward the response to the client. In the absence of any Byzantine Faults or link failures or the $common - case$, the client receives $3t + 1$ responses and commits the request. If however, the client does not receive responses from all the processes the second phase is executed depending on the number of responses received. This second phase behaves similar to $PBFT$ albeit with certain modifications to manage the speculative execution logs produced in the $common - case$.

2.4.3 Cross Fault Tolerance (XFT)

XPaxos [6] was the first protocol to use the Cross Fault Tolerance (XFT) model. The XFT model provides stronger consistency and availability guarantees than both the CFT and

BFT.

Following our discussion on PBFT in Section 2.4.2.1, BFT protocols require a Quorum of size $2t+1$. The $2t+1$ Quorums intersect at a minimum of $t+1$ acceptors hence at least one correct acceptor would be present which ensures the validity of the protocols. This follows from the fact that if t processes are *slow* to respond in case the network is asynchronous or due to the presence of crashes, we require $t+1$ messages that contain the same value to offset t messages received from faulty processes so that a process can discern *correct* messages.

However, if we assume that a majority of processes are *correct* that is they are not crash faulty nor byzantine and can communicate with each other synchronously then the protocol can deliver commands in three communication delays to the client. XPaxos works precisely on finding such $t+1$ groups and if the current group misbehaves a $view-change$ is initiated to discover a new group. Hence, XPaxos works if,

$$t_{nc} + t_c + t_p \leq \left\lfloor \frac{N-1}{2} \right\rfloor$$

where t_{nc} are the number of non crash-faulty or byzantine processes, t_c is the number of crash-faulty processes and t_p is the number of partitioned processes. In any other case the system is considered to be in *anarchy* and a $fault-detection$ mechanism is used to find and possibly correct these faults.

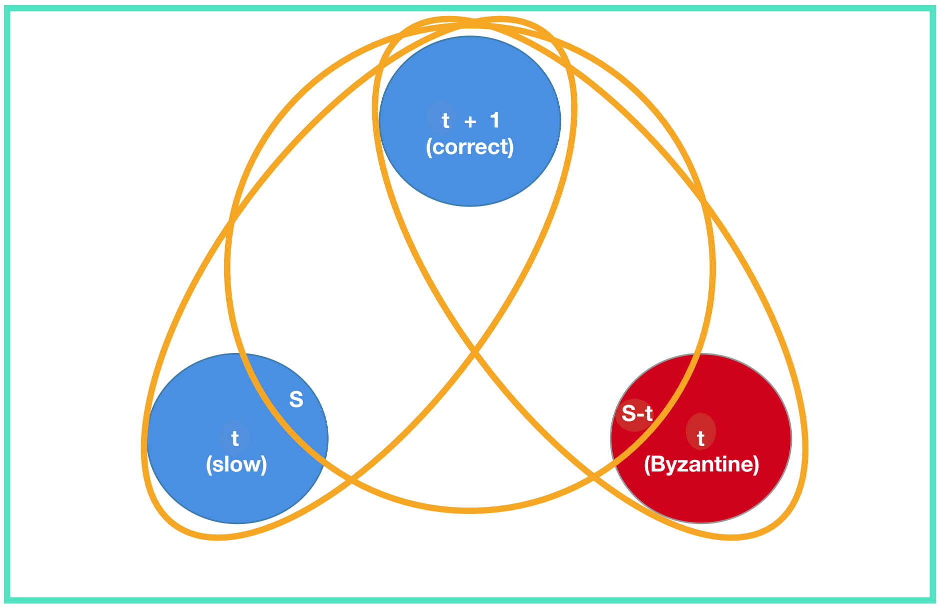

Figure 2.3: BFT vs XFT: The box (in green) represents BFT which requires $3t+1$ acceptors because t acceptors could be slow and cannot be differentiated from t Byzantine acceptors. In contrast, XFT assumptions result in $t+1$ correct processes with a total of t faulty and/or slow processes represented by ovals from which $t+1$ correct messages can be differentiated from.

Chapter 3

System Model and Problem Formulation

This chapter specifies the system assumptions used for designing Elpis, the contribution of this thesis. There exists a set $\Pi = \{p_1, p_2, ..., p_N\}$ of processes that communicate by message passing and do not have access to shared memory. Additionally, there exist clients c which can communicate with any process in the system.

3.1 Cross Fault-Tolerance (XFT) Model

The processes may be faulty, they may fail by crashing (t_c) or be Byzantine (t_{nc}). However, the faulty processes do not break cryptographic hashes, digital signatures, and MACs. A process that is not faulty is *correct*. The network is complete and each pair of processes is connected using a *reliable* point-to-point bidirectional link. The network can be *asynchronous* that is the processes might not be able to receive messages in a timely manner. In this case that the network is *partitioned* and the system model abstracts these partitions as *network faults* (t_p).

Following the XFT model [6], the total number of faults are bounded by,

$$t_{nc} + t_c + t_p \leq \left\lfloor \frac{N-1}{2} \right\rfloor \tag{3.1}$$

where t_{nc} are the number of non crash-faulty or byzantine processes, t_c is the number of crash-faulty processes and t_p is the number of partitioned processes. In any other case the system is considered to be in *anarchy*. For the discussion in this thesis, the system is assumed to be never in *anarchy*, that is, there always exists at least a majority of correct and synchronous processes.

The Generalized Consensus [11] specification is used where the processes aim to reach consensus on a collection of commands, the *C-Struct*. The Consensus algorithm orders non-commutative commands before deciding and decides commutative commands directly. (Ref. 2.4.1.4). Every process can propose commands using the C-PROPOSE interface and the processes decide command structures *C-struct* using the C-DECIDE*(C-struct cs)* interface.

Finally, the identifiers for the objects accessed by the commands are known apriori and is represented with the *LS* attribute in every command. That is, for a command c, the identifiers for its set of objects is given by $c.LS$.

3.2 Thesis Problem Statement

The thesis problem is formulated as follows:

> *How to implement State Machine Replication (SMR) using Generalized Consensus in the Cross Fault-Tolerance (XFT) model?*

The SMR clients invoke commands by sending a request to a process which then uses the C-PROPOSE interface to propose, decides and then applies the $C-Structs$ to the State Machine and generate a reply which is returned to the client. Given that the majority of processes are correct and communicate synchronously (Equation 3.1), the following properties should be guaranteed.

Non-triviality - Commands that are included in the decided *C-structs* must have been proposed;

Stability - if a node decided a *C-struct cs* at any time, then it can only decide $cs \bullet \sigma$, where σ is a sequence of commands, at all later times;

Consistency: Two *C-structs* decided by two different nodes are prefixes of the same *C-struct*.

Liveness: The command c will eventually be decided in some *C-struct*.

In Chapter 4 we illustrate how Elpis achieves State Machine Replication and in Chapter 5 we prove the properties listed above.

Chapter 4

Protocol Description

4.1 Overview

Interestingly, Elpis derives the inspiration of implementing Generalized Consensus from M^{2}Paxos [10] which does not tolerate Byzantine faults. To avoid contention among multiple processes that propose non-commutative (Ref. 2.4.1.4) commands C, a unique *owner* is chosen. This owner orders all commands which interfere with the each other. Once an owner is chosen, other processes forward any command in C to the owner. The interference of commands is interpreted dynamically by the *objects* on which the command operates (Section 2.4.1.6). If a process does not have the ownership of the *objects* accessed by a command, it first tries to acquire the ownership by running the ownership acquisition phase (Section 4.3.4). If the process acquires the *ownership*, it tries to decide the command. The first challenge in designing Elpis is to avoid Byzantine processes from acquiring ownership.

For Byzantine Fault-Tolerant algorithms a quorum of size $2t + 1$ out of $3t + 1$ processes is required. The two Byzantine quorums intersect at $t+1$ processes, one of which is guaranteed to be *correct*. Elpis on the other hand uses $2t + 1$ processes. A set of $2t + 1$ processes would include t faulty processes which is determinant to the *liveness* since these processes may not respond. Hence, a quorum of the same size $2t + 1$ seems implausible. Elpis takes a different approach wherein if a $faulty$ process is detected, clients switch to another proposer after receiving $t + 1$ *Aborts* in the *commit phase* (Section 4.3.3). Since, a majority of processes are correct and communicate synchronously (Section 3.1), when an honest proposer is found this $t + 1$ synchronous set of correct processes form a quorum of size $2t + 1$ with the $t + 1$ processes in the iteration which last aborted. It is easy to see that the client can send requests to a maximum of t faulty proposers and eventually on the $t + 1$ try the request would be committed.

The ownership acquisition can also fail if multiple nodes try to acquire ownership concurrently. In this case the acceptors reply with a *Defer* message which includes the tag of the process it last sent an acknowledgment for in reply to an ownership acquisition message. Upon receiving t+1 *Defer* messages the node starts a coordinated collision-recovery phase by using a tag picked in a predetermined fashion (Section 4.3.2). At the end of the collision-recovery phase processes forward the command to the process picked in the collision-recovery phase and continue to process other commands.

Initially a client sends a digitally signed request to it is *home process* which is the process that is geographically closest to the client and starts a timer.[1] The client waits responses from the processes. Each correct process responds with either a signed *Reply* message or a signed *Abort* message. If the client receives t+1 messages containing the same reply then the client is sure that the request has been replicated. Alternatively, if the client receives t+1 *Abort* messages that implies that the process to which the client sent the request is Byzantine and the client retries with a different process. In the worst case the client tries t+1 processes. However, if the timer expires before receiving t+1 messages the client initiates a phase switch protocol.

During the phase switch protocol processes stop processing any commands that access the object for which the phase switch protocol is initiated and switch to a synchronous group along with a fixed Primary and the nodes run the XPaxos protocol to *Decide* the commands.

This section presented a birds-eye view of Elpis and an insight into its core components. In summary, Elpis consists of three major components

1. A common-case protocol which allows processes to acquire ownership of the objects, decide the commands and return responses to the clients.
2. A collision recovery protocol which resolves the ownership if multiple processes try to acquire the ownership concurrently.
3. A phase switch protocol which includes the clients to transition the processes to a synchronous group with a predetermined primary during period of asynchrony.

4.2 State maintained by a process p_i

Each process p_i maintains the following data structures.

Decided and *LastDecided*. The former is a multidimensional arrays that maps a pair of $\langle l, in\rangle$ to a request where l is the object and in is the consensus instance. $Decided[l][in]$ $= r$ if r has been decided in the consensus instance in (i.e., in position in) of the object l. The latter is a unidimensional array which maps the consensus instance in this

[1]The client is allowed to send the request to any process, however the home process would result in lower latency.

process last observed for an object l. The initial value for *Decided* is NULL while the initial value for *LastDecided* is 0.

Epoch. It is an array that maps an object to an epoch number (a non-negative integer). $Epoch[l] = e$ means that e is the current epoch number that has been observed by p_i for the the object l. The initial values are 0.

Owners. It is an array that maps an object to a node. Owners[l] $= p_j$ means that p_j is the current owner of the object l. The initial values are NULL.

Rnd, *CommitLog*, *StatusLog* These are three multidimensional arrays. The first one maps a pair of $\langle l, in\rangle$ to an epoch number; In particular, $Rnd[l][in] = e$ if e is the highest epoch number in which p_i has participated in the consensus instance in of object l. Therefore, $CommitLog[l][in] = \langle r, e\rangle$ implies that the process received a Quorum of *Commits* for request r and epoch e. The StatusLog maintains the *valid* $\langle r, e\rangle$ that the process is willing to commit on. Hence, $StatusLog[l][in] = \langle r, e\rangle$ implies that p_i would accept a replicate message for r in epoch e and reject others.

Tags An array which maps a process p_i to it is tag. The tag of a process p_i is equal to $Tag[p_i] \in S$ where S is a totally ordered set. The tag is used during Collision Recovery. This mapping has to be predefined by the application layer during setup and is not modified during the protocol execution.

Estimated. It is a multidimensional array which maps the $\langle l, in\rangle$ to the address of the process which this process estimates to be the owner of the object l for an *epoch e*. Hence, $Estimated[l][in] = \langle e, t_{p_e}, p_e\rangle$ implies that for the *epoch e* this process estimates p_e to be the owner where t_{p_e} is the tag of the process.

statusList, *commitList*, *decideList*, *trustList.* These are four multidimensional arrays which are use to store COMMIT, STATUS, DECIDE and TRUST messages respectively. The initial value is $NULL$.

Leader This is a multidimensional array which maps the $\langle l, in\rangle$ pairs to the $\langle e, p_t\rangle$ pairs for which the *Collision recovery* (Section 4.3.6) decided ownership. The value of this array is updated only during the *Collision recovery.* The initial value is $NULL$.

4.3 Detailed Protocol

It is assumed that all processes including the clients possess public keys P_k of all the processes. Each message m includes the digest of the message *D(m)* and a signed message sent by p along with it is digest is represented as $\langle m\rangle_{\sigma_p}$. Unless otherwise stated, each process validates the messages received by first verifying the signatures using the corresponding public key in P_k and then by verifying the message by using a checksum mechanism by comparing it against the message digest. Any message parameter which includes object l as the key can be verified to be for the correct l by matching the objects in *req.LS*. In other words, an object l' cannot exist in the message which does not exist in *req.LS*, otherwise the message is deemed to be invalid.

A client *c* sends a signed request $req = \langle \text{REQUEST}, o,\ t,\ ls,\ c\rangle_{\sigma_c}$ where *o* represents the command to be executed, *t* is the client's timestamp and *ls* contains the objects accessed by the operation *o* and sets a timer. The timer is useful if the client sends a request to a process which has already crashed or is partitioned from other processes.

4.3.1 Coordination Phase

When a request *req* is proposed by process p_i using the C-Propose interface, Elpis coordinates the decision for *req*. In Coordination phase (Algorithm 1), p_i reads the ownership of objects in the system. Depending on the current ownership configuration the process either chooses invoke the *replication phase* (Section4.3.2), forwards the request to the owner process or tries to acquire the ownership for all the objects accessed by the *req* by executing the *ownership acquisition* (Section 4.3.4).

Algorithm 1 Elpis: *Coordination phase* (node p_i).

```
1:  upon C-PROPOSE(Request r)
2:      Set ins ← $\{\langle l, LastDecided[l]+1\rangle : l \in r.ls \wedge \nexists in : Decided[l][in] = c\}$
3:      if $ins = \emptyset$ then
4:          return
5:      if IsOWNER($p_i$, ins) = $\top$ then
6:          Array eps
7:          $\forall\langle l, in\rangle \in ins,\ eps[l][in] \leftarrow Epoch[l]$
8:          $\forall\langle l, in\rangle \in ins\ Estimated[l][in] \leftarrow \langle eps[l][in], Tag[p_i], p_i\rangle$
9:          REPLICATE(req, ins, eps)
10:     else if |GETOWNERS(ins)| = 1 then
11:         send PROPOSE(c) to $p_k \in$ GETOWNERS(ins)
12:         wait(timeout) until $\forall l \in c.LS, \exists in : Decided[l][in] = c$
13:         if $\exists l \in c.LS, \nexists in : Decided[l][in] = c$ then
14:             trigger C-PROPOSE(r) to $p_i$
15:     else
16:         ACQUISITIONPHASE(c)
17:
18: function Bool IsOWNER(Replica $p_i$, Set ins)
19:     for all $\langle l, in\rangle \in ins$ do
20:         if $Owners[l] \neq p_i$ then
21:             return $\bot$
22:     return $\top$
23:
24: function Set GETOWNERS(Set ins)
25:     Set res ← $\emptyset$
26:     for all $\langle l, in\rangle \in ins$ do
27:         $res \leftarrow res \cup \{Owners[l]\}$
28:     return res
```

The process p_i finds the consensus instance it last decided for each object in *LS* and which is not decided for *req*. For every such object, p_i sets *in* equal to *LastDecided[l]+1* and adds it to the *ins* set (line 2). If the process has the ownership of all objects in *req.LS* then the process tries to achieve a *fast decision* by executing the *replication phase* without changing the epoch. If the *replication phase* succeeds, p_i is able to execute the *req* in two communication delays and returns the response to the client.

Alternatively, If p_i detects that p_k has the ownership of all objects in *ins* then it forwards the *req* to the p_k. To avoid blocking in case p_k crashes or is partitioned, p_i also sets a time. Upon expiration of the configurable timer, if the p_i detects that the *req* has not been decided, it takes charge of the *req* and tries to $C - Propose$ the *req* (lines 10-14).

In the last case, if p_i detects no owners for all objects in *ins*, it tries to acquire the ownership by executing the *acquisition phase* (4.3.4) (line 14). A different process, p_k can have the ownership of some subset of objects in *req.LS*, however this process proceeds to *steal ownership* as complete ownership is necessary for setting the correct instance number *ins* for proper *linearization* of commands during execution.

4.3.2 Replication Phase

In the *replication phase*(Algorithm 2), the p_i requests the replication of request *req* for instance *ins* and epochs *eps*. It sends a signed REPLICATE message to all processes in Π. Upon receiving a REPLICATE message the processes check if the received message contains a epoch higher than or equal to the last observed *Rnd*$[l][in]$ for all the objects in the request and checks if p_i is in fact the owner of all the objects. However, if this is not the case then the receiving process sends a *Nack* message along with the information about the last process it sent an *Ack* for the same epoch for one or more $\langle l, in\rangle$ pairs (lines 8-10). This information returned with the *Nack* message is relevant for the *collisionrecovery* and it is discussed in detail in Section 4.3.6. Otherwise, the process starts the *commit phase* (Section 4.3.3) for the request with the received *ins* and *eps* values. The empty parameters represented by $\{,\}$ are used if some request *r'* has to be prioritized over the current request. This scenario is further explained in the *acquisition phase* (Section 4.3.4).

Algorithm 2 Elpis: *Replication phase* (node p_i).

```
1: function Bool REPLICATE(Request r, Set ins, Array eps)
2:     send $\langle \text{REPLICATE}, r, ins, eps\rangle_{\sigma_{p_i}}$ to all $p_k \in \Pi$
3:
4: upon REPLICATE($\langle r$, Set ins, Array eps$\rangle$) from $p_j$
5:     if $\forall\langle l, in\rangle \in ins$, $Rnd[l][in] \le eps[l][in] \wedge \text{ISOWNER}(p_j, ins) = \top$ then
6:         COMMITPHASE($p_j, \{\}\{\}, r, ins, eps$)
7:     else
8:         for all $\langle l, in\rangle \in ins \wedge (Rnd[l][in] > eps[l][in])$ do
9:             Set $deferTo \leftarrow Estimated[l][in]$
10:        send $\langle \text{COMMIT}, -, -, ins, eps, deferTo, NACK\rangle_{\sigma_{p_i}}$ to $p_j$
```

4.3.3 Commit Phase

In the *Commit phase* (Algorithm 3) p_i tries to pick the *req* for instances in *ins* and for the epochs in *eps*. Commit phase can be invoked as part of the the *Acquisition phase* (as discussed later in Section 4.3.4) or as part of the Replicate phase (as discussed earlier in Section 4.3.2). In both of these cases it is possible that the process sending this request is Byzantine and hence possibly equivocate by sending different requests to different processes (which affects $safety$) or send invalid requests req' (which violates integrity).

Algorithm 3 Elpis: *Commit phase* (node p_i).

```
function Void CommitPhase(Replica $p_o$, Array toForce, Request req, Set ins, Array eps)
    Array toDecide
    for all $\langle l, in\rangle \in ins : toForce[l][in] = \langle req', -\rangle : req' \neq NULL$ do
        $toDecide[l][in] \leftarrow req'$
    if $\forall\langle l, in\rangle \in ins, toDecide[l][in] = NULL$ then
        for all $\langle l, in\rangle \in ins$ do
            $toDecide[l][in] \leftarrow req$
    send $\langle$Commit$, p_o, toDecide, ins, eps, -, -\rangle_{\sigma_{p_i}}$ to all $p_k \in \Pi$

upon Commit($\langle$ Replica $p_o$, Array toDecide, Set ins, Array eps, Array $deferTo$,
Value ack$\rangle$) from $p_j$
    if $\forall\langle l, in\rangle \in ins$, $Rnd[l][in] \leq eps[l][in]$ then
        for all $\langle l, in\rangle \in ins$ do
            Set $commitList[l][in][eps[l][in]] \leftarrow commitList[l][in][eps[l][in]]$
                                    $\cup \{\langle toDecide[l][in], p_o, deferTo, ack, j\rangle\}$
        if $\forall\langle l, in\rangle \in ins, |commitList[l][in][eps[l][in]]| \geq sizeof(Quorum)$ then
            if $\exists\langle -, -, deferTo, NACK, -\rangle : commitList[l][in][eps[l][in]]$ then
                $\forall\langle l, in\rangle \in ins$, Set $defers[l][in] \leftarrow deferTo$ :
                                    $\langle -, -, ins, eps, deferTo, NACK, -\rangle$
                trigger Defer$(ins, eps, defers)$
        else if $\forall\langle l, in\rangle \in ins$,
                $\exists\langle r, p_o\rangle \ni \langle r, p_o\rangle = \langle r', p'_o\rangle$ : Valid$(ins, commitList)$ then
            $\forall\langle l, in\rangle \in ins$, $Owners[l] \leftarrow p_o$
            $\forall\langle l, in\rangle \in ins$, $CommitLog[l][in] \leftarrow \langle r, eps[l][in]\rangle$
            $\forall\langle l, in\rangle \in ins$, $Rnd[l][in] \leftarrow eps[l][in]$
            send $\langle$Decide$, p_o, r, ins, eps\rangle_{\sigma_{p_i}}$ to all $p_k \in \Pi$
        else
            send $\langle$Abort$, ins, eps, req\rangle_{\sigma_{p_i}}$ to all $p_k \in \Pi$, $req.c$
    else
        for all $\langle l, in\rangle \in ins \wedge (Rnd[l][in] > eps[l][in])$ do
            Set $deferTo \leftarrow Estimated[l][in]$
        send $\langle$Commit$, -, -, ins, eps, deferTo, NACK\rangle_{\sigma_{p_i}}$ to $p_o$
function Array Valid(Set ins,Set eps, Set Commits)
    Array toCommit
    for all $\langle l, in\rangle \in ins$ do
        Set requests $\leftarrow \langle r', p'_o\rangle : \langle p'_o, r', -, -, -\rangle \in Commits[l][in][eps[l][in]]$
        if $\exists\langle r, p_o\rangle \ni |\langle r, p_o\rangle = \langle r', p'_o\rangle$ : requests$| \geq sizeof(Quorum)$ then
            $toCommit[l][in] \leftarrow \langle r, p_o\rangle$
    return toCommit
```

To avoid this scenario, a *req* is matched with the value in the *StatusLog*. The *StatusLog* is populated by *Status* messages in the *Acquisition phase* or by the *Replicate* messages in the *Replicate Phase*. This *req* for $StatusLog[l][in]$ is considered to be a valid request. Furthermore, any *t committing* processes could also be Byzantine and hence, equivocate and send pick different requests, this phase is executed by all processes. All processes wait for *t+1* messages and if the processes find a request which is common to *t+1* messages and matches the *valid* request this *req* is picked to be *committed*.

After receiving a COMMIT from p_j, for all $\langle l, in \rangle$ pairs the process adds the *req* in toDecide to the Commits set for all. Since, a Byzantine node can equivocate and send an invalid *req* this process has to find a valid *req*. If more than a Quorum of Commits received contain a *req* then we can be sure that the req is infact valid (line 18). If there exists a valid *req* for all $\langle l, in \rangle$ pairs then this phase successfully concludes by sending a Decide (line 22). Otherwise if this process has received a COMMIT from a Quorum of processes then this process Aborts by sending an Abort message to every process including the Client (line 24).

4.3.4 Acquisition Phase

In the *Acquisition phase* (Algorithm 4) the process p_i tries to acquire the ownership of the objects in *req.ls* and also assure that a faulty process is not able to acquire the ownership.

Similar to the Coordination phase, for each object in ls of the request *req* the process p_i finds the consensus instance *LastDecided[l]* it last decided for the object and which is not decided for c and finds the next position by setting *in* equal to *LastDecided[l]+1* and adds it to the *ins* set. Additionally for each pair $(l, in) \in ins$, it increments the current epoch number for l and sets the *Estimated[l][in]* to it is own tag and epoch. After that the p_i sends the PREPARE message to all nodes in Π(lines 1-6).

Upon receiving a $\langle$PREPARE, *ins, eps, req*$\rangle_{\sigma_{p_i}}$ message with a higher epoch for all objects than the last observed then the process sends it is state in the STATUS message to all the processes (lines 8-12). If the epoch of the received message is lower then it sends the *Nack* message with the information about process it send an *Ack* for (lines 14-16). The STATUS message includes the *CommitLog* for the object, instance pairs. Upon receiving STATUS messages from a majority, the process decides if there is a request left to be committed from a concurrently executed or an aborted Commit Phase from an earlier epoch. For this, for all $\langle request, epoch \rangle$ entries present in the *CommitLog* the process first calculates the highest epoch present in the entries and returns the $\langle request, epoch \rangle$ pair present in the entries and stores it in the *epochhighest* set (lines 20-27).

However, the request in this log could be from a Byzantine process. To eliminate such requests the process also calculates a valid $\langle request, epoch \rangle$ pair by checking across the *CommitLogs* of all processes it received a *Status* message from. If a pair is present in more than the number of faulty processes (since the correct processes are in the majority) then this $\langle request, epoch \rangle$ pair is validated. If a $\langle request, epoch \rangle$ is present in the *epochhighest* set and is also present in the validated set then process starts a commit phase with a *toForce* array which contains this $\langle request, epoch \rangle$ pair.

However, if it is not present in either of those sets then the process starts the *Commit Phase* with an empty array. If however, a pair exists in the *quorumhighest* set and is not present in the validated set or vice versa the leader has equivocated and the phase *Aborts* by sending an ABORT message to all processes including the client. Upon receiving $t+1$ such ABORT messages the client tries to $C - Propose$ a request to some other process.

4.3.5 Decision Phase

In the *Decision phase* (Algorithm 5) a process p_i tries to learn a request. Upon receiving a DECIDE message the process stores the message in the *decides* array indexed by the $\langle l, in\rangle$ pair and the *epoch* e. If there exists a request in a Quorum of messages then the process p_i assumes this request to be decided for the object l and instance in (lines 2-6). When a request is decided for all the objects accessed by the request, p_i appends it to its $Cstruct$, executes the request and returns the response to the client as a REPLY message and increments the *LastDecided* for all objects and at this point the protocol is ready for the next instance (lines 7-13).

Algorithm 5 Elpis: *Decision phase* (node p_i).

```
 1: upon DECIDE($\langle Set\ toDecide, Set\ ins, Array\ eps\rangle$) from $p_j$
 2:     $Set\ decideList[l][in][eps[l][in]] \leftarrow decideList[l][in][eps[l][in]] \cup$
                                $\{\langle toDecide[l][in], j\rangle\}$
 3:     if $\forall\langle l, in\rangle \in ins, \exists r \ni |r = r' : \langle r', -\rangle : Decides[l][in][eps[l][in]]|$
                                $\geq sizeof(Quorum)$ then
 4:         for all $\langle l, in\rangle \in ins$ do
 5:             if $Decided[l][in] = NULL$ then
 6:                 $Decided[l][in] \leftarrow r$
 7:
 8: upon $(\exists r : \forall l \in r.LS, \exists in : Decided[l][in] = r \wedge$
                                $in = LastDecided[l] + 1)$
 9:     $Cstructs \leftarrow Cstructs \bullet r$
10:     $Reply\ rep$ = C-DECIDE($Cstructs$)
11:     send REPLY($reply$) to $r.c$
12:     for all $l \in r.LS$ do
13:         $p_i.lastDecided[l]++$
```

4.3.6 Collision Recovery

Collision recovery (Algorithm 6) is an *uncoordinated recovery* protocol which is used to reduce the number of conflicting processes. It follows from how *deferTo* is returned that the $p_k : \langle -, tag, p_k\rangle$ in *deferTo* is a process which is running a phase for the an *epoch* equal to or higher than the $eps[l][in]$ for some $\langle l, in\rangle$ pair this process sent a message for or has the highest *tag* amongst the conflicting. This phase is used to find such a process.

If p_i receives $Defer$ messages, the *collision recovery* is used to conform to the current ownership reconfiguration taking place in the system. Adopting a similar mechanism as in [19] the process tries to PICK a process to defer to which either occurs in a Quorum or otherwise

Algorithm 4 Elpis: *Acquisition Phase* (node p_i).

```
1: function Void AcquisitionPhase($Request\ req$)
2:     $Set\ ins \leftarrow \{\langle l, LastDecided[l]+1\rangle : l \in c.LS \wedge \nexists in : Decided[l][in] = c\}$
3:     $Array\ eps$
4:     $\forall\langle l, in\rangle \in ins,\ eps[l][in] \leftarrow ++Epoch[l]$
5:     $\forall\langle l, in\rangle \in ins,\ Estimated[l][in] \leftarrow \langle eps[l][in], Tag[p_i], p_i\rangle$
6:     send Prepare($\langle ins, eps\rangle$) to all $p_k \in \Pi$
7:
8: upon Prepare($\langle Set\ ins,\ Array\ eps\rangle$) from $p_j$
9:     if $\forall\langle l, in\rangle \in ins, Rnd[l][in] < eps[l][in]$ then
10:        $\forall\langle l, in\rangle \in ins,\ Rnd[l][in] \leftarrow eps[l][in]$
11:        $Set\ decs \leftarrow \{\langle l, in, CommitLog[l][in]\rangle : \langle l, in\rangle \in ins\}$
12:        send Status($\langle ins, eps, decs, -, -\rangle$) to all $p_k \in \Pi$
13:    else
14:        for all $\langle l, in\rangle \in ins,\ Rnd[l][in] \geq eps[l][in]$ do
15:            $Set\ deferTo \leftarrow \langle Rnd[l][in], Tag[p_i]\rangle$
16:        send Status($\langle ins, eps, decs, deferTo, NACK\rangle$) to $p_j$
17:
18: upon Status($\langle Set\ ins,\ Array\ eps,\ Array\ decs,\ Array\ deferTo,\ Value\ ack\rangle$) from $p_j$
19:    for all $\langle l, in\rangle \in ins$ do
20:        $Set\ statusList[l][in][eps[l][in]] \leftarrow statusList[l][in][eps[l][in]] \cup$
                                   $\{\langle decs[l][in], deferTo, ack, j\rangle\}$
21:    if $\forall\langle l, in\rangle \in ins, |statusList[l][in][eps[l][in]]| \geq sizeof(Quorum)$ then
22:        if $\exists\langle -, deferTo, NACK, -\rangle : statusList[l][in][eps[l][in]]$ then
23:            $\forall\langle l, in\rangle \in ins,\ Set\ defers[l][in] \leftarrow deferTo :$
                                   $\langle -, deferTo, NACK, -\rangle$
24:            trigger Defer($ins, eps, defers$)
25:            return
26:    $Set\ epochighest \leftarrow$ Select($ins, statuses$)
27:    $Set\ valids \leftarrow$ Valid($ins, statuses$)
28:    if $epochighest = \emptyset \wedge valids = \emptyset$ then
29:        $\forall\langle l, in\rangle \in ins,\ StatusLog[l][in] \leftarrow \langle req, eps[l][in]\rangle$
30:        if $p_i =$ Proposer then
31:            Replicate($req, ins, eps$)
32:    else if $\exists\ \langle r, e, l, in\rangle \ni \langle r, e, l, in\rangle \in epochhighest$
                   $\wedge\langle r, e, l, in\rangle \in valids$ then
33:        $StatusLog[l][in] \leftarrow \langle r, e\rangle : \langle r, e, l, in\rangle$
34:        if $p_i =$ Proposer then
35:            $Array\ toForce[l][in] \leftarrow \langle r, e\rangle : \langle r, e, l, in\rangle$
36:            Replicate($toForce[l][in], ins, eps$)
37:    else
38:        send Abort($ins, eps, req$) to all $p_k \in \Pi$, $req.c$
39: function Set Select($Set\ ins,\ Set\ statuses$)
40:    $Array\ toForce$
41:    for all $\langle l, in\rangle \in ins$ do
42:        $Epoch\ k \leftarrow max(\{k : \langle -, k\rangle \in decs \wedge \langle decs, -, -, -\rangle \in statuses\})$
43:        $Request\ r \leftarrow r : \langle r, k\rangle \in decs \wedge \langle decs, -, -, -\rangle \in statuses$
44:        $toForce \leftarrow \langle r, k, l, in\rangle$
45:    return $toForce$
46:
47: function Array Valid($Set\ ins,\ Set\ statuses$)
48:    $Array\ valid$
49:    for all $\langle l, in\rangle \in ins,\ statuses[l][in] = \langle decs, -, -, j\rangle : \langle decs, j\rangle$ do
50:        $Set\ requests[j] \leftarrow \langle v, k\rangle \in decs : \langle decs, j\rangle$
51:        if $\exists\langle r, k\rangle \ni |\langle r, k\rangle : requests| \geq t$ then
52:            $valid \leftarrow \langle r, k, l, in\rangle$
53:    return $valid$
```

Algorithm 6 Elpis: *Collision Recovery* (node p_i).

```
 1: function DEFER($\langle Set\ ins,\ Array\ eps,\ Set\ defers\ \rangle$)
 2:     $Array\ deferTo \leftarrow$ PICK$(ins, defers)$
 3:     RECOVERY$(ins, deferTo)$
 4:     $\forall \langle l, in \rangle \in ins,\ Owners[l] \leftarrow Leader[l][in]$
 5:     trigger C-PROPOSE$(r)$ to $p_i$
 6:
 7: function Void RECOVERY($\langle Set\ ins\ ,Array\ eps,\ Array\ deferTo \rangle$)
 8:     if $\forall \langle l, in \rangle \in ins,\ \exists \langle e, p_l \rangle \ni Leader[l][in] : \langle e, p_l \rangle : e \geq eps[l][in]$ then
 9:         return
10:     else
11:         send $\langle$TRUST$, ins, deferTo\rangle_{\sigma_{p_i}}$ to all $p_k \in \Pi$
12:
13: function Array PICK($Set\ ins,\ Set\ defers$)
14:     $Array\ deferTo$
15:     for all $\langle l, in \rangle \in ins$ do
16:         COUNT$(\langle e, t_{p_l}, p_l \rangle) = |\langle e, t_{p_l}, p_l \rangle = \langle e', t'_{p_l}, p'_l \rangle : defer[l][in]|$
17:         if $(\exists \langle e, t_{p_l}, p_l \rangle \ni$ COUNT$(\langle e, t_{p_l}, p_l \rangle) \geq sizeof(Quorum)) \vee$
              $(\exists \langle e, t_{p_l}, p_l \rangle \ni$ COUNT$(\langle e, t_{p_l}, p_l \rangle) =$
                              $\mathbf{max}(\{$COUNT$(\langle e', t'_{p_l}, p'_l \rangle) : defer[l][in]\})) \vee$
              $(\langle e, t_{p_l}, p_l \rangle : t_{p_l} = \mathbf{max}\ (t'_{p_l} : \langle -, t'_{p_l}, - \rangle : defers[l][in])$ then
18:             $deferTo[l][in] \leftarrow \langle e, t_{p_l}, p_l \rangle$
19:     return $deferTo$
20:
21: upon TRUST($\langle Set\ ins,\ Array\ deferTo \rangle$) from $p_j$
22:     if $isHigher(ins, deferTo)$ then
23:         for all $\langle l, in \rangle \in ins$, do
24:             $Estimated[l][in] \leftarrow deferTo[l][in]$
25:             $trustList[l][in] \leftarrow trustList[l][in] \cup$
                                      $\{\langle e, p_r \rangle : \langle e, -, p_r \rangle : deferTo[l][in], j\}$
26:             if $\exists \langle e, p_o \rangle \ni |\langle e, p_o, - \rangle : trustList[l][in]|$
                                      $\geq sizeof(Quorum)$ then
27:                 $Leader[l][in] \leftarrow \langle e, p_o \rangle$
28:         send $\langle$TRUST$, ins, deferTo\rangle_{\sigma_{p_i}}$ to all $p_k \in \Pi$
29:     else
30:         $\forall \langle l, in \rangle \in ins,\ Set\ estimate \leftarrow Estimated[l][in]$
31:         send $\langle$DOUBT$, ins, estimate\rangle_{\sigma_{p_i}}$ to $p_i$
32:     if $\forall \langle l, in \rangle \in ins,\ Leader[l][in] \neq NULL$ then
33:         return
34:
35: upon DOUBT($\langle Set\ ins,\ Array\ estimate \rangle$) from $p_j$
36:     $\forall \langle l, in \rangle \in ins,\ Estimated[l][in] \leftarrow deferTo[l][in]$
37:
38: function Bool ISHIGHER($Set\ ins,\ Set\ Received$)
39:     for all $\langle l, in \rangle \in ins, Estimated[l][in] = \langle e, t_{p_e}, - \rangle : \langle e, t_{p_e} \rangle,$
                          $Received[l][in] = \langle e', t_{p_r}, - \rangle : \langle e', t_{p_r} \rangle$ do
40:         if $e > e' \vee (e = e' \wedge t_{p_e} > t_{p_r})$ then
41:             return $\bot$
42:     return $\top$
```

a majority of *Nack* messages. After the completion of the Collision recovery the process proposes the request using $C - propose$ with the new learned configuration (lines 13-16). If the p_i has the ownership of all objects in *ins* then the p_i can start the *replicate phase* (4.3.2) for the request. If the phase succeeds then the request has been successfully decided and the client will receive the reply. If however this phase fails then p_i starts the *Collision Recovery* (4.3.6) and retries the request *req* from Step 1 (lines 5-9).

Before starting this phase we check if an instance of Collision Recovery has already been completed by the system. In this case, no additional run is required and we conclude the recovery (line 2-3). However, if no such instance has been completed then we start this recovery by sending a $\langle$TRUST, *ins, eps, leader*$\rangle_{\sigma_{p_i}}$ message to all the nodes (line 5). Upon receiving a TRUST message the process compares the current estimated leader value to the received value. The values are ordered by using their *epochs* first and then by their tags. That is, a value is *Higher* if it has a higher epoch. If the epochs are equal then the node tags are used to break the symmetry (lines 8, 24-28). Therefore, if the received value is *Higher*, then the process sets this as the new estimated value, stores the value in its*trustList* and forwards the TRUST message to all the processes with the received value. If the value is *lower* however, the process sends a DOUBT message with the higher value (line 9-14).

Upon receiving a DOUBT message the process sets its *Estimated* to the value received in the defer message (line 22). Henceforth, for this instance of the *recovery* this process will act in response to *Trust* messages from other processes. When the cardinality of $statusList[l][in]$ equals the Quorum for some $\langle l, in\rangle$ pair then the process $p_l : trustList[l][in]$ is trusted to be the owner of this $\langle l, in\rangle$ pair and when there is a trusted owner for all $\langle l, in\rangle \in ins$ then the recovery concludes.

4.4 Elpis vs. XPaxos and M^{2}Paxos

This section gives a comparison of Elpis with XPaxos and M^{2}Paxos . Following the discussion of Generalized Consensus in 2.4.1.4, the processes have to learn the ordering between non-commutative commands before the *c-struct* is applied to the state machine. Algorithms like EPaxos decide this ordering at the time of execution. Therefore, processes exchange dependencies and create a directed graph and resolve any cycles in a predetermined fashion. However, this dependency exchange loads the network due to the increase in the size of messages exchanged and loads the CPUs at processes for resolving the cycles contributing to reduced performance.

Elpis takes an alternate approach by deriving inspiration from the Generalized Consensus of M^{2}Paxos. M^{2}Paxos manages dependencies by mapping an object o to a process p_o. Consequently, p_o orders all the commands which touch o and once a complete order is found the processes execute all the commands. Agreeing on p_o ownership of o is a consensus problem in itself and M^{2}Paxos leverages Paxos. We treat M^{2}Paxos like an extended specification of Generalized Consensus and inherit a portion of data structures and interfaces. However, M^{2}Paxos is designed in the Crash Fault Tolerant (CFT) model and does not tolerate any non-crash faults.

Let us illustrate with an example. As discussed in Section 2.4.1.2 Paxos does not guarantee liveness when multiple processes try to commit commands concurrently. Since M^{2}Paxos uses Paxos for the ownership acquisition, we illustrate the example discussed in Section 2.4.1.2

in the context of M^{2}Paxos . Let us consider, two Proposers p_i and p_j which are trying to acquire the ownership of an object l. Consequently, they send a prepare for $\langle l, e_i \rangle$ and $\langle l, e_j \rangle$ respectively where l is object id and e is the epoch such that $e_j > e_i$. Lets assume a majority of Acceptors send acknowledgments for e_j. As a result, p_i is now ready to send an Accept for $\langle l, e_i \rangle$. Meanwhile, p_j completes its Prepare phase as well. At this point, the highest proposal number that a majority of Acceptors has seen is e_i. Now Acceptors will reject the Accept phase for p_i because its proposal number $e_i < e_j$. The p_i would retry Prepare with a higher proposal number $e_k > e_j$. Hence, p_j would not be able to complete the Accept phase. It is straightforward to see that this can go on forever. Hence, the ownership acquisition can live-lock. A Byzantine process can contribute to this problem by continuously sending conflicting requests preventing any progress. Additionally, the Byzantine faulty process can acquire ownership acquisition by proposing a very high epoch and violate safety by sending incorrect Accept requests. Elpis deals with this problem by curbing the conflicts in Section 4.3.6 and by preventing a Byzantine process from acquiring the ownership in Section 4.3.4.

Elpis uses the Cross Fault Tolerance (XFT), the same system model as XPaxos but the leaderless protocol of Elpis with all $2t + 1$ active processes differs from XPaxos which uses fixed synchronous groups (sg) of size $t+1$ with a fixed leader. XPaxos works by determining $\binom{n}{t+1}$ sg groups with active groups switching via a view-change mechanism in case of faults until a sg with correct processes is not found. For higher n and t, the number of such groups is exponential. However, in the worst case of Elpis, a client has to contact a maximum of $t + 1$ processes.

Chapter 5

Correctness

In this chapter, we present the correctness arguments for Elpis.

Only the owner of an object l in epoch e successfully commits the requests, and thus increments in. We prove in this section that a Byzantine process does not acquire the ownership. Since, the correct processes initially start with the same value of $LastDecided[l]$ and only increment it when a command is decided for $\langle l, in \rangle$ we can see that the valid requests proposed by a correct owner of l in e will follow a complete order for in throughout the execution of the protocol and would not diverge for any correct process. Thus, in the rest of the section we refer to $StatusLog[l][in]$ and $CommitLog[l][in]$ as $StatusLog$ and $CommitLog$ for brevity which denote the value of the logs for some $\langle l, in \rangle$. The proofs can be generalized for any instance in of the object for which the process acquires ownership of the object l.

Validity: *Only a proposed command c is decided.*

A process only appends a command c to the $C-Struct$ if it receives $Commit$ messages from a majority of processes for c and no other command can exist. Correct processes only send $Commit$ messages for the value c if they receive c in the $Replicate$ message.

Agreement: *Every correct process agree on the same c.*

Lets assume some process p_i decides a command c for some $\langle l, in \rangle$ and epoch e. This must imply that this process received Decide messages from $\geq \frac{N}{2}$ processes with the command c and $\langle l, in \rangle$ and epoch e. Hence, there must be a set X of size $\frac{N}{2} > t$ which received $\geq \frac{N}{2}$ Commit messages for the command c for $\langle l, in \rangle$ and epoch e. All processes in X set $CommitLog = \langle c, e \rangle$. The state of at least one correct process is contained in the quorum and because all processes in X include $\langle c, e \rangle$, the $StatusLog = \langle c, e \rangle$ in the next epoch.

We argue that if a correct process in X commits request c' in the epoch e', and $StatusLog = \langle c, e \rangle$ then for $e' : \langle c, e' \rangle = StatusLog \wedge e' \geq e$, $c' = c$.

We prove this by induction on the epoch e'. For the base case, lets suppose $StatusLog =$

$\langle c, e' \rangle$ at some correct process p_i. If p_i commits c in e' then it must receive a *Replicate* request for c. Otherwise if it receives a request for $c' \neq c$ it would detect that the process contending for ownership has equivocated and *Abort*. Hence, $c' = c$. For e', p_i commits on c', sets $CommitLog[l][in] = \langle c', e' \rangle$ sets the process which sends c' as the owner (which is in fact correct).

Lets suppose that for any epochs in between e' and e, $StatusLog = \langle c', - \rangle$. We have to prove that if $StatusLog = \langle c, e+1 \rangle$ then $c = c'$. The $StatusLog = \langle c, e+1 \rangle$ consists of valid *CommitLogs* for c in e. Since, the $StatusLog[l][in] = \langle c', e \rangle$ any correct process that commits c and sets its *CommitLog* to $\langle c, e \rangle$ can only do that if c and c' are equal. Hence, c $= c'$. By induction we can say that this is true for all $e' \geq e$.

We use this argument to prove *agreement*: If two correct processes commit c and c' then $c = c'$.

If a correct process initially commits c in e, then $StatusLog = \langle c, e \rangle$. If another correct process commit c' in e', then we know that for any $e > e'$, $c = c'$. Hence, the correct processes must agree.

Liveness: *Every correct process eventually decides some value.*

Under the assumptions of the XFT model, there always exists at least a majority of processes that are correct and synchronous, and thus can decide on the order of commands. We see that in the case of a malicious leader, every correct process detects equivocation and sends *Abort* messages to the client. After receiving $t+1$ messages the client switches to a new process. If the process is Byzantine it would again receive the *Abort* messages or timeout. This can only happen a maximum of t times. Finally, when a correct owner exists for l in epoch e, it sends a Replicate message with c where $StatusLog = \langle c, e \rangle$. Consequently, all processes start the Commit phase with c and send Commit messages to each other with this value c. Since, a majority of processes are correct, every process will receive $\geq \frac{N}{2}$ such messages, *Decide* and terminate.

Once the leader election is triggered, nodes use the generic PREPARE phase to acquire ownership. If a command c has been proposed by a correct node p_i, eventually, if there is no other concurrent and conflicting command with c in the system, p_i succeeds the execution of all the phases of the protocol for c, since no other node attempts to become the owner of any of the objects in *c.LS*, and there always exists a quorum of nodes that acknowledge for messages. However, when there exists conflicts and consequently *collision recovery* is triggered, convergence happens eventually but the chances increases at each step.

To show the eventual convergence, notice that every correct process sends a TRUST message whenever it trusts a leader; it sends a DOUBT message otherwise to transfer its view with higher epoch. Thus, eventually all the nodes will only trust the node with the highest epoch, and hence send TRUST message for that epoch. Hence, there exists a time when the *collision recovery* has caused every correct process to trust the same process $\bar{p}_i$ forever. Consequently, no correct process sends any further TRUST or DOUBT messages (lines 18–19 in Algorithm 6).

Chapter 6

Evaluation

We evaluate Elpis by comparing it against four other consensus algorithms: XFT, PBFT, Zyzzyva and M^{2}Paxos . We evaluate the latencies in a Geo-Replicated setup by setting up five processes on the Amazon EC2 setup and throughput by placing the processes in a single placement group (`us-east-1`) so as to not skew the data due to a greater variance in latencies in case of a Geo-Replicated setup.

We implemented Elpis, XFT and M^{2}Paxos using the reliable messages toolkit Jgroups [28], in Java 8. We implemented Zyzzyva using the BFT-SMaRt library (also in Java) [28] and used the default BFT-SMaRt protocol as an approximation of the PBFT protocol as it is an implementation of the PBFT protocol while providing comparable performance. Unless otherwise stated, each node is a c3.4xlarge instance (Intel Xeon 2.8GHz, 16 cores, 30GB RAM) running Ubuntu 16.04 LTS - Xenial (HVM).

6.1 Experimental Setup

For the Geo-replicated processes are placed in regions as shown in Table 6.1. The inter-region latencies are shown in Table 6.2. For PBFT, Zyzzyva and XFT the primary is place at Frankfurt. Additionally, the initial synchronous group for XFT consists of {Frankfurt, Ireland, Ohio} and {Frankfurt, Ireland, Ohio, Virginia} in line with the latencies recorded in Table 6.2. For processes in the single placement group of Virginia the latency for communicating with other processes in the group was observed to be close to 2 ms.

To properly load the system, we injected commands into an open-loop using client threads placed at each node. Commands are accompanied by a 16-byte payload. However, to not overload the system we limit the number of in flight messages by introducing a sleep time where every client sleeps for a predetermined duration after proposing a request. This is tuned so as to get the best possible performance for the setup. We do not show performance

AWS Region	Name
`us-east-1`	Virginia
`us-east-2`	Ohio
`eu-central-1`	Frankfurt
`eu-west-1`	Ireland
`ap-south-1`	Mumbai
`us-west-1`	California
`sa-east-1`	Sao Paulo

Table 6.1: AWS regions mapped to region names.

in case of faults for Elpis as since there are multiple processes proposing the requests the performance will be similar to that of a different process taking command of ordering the requests for a client which was initially proposing to a failed process.

6.2 Latency

Figure 6.1 shows the comparison of latencies in a Geo-Replicated setup with five processes for clients proposing requests where the requests have 100% locality which implies that the requests in different regions access different objects. We notice that the CFT M^{2}Paxos achieves the lowest latencies for all regions which is expected due to a lower quorum size. Elpis achieves latencies close albeit slightly higher than M^{2}Paxos which is expected due to the additional overhead of sending signed messages which include the digest of the message. This overhead is inherent to all other protocols including XFT. However, except for Frankfurt we notice that the latencies are higher for clients placed in other regions. This is due to the fact that every client present in the other regions has requests forwarded to Frankfurt resulting in higher response time.

Region	Virginia	Ohio	Frankfurt	Ireland	Mumbai	California	Sao Paulo
Virginia	-	10.792	88.331	74.572	182.368	59.5	140
Ohio	10.8	-	97.321	84.645	191.224	49	149
FrankFurt	88	97.407	-	21	109	145	226
Ireland	74.6	84.6	39	-	122	129	183
Mumbai	182.369	191	109	120	-	241	320
California	59.5	49	145	129	241	-	197
Sao Paulo	140	149	226	183	320	197	-

Table 6.2: Inter region average latency (ms)

In contrast the primary/owner for every client in case of Elpis is present in the same region as the client which provides lower response time. PBFT and Zyzzyva incur similar problems in

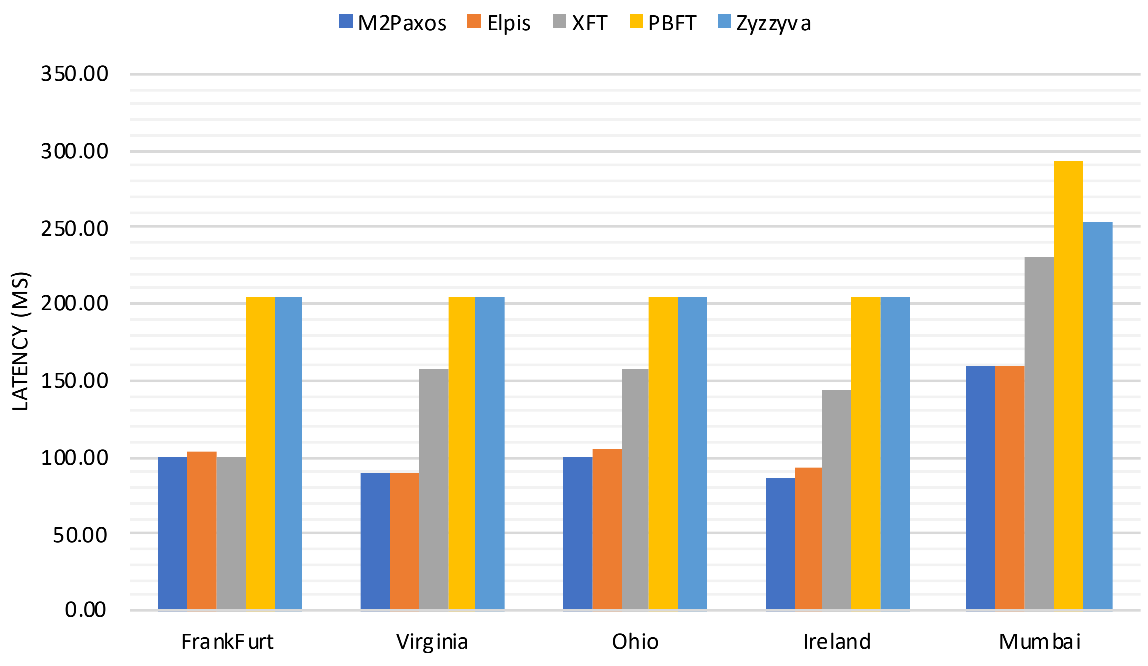

Figure 6.1: Latency comparison for 5 processes in Geo-replication

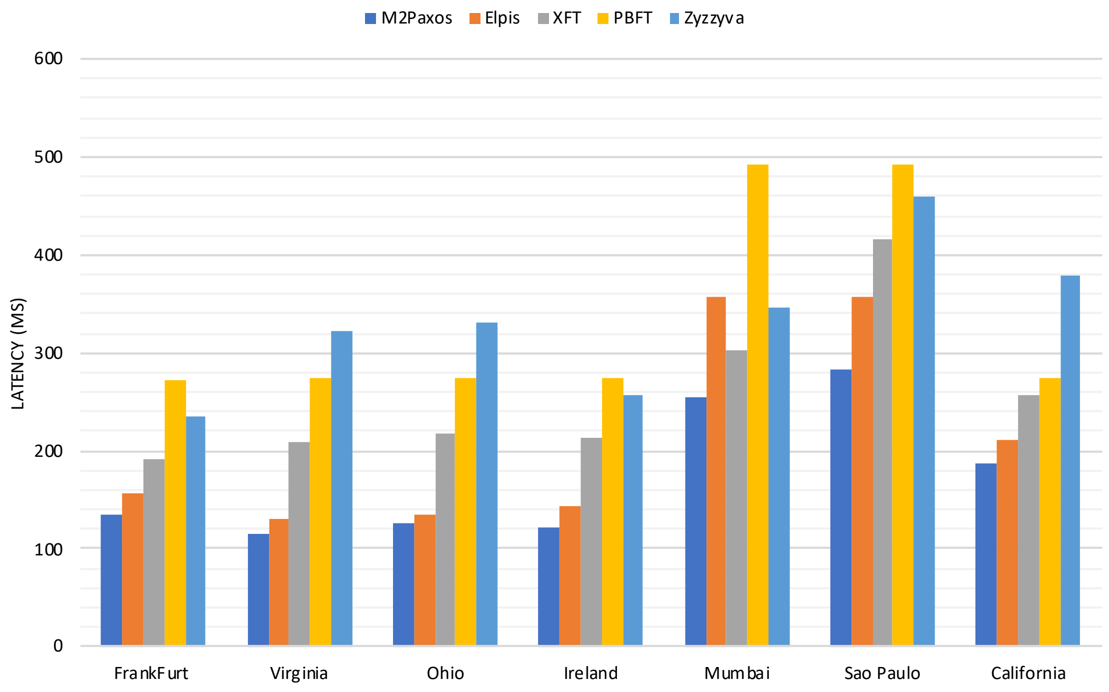

Figure 6.2: Latency comparison for 7 processes in Geo-replication. t=3 for Elpis and M^{2}Paxos and t=2 for rest.

addition to the fact that the quorum size is bigger. Hence, at for each communication delay the primary has to wait for a greater number of messages thus incurring longer response times which is reflected in the chart.

6.3 Throughput

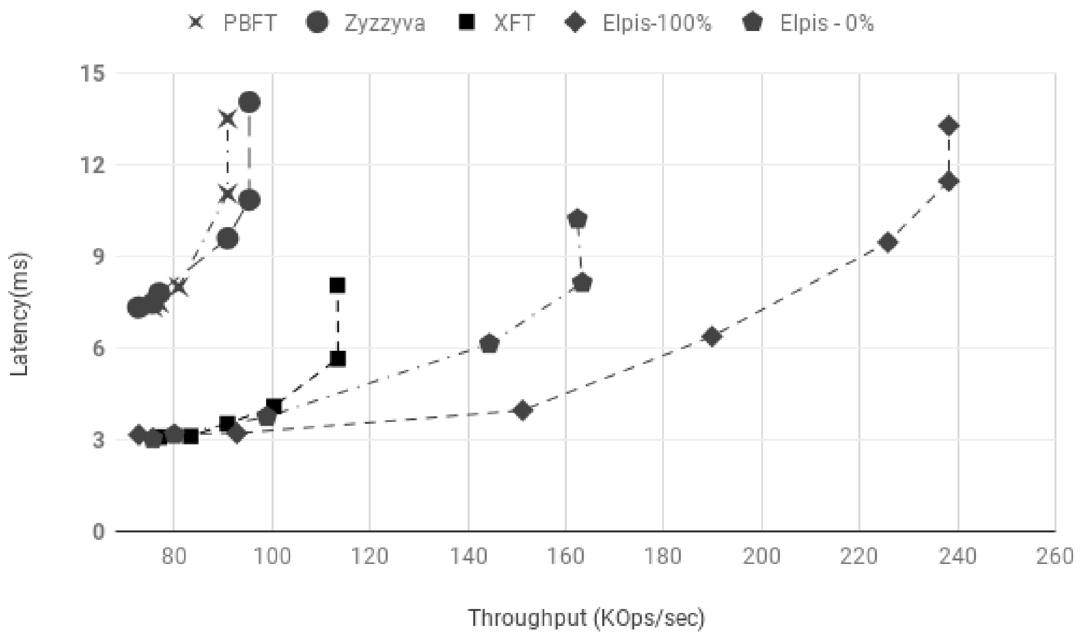

Figure 6.3: Latency vs Throughput in a cluster

Figure 6.3 shows the throughput comparison in a single placement group as the system is pushed closer to saturation to achieve the maximum throughput possible. PBFT and Zyzzyva peak at under 1x10^5 operations/sec due to complicated message patterns resulting in higher bandwidth usage. XFT and Elpis perform significantly better as they try to replicate requests to lower number of followers. However, since Elpis relies on multiple owners the inherent load-balancing in the protocol results in higher throughput as compared to XFT where leader becomes a bottleneck.

6.4 Conflicts

Figure 6.4 shows the throughput vs latency comparison as the number of conflicts are increased. The throughputs are reduced when the percentage of conflicts arise. In case of a conflict, the collision recovery is initiated which results in more messages exchanged hence burdening the network until a single leader exists for each object for the conflicting commands. Progressively, lesser requests are concurrently completed which is reflective of the workload as the commands which touch the same objects cannot be parallelized. However, even in the case of 100% conflicts the throughput is not reduced below about 1.6x10^5 operations/sec which is still higher than the protocols in comparison as shown in 6.3 by Elpis-100%.

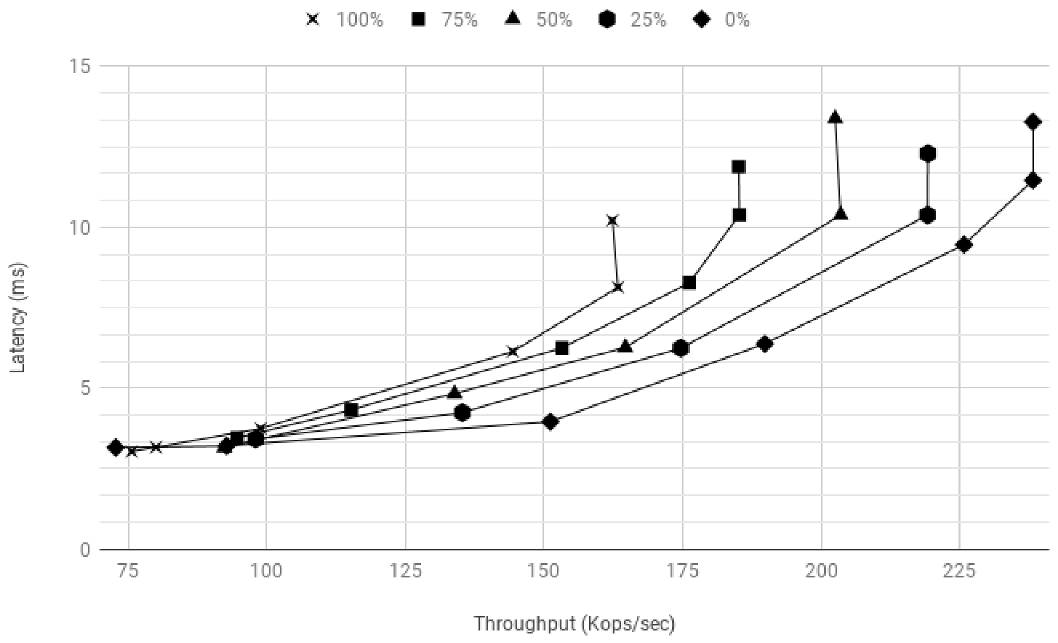

Figure 6.4: Latency vs Throughput as the percentage of conflicting commands is varied.

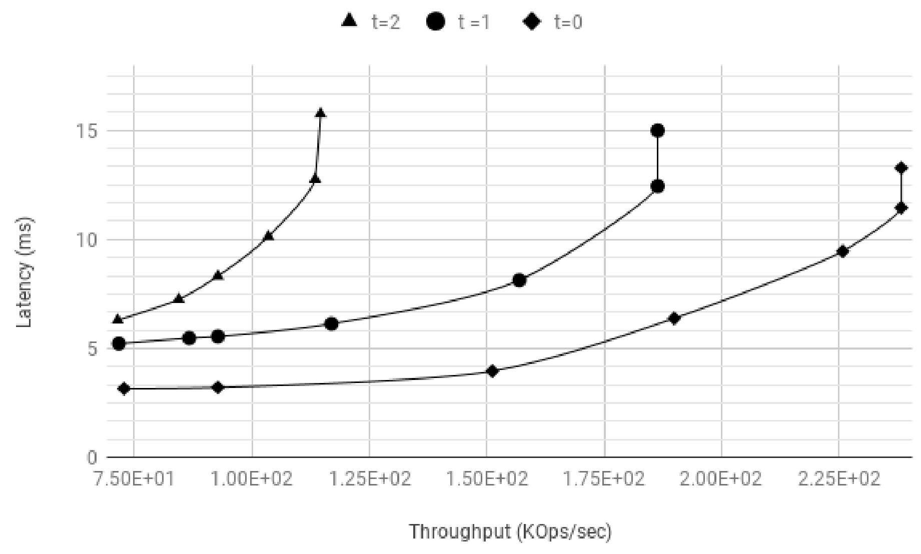

Figure 6.5: Latency vs Throughput as the percentage of conflicting commands is varied.

6.5 Faults

Figure 6.5 shows the throughput vs latency comparison in the presence of faults (t). A Byzantine faulty process is simulated by adding a Byzantine layer beneath the protocol implementation which changes the messages arbitrarily. As shows in the figure, the latency is increased and the throughput decreases considerably when the faults are increased. When a Byzantine process is detected, the clients forward their requests to other processes. This mechanism increases the latency. The throughputs are decreased because the faulty processes can no longer successfully commit any commands.

Chapter 7

Conclusion and Future Work

This thesis presented the first leaderless XFT protocol that overcomes the drawbacks of XPaxos single-leader based protocol. We showed with evaluation the significant performance gains that can be achieved over XPaxos and also other BFT protocols.

7.1 Conclusion

State Machine Replication (SMR) is a technique widely employed to support hundreds of millions of users while providing high availability. SMR is achieved by using Consensus algorithms provide consistency and total order following a set of fault assumptions specified by a fault model. In Chapter 2, we provide a background on several Crash Fault-Tolerant algorithms such as Paxos and Raft which are one of the most popularly implemented Consensus algorithms, and Generalized Paxos, Fast Paxos, EPaxos and M^2Paxos which strive to improve performance in practice. We also introduce the Byzantine Fault-Tolerant model which covers any non-crash faults due to which node behave arbitrarily. This could be accidental or the result of adversarial behavior. To this end, we briefly discuss PBFT and Zyzzyva. However, these algorithms require extra resources (namely 3t+1 processes) to tolerate the same number of faults (t) as the Crash Fault-Tolerant model. To this end, we review the Cross Fault Tolerance (XFT) model and XPaxos which allows Byzantine node behavior but assumes that there exists a correct majority of nodes that observe synchronous behavior. This assumption is especially suitable for geo-replicated systems where coordinated attacks on multiple servers and network are highly unlikely.

In the rest of the thesis, we present Elpis which unlike XPaxos implements generalized consensus. Elpis assigns different and independent objects to different nodes, such that the need for ordering is limited to local scope, each governed by one of the nodes, and transfers ownership when needed. This way operations on disjoint collections of objects trivially commute, and Elpis can decide on such commands in just two message delays while ownership transfer adds one additional message delay. In Chapter 6, we evaluate the implementation in the wide area setting and demonstrate the throughput and latency improvements over existing BFT algorithms. For the geo-replicated setting, Elpis achieves low latency for clients due to the ownership of objects accessed by the clients at the local process and high throughput due to the leaderless approach providing inherent load balancing.

Elpis is an attractive option for building geo-replicated fault-tolerant systems as not only does it provides better performance than BFT algorithms but, it also offers higher reliability than CFT algorithms while requiring no extra resources.

7.2 Key Insights

The leader based approach to designing consensus protocols is relatively easy to develop, understand and implement. All processes forward client requests to a single leader process which commits in one communication delay and sends the responses to the client. These protocols orchestrate operation in increasing order of views with a view-change mechanism required when the current leader fails. In the non-faulty execution, the extra processing load on the leader thwarts the performance. When the processes are faulty, and even in the presence of perfect failure detectors, the view-change mechanism is not transparent. The throughput is reduced to zero as no commands can be committed until the view-change finishes. On the other hand, the leaderless approach of generalized consensus is harder to design and implement and is only limited by the technique used to handle dependencies to order the non-commutative commands. But, the performance benefits are tremendous in both faulty and non-faulty cases due to better load distribution and reduced latencies, particularly visible in the geo-replicated setting. Dependencies are tracked using graphs [20, 29] where any cycles are broken using a pre-proposed mechanism. However, this results in increased message size requiring higher network bandwidth and increased CPU loads to break these cycles. On the other hand, algorithms like M^{2}Paxos [10] and Elpis prevent this scenario by binding dependent commands to processes and hence, provide straightforward ordering.

Byzantine Fault-Tolerance (BFT) is too expensive for the geo-replicated setting due to its complex message patterns which burden the network bandwidth. Additionally, BFT considers the kind of coordinated attacks on data centers which are unlikely as they require considerable adversarial resources. This realization has prompted research on alternate Byzantine models. While the Byzantine processes affect safety, the network faults jeopardize the liveness. Hybrid Fault Models use precisely this observation and place thresholds on these individual faults to guarantee the corresponding properties. A prominent member of the Hybrid Fault Model is Visigoth Fault Tolerance (VFT) [30]. However, the VFT model requires one to place guarantees on the network faults irrespective of the Byzantine faults. On the other hand, the XFT model needs that the total number of failures to be less than the majority.

7.3 Future Work

The immense popularity of Blockchains has reinvigorated the interest in researching protocols that use the Byzantine Fault-Tolerant model. Permissionless protocols like Bitcoin have made evident the performance limitations of their approach. The traditional BFT models which form the basis for most Permissioned Blockchains have a chance to be looked at through a new prism and to be tested in real-world deployments. To that end, this thesis holds immense potential for future work as it pertains to achieving higher performance by using alternate assumptions about the BFT adversarial model.

The Crash Fault Tolerant (CFT) is well studied and battle-tested. Distributed coordination services like Yahoo! Zookeeper use Zab [31] (a Paxos variant) for atomic broadcast. A plethora of services implements the Raft consensus algorithm [22]. While Byzantine Fault Tolerant (BFT) models have classically seen low adoption except for absolutely safety-critical systems like Aircraft information systems they are still well prototyped and studied in practical scenarios. The BFT-Smart library is immensely popular amongst researchers, and several studies report promising performance with throughputs up to 80,000 ops/sec. It remains to be seen how the systems can be parameterized to make the XFT assumptions tenable in different environments. For instance, the XFT assumptions of a weaker adversary are open to being explored in existing environments which have external security measures, such as firewalls and reverse proxies where coordinated attacks by adversaries are unlikely.

We present the Generalized Consensus which offers the highest performance when the workloads have 100% locality. While locality in database workloads has been quite well explored, analyzing the locality in Blockchain workloads could be an interesting avenue of research. Tools like Blocksci [32] which provide a means to analyze Blockchain data can be a good starting point.

Bibliography

[1] Stephen Elliot. DevOps and the Cost of Downtime: Fortune 1000 Best Practice Metrics Quantified. page 13.

[2] Nicola Dragoni, Ivan Lanese, Stephan Thordal Larsen, Manuel Mazzara, Ruslan Mustafin, and Larisa Safina. Microservices: How To Make Your Application Scale. In *Perspectives of System Informatics*, Lecture Notes in Computer Science, pages 95–104. Springer, Cham, June 2017.

[3] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: speculative byzantine fault tolerance. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 45–58. ACM, 2007.

[4] Pierre-Louis Aublin, Rachid Guerraoui, Nikola Knežević, Vivien Quéma, and Marko Vukolić. The Next 700 BFT Protocols. *ACM Trans. Comput. Syst.*, 32(4):12:1–12:45, January 2015.

[5] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[6] Shengyun Liu, Paolo Viotti, Christian Cachin, Vivien Quéma, and Marko Vukolic. XFT: Practical Fault Tolerance beyond Crashes. In *OSDI*, pages 485–500, 2016.

[7] Diego Ongaro and John K. Ousterhout. In search of an understandable consensus algorithm. In *USENIX Annual Technical Conference*, pages 305–319, 2014.

[8] Iulian Moraru, David G. Andersen, and Michael Kaminsky. There is more consensus in Egalitarian parliaments. pages 358–372. ACM Press, 2013.

[9] Balaji Arun, Sebastiano Peluso, Roberto Palmieri, Giuliano Losa, and Binoy Ravindran. Speeding up Consensus by Chasing Fast Decisions. *arXiv:1704.03319 [cs]*, April 2017. arXiv: 1704.03319.

[10] S. Peluso, A. Turcu, R. Palmieri, G. Losa, and B. Ravindran. Making Fast Consensus Generally Faster. In *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 156–167, June 2016.

[11] Leslie Lamport. Generalized Consensus and Paxos. page 63.

[12] Redis.

[13] Open Source Search & Analytics · Elasticsearch.

[14] Christian Cachin, Rachid Guerraoui, and Luis Rodrigues. *Introduction to Reliable and Secure Distributed Programming (2. ed.).* January 2011.

[15] J FISCHER and A LYNCH. Impossibility of Distributed Consensuswith One Faulty Process. page 9.

[16] Leslie Lamport. The Part-time Parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.

[17] Brian M. Oki and Barbara H. Liskov. Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, PODC '88, pages 8–17, New York, NY, USA, 1988. ACM.

[18] Jim Gray. Notes on Data Base Operating Systems. In *Operating Systems, An Advanced Course*, pages 393–481, London, UK, UK, 1978. Springer-Verlag.

[19] Leslie Lamport. Fast Paxos. *Distributed Computing*, 19, October 2006.

[20] Iulian Moraru, David G. Andersen, and Michael Kaminsky. Egalitarian paxos. In *ACM Symposium on Operating Systems Principles*, 2012.

[21] Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance. In *OSDI*, volume 99, pages 173–186, 1999.

[22] etcd: Distributed reliable key-value store for the most critical data of a distributed system, July 2018. original-date: 2013-07-06T21:57:21Z.

[23] cockroach: CockroachDB - the open source, cloud-native SQL database, June 2018. original-date: 2014-02-06T00:18:47Z.

[24] rethinkdb: The open-source database for the realtime web, July 2018. original-date: 2012-10-30T05:37:47Z.

[25] raft-leader-election: Raft Consensus algorithm to allow a collection of nodes / members to work as a coherent group that can survive the failures of some of its members, June 2018. original-date: 2017-03-27T08:52:50Z.

[26] RavenDB - Documentation - 4.0 - Glossary : Raft Consensus Algorithm.

[27] Christian Cachin and Marko Vukolic. Blockchain consensus protocols in the wild. *CoRR*, abs/1707.01873, 2017.

[28] Bela Ban. JGroups, a toolkit for reliable multicast communication. 2002.

[29] Shuai Mu, Lamont Nelson, Wyatt Lloyd, and Jinyang Li. Consolidating Concurrency Control and Consensus for Commits under Conflicts. page 16.

[30] Daniel Porto, João Leitão, Cheng Li, Allen Clement, Aniket Kate, Flavio Junqueira, and Rodrigo Rodrigues. Visigoth fault tolerance. In *Proceedings of the Tenth European Conference on Computer Systems - EuroSys '15*, pages 1–14, Bordeaux, France, 2015. ACM Press.

[31] Patrick Hunt, Mahadev Konar, Flavio P Junqueira, and Benjamin Reed. ZooKeeper: Wait-free coordination for Internet-scale systems. page 14.

[32] Harry Kalodner, Steven Goldfeder, Alishah Chator, Malte Möser, and Arvind Narayanan. BlockSci: Design and applications of a blockchain analysis platform. *arXiv:1709.02489 [cs]*, September 2017. arXiv: 1709.02489.