Generalized Consensus for Practical Fault Tolerance

Mohit Garg
Virginia Tech
mohitce@vt.edu

Sebastiano Peluso
Virginia Tech
peluso.sebastiano@gmail.com

Balaji Arun
Virginia Tech
balajia@vt.edu

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 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. 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 XPaxos 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

ACM Reference Format:
Mohit Garg, Sebastiano Peluso, Balaji Arun, and Binoy Ravindran. 2019. Generalized Consensus for Practical Fault Tolerance. In *Middleware '19: Middleware '19: 20th International Middleware Conference, December 8–13, 2019, Davis, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3361525.3361536

Middleware '19, December 8–13, 2019, Davis, CA, USA

ACM ISBN 978-1-4503-7009-7/19/12.
https://doi.org/10.1145/3361525.3361536

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^{2}Paxos [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,

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

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 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 $cs \bullet \sigma$, where σ 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 $\langle l, in\rangle$ 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 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 $\langle l, in\rangle$ 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, $CommitLog[l][in] = \langle r, e\rangle$ 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.
- *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 $\langle l, in\rangle$ 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 *epoch e* this

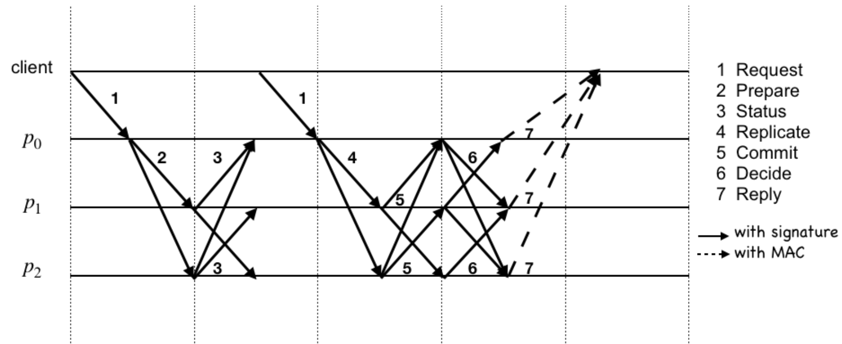


Figure 1. Elpis: p_0 sends a **Prepare** first to acquire the ownership or **Replicate** directly if it has the ownership of all the objects in the request. Client expects $t + 1$ matching **Reply** messages.

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 $\langle l, in \rangle$ pairs to the $\langle e, p_t \rangle$ 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 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 $req = \langle \text{REQUEST}, o, t, ls, c \rangle_{\sigma_c}$ 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 $LastDecided[l]$+1 and

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

1: **upon** C-PROPOSE(*Request r*)
2:     $Set\ ins \leftarrow \{\langle l, LastDecided[l] + 1 \rangle : l \in r.ls \wedge \nexists in : Decided[l][in] = c\}$
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:    $Set\ res \leftarrow \emptyset$
26:    **for all** $\langle l, in \rangle \in ins$ **do**
27:         $res \leftarrow res \cup \{Owners[l]\}$
28:    **return** res

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 $C - Propose$ 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 *ins* for proper *linearization* of commands during execution.

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

1: **function** *Bool* REPLICATE(*Request r, Set ins, Array eps*)
2:     **send** $\langle$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** $\forall\langle l, in\rangle \in ins$, $Rnd[l][in] \leq eps[l][in] \wedge$ (ISOWNER(p_j, ins) $= \top$ $\vee$STATUSLOG$[l][in] = \langle r, eps[l][in]\rangle$)) **then**
6:         COMMITPHASE(p_j, $-$, r, ins, eps)
7:     **else if** STATUSLOG$[l][in] \neq \bot \wedge$STATUSLOG$[l][in] \neq \langle r, eps[l][in]\rangle$ **then**
8:         **send** $\langle$ABORT, $r\rangle_{\sigma_{p_i}}$ **to** $r.c$
9:     **else**
10:         **for all** $\langle l, in\rangle \in ins \wedge (Rnd[l][in] > eps[l][in])$ **do**
11:             $Set\ deferTo \leftarrow Estimated[l][in]$
12:         **send** $\langle$COMMIT, $-$, $-$, ins, eps, $deferTo$, $NACK\rangle_{\sigma_{p_i}}$ **to** p_j

4.3.2 Replication Phase

In the *Replication phase* (Algorithm 2), 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 process p_j checks if the received message is for an epoch greater 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 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 *ins* and *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 *req.LS*. Otherwise, a *StatusLog* is constructed by collecting *Status* messages in the *Acquisition phase*. This request, epoch pair in the $StatusLog[l][in]$ is considered to be *valid*. This is discussed further in (Section 4.3.5). Hence, the $\langle r, eps[l][in]\rangle$ 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_j has already acknowledged a message for $eps[l][in]$ from the owner of the objects in the message and sends a *Nack* message along with the information about the last process it sent an *Ack* for the highest epoch for one or more $\langle l, in\rangle$ pairs (lines 8-10). This information returned with the *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 *ins* and for the epochs in *eps* in the presence of Byzantine processes. The request *req* 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_o, *Array toForce, Request req, Set ins, Array eps*)
2:     $Array\ toDecide$
3:     **for all** $\langle l, in\rangle \in ins : toForce[l][in] = \langle req', -\rangle : req' \neq NULL$ **do**
4:         $toDecide[l][in] \leftarrow req'$
5:     **if** $\forall\langle l, in\rangle \in ins$, $toDecide[l][in] = NULL$ **then**
6:         **for all** $\langle l, in\rangle \in ins$ **do**
7:             $toDecide[l][in] \leftarrow req$
8:     **send** $\langle$COMMIT, p_o, $toDecide$, ins, eps, $-$, $-\rangle_{\sigma_{p_i}}$ **to all** $p_k \in \Pi$
9:
10: **upon** COMMIT($\langle$ *Replica* p_o, *Array* $toDecide$, *Set* ins, *Array* eps, *Array* $deferTo$, *Value ack*$\rangle$) **from** p_j
11:     **if** $\forall\langle l, in\rangle \in ins$, $Rnd[l][in] \leq eps[l][in]$ **then**
12:         **for all** $\langle l, in\rangle \in ins$ **do**
13:             $e \leftarrow eps[l][in]$
14:             $Set\ commitList[l][in][eps[l][in]] \leftarrow commitList[l][in][e] \cup \{\langle toDecide[l][in], p_o, deferTo, ack, j\rangle\}$
15:         **if** $\forall\langle l, in\rangle \in ins$, $|commitList[l][in][e]| \geq sizeof(Quorum)$ **then**
16:             **if** $\exists\langle -, -, deferTo, NACK, -\rangle : commitList[l][in][e]$ **then**
17:                 $\forall\langle l, in\rangle \in ins$, $Set\ defers[l][in] \leftarrow deferTo : \langle -, -, ins, eps, deferTo, NACK, -\rangle$
18:                 **trigger** DEFER(ins, eps, $defers$)
19:         **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**
20:             $\forall\langle l, in\rangle \in ins$, $Owners[l] \leftarrow p_o$
21:             $\forall\langle l, in\rangle \in ins$, $CommitLog[l][in] \leftarrow \langle r, eps[l][in]\rangle$
22:             $\forall\langle l, in\rangle \in ins$, $Rnd[l][in] \leftarrow eps[l][in]$
23:             **send** $\langle$DECIDE, p_o, r, ins, $eps\rangle_{\sigma_{p_i}}$ **to all** $p_k \in \Pi$
24:         **else**
25:             **send** $\langle$ABORT, $req\rangle_{\sigma_{p_i}}$ **to** $req.c$
26:     **else**
27:         **for all** $\langle l, in\rangle \in ins \wedge (Rnd[l][in] > eps[l][in])$ **do**
28:             $Set\ deferTo \leftarrow Estimated[l][in]$
29:         **send** $\langle$COMMIT, $-$, $-$, ins, eps, $deferTo$, $NACK\rangle_{\sigma_{p_i}}$ **to** p_o
30: **function** *Array* VALID($Set\ ins$, $Set\ eps$, $Set\ Commits$)
31:     $Array\ toCommit$
32:     **for all** $\langle l, in\rangle \in ins$ **do**
33:         $e \leftarrow eps[l][in]$
34:         $Set\ requests \leftarrow \langle r', p'_o\rangle : \langle p'_o, r', -, -, -\rangle \in Commits[l][in][e]$
35:         **if** $\exists\langle r, p_o\rangle \ni |\langle r, p_o\rangle = \langle r', p'_o\rangle : requests| \geq sizeof(Quorum) \wedge r = req$ **then**
36:             $toCommit[l][in] \leftarrow \langle r, p_o\rangle$
37:     **return** $toCommit$

There are two cases for Byzantine processes: (1) Any t acceptors could send arbitrary request values rather than forwarding *req*, (2) The proposer which has the *ownership* can equivocate by sending *req* to some processes and some request *req'* to the rest of the processes. To tackle both of these scenarios a request r is *valid* if p_i 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 $\langle l, in\rangle$ 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 atleast $t+1$ 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 DECIDE(⟨Set toDecide, Set ins, Array eps⟩) from p_j
2:      for all ⟨l, in⟩ ∈ ins do
3:          e ← eps[l][in]
4:          Set decideList[l][in][e] ← decideList[l][in][e] ∪
                                      {⟨toDecide[l][in], j⟩}
5:      if ∀⟨l, in⟩ ∈ ins,∃r ∋|r = r′ : ⟨r′, −⟩ : Decides[l][in][eps[l][in]]|
                                      ≥ sizeof(Quorum) then
6:          for all ⟨l, in⟩ ∈ ins do
7:              if Decided[l][in] = NULL then
8:                  Decided[l][in] ← r
9:
10: upon (∃r : ∀l ∈ r.LS, ∃in : Decided[l][in] = r ∧
                                      in = LastDecided[l] + 1)
11:     Cstructs ← Cstructs • r
12:     Reply rep = C-DECIDE(Cstructs)
13:     send REPLY(reply) to r.c
14:     for all l ∈ r.LS do
15:         p_i.lastDecided[l] + +
```

4.3.4 Decision Phase

In the *Decision phase* (Algorithm 4) 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 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 Π(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 ← {⟨l, LastDecided[l] + 1⟩ : l ∈ c.LS ∧ ∄in :
    Decided[l][in] = c}
3:      Array eps
4:      ∀⟨l, in⟩ ∈ ins, eps[l][in] ← + + Epoch[l]
5:      ∀⟨l, in⟩ ∈ ins, Estimated[l][in] ← ⟨eps[l][in], Tag[p_i], p_i⟩
6:      send PREPARE(⟨ins, eps⟩) to all p_k ∈ Π
7:
8:  upon PREPARE(⟨Set ins, Array eps⟩) from p_j
9:      if ∀⟨l, in⟩ ∈ ins, Rnd[l][in] < eps[l][in] then
10:         ∀⟨l, in⟩ ∈ ins, Rnd[l][in] ← eps[l][in]
11:         Set decs ← {⟨l, in, CommitLog[l][in]⟩ : ⟨l, in⟩ ∈ ins}
12:         send STATUS(⟨ins, eps, decs, −, −⟩) to all p_k ∈ Π
13:     else
14:         for all ⟨l, in⟩ ∈ ins, Rnd[l][in] ≥ eps[l][in] do
15:             Set deferTo ← ⟨Rnd[l][in], Tag[p_i]⟩
16:         send STATUS(⟨ins, eps, decs, deferTo, NACK⟩) to p_j
17:
18: upon STATUS(⟨Set ins, Array eps, Array decs, Array deferTo, Value ack⟩)
    from p_j
19:     for all ⟨l, in⟩ ∈ ins do
20:         e ← eps[l][in]
21:         Set statusList[l][in][e] ← statusList[l][in][eps[l][in]] ∪
                                     {⟨decs[l][in], deferTo, ack, j⟩}
22:     if ∀⟨l, in⟩ ∈ ins,|statusList[l][in][eps[l][in]]| ≥ N − t then
23:         e ← eps[l][in]
24:         if ∃⟨−, deferTo, NACK, −⟩ : statusList[l][in][e] then
25:             ∀⟨l, in⟩ ∈ ins, Set defers[l][in] ← deferTo :
                                     ⟨−, deferTo, NACK, −⟩
26:             trigger DEFER(ins, eps, defers)
27:             return
28:     Set epochighest ← SELECT(ins, statuses)
29:     Set valids ← VALID(ins, statuses)
30:     if epochighest = ∅ ∧ valids = ∅ then
31:         ∀⟨l, in⟩ ∈ ins, StatusLog[l][in] ← ⟨req, eps[l][in]⟩
32:         if p_i = PROPOSER then
33:             REPLICATE(req, ins, eps)
34:     else if ∃ ⟨r, e, l, in⟩ ∋ ⟨r, e, l, in⟩ ∈ epochhighest
                ∧⟨r, e, l, in⟩ ∈ valids then
35:         StatusLog[l][in] ← ⟨r, e⟩ : ⟨r, e, l, in⟩
36:         if p_i = PROPOSER then
37:             Array toForce[l][in] ← ⟨r, e⟩ : ⟨r, e, l, in⟩
38:             REPLICATE(toForce[l][in], ins, eps)
39:             C-PROPOSE(req)
40:     else
41:         send ABORT(ins, eps, req) to all p_k ∈ Π, req.c
42: function Set SELECT(Set ins, Set statuses)
43:     Array toForce
44:     for all ⟨l, in⟩ ∈ ins do
45:         Epoch k ← max({k : ⟨−, k⟩ ∈ decs ∧ ⟨decs, −, −, −⟩ ∈
    statuses})
46:         Request r ← r : ⟨r, k⟩ ∈ decs ∧ ⟨decs, −, −, −⟩ ∈ statuses
47:         toForce ← ⟨r, k, l, in⟩
48:     return toForce
49:
50: function Array VALID(Set ins, Set statuses)
51:     Array valid
52:     for all ⟨l, in⟩ ∈ ins, statuses[l][in] = ⟨decs, −, −, j⟩ : ⟨decs, j⟩ do
53:         Set requests[j] ← ⟨v, k⟩ ∈ decs : ⟨decs, j⟩
54:         if ∃⟨r, k⟩∋|⟨r, k⟩ : requests|≥t then
55:             valid ← ⟨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 $N - t$), the process decides if there is a request to be committed from an aborted *Commit Phase* from an earlier epoch. For this, for all $\langle request, epoch \rangle$ entries present in the *CommitLog* received from a process p_j, the process first calculates the highest *epoch* values present in the entries for which a request is present and adds such $\langle request, epoch \rangle$ 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 $\langle request, epoch \rangle$ 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 $\langle request, epoch \rangle$ 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 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 $p_k: \langle -, tag, p_k \rangle \in$ *deferTo* is a process which is executing Elpis for an *epoch* equal to or higher than the *Epoch*[l] at p_i for some object l. 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

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

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

run is required and p_i concludes the recovery (line 8-9). However, if no such instance has been completed then p_i 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 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).

Upon receiving a Doubt message the process sets its *Estimated* to the value received in the defer message (line 22). 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.

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.

Stability: Only the owner of an object l in epoch e successfully commits the requests, and thus increments in. 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 $LastDecided[l]$ and only increment it when a command is decided for $\langle l, in\rangle$, the valid requests proposed by a correct owner of l in e would follow a complete order for in throughout the execution of the protocol and would not diverge for any correct process.

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.

Non-triviality: 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.

Consistency: 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 $CommitLog = \langle c', e'\rangle$ 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 $StatusLog = \langle c', e\rangle$ any correct process that commits c and sets its *Commit Log* 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.

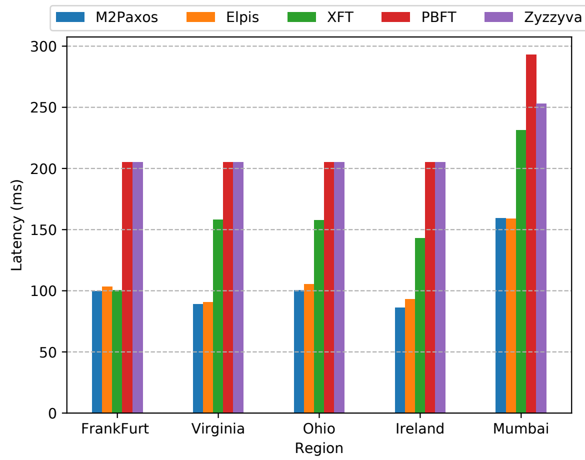
Liveness: Under the assumptions of the XFT model, there always exists at least a majority of processes that are correct and synchronous. We see that in the case of a malicious leader, every correct process detects equivocation and sends *Abort* messages to the client. After receiving $t+1$ messages the 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.

We show liveness by proving that a correct process in $e'(> e)$ is able to collect $N - t$ *Status* messages and calculate a $StatusLog$ to find the $\langle c, e\rangle$ values. If a $CommitLog$ contains a value $\langle c, e\rangle$, this means it must have received *Commit* messages for this value from $\frac{N}{2}$ processes (line 21 Algorithm 3). Since, every correct node broadcasts *Commit* messages, it's easy to see that all $\frac{N}{2} > t$ nodes in the system contain the same value in their $CommitLog$ 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 $N - t$ $CommitLogs$ and set $StatusLog$ to $\langle c, e\rangle$. The process p_i now sends a value matching the $StatusLog$. At least $t+1$ processes share *Commit* messages and *Decide* the value.

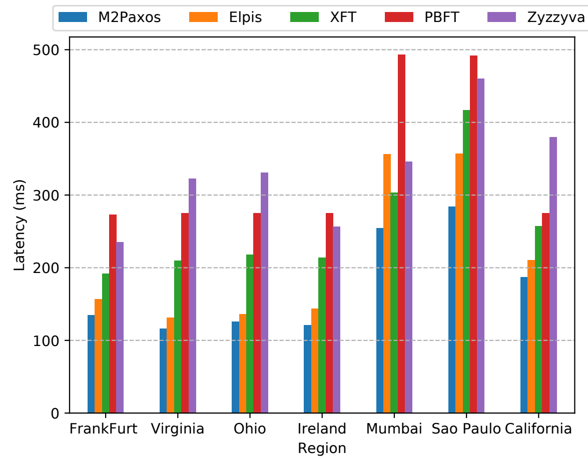
6 Evaluation

We evaluate Elpis by comparing it against four other consensus algorithms: XPaxos, PBFT, Zyzzyva and M^2Paxos . 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.

[1]Elpis TLA+ Specification: http://bit.ly/elpistla
[2]Elpis Collision Recovery TLA+: http://bit.ly/elpiscr



(a) 5 nodes



(b) 7 nodes

Figure 2. Latency comparison across regions.

We implemented Elpis, XFT and M^2Paxos 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).

AWS Region	Name	Virginia	Ohio	Frankfurt	Ireland	Mumbai	California	Sao Paulo
`us-east-1`	**Virginia**	-	11	88	74	182	59	140
`us-east-2`	**Ohio**	10	-	97	84	191	49	149
`eu-central-1`	**Frankfurt**	88	97	-	21	109	145	226
`eu-west-1`	**Ireland**	74	84	39	-	122	129	183
`ap-south-1`	**Mumbai**	182	191	109	120	-	241	320
`us-west-1`	**California**	59	49	145	129	241	-	197
`sa-east-1`	**Sao Paulo**	140	149	226	183	320	197	-

Table 1. Average inter-region RTT latencies in ms.

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 M^2Paxos achieves the best response time for all regions due to a lower quorum size. Elpis expectedly achieves close but slightly higher latencies than M^2Paxos 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 M^2Paxos 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%

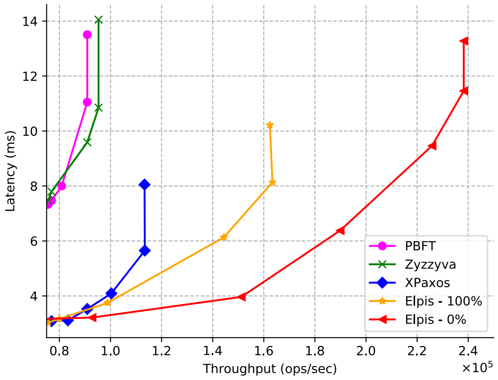


Figure 3. Latency vs. throughput in a cluster.

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 synchronous group consists of $t + 1$ nodes for XPaxos) participants are active.

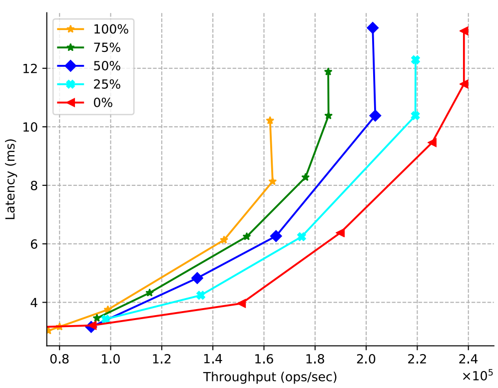


Figure 4. Performance under contention.

6.3.1 Performance under contention

Figure 4 shows the throughput vs latency comparison for Elpis as the percentage of contention is varied from 0% to 100%. Throughput decreases as the percentage of contention increases. In the case of contention, collision recovery is invoked which increases the number of messages exchanged. 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

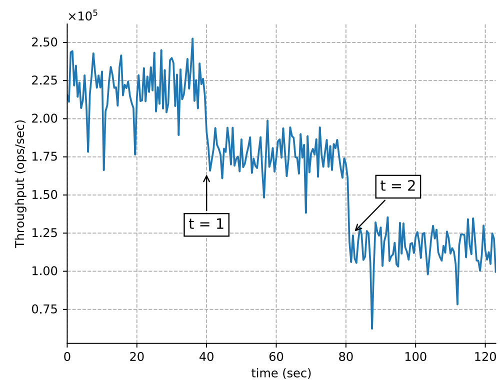


Figure 5. Performance under faults. Initially, $t = 0$.

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*

Protocol	PBFT	Zyzzyva	M²Paxos	XPaxos	Elpis
Resilience	$t < n/3$	$t < n/3$	$t^{\star} < n/2$	$t < n/2$	$t < n/2$
Best case communication steps	5	3	2	2	3
Base case in the absence of	t_{nc}	t_{nc}, t_p, $contention$	t_c, t_p, $contention$	t_{nc}, t_c t_p	t_{nc}, t_c, t_p, $contention$
Slow-path steps during contention	-	2	$2^{\dagger}$	-	2
Leader	Single	Single	Multi-Leader	Single	Multi-Leader
Fault Model	BFT	BFT	CFT	XFT	XFT

Table 2. Comparison of existing protocols and Elpis.
⋆ only tolerates crash faults † may require multiple runs of this step.

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 M²Paxos [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_o similar to M²Paxos . However, it innovates on how it manages contention. Agreeing on ownership of o is a consensus problem in itself and M²Paxos 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_j (could be p_2 itself). We piggyback this information ($tag(p_j)$, and *epoch* of ACK ($p2 \rightarrow p_j$)) in $Defer(p2 \rightarrow p1)$ (Algo 5 line 24, Algo 3 line 16) messages and let $p1$ 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 $\{p_3, p_1, p_2\}$. (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 M²Paxos 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.

8 Conclusion

This paper presented Elpis, the first multi-leader XFT protocol that overcomes the drawbacks of XPaxos, a single-leader protocol. Elpis implements generalized consensus. By assigning different and independent objects to different processes, such that the need for ordering is limited to local scope, each governed by one of the processes, 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 communication delays while ownership transfer adds an additional delay. The efficacy of this approach is further validated by evaluation as it leads to significant performance gains over XPaxos and other BFT protocols. 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. Hence, Elpis is an attractive option for building geo-replicated fault-tolerant systems as not only does it provides better performance than BFT protocols but, it also offers higher reliability than CFT protocols.

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

[1] 2012. AWS Service Event in the US-East Region: October 22, 2012. (2012). https://aws.amazon.com/message/680342/

[2] 2018. cockroach: CockroachDB - the open source, cloud-native SQL database. https://github.com/cockroachdb/cockroach original-date: 2014-02-06T00:18:47Z.

[3] 2018. etcd: Distributed reliable key-value store for the most critical data of a distributed system. https://github.com/coreos/etcd original-date: 2013-07-06T21:57:21Z.

[4] 2019. Google App Engine: 02 January 2019. (2019). https://status.cloud.google.com/incident/appengine/19001

[5] 2019. Google Compute Engine: November 05, 2018. (2019). https://status.cloud.google.com/incident/compute/18012

[6] 2019. Home | YugaByte DB. https://www.yugabyte.com/. Accessed: 2019-09-04.

[7] Balaji Arun, Sebastiano Peluso, Roberto Palmieri, Giuliano Losa, and Binoy Ravindran. 2017. Speeding up Consensus by Chasing Fast Decisions. *arXiv:1704.03319 [cs]* (April 2017). http://arxiv.org/abs/1704.03319 arXiv: 1704.03319.

[8] Pierre-Louis Aublin, Rachid Guerraoui, Nikola Kneevi, Vivien Quéma, and Marko Vukoli. 2015. The Next 700 BFT Protocols. *ACM Trans. Comput. Syst.* 32, 4 (Jan. 2015), 12:1–12:45. https://doi.org/10.1145/2658994

[9] Bela Ban. 2002. JGroups, a toolkit for reliable multicast communication. (2002).

[10] Alysson Neves Bessani and Marcel Santos. 2011. *Bft-smart-high-performance byzantine-faulttolerant state machine replication.*

[11] Eric A Brewer. 2000. Towards robust distributed systems. In *PODC*, Vol. 7.

[12] Mike Burrows. 2006. The Chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th symposium on Operating systems design and implementation*. USENIX Association, 335–350.

[13] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, et al. 2011. Windows Azure Storage: a highly available cloud storage service with strong consistency. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. ACM, 143–157.

[14] Apache Cassandra. 2014. Apache cassandra. *Website. Available online at http://planetcassandra. org/what-is-apache-cassandra* (2014), 13.

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

[16] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. 2013. Spanner: Googles Globally Distributed Database. *ACM Trans. Comput. Syst.* 31, 3 (Aug. 2013), 8:1–8:22. https://doi.org/10.1145/2491245

[17] Patrick Hunt, Mahadev Konar, Flavio P Junqueira, and Benjamin Reed. 2010. ZooKeeper: Wait-free coordination for Internet-scale systems. (2010), 14.

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

[19] Leslie Lamport. 2001. Paxos made simple. *ACM Sigact News* 32, 4 (2001), 18–25.

[20] Leslie Lamport. 2002. *Specifying systems: the TLA+ language and tools for hardware and software engineers.* Addison-Wesley Longman Publishing Co., Inc.

[21] Leslie Lamport. 2005. Generalized Consensus and Paxos. (2005), 63.

[22] Leslie Lamport. 2006. Fast Paxos. *Distributed Computing* 19 (Oct. 2006). https://www.microsoft.com/en-us/research/publication/fast-paxos/

[23] Leslie Lamport, Robert Shostak, and Marshall Pease. 1982. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems* 4, 3 (July 1982), 382–401. https://doi.org/10.1145/357172.357176

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

[25] J-P Martin and Lorenzo Alvisi. 2006. Fast byzantine consensus. *IEEE Transactions on Dependable and Secure Computing* 3, 3 (2006), 202–215.

[26] Iulian Moraru, David G. Andersen, and Michael Kaminsky. 2013. There is more consensus in Egalitarian parliaments. ACM Press, 358–372. https://doi.org/10.1145/2517349.2517350

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

[28] S. Peluso, A. Turcu, R. Palmieri, G. Losa, and B. Ravindran. 2016. Making Fast Consensus Generally Faster. In *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 156–167. https://doi.org/10.1109/DSN.2016.23

[29] Fred B Schneider. 1993. Replication management using the state-machine approach, Distributed systems. (1993).

[30] Swaminathan Sivasubramanian. 2012. Amazon dynamoDB: a seamlessly scalable non-relational database service. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. ACM, 729–730.

[31] Mohammad Reza Khalifeh Soltanian and Iraj Sadegh Amiri. 2016. Chapter 1 - Introduction. In *Theoretical and Experimental Methods for Defending Against DDOS Attacks*, Mohammad Reza Khalifeh Soltanian and Iraj Sadegh Amiri (Eds.). Syngress, 1 – 5. https://doi.org/10.1016/B978-0-12-805391-1.00001-8