Generalized Consensus for Practical Fault Tolerance

Mohit Garg  
Virginia Tech  
mohitce@vt.edu

Balaji Arun  
Virginia Tech  
balajia@vt.edu

Sebastiano Peluso  
Virginia Tech  
peluso.sebastiano@gmail.com

Binoy Ravindran  
Virginia Tech  
binoy@vt.edu

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 nodes are correct and communicate synchronously. Although XPTaos 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. We present Elpis, the first multi-leader XFT consensus protocol. By adopting the Generalized Consensus specification, 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 replicas during periods of synchrony. Subsequently, these replicas order all commands which access these objects. The experimental evaluation confirms the effectiveness of this approach: Elpis achieves up to 2x speedup over XPTaos and up to 3.5x speedup over state-of-the-art Byzantine Fault-Tolerant Consensus Protocols.

CCS Concepts: • Security and privacy → Distributed systems security; • Software and its engineering → Software fault tolerance;

Keywords  Consensus; Generalized Consensus; Byzantine Fault Tolerance; Collision Recovery; Blockchain

1 Introduction

Consensus solutions underpin numerous distributed systems – from horizontally scalable databases [2, 13, 16] and key-value stores [3] to distributed synchronization services [12, 17] – providing strong consistency, fault-tolerance, and high availability. These systems employ State Machine Replication [29] where Consensus algorithms are used to achieve agreement on a common order among concurrent client requests that each node in a distributed system should execute, even in the presence of faults.

Consensus algorithms are designed using two prominent fault models: the Crash Fault Tolerance (CFT) model and the Byzantine Fault Tolerance (BFT) model [23]. CFT protocols do not tolerate any non-crash faults – even accidental faults like hardware errors, misconfigurations, and software bugs, that are increasingly common in production systems today [1, 4, 5]. BFT protocols, on the other hand, shield applications from non-crash faults, including malicious actors, but are expensive, requiring more resources and complex messaging patterns. Notably, in geo-scale deployments where round-trip timings (RTT) are high, BFT protocols have significantly higher client-perceived latencies, discouraging widespread adoption. Various approaches [8, 18, 25] improve the performance of BFT protocols, but the lower bounds of the BFT model [23] prevents from reducing both the number of communication steps as well as the quorum size, which is essential for reducing wide-area latencies.

Most practical systems today operate in secure networks with solutions in place to thwart malicious attacks like distributed denial-of-service [31]. For these systems, the Cross Fault Tolerance (XFT) model [24] achieves a favorable trade-off between the CFT and BFT models. Mainly, the XFT model relaxes the assumption that the adversary can launch coordinated attacks, which is unlikely in geo-scale deployments but is sufficient to shield applications from crash faults, network faults, and non-crash non-malicious faults. This enables the XFT model to use the same quorum size and the same number of communication steps as the CFT model by assuming that a majority of processes are correct and synchronous.

XPaxos [24], the lone XFT protocol, is leader-based with performance similar to CFT-based Raft/Multi-Paxos [27]
protocols. While the XFT model is built with an assumption that befits the geo-replicated setting, the accompanying algorithm, XPaxos, provides poor scalability and performance. XPaxos inherits the shortcomings of leader-based approaches: imbalanced load distribution, where the leader does more work than other nodes; high latency for requests originating from non-leader nodes due to the requirement of forwarding requests to the leader; and the inability to deliver any commands whenever the current leader is slow or Byzantine pending a leader/view-change.

To address these aspects, we present Elpis, the first multi-leader XFT consensus protocol that exploits the underlying commutativity of commands to provide fast decisions in three communication steps from any node, in the common case. We achieve this by exploiting workload locality that is very common in geo-scale deployments. The core idea of Elpis is enabling 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 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 three communication steps can be achieved from the owner nodes. Furthermore, clients benefit from workload locality by sending requests to the closest node (with ownership) and observe optimal latencies.

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 between multiple nodes trying to acquire the ownership of the same object(s) concurrently. We address this using a rectification mechanism that follows the ownership acquisition phase during conflicts and minimizes the number of retries to acquire ownership. Additionally, while XPaxos and BFT protocols like PBFT [15] use a three-phase view-change/leader election sub-protocol in addition to the normal operation phases, Elpis requires just one additional phase (two for regular operation) for liveness. This linear procedure of Elpis also improves on the combinatorial view-change mechanism of XPaxos.

We implemented Elpis and competitors - M^2Paxos [28], XPaxos, PBFT, Zyzzyva [18] - in Java, using the JGroups messaging toolkit for the first two protocols and BFT-SMaRT [10] a highly optimized implementation of PBFT to implement Zyzzyva. 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 the state-of-the-art BFT Consensus Protocols.

The core contributions of this paper are:
1. The design and implementation of the first multi-leader Cross Fault Tolerant (XFT) consensus protocol
2. An ownership conflict resolution protocol that minimizes retries due to proposer contention using a more cohesive algorithm.
3. An extensive evaluation and comparison to the existing state-of-the-art in the BFT space.

The rest of the paper is organized as follows: In Section 2, we discuss the desirable properties which are seemingly amiss from production systems today. Section 3 presents the system model and assumptions. In Section 4.1, we introduce Elpis at a high level, while in Section 4.3, we delve into the details and present the algorithm pseudocode. Section 5 presents the correctness proof of Elpis supported by a TLA+ specification [20] for the algorithm. We evaluate our solution and competitors in Section 6, and we summarize the existing state-of-the-art solutions that are related to our contribution, Elpis in Section 7. We conclude in Section 8.

2 Motivation

One of the primary motivations for Elpis is to provide strong consistency in geo-replication. Under the CAP theorem [11], only one out of two – Consistency or Availability – can be guaranteed in the presence of a network partition. Distributed systems like Cassandra [14] and DynamoDB [30] choose availability over consistency under a network partition. While availability can not be assured in real-world systems, these systems hand-over the burden of ensuring consistency to application developers. Additionally, such systems cannot safeguard applications from faults such as data corruptions without additional mechanisms. In contrast, with Elpis, the objective is to favor strong consistency, while striving to provide high availability under faults. Specifically, using the localized ownership mechanism, we ensure that a faulty node does not bring the system to a standstill (pending leader election), unlike single-leader based protocols. As long as a majority of nodes are up, clients requests are executed.

Moreover, Elpis empowers shard-based systems to guarantee linearizability on multi-shard operations. Most geographically distributed systems run a per-shard consensus protocol (e.g., CockroachDB [2], Spanner [16]) to achieve scalability with the number of nodes. However, guaranteeing linearizability on multi-shard transactions is non-trivial and requires additional hardware components such as GPS clocks which timestamp transactions to establish order. Example, CockroachDB uses HLC (hybrid logical clocks), and YugaByte [6] uses Hybrid Time. Such systems can instead depend on a single instance of Elpis to guarantee linearizability as well as scalability without the need for additional mechanisms. Multi-shard operations can be easily executed without complicated cross-shard transactions by the application layer.
Furthermore, Elpis provides an appealing trade-off between CFT and BFT protocols. While CFT protocols cannot tolerate non-crash faults, BFT protocols require more nodes and larger quorums to tolerate the same number of faults as CFT protocols. The need for bigger quorums in the BFT model is due to the assumption that \( t \) non-faulty processes could be slow in responding in the presence of \( t \) Byzantine processes. Therefore, an additional \( t \) non-faulty processes are required to distinguish between messages from non-faulty and Byzantine processes. Hence, a quorum of size \( 2t+1 \) out of \( 3t+1 \) processes is required for consensus protocols that employ the BFT model. In practice, this assumption implies that an adversary can affect the network on a wide scale as well as attack multiple nodes all in a coordinated fashion. This is a strong assumption, especially for geo-replicated systems where data-centers are distributed around the world and linked using secure networks. The XFT model, instead, provides the same quorum size and uses the same number of communication steps as the CFT model.

Multi-leader consensus solutions [7, 26, 28] have been proposed for the CFT model in recent years to address the aforementioned issues with the single leader algorithms. Such solutions adopt the Generalized form of Consensus [21], 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 processes. Implementing this in the XFT model is non-trivial due to the addition of Byzantine nodes wherein commands originating from these nodes cannot be committed. Our goal is to provide a Generalized Consensus algorithm in the XFT model which achieves high performance in the geo-replicated setting using XFT, an adversary model which befits the geo-replicated setting.

3 System Model and Problem Formulation

This section specifies the system assumptions used for designing Elpis, the contribution of this paper. 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 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, 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, we say that the network is partitioned and the system model abstracts these partitions as network faults \((t_p)\). Following the XFT model [24], the total number of faults are bounded by,

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

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. For discussion in this paper, the system is assumed to be never in anarchy – there always exists at least a majority of correct and synchronous processes.

The Generalized Consensus [21] specification is used where the processes try to reach consensus on a sequence of commands, the \(C\)-Struct. The Consensus algorithm orders non-commutative commands before deciding and decides commutative commands directly. 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 \(c.LS\).

3.2 Problem Statement

Given the system model, the 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. When the process decides, it applies the \(C\)-Struct to the State Machine and generates a reply which is returned to the client. Given that the majority of processes are correct and communicate synchronously (Equation 1), the following properties should be guaranteed.

- **Non-triviality** Only proposed commands are decided and added to the \(C\)-structs.
- **Stability** If a process decided a \(C\)-struct \(cs\) at any time, then it can only decide \(cs \ast \sigma\), where \(\sigma\) is a sequence of commands, at all later times.
- **Consistency** Two \(C\)-structs decided by two different processes are prefixes of the same \(C\)-struct.
- **Liveness** If a command \(c\) is proposed it will eventually be decided and added to the \(C\)-struct.

In Section 4, we illustrate how Elpis achieves State Machine Replication, and in Section 5, we prove that Elpis satisfies all of the properties listed above.

4 Protocol Description

4.1 Overview

Interestingly, Elpis derives the inspiration for implementing Generalized Consensus from \(M^2\)Paxos [28] which does not tolerate Byzantine faults. The core idea of \(M^2\)Paxos is to
avoid contention among multiple processes that propose non-commutative commands \( C \) by dynamically choosing a unique owner for the objects on which the commands operate. This owner now orders all commands which access the objects for this epoch. Once an owner is chosen, other processes forward any command in \( C \) to the owner. 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 4.3.5). If the process acquires the ownership, it tries to decide the command.

For Byzantine Fault Tolerant algorithms a quorum of size \( 2t + 1 \) out of \( 3t + 1 \) processes is required where \( t \) is the number of faulty processes. 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 of the protocol. Hence, a quorum of size \( 2t + 1 \) seems implausible. Elpis takes the 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. In the worst case the client retries request with a maximum of \( t \) faulty proposers and on the \( t + 1 \) try the request is decided.

The ownership acquisition can affect the progress of the protocol if multiple processes try to acquire the ownership by issuing an increasing sequence of epoch values similar to Paxos [19]. To attenuate this scenario all processes are allotted tag values picked from a set of totally ordered elements and prioritized as such when proposing. The acceptor which replies with a Nack message includes the tag of the process it last sent an Ack for the highest epoch for the objects in the message received. Upon receiving \( t + 1 \) Nack messages the proposer 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 (CR), the failing processes reevaluate the ownership configuration in the system depending on the result of CR and either take command of the objects or else forward the command to the process picked.

Initially, a home process \( p_i \) that is closest (incurs lowest latency) to the client receives the request. The client sets a timer and waits for responses. Each correct process responds with either a signed Reply message or a signed Abort message. If the client receives \( t + 1 \) messages with matching replies then the client is sure that the request is replicated. Alternatively, if the client receives \( t + 1 \) Abort messages or the timer expires, then \( p_i \) is not part of the correct and synchronous group and it retries with another process.

In summary, Elpis solves two challenges of implementing Generalized Consensus in the XFT model: (1) How to tolerate Byzantine faults with \( 2t + 1 \) processes with no predetermined leaders (2) How to reliably acquire the ownership as measured in terms of the number of retries required in the presence of multiple processes vying to take the ownership of objects by using two 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 helps resolve the ownership if multiple processes try to acquire the ownership concurrently.

### 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 array that maps a pair of \( (l, in) \) 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 uni-dimensional array which maps the consensus instance \( in \) that \( p_i \) 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 (non-negative integer). Epoch\([l] = e \) means that \( e \) is the current epoch number that has been observed by \( p_i \) for the object \( l \). The initial value is 0.
- **Owners** It is an array that maps an object to a process. 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 \( (l, in) \) to an epoch number. 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\([l][in] = (r, e) \) implies that the process received a quorum of 1 in the commit phase for request \( r \) and epoch \( e \). The StatusLog maintains the valid \( (r, e) \) that the process is willing to commit on. Hence, StatusLog\([l][in] = (r, e) \) implies that \( p_i \) would accept a replicate message for \( r \) in epoch \( e \) and reject others.
- **statusList, commitList, decideList, trustList**. These are four multidimensional arrays which are used to store COMMIT, STATUS, DECIDE and TRUST messages respectively. The initial value is NULL.
- **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 (Section 4.3.6). This mapping has to be predefined by the application layer during setup and is static during the protocol execution.
- **Estimated** A multidimensional array which maps the \( (l, in) \) to the address of the process which this process estimates to be the owner of the object \( l \) for an epoch \( e \). Hence, Estimated\([l][in] = (e, t_p, p_e) \) implies that for epoch \( e \) this
process estimates \( p_e \) to be the owner where \( t_{p_e} \) is the tag of the process.

- **Leader** This is a multidimensional array which maps the \((l, in)\) pairs to the \((e, p_i)\) pairs for which the collision recovery decides 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 some process \( p \) along with it 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 \( req = \langle \text{Request}, o, t, ls, c \rangle \sigma_c \) to a process \( p_i \) 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 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 the Coordination phase (Algorithm 1), \( p_i \) reads the ownership of objects in the system. Depending on the current ownership configuration, the process either invokes the replication phase (Section 4.3.2), forwards the request to the owner or tries to acquire the ownership for all the objects accessed by the \( req \) by executing ownership acquisition (Section 4.3.5).

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 \( \text{LastDecided}[l] + 1 \) 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 \( ins \), it forwards the \( req \) to the \( p_k \). To avoid blocking in case \( p_k \) crashes or is partitioned, \( p_i \) also sets a timer. Upon expiration of the timer, if the \( p_i \) detects that the \( req \) has not been decided, it takes charge of the \( req \) and tries to \( C \rightarrow \text{Propose} \) the \( req \) (lines 10-14).

Finally, if \( p_i \) detects no owners for all objects in \( ins \), it tries to acquire the ownership by executing the acquisition phase (4.3.5) (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 \(\text{ins}\) for proper linearization of commands during execution.

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

1: function Bool Replicate(req, Set ins, Array eps)
2: send (Replicate, \(r, \text{ins}, \text{eps}\)) to \(p_j\) to all \(p_j \in \Pi\)
3: if \(\forall (l, \text{in}) \in \text{ins}, \text{Rnd}[l][\text{in}] \leq \text{eps} = \text{eps}(\text{in})\) \& \(\geq \text{Rown}(p_j, \text{ins}) = T\)
4: CommitPhase\(p_j, \rightarrow, \text{r, ins, eps}\)
5: else if StatusLog\(l, \text{in}] \neq \text{ins}\) then
6: send (Abort, \(r, \text{eps}\)) to \(r, c\)
7: \(\forall (l, \text{in}) \in \text{ins, Rnd}[l][\text{in}] \geq \text{eps} = \text{eps}(\text{in})\) do
8: CommitPhase\(p_j, \rightarrow, \text{ins, eps}\)
9: \(\forall (l, \text{in}) \in \text{ins, Rnd}[l][\text{in}] \geq \text{eps} = \text{eps}(\text{in})\) then
10: send (Commit, \(\rightarrow, \rightarrow, \text{eps, deferTo, NACK}\)) to \(p_j\)

4.3.2 Replication Phase

In the Replication phase (Algorithm 2), \(p_i\) requests the replication of request \(r\) for instance \(\text{ins}\) and epochs \(\text{eps}\). It sends a signed Replicate message to all processes in \(\Pi\). Upon receiving a Replicate message, the process \(p_j\) checks if the received message is for an epoch greater than or equal to the last observed \(\text{Rnd}[l][\text{in}]\) for all the objects in the request and checks if \(p_j\) is, in fact, the owner of all the objects in the message (line 5). If both of these conditions are satisfied, \(p_j\) starts the Commit phase (Section 4.3.3) for the request with the received \(\text{ins}\) and \(\text{eps}\) values (line 6).

The Replication phase is invoked during either the Acquisition phase (Section 4.3.5) where a process is trying to acquire the ownership or invoked directly by a process which already has the ownership of all objects in the request \(r\) \& \(\text{LS}\). Otherwise, a StatusLog is constructed by collecting Status messages in the Acquisition phase. This request, epoch pair in the StatusLog\(l, \text{in}]\) is considered to be valid. This is discussed further in (Section 4.3.5). Hence, the \((r, \text{eps} = \text{eps}(\text{in})\) in the Replicate message should match the values in the StatusLog for a process \(p_i\) which is trying to acquire the ownership. If this is not the case then \(p_i\) has equivocated and hence, this phase concludes by sending an Abort message to the client.

Otherwise, if the message does not fall under either of the cases mentioned above then \(p_i\) has already acknowledged a message for \(\text{eps} = \text{eps}(\text{in})\) from the owner of the objects in the message and sends a \(\text{Nack}\) message along with the information about the last process it sent an \(\text{Ack}\) for the highest epoch for one or more \((l, \text{in})\) pairs (lines 8-10). This information returned with the \(\text{Nack}\) message is relevant for the collision recovery and it is discussed in detail in Section 4.3.6.

4.3.3 Commit Phase

In the Commit phase (Algorithm 3), correct processes coordinate to pick a valid request for instances in \(\text{ins}\) and for the epochs in \(\text{eps}\) in the presence of Byzantine processes.

The request \(r\) received in the Replicate message is broadcasted using Commit messages and each received Commit is collected in the commitList (lines 11-14).

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

1: function Void CommitPhase(Replica \(p_j\), Array toForce, Request req, Set ins, Array eps)
2: Array toDecide
3: \(\forall (l, \text{in}) \in \text{ins}, \text{toForce}[l][\text{in}] = (\text{req}, \rightarrow) : \text{req} \neq \text{NULL}\) do
4: \(\text{toDecide}[l][\text{in}] = \text{req}\)
5: \(\forall (l, \text{in}) \in \text{ins}, \text{toDecide}[l][\text{in}] = \text{NULL}\) then
6: \(\forall (l, \text{in}) \in \text{ins}\) do
7: \(\text{toDecide}[l][\text{in}] = \text{req}\)
8: send (Commit, \(p_j\), toDecide, ins, eps, \(\rightarrow\), \(\rightarrow\)) to all \(p_k \in \Pi\)
9: \(\forall (l, \text{in}) \in \text{ins}, \text{Rnd}[l][\text{in}] \leq \text{eps} = \text{eps}(\text{in})\) then
10: \(\forall (l, \text{in}) \in \text{ins}\) do
11: \(\forall (l, \text{in}) \in \text{ins}\) do
12: Decide\(p_j, \text{r}, \text{eps}, \text{defeTo, NACK}\) to \(p_j\)
13: \(\forall (l, \text{in}) \in \text{ins}, \text{Rnd}[l][\text{in}] \geq \text{eps} = \text{eps}(\text{in})\) then
14: CommitPhase\(p_j, \rightarrow, \text{eps, deferTo, NACK}\)
15: trigger Decide\(p_j, \text{eps, eps, defeRs}\)
16: else if \(\forall (l, \text{in}) \in \text{ins}, \text{Rnd}[l][\text{in}] \geq \text{eps} = \text{eps}(\text{in})\) then
17: send \(\text{Decide}\) \(p_j, \text{r}, \text{eps, defeTo, NACK}\) to \(p_j\)
18: \(\forall (l, \text{in}) \in \text{ins}\) do
19: \(\forall (l, \text{in}) \in \text{ins}\) do
20: \(\forall (l, \text{in}) \in \text{ins}\) do
21: \(\forall (l, \text{in}) \in \text{ins}\) do
22: \(\forall (l, \text{in}) \in \text{ins}\) do
23: \(\forall (l, \text{in}) \in \text{ins}\) do
24: return toCommit

There are two cases for Byzantine processes: (1) Any \(t\) acceptors could send arbitrary request values rather than forwarding \(r\), (2) The proposer which has the ownership can equivocate by sending req to some processes and some request \(r'\) to the rest of the processes. To tackle both of these scenarios a request \(r\) is valid if \(p_j\) receives \(t+1\) matching Commit messages (Ack) for \(r\) and \(r\) matches the request for which this phase was invoked. If there exists a valid req for all \((l, \text{in})\) pairs, then this phase successfully concludes by setting the owners array to the process which sent the Replicate message which invoked this Commit phase, adds the values to the CommitLog and sends a Decide message to all the processes (line 19 - 22). Otherwise, if either there exists no common request req in at least \(t+1\) messages or req does not match the request for which this phase was invoked,
then this process aborts by sending an abort message to the client (line 24).

If $p_i$ does not acquire $t + 1$ commit messages (ack) and there exist nack commit messages, then some other process has stolen the ownership. In this case, $p_i$ triggers the collision recovery phase (Section 4.3.6).

**Algorithm 4 Elpis: Decision phase (node $p_i$).**

1. upon Decides((Set to Decide, Set ins, Array eps)) from $p_j$
2. for all $(l, in) \in ins$
3. $\epsilon \leftarrow$ decideList[((l, in)[e] ← decideList[((l, in)[e] ∪ ((toDecide)((l, in)), j))
5. if \( \forall (l, in) \in ins, \exists r : (r', -) : Decides[((l, in)[eps][l][in])] \geq \text{sizeof(Quorum)} \)
6. for all $(l, in) \in ins$
7. if Decided[l][in] = NULL then
8. Decided[l][in] $\leftarrow r$
9. 
10. upon $(\exists l : r \in l, \forall s : Decides[l][s][i] = r \land in = \text{LastDecided}[l] + 1)$
11. Cstructs $\leftarrow$ Cstructs • r
12. Reply req $\leftarrow$ Decide(Cstructs)
13. send Reply(req) to $r$
14. for all $l \in l, r, LS do
15. $p_i.\text{LastDecided}[l] + 1$

4.3.4 Decision Phase

In the Decision phase (Algorithm 4) a process $p_i$ tries to learn a request. Upon receiving a Decides message the process stores the message in the decides array indexed by the $(l, in)$ pair and the epoch $e$. If $p_i$ receives $t + 1$ matching 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 (lines 7-13).

4.3.5 Acquisition Phase

In the Acquisition phase (Algorithm 5) 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 the object $l$. The process $p_i$ now sends a Prepare message to all processes in P(lines 1-6).

Before sending a Prepare message, the process also sets Estimated[l][in] to its tag and epoch thus estimating itself to acquire the ownership. This is relevant if this process receives a Prepare message for the same or a lower epoch for the objects it is trying to acquire the ownership. In that case, this process will send a Nack message using the Estimated[l][in] values.

**Algorithm 5 Elpis: Acquisition Phase (node $p_i$).**

1. function Void AcquisitionPhase(Request req)
2. Set ins $\leftarrow$ (l, LastDecided[l] + 1) : l \in \text{c.LS} \land \mathcal{I}_{\text{in}} : \text{Decided[l][in]} = c$
3. Array eps
4. $\forall (l, in) \in ins, \text{eps}[l][in] \leftarrow + + \text{Epoch}[l]$
5. $\forall (l, in) \in \text{ins}, \text{Estimated[l][in]} \leftarrow (\text{eps}[l][in], \text{Tag}[p_i], p_i)$
6. send Prepare(ins, eps) to all $p_j \in P$
7. 
8. upon Prepare((Set ins, Array eps)) from $p_j$
9. if $\forall (l, in) \in \text{ins}, \text{Rnda}[l][in] < \text{eps}[l][in]$ then
10. $\forall (l, in) \in \text{Rsnda}[l][in] \leftarrow \text{eps}[l][in]$
11. Set deco $\leftarrow (\{(l, in, \text{CommitLog}[l][in]) : (l, in) \in \text{ins} \}$
12. send Status(ins, eps, deco, $\neg, \neg, \neg)$ to all $p_j \in P$
13. else
14. for all $(l, in) \in \text{ins}, \text{Rnda}[l][in] \geq \text{eps}[l][in]$ do
15. Set deferTo $\leftarrow (\text{Rnda}[l][in], \text{Tag}[p_i])$
16. send Status(ins, eps, deco, deferTo, NACK) to $p_j$
17. 
18. upon Status((Set ins, Array eps, Array deco, Array deferTo, Value ack)) from $p_j$
19. for all $(l, in) \in \text{ins}$ do
20. $e \leftarrow \text{eps}[l][in]$
21. Set statusList[l][in][e] $\leftarrow$ statusList[l][in][eps[l][in]] $\cup$ \{(l, in), deferTo, ack, j)$
22. $\forall (l, in) \in \text{ins, statusList}[l][in][\text{eps}[l][in]] \geq N - t$ then
23. $e \leftarrow \text{eps}[l][in]$
24. if $\exists (-, \text{deferTo, NACK}, -)$ : statusList[l][in][e] then
25. $\forall (l, in) \in \text{ins, set defer}[l][in] \leftarrow$ deferTo $= (\neg, \text{deferTo, NACK}, -)$
26. trigger Decide(ins, eps, deco)
27. return
28. 
29. Set epochHighest $\leftarrow$ Select(ins, statuses)
30. Set valids $\leftarrow$ Valid(ins, statuses)
31. if epochHighest $\neq 0$ $\land$ valids $\neq 0$ then
32. Set StatusLog[l][in][e] $\leftarrow$ \{(req, eps[l][in])
33. if $p_i$ $\neq$ Proposer then
34. Replicate(req, ins, eps)
35. else if $\exists (r, e, l, in) \in \text{epochHighest} \land (r, e, l, in) \in \text{valids}$ then
36. StatusLog[l][in][e] $\leftarrow$ $(r, e, l, in)$
37. Set statusLog[l][in][e] $\leftarrow$ $(r, e, l, in)$
38. Replace(ToForce[l][in][e] $\leftarrow$ $(r, e, l, in)$
39. C: Proposer(req)
40. else
41. send Abort(ins, eps, req) to all $p_j \in P, req.c$
42. function Set Select(Set ins, Set statuses)
43. return ToForce
44. for all $(l, in) \in \text{ins}$ do
45. Epoch $k$ $\leftarrow$ $max(k : (-, k) \in \text{dec} \land (\text{dec}, \neg, \neg) \in \text{statuses})$
46. Request $r \leftarrow$ $r : (r, k) \in \text{dec} \land (\text{dec}, \neg, \neg) \in \text{statuses}$
47. return ToForce $= (r, k, l, in)$
48. 
49. function Array Value(Set ins, Set statuses)
50. Array valid
51. return ToForce
52. for all $(l, in) \in \text{ins, statuses}[l][in] = (\text{dec}, \neg, \neg) : (\text{dec}, j)$ do
53. Set requests[j][e] $\leftarrow$ $(r, e) \in \text{dec}$
54. if $\exists (r, e) \in \text{dec} \land \text{requests}[\geq t]$
55. valid $\leftarrow (r, k, l, in)$
56. return valid

Upon receiving a Prepare message with a higher epoch for all objects than the last observed, each process sends its CommitLog in the Status message to all the processes (lines 8-12). If the received message has a lower epoch, it sends a Nack message with the information about process it last sent an Ack for (lines 14-16). The Status message includes the CommitLog for the (object, instance) pairs. Upon receiving Status messages from enough processes (at least $N - 1$), the
process decides if there is a request to be committed from an aborted Commit Phase from an earlier epoch. For this, for all ⟨request, epoch⟩ entries present in the CommitLog received from a process \( p_1 \), the process first calculates the highest epoch values present in the entries for which a request is present and adds such ⟨request, epoch⟩ pairs to the epochhighest set (lines 20-27).

However, the request in this log could be from a Byzantine process. To eliminate such requests, each process also calculates a valid ⟨request, epoch⟩ pair by reading the CommitLogs it received as part of the Status message. If a pair is present in more than the number of faulty processes then this ⟨request, epoch⟩ pair is validated. If a ⟨request, epoch⟩ is present in the epochhighest set and is also present in the validated set then process starts a commit phase with toForce array which contains this ⟨request, epoch⟩ 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 retries the request with a different process.

### 4.3.6 Collision Recovery

Collision recovery (Algorithm 6) is used to reduce the number of processes contending to acquire the ownership of some object(s). When a process \( p_i \) receives Nack messages (line 24 in Algorithm 5 and line 16 in Algorithm 3), the process \( p_k : (\neg, \text{tag}, p_k) \in \text{deferTo} \) is a process which is executing Elpis for an epoch equal to or higher than the Epoch\([\ell]\) at \( p_i \) for some object \( \ell \). Hence, some subset of acceptors return a Nack message as they have already sent an Ack to process \( p_k \). There could be multiple processes like \( p_k \) at any given time. For instance, if all processes propose simultaneously for the same epoch. This phase provides a coordinated mechanism to find a process which is executing ownership acquisition for the highest epoch with any ties broken by using the tags of the processes. Succinctly stated, \( p_i \) uses collision recovery to conform to the current ownership reconfiguration taking place in the system as opposed to contending by proposing higher epoch values.

The process \( p_i \) may receive multiple Nack messages. In this case, a set of rules (similar to Fast Paxos [22]) are used where \( p_i \) tries to Pick a process to defer to. A deferTo value is picked if it exists in a majority of Nack messages or has the maximum count with any ties broken by using the tags. After the completion of the Collision recovery (CR), the process sets its owners to the one learned from CR and retries the request with the coordination phase (lines 1-5).

Before starting the phase \( p_i \) checks if an instance of Collision Recovery has already been completed by the system (invoked by some other process). In this case, no additional run is required and \( p_i \) concludes the recovery (line 8-9). However, if no such instance has been completed then \( p_i \) starts this recovery by sending a \( \{\text{Trust, ins, eps, leader}\}_{\alpha p_i} \) 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 38-42). 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 (lines 22-26). If the value is lower however, the process sends a Doubt message with the higher value (line 30-31).

---

### Algorithm 6: Elpis: Collision Recovery (node \( p_i \))

1: function Defe x(\( \text{Set ins, Array eps, Set deferTo} \))
2: \( \text{Array deferTo} \leftarrow \text{Pick}(\text{ins, deferTo}) \)
3: \( \text{Recovery(\text{ins, deferTo})} \)
4: \( \forall (l, \text{in}) \in \text{ins}, \text{Owners}[l] \leftarrow \text{Leader}[l][\text{in}] \)
5: \( \text{trigger C-Process}(\text{r}) \) to \( p_i \)
6: \( \text{function Void(\text{Set ins, Array eps, Array deferTo})} \)
7: \( \text{if } \forall (l, \text{in}) \in \text{ins}, \exists (e, p_i) \supseteq \text{Leader}[l][\text{in}] : (e, p_i) : e \geq \text{eps}[l][\text{in}] \) then
8: \( \text{return} \)
9: \( \text{else } \)
10: \( \text{send } (\text{Trust, ins, deferTo})_{\alpha p_i} \) to all \( p_k \in \Pi \)
11: \( \text{else } \)
12: \( \text{function Array Pick}(\text{Set ins, Set deferTo}) \)
13: \( \text{Array deferTo} \)
14: \( \forall (l, \text{in}) \in \text{ins} \) do
15: \( \text{Count}(\langle e, t_{p_i}, p_i \rangle) = |(e, t_{p_i}, p_i) = (e', t_{p_i}', p_i') : \text{deferTo}[l][\text{in}]| \)
16: \( \text{if } \exists (e, t_{p_i}, p_i) \supseteq \text{Count}(\langle e, t_{p_i}, p_i \rangle) = \text{sizeof}(\text{Quorum}) \land\) (\( \langle e, t_{p_i}, p_i \rangle \supseteq \text{Count}(\langle e, t_{p_i}, p_i \rangle) = \max(|\{\text{Count}(\langle e, t_{p_i}, p_i \rangle) : \text{deferTo}[l][\text{in}]\}|) \) then
17: \( \text{ deferTo}[l][\text{in}] \leftarrow (e, t_{p_i}, p_i) \)
18: \( \text{return deferTo} \)
19: \( \text{else } \)
20: \( \text{function Void(\text{Set ins, Array deferTo})} \) from \( p_j \)
21: \( \text{if } \text{isHigher}(\text{ins, deferTo}) \) then
22: \( \forall (l, \text{in}) \in \text{ins} \text{ do } \)
23: \( \text{Estimated}[l][\text{in}] \leftarrow \text{deferTo}[l][\text{in}] \)
24: \( \text{trustList}[l][\text{in}] \leftarrow \text{trustList}[l][\text{in}] \cup \text{\{p_j \}} \leftarrow \text{deferTo}[l][\text{in}] \)
25: \( \text{if } \exists (e, p_i) \supseteq \text{\{p_j \}} : \text{trustList}[l][\text{in}] \land \text{sizeof}(\text{Quorum}) \) then
26: \( \text{Leader}[l][\text{in}] \leftarrow (e, p_i) \)
27: \( \text{return deferTo} \)
28: \( \text{else } \)
29: \( \forall (l, \text{in}) \in \text{ins}, \exists \text{ estimate } : \text{Estimated}[l][\text{in}] \)
30: \( \text{send } (\text{Doubt, ins, estimate})_{\alpha p_i} \) to \( p_i \)
31: \( \text{if } \forall (l, \text{in}) \in \text{ins}, \text{Leader}[l][\text{in}] \neq \text{NULL} \) then
32: \( \text{return} \)
33: \( \text{else } \)
34: \( \forall (l, \text{in}) \in \text{ins}, \exists \text{ estimate } : \text{Estimated}[l][\text{in}] \leftarrow \text{deferTo}[l][\text{in}] \)
35: \( \text{send } (\text{Doubt, ins, Array estimate})_{\alpha p_i} \) from \( p_j \)
36: \( \forall (l, \text{in}) \in \text{ins}, \exists \text{ estimate } : \text{Estimated}[l][\text{in}] \leftarrow \text{deferTo}[l][\text{in}] \)
37: \( \text{function Bool IsHigher}(\text{Set ins, Set Received}) \)
38: \( \text{for all } (l, \text{in}) \in \text{ins}, \exists \text{ estimate } : \text{Estimated}[l][\text{in}] = (e, t_{p_i}, \neg e) : (e, t_{p_i}, p_i) \)
39: \( \text{Received}[l][\text{in}] = (e', t_{p_i}', \neg e') : (e', t_{p_i}') \) do
40: \( \text{if } e > e' \lor (e = e' \land t_{p_i} > t_{p_i}') \) then
41: \( \text{return } \) false
42: \( \text{return } \) true
Upon receiving a \textit{Doubt} message the process sets its \textit{Estimated} to the value received in the \textit{defer} message (line 22). When the cardinality of \textit{statusList}[l][i] equals the \textit{Quorum} for some \((l, i)\) pair then the process \(p_l : trustList[l][i]\) is trusted to be the owner of this \((l, i)\) pair and when there is a trusted owner for all \((l, i) \in \text{ins}\) then the recovery concludes.

5 Correctness
We formally specified Elpis in TLA+ [20], and model-checked with the TLC model checker for correctness as a decision on the correct value despite the presence of Byzantine acceptors and abort when the proposer equivocates. The TLA+ specification is provided in two anonymous technical reports\(^1\)\(^2\) for the algorithm component and the collision recovery component. In this section, we provide an intuition of how Elpis satisfies the protocol’s guarantees.

\textbf{Stability:} Only the owner of an object \(l\) in epoch \(e\) successfully commits the requests, and thus increments \(\text{ins}\). A Byzantine process does not acquire ownership as the proposer equivocation is detected and the execution is aborted. Since, the correct processes initially start with the same value of \textit{LastDecided}\([l]\) and only increment it when a command is decided for \((l, i)\), the valid requests proposed by a correct owner of \(l\) in \(e\) would follow a complete order for \(\text{ins}\) throughout the execution of the protocol and would not diverge for any correct process.

In the rest of the section we refer to \textit{StatusLog}\([l][i]\) and \textit{CommitLog}\([l][i]\) as \textit{StatusLog} and \textit{CommitLog} for brevity which denote the value of the logs for some \((l, i)\). The proofs can be generalized for any instance \(i\) of the object for which the process acquires ownership of the object \(l\).

\textbf{Non-triviality:} A process only appends a command \(c\) to the \textit{C-Struct} if it receives \textit{Commit} messages from a majority of processes for \(c\) and no other command can exist. Correct processes only send \textit{Commit} messages for the value \(c\) if they receive \(c\) in the \textit{Replicate} message.

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

We argue that if a correct process in \(X\) commits request \(c'\) in the epoch \(e'\), and \textit{StatusLog} = \((c, e)\) then for \(e' : (c, c') = \textit{StatusLog} \land e' \geq e, c' = c\). We prove this by induction on the epoch \(e'\). For the base case, lets suppose \textit{StatusLog} = \((c, e')\) at some correct process \(p_i\). If \(p_i\) commits \(c\) in \(e'\) then it must receive a \textit{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 \textit{Abort}. Hence, \(c' = c\). For \(e'\), \(p_i\) commits on \(c'\), sets \textit{CommitLog} = \((c', e')\) sets the process which sends \(c'\) as the owner (which is in fact correct).

Let’s suppose that for any epochs in between \(e'\) and \(e\), \textit{StatusLog} = \((c', e')\). We have to prove that if \textit{StatusLog} = \((c, e + 1)\) then \(c = c'\). The \textit{StatusLog} = \((c, e + 1)\) consists of valid \textit{CommitLogs} for \(c\) in \(e\). Since, the \textit{StatusLog} = \((c', e)\) any correct process that commits \(c\) and sets its \textit{CommitLog} to \((c, e)\) 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 \textit{agreement}: If two correct processes commit \(c\) and \(c'\) then \(c = c'\). If a correct process initially commits \(c\) in \(e\), then \textit{StatusLog} = \((c, e)\). If another correct process commit \(c'\) in \(e'\), then we know that for any \(e > e', c = c'\). Hence, the correct processes must agree.

\textbf{Liveness:} Under the assumptions of the XFT model, there always exists at least a majority of processes that are correct and synchronous. We see that in the case of a malicious leader, every correct process detects equivocation and sends \textit{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 \textit{Abort} messages or timeout. This can only happen a maximum of \(t\) times.

We show liveness by proving that a correct process in \(e' (> e)\) is able to collect \(N - t\) \textit{Status} messages and calculate a \textit{StatusLog} to find the \((c, e)\) values. If a \textit{CommitLog} contains a value \((c, e)\), this means it must have received \textit{Commit} messages for this value from \(\geq \frac{N}{2}\) processes (line 21, Algorithm 3). Since, every correct node broadcasts \textit{Commit} messages, it’s easy to see that all \(\frac{N}{2} > t\) nodes in the system contain the same value in their \textit{CommitLog} as well. Using this and the consistency property above we can see that in round \(e'\), if a correct process \(p_i\) receives the client request, then every correct process is able to collect the same \(N - t\) \textit{CommitLogs} and set \textit{StatusLog} to \((c, e)\). The process \(p_i\) now sends a value matching the \textit{StatusLog}. At least \(t + 1\) processes share \textit{Commit} messages and \textit{Decide} the value.

6 Evaluation
We evaluate Elpis by comparing it against four other consensus algorithms: XPaxos, PBFT, Zyzzyva and M\(^4\)Paxos. We take the latency measurements in a geo-replicated setup by setting up seven nodes using Amazon EC2 (Table 1) and throughput by placing the nodes in a single placement group us-east-1 so as to avoid skewing the data due to a greater variance in latencies in case of the geo-replicated setup. Additionally, all the clients are placed at respective nodes to simulate real-world implementations where requests are served by the closest data center.
We implemented Elpis, XFT and M²Paxos using the reliable messages toolkit Jgroups [9], in Java 8. We used the ClusterPartition MBean configuration and leveraged ASYM_ENCRYPT protocol configured with RSA 512 for asymmetric and AES/ECB/PKCS5Padding with 128 bit key size for symmetric encryption for Elpis. We implemented Zyzzyva using the BFT-SMaRt library (also in Java 8) and used the default highly optimized PBFT implementation. 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 PBFT, Zyzzyva, and XPaxos the primary is placed at Frankfurt. Additionally, the initial synchronous group for XPaxos consists of [Frankfurt, Ireland, Ohio] and [Frankfurt, Ireland, Ohio, Virginia] for the five node and seven node experiment respectively. 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 implemented a synthetic application that generates a workload which covers partitionable case with no inter-node conflicts (objects are locally accessed), to when command forwarding is required (a remote owner present for the objects), and to when multiple nodes have to acquire the ownership. Since, we are just testing the Consensus layer we do not execute any commands.

### 6.2 Latency

Figure 2 shows the comparison of latencies in a geo-replicated setup where the requests have 100% locality which implies that the requests in different regions access different objects. We notice that M²Paxos achieves the best response time for all regions due to a lower quorum size. Elpis expectedly achieves close but slightly higher latencies than M²Paxos due to the additional overhead of message digests and message broadcasts. This overhead is inherent to all the other protocols including XPaxos. XPaxos achieves best response time for Frankfurt (the primary). For clients present in all the other regions the request forwarding to Frankfurt results in higher response times. In contrast, the primary/owner for every client in the case of Elpis is present in the same region as the client, which provides lower response time. PBFT and Zyzzyva latencies are much higher due to the same reasons as XPaxos compounded by the requirement of a greater quorum size. Hence, at each step, the primary has to wait for more messages and thus incurs longer response times. In summary, Elpis achieves response times close to M²Paxos while promising better resilience.

### 6.3 Throughput

Figure 3 shows the throughput comparison in the single placement group us-east-1 as the system is pushed closer to saturation to achieve the maximum throughput possible. Elpis-x shows the performance under x% conflict where x%
implies that the commands issued by a node access objects out of which x% are shared with all the other nodes. Hence, Elpis-0% implies no conflict and Elpis-100% implies that all the clients across all nodes propose commands that access shared objects. PBFT and Zyzzyva peak at under 1x10^5 operations/sec due to complicated message patterns resulting in higher bandwidth usage. XPaxos and Elpis perform significantly better as they replicate requests to lower number of followers as compared to Zyzzyva (t acceptors vs all 3t nodes) and have less communication steps as compared to PBFT. Since Elpis relies on multiple owners the inherent load-balancing in the protocol results in higher throughput as compared to XPaxos where the primary becomes a bottleneck. As such even the 100% conflict case achieves higher throughput than XPaxos because for Elpis all the nodes are active as compared to XPaxos where only three (the active group consists of t + 1 nodes for XPaxos) participants are active.

Hence, the network is stressed until a single owner emerges for each object for the conflicting commands which orders all these commands. Hence, a lower percentage of requests are concurrently executed across nodes. However, even in the presence of 100% contention Elpis outperforms all the competitors as shown in Figure 3.

### 6.3.2 Performance under faults

In this section, we show the behavior of Elpis in the presence of faults (t). Figure 5 shows throughput as a function of time. A Byzantine process is simulated by adding a Byzantine layer in the Jgroups protocol stack under the Elpis implementation which when activated intercepts the Replicate messages and changes the req value in the message to an arbitrary value. At time 40 secs the message interception is activated at one of the nodes. This Byzantine node is detected by the other nodes and the clients start forward their requests to a different node after receiving t + 1 Abort messages. The node to which the request is forwarded is pre-configured for this experiment. As a result, throughput is decreased as this node is no longer completing the requests however, the remaining nodes continue to serve the client requests. In the presence of a single fault PBFT, Zyzzyva and XPaxos would start a view-change as a result of which the throughput would be effectively reduced to zero as no requests can be processed until a new leader emerges. At time 80 secs, message interception at another node is triggered. At this point, Elpis continues to decide the client requests via the active nodes while tolerating the maximum number of faults outside anarchy.

### 7 Related Work

PBFT [15] was the first efficient solution to solve consensus in the BFT model. The protocol requires 3t + 1 processes to tolerate t faults and uses a quorum of size 2t + 1 to return the result in five message delays. Zyzzyva [18] requires the same number of processes but achieves consensus in three message delays when no processes are slow or faulty. As such it requires bigger quorums of 3t + 1 for the single-phase
execution in contrast to $2t + 1$ required for PBFT. Zyzzyva is not particularly suited for heterogeneous networks like the ones in geo-replicated systems as even a single constrained network link at any node can make it switch to a slower two-phase operation. Furthermore, these protocols rely on a single leader which is a bottleneck for throughput in geo-replicated systems during normal operation and incurs downtime during view-change.

Elpis treats $M^2$Paxos [28] as an extended specification of Generalized Consensus and inherits a portion of data structures and interfaces. Elpis manages dependencies by mapping an object $o$ to a process $p_0$ similar to $M^2$Paxos. However, it innovates on how it manages contention. Agreeing on ownership of $o$ is a consensus problem in itself and $M^2$Paxos uses a mechanism similar to Phase 1 of Paxos [19].

As such, it does not guarantee liveness when multiple processes try to propose commands concurrently. This becomes even more evident when there exist cyclic dependencies in compound commands (that access multiple objects) such as $C_1 : \{a, b\}$, $C_2 : \{b, c\}$ and $C_3 : \{c, a\}$ where $a, b, c$ are the object ids.

In the case of Elpis, however, the nodes submit to the ownership transition taking place in the system as they learn about it and thereby converge on the contention set. The nodes learn in two phases. (1) If a node $p_1$ receives a NACK message for a Prepare or a Commit message from $p_2$, that implies $p_2$ must have sent an ACK to some node $p_1$ (could be $p_2$ itself). We piggyback this information ($tag(p_2)$, and $epoch$ of $ACK (p_2 \rightarrow p_1)$) in $Defer(p_2 \rightarrow p_1)$ (Algo 5 line 24, Algo 3 line 16) messages and let $p_1$ pick the best node to $deferTo$ (Algo 6 line 13). Without collision recovery (CR), however, this can result in a case where the set $\{p_1, p_2, p_3\}$ defers to $\{p_1, p_2, p_3\}$. (2) Using CR, we force at least a quorum of nodes (Algo 6 line 26) to Trust the same node to have a chance to acquire the ownership. If the set now is $\{p_1, p_2, p_3\}$ defer to $\{p_1, p_1, p_3\}$, $p_2$ would not retry ownership acquisition for $ab$, but $p_1$ and $p_3$ would (contending on $c$) and either acquire the ownership by completing Prepare and Commit or follow (1) and (2) as above to eventually have a single owner. Additionally, Elpis guarantees liveness and consistency in the presence of Byzantine faults, whereas $M^2$Paxos does not.

As part of the Cross Fault Tolerance (XFT) [24], model the authors propose XPaxos which uses $2t + 1$ processes. The protocol is executed by a synchronous group of $t + 1$ active processes with a fixed leader for the group. In the presence of faults, XPaxos transitions to a new group of $t + 1$ processes using a view – change mechanism. XPaxos provides similar performance to Paxos while providing higher reliability by tolerating Byzantine faults and is optimized for the $t = 1$ case but does not scale well with the number of faults.

Elpis uses 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 found. For higher $n$ and $t$, the number of such groups increases exponentially. However, in the worst case of Elpis, a client has to contact a maximum of $t + 1$ processes.

### Acknowledgments

We thank the anonymous reviewers for their valuable comments which have significantly improved the paper. This work is supported in part by NSF under grant CNS 1523558 and AFOSR under grant FA9550-15-1-0098.
References


