SeeMoRe: A Fault-Tolerant Protocol for Hybrid Cloud Environments

Mohammad Javad Amiri   Sujaya Maiyya   Divyakant Agrawal   Amr El Abbadi
Department of Computer Science, University of California Santa Barbara
Santa Barbara, California
{amiri, sujaya_maiyya, agrawal, amr}@cs.ucsb.edu

ABSTRACT

Large scale data management systems utilize State Machine Replication to provide fault tolerance and to enhance performance. Fault-tolerant protocols are extensively used in the distributed database infrastructure of large enterprises such as Google, Amazon, and Facebook, as well as permissioned blockchain systems like IBM's Hyperledger Fabric. However, and in spite of years of intensive research, existing fault-tolerant protocols do not adequately address all the characteristics of distributed system applications. In particular, hybrid cloud environments consisting of private and public clouds are widely used by enterprises. However, fault-tolerant protocols have not been adapted for such environments. In this paper, we introduce *SeeMoRe*, a hybrid State Machine Replication protocol to handle both crash and malicious failures in a public/private cloud environment. SeeMoRe considers a private cloud consisting of non-malicious nodes (either correct or crash) and a public cloud with both Byzantine faulty and correct nodes. SeeMoRe has three different modes which can be used depending on the private cloud load and the communication latency between the public and the private cloud. We also introduce a dynamic mode switching technique to transition from one mode to another. Furthermore, we evaluate SeeMoRe using a series of benchmarks. The experiments reveal that SeeMoRe's performance is close to the state of the art crash fault-tolerant protocols while tolerating malicious failures.

1. INTRODUCTION

Today's enterprises mostly rely on cloud storage to run their business applications. Cloud computing has many benefits in terms of cost savings, scalability, and easy access [59]. However, storing data on a single cloud may reduce robustness and performance [14, 26, 31]. Robustness is the ability to ensure availability (liveness) and one-copy semantics (safety) despite failures, while performance deals with the response time of requests (latency) and the number of processed requests per time unit (throughput) [7].

Fault-tolerant protocols are designed to satisfy both robustness and performance concerns using State Machine Replication (SMR) [35] techniques. SMR regulates the deterministic execution of client requests on multiple copies of a server, called replicas, such that every non-faulty replica must execute every request in the same order [46] [35].

Large scale data management systems utilize SMR to provide fault tolerance and to increase the performance of the system. Fault-tolerant protocols are extensively used in distributed databases such as Google's Spanner [20], Amazon's

Dynamo [23], and Facebook's Tao [13], thus highlighting the critical role of SMR in data management. SMR is also the core component in the more recently developed, highly popular set of technologies – *Blockchain*. In particular, permissioned blockchain systems extensively use fault-tolerant protocols to establish consensus on the order of transactions between a set of known, identified nodes that do not fully trust each other.

While large enterprises might have their own Geo-replicated fault-tolerant cloud storage around the world, smaller enterprises may only have a local private cloud that is lacking in resources to guarantee fault tolerance. One solution is to store all the data on third-party public cloud providers [5, 8, 57]. Public clouds provide several advantages like elasticity and durability, but they often suffer from security concerns, e.g., malicious attacks [2]. Whereas private clouds are considered more secure but may not provide sufficient elasticity and durability. The trustworthiness of a private cloud allows an enterprise to build services that can utilize crash fault-tolerant protocols, i.e., protocols that make progress when a bounded number of replicas only fail in a benign manner, for example by either crashing or being unresponsive. But due to lack of private resources, if a third-party public cloud is used, the nodes of the public cloud *may* behave maliciously, in which case a more robust fault-tolerant protocol is needed that allows the system to continue operating correctly, even when some replicas exhibit arbitrary, possibly malicious behavior. Current Byzantine fault-tolerant protocols (e.g., PBFT [15]) introduce significant communication and latency overheads in order to tolerate failures since they consider all failures as malicious.

An alternative solution to storing all the data in public cloud is to use a hybrid cloud storage system consisting of both private and public clouds [26]. In a hybrid cloud, the nodes in the private cloud are trusted and may crash but do not behave maliciously whereas the nodes in the public cloud(s) might be malicious. Hybrid clouds address the *security* concerns of using *only* public clouds by giving enterprises the ability to still use their private clouds. In addition, storing data on multiple clouds is more *reliable*, e.g., if a cloud outage happens, the system might still be able to process requests. Moreover, while the small private cloud may represent a *scalability* bottleneck, the system can rent as many servers as required wants from the public clouds. The benefits of a hybrid cloud necessitates designing new protocols that can leverage the trust of private clouds and the scalability of public clouds.

Despite years of intensive research, existing fault-tolerant

protocols do not adequately address all the characteristics of hybrid cloud environments. On one hand, the existing Byzantine fault-tolerant protocols [7, 15, 32, 34, 40, 41, 56] do not distinguish between crash and malicious failures, and consider all failures as malicious, thus incurring a higher cost in terms of performance. On the other hand, the hybrid protocols [18, 47] that have been designed to tolerate both crash and malicious failures, make no assumption on *where* the crash or malicious failures may occur. As a result using these protocols in a hybrid cloud environment, where all machines in the private cloud are known to be trusted while machines in the public cloud could be compromised and hence malicious, results in an unnecessary performance overhead.

A hybrid fault tolerant protocol can be beneficial in a distributed database such as Spanner [20]. Spanner consists of a layered architecture where transaction management is oblivious to the underlying fault-tolerant protocol used. If a small enterprise wants to deploy Spanner in a hybrid cloud, it can plug a hybrid fault-tolerant protocol instead of the crash fault-tolerant protocol used in Spanner to achieve more reliable replication in this heterogeneous cloud setting. Permissioned blockchain systems can also take advantage of a hybrid protocol. For example, in IBM's Hyperledger Fabric [4] the fault-tolerant (consensus) protocol is pluggable. Hence, if some nodes are trusted but not all, Fabric can benefit from a hybrid fault-tolerant protocol.

In this paper, we present *SeeMoRe*[1]: a State Machine Replication protocol that leverages the localization of crash and malicious failures in a *hybrid cloud environment*. SeeMoRe considers a private cloud consisting of trusted replicas, a subset of which may fail-stop, and a public cloud where a subset of the replicas may behave maliciously. SeeMoRe takes explicit advantage of this knowledge to improve performance by reducing the number of communication phases and messages exchanged and/or the number of required replicas. SeeMoRe has three different modes of operation which can be used depending on the load on the private cloud, and the latency between the public and the private cloud. We also introduce a dynamic and elastic technique to transition from one mode to another.

A key contribution of this paper is to show how being aware of *where* different types of failures (crash and malicious) may occur in hybrid cloud environments, results in designing more efficient protocols. In particular, this paper makes the following contributions:

- A model for hybrid cloud environments is presented which can be used by enterprises that do not have enough servers in their trusted private cloud to run fault-tolerant protocols and gives them the option of renting from untrusted public clouds.

- SeeMoRe, a hybrid protocol that tolerates both crash and malicious failures, is developed in three different modes. Being aware of where the crash faults may occur and where the malicious faults can occur results in reducing the number of communication phases, messages exchanged and/or required replicas. In addition,

---

[1] SeeMoRe is derived from Seemorq, a benevolent, mythical bird in Persian mythology which appears as a peacock with the head of a dog and the claws of a lion. Seemorq in Persian literature also refers to a group of birds who flew together to achieve a common goal.

a technique to dynamically switch between different modes of SeeMoRe is presented.

The rest of this paper is organized as follows. Section 2 presents related work. The system model is introduced in Section 3. Section 4 presents a method to compute the required number of replicas from a public cloud. The design of SeeMoRe is proposed in Section 5, elaborating on the three modes, as well as the dynamic switching of modes. Section 6 shows the performance evaluation, and Section 7 concludes the paper.

2. RELATED WORK

State machine replication (SMR) is a technique for implementing a fault-tolerant service by replicating servers [35]. Several approaches [36, 43, 46] generalize SMR to support crash failures among which Paxos is the most well-known [36]. Paxos guarantees safety in an asynchronous network using $2f+1$ processors despite the simultaneous crash failure of any f processors. Many protocols are proposed to either reduce the number of phases, e.g., Multi-Paxos which assumes the leader is relatively stable or Fast Paxos [37] and Brasileiro et al. [12] which add f more replicas, or reduce the number of replicas, e.g., Cheap Paxos [38] which tolerates f failures with $f+1$ active and f passive processors.

Byzantine fault tolerance refers to servers that behave arbitrarily after the seminal work by Lamport, et al. [39]. Practical Byzantine fault tolerance protocol (PBFT) [15] is one of the first and probably the most instructive state machine replication protocol to deal with Byzantine failures. Although practical, the cost of implementing PBFT is quite high, requiring at least $3f + 1$ replicas, 3 communication phases, and a quadratic number of messages in terms of the number of replicas. Thus, numerous approaches have been proposed to explore a spectrum of trade-offs between the number of phases/messages (latency), number of processors, the activity level of participants (replicas and clients), and types of failures.

On latency, FaB [41] and Bosco [49] reduce the communication phases by adding more replicas. Speculative protocols, e.g., Zyzzyva [34], HQ [22], and Q/U [1], also reduce the communication by executing requests without running any agreement between replicas and optimistically rely on clients to detect inconsistencies between replicas.

To reduce the number of replicas, some approaches rely on a trusted component (a counter in A2M-PBFT-EA [17], MinBFT [54] and, EBAWA [53], a hypervisor [45], or a whole operating-system instance [21]) that prevents a faulty replica from sending conflicting (i.e., asymmetric) messages to different replicas without being detected.

In addition, optimistic approaches reduce the required number of replicas during the normal-case operation by either utilizing the Cheap Paxos [38] solution and keeping f replicas in a passive mode (REPBFT [24]), or by separating agreement from execution [58]. In ZZ [56] both passive replicas and separating agreement from execution are employed. Note that all these approaches need $3f + 1$ replicas upon occurrence of failures. REMINBFT [24], SPARE [25], and CheapBFT [32] use a trusted component to reduce the network size to $2f + 1$ and then keep f of those replicas passive during the normal-case operation. In contrast to optimistic approaches, robust protocols (Prime [3], Aardvark [19], Spinning [52], RBFT [6]) consider the system to be

under attack by a very strong adversary and try to enhance the performance of the protocol during periods of failures.

In this paper we focus on Hybrid fault tolerance. Such consensus with multiple failure modes were initially addressed in synchronous protocols [33, 42, 48, 51]. Many recent efforts have focused on partial synchrony, a technique that defines a threshold on the number of slow (partitioned) processes. Let m, c, and s denote the number of malicious, crash, and slow servers respectively, VFT [44] and XFT [40] require $2m + c + min((m+c), s) + 1$ and $2(m + c + s) + 1$ servers respectively, and SBFT [30] needs $3m + 2c + 1$ servers as the minimum size of the network from which $3m + c + 1$ are participating in each quorum. VFT is very similar to PBFT regarding the number of phases and massage exchanges. XFT however, optimistically assumes that an adversary cannot fully control the Byzantine nodes and as a result, reduces the phases of communication and message exchanges. SBFT also reduces the number of message exchanges by assuming the adversary controls only the crash failures.

Finally, Scrooge [47] and UpRight [18] are two asynchronous hybrid approaches that use optimistic solutions. Scrooge [47] uses a speculative technique to reduce the latency in the presence of $4m + 2c$ replicas. UpRight [18], which is the closest protocol to SeeMoRe, requires $3m + 2c + 1$ nodes as the minimum network size from which $2m + c + 1$ are required to participate in each communication quorum. In addition, UpRight utilizes the agreement routines of PBFT [15], Aardvark [19], and Zyzzyva [34] and similar to [58], separates agreement from execution. However, UpRight is not aware which nodes may crash and which may be malicious, therefore, does not take advantage of this knowledge by placing particular processes executing specific protocol roles on crash-only or malicious sites. On the other hand, SeeMoRe knows where the crash or malicious faults may occur, thus, it either reduces the number of communication phases and message exchanges by placing the primary in the crash-only private cloud, or decreases the number of required nodes by placing the primary in the untrusted public cloud.

Storing data on multiple clouds to enhance fault tolerance is addressed for both crash (ICStore [8], SPANStore [57]) and malicious (DepSky [9], SCFS [10]) failures which rely on $2f + 1$ and $3f + 1$ servers respectively. DAPCC [55] assumes a synchronous cloud environment and solves the consensus in a dual failure mode with $n \geq \lfloor (n-1)/3 \rfloor + 2m + c$ nodes (similar to [48]). However, DAPCC needs $\lfloor (n-1)/3 \rfloor + 1$ rounds of communication. Hypris [26] on the other hand reduces the number of required servers to $2f + 1$ ($f + 1$ when the system is synchronous and no faults happen) by keeping the metadata in a private cloud assumed to be partially synchronous.

3. SYSTEM MODEL

In this section, we introduce the system model wherein an application layer, such as a distributed database management system, relies on a replication service to store copies of data across a cloud environment consisting of private and public clouds. The replication service aims to replicate the data across some trusted and some untrusted servers. The replicas can be geo-distributed to provide low data access latency to clients across the globe or they can be geographically confined to tolerate both crash or malicious failures. Such a replication service can use SeeMoRe and we specify the assumptions on which SeeMoRe is built in this section.

3.1 Basic Assumptions

We consider a hybrid failure model that admits both crash and malicious failures where crash failures may occur in the private cloud and malicious failures may *only* occur in the public cloud. Note that a malicious failure can encompass a crash failure but since the trust assumptions are low, we do not distinguish between a crash or a malicious failure in the public cloud. In a crash failure model, replicas operate at arbitrary speed, may fail by stopping, and may restart, however they may not collude, lie, or otherwise attempt to subvert the protocol. Whereas, in a malicious failure model, faulty nodes may exhibit arbitrary, potentially malicious, behavior. We assume that a strong adversary can coordinate malicious nodes and delay communication to compromise the replicated service. However, the adversary cannot subvert standard cryptographic assumptions about collision-resistant hashes, encryption, and signatures, e.g., the adversary cannot produce a valid signature of a non-faulty node.

Each pair of replicas is connected with point-to-point bidirectional communication channels and each client can communicate with any replica. Network links are pairwise authenticated, which guarantees that a malicious replica cannot forge a message from a correct replica, i.e., if replica i receives a message μ in the incoming link from replica j, then replica j sent message μ to i beforehand.

We use the state machine replication algorithm [46] where replicas agree on an ordering of incoming requests and all replicas execute the requests in the same order. Our system ensures safety in an asynchronous network that can drop, delay, corrupt, duplicate, or reorder messages. Liveness is guaranteed only during periods of synchrony when there is a finite but possibly unknown bound on message delivery time. The model puts no restrictions on clients, except that their numbers must be finite, however, safety and liveness require some constraints on the number of faulty servers.

Depending on the role of a node and the type of message it wants to send, messages may contain public-key signatures and message digests [15]. A *message digest* is a numeric representation of the contents of a message produced by collision-resistant hash functions. Message digests are used to protect the integrity of a message and detect changes and alterations to any part of the message. We denote a message μ signed by replica r as $\langle \mu \rangle_{\sigma_r}$ and the digest of a message μ by $D(\mu)$. For signature verification, we assume that all machines have the public keys of all other machines. In Section 5, we explain when signatures and digests are needed.

3.2 Quorum and Network Size

We consider a cloud environment consisting of private and public clouds. The system is an asynchronous distributed system containing a set of N replicas where S of them exist in a private cloud and P of them are in a public cloud. Nodes in the private cloud are non-malicious: either non-faulty (correct) nodes or crashed nodes. The bound on the maximum number of crashed nodes is assumed to be c. Similarly, nodes in the public cloud can be either non-faulty nodes or Byzantine nodes. The bound on the maximum number of Byzantine nodes is m. We call the nodes in the private cloud *trusted* and the nodes in the public cloud *untrusted*. All the clients and the replicas know which replicas are trusted and which are untrusted.

Failures are divided into two disjoint classes: malicious and crash faults. In crash fault-tolerant models, e.g., Paxos

[36], given that c nodes can crash, a request is replicated to a quorum consisting of at least $c+1$ nodes to provide fault tolerance and to guarantee that a value once decided will remain decided in spite of failures (safety). Furthermore, any two quorums intersect on at least *one* node and as a result, $2c+1$ is the minimum number of nodes that allows an asynchronous system to provide the safety and liveness properties.

In the Byzantine failure models, e.g., PBFT [15], given that m nodes can be malicious, the quorum size should be at least $2m+1$ to ensure that non-faulty replicas outnumber the malicious ones, i.e., a request is replicated in enough non-faulty nodes to guarantee safety in the presence of m failures. This implies that any two quorums intersect with at least $m+1$ nodes to ensure one correct node in the intersection, thus the minimum network size is $3m+1$ [11].

Likewise, in the hybrid model the quorum size must include at least $2m+c+1$ nodes to tolerate c crash and m malicious failures [18]. This also guarantees that the intersection of any two quorums has to be at least $m+1$ nodes. Since the quorum size is $2m+c+1$ and the intersection of any two quorum Q and Q' is $m+1$ nodes, $|Q|+|Q'| = N+m+1 = 4m+2c+2$, thus the (minimum) network size, N, will be [18]

$$N = 3m + 2c + 1 \tag{1}$$

Intuitively, if there are f failures (of any type) in a network, the network size has to be at least f larger than the quorum size, as any network with smaller size could lead to a deadlock situation where none of the f faulty servers are participating. Since, $f = m + c$ and the quorum size Q is $2m + c + 1$, the network size should be at least $Q + f$ i.e., $3m + 2c + 1$.

4. PUBLIC CLOUD

The hybrid failure model presented in Section 3 can be used by enterprises that own private clouds with a limited number of trusted servers which is *insufficient* to run a fault-tolerant protocol. This model gives them the option of renting from untrusted public clouds. In this section, we present two methods to identify the number of servers an enterprise needs to rent from a public cloud.

A business that owns an insufficient number of trusted servers (servers that might crash but are not malicious) needs to rent more servers from some untrusted public clouds to satisfy the minimum network size constraints ($3m+2c+1$) of the protocol. Public clouds might provide some statistics that show the percentage of faulty nodes in the cloud. If there is no information on the type of failures, i.e. crash or malicious, within the public cloud, we consider all the faulty nodes as malicious and assume that the ratio of malicious nodes in public cloud (m) to the size of public cloud ($\mathcal{P}$) is known and is equal to $\alpha = \frac{m}{\mathcal{P}}$. Note that, we assume a uniform distribution of malicious nodes in public cloud, i.e., in any set $\pi \subseteq \mathcal{P}$, at most $\alpha \times \pi$ nodes are malicious.

Given the size of the private cloud S, the bound on the maximum number of crashed nodes c in the private cloud, and the ratio α of malicious nodes (m) in the public cloud to the size of the public cloud ($\mathcal{P}$), the task is to identify the required number of nodes P to be rented from the public cloud that allows satisfying the protocol constraints.

The total number of nodes in the network is $N = S + P$. Given our assumption of α, we get $m = \alpha P$. Replacing m

in Equation 1, we get $N = 3\alpha P + 2c + 1$, which means, $(3\alpha - 1)P = S - (2c + 1)$, thus:

$$P = \lceil \frac{S - (2c+1)}{3\alpha - 1} \rceil \tag{2}$$

As an example consider the situation that a private cloud has 2 servers where one of them might be faulty, i.e., $S = 2$, and $c = 1$, and we want to rent servers from a public cloud with $\alpha = 0.3$. Here, $P = \frac{2-2-1}{3*0.3-1} = \frac{-1}{-0.1} = 10$, which means we need to rent 10 servers from the public cloud to provide the safety constraints of the replication protocol.

In Equation 2, if the size of the private cloud (S) is equal or greater than $2c+1$, then the private cloud does not need to rent any nodes and can run a crash-fault tolerant protocol like Paxos [36] by itself. If there is no private cloud ($S = 0$) or all the nodes in the private cloud are faulty ($S = c$), using the private cloud has no advantage and it is more reasonable to rent all the required nodes from the public cloud and run a Byzantine fault-tolerant protocol in the public cloud. However, if $c < S < 2c+1$, renting some nodes from a public cloud might be helpful.

Similarly, if $\alpha \geq \frac{1}{3}$, (i.e., more than $\frac{1}{3}$ of the nodes in the public cloud are malicious), then the public cloud cannot satisfy the network size constraint for Byzantine fault-tolerance. Hence, an enterprise will need to rent servers if its private cloud size, S, is between $c+1$ and $2c$, and it can rent servers from public cloud providers that satisfy $\alpha < \frac{1}{3}$. It should be noted that even if the size of the private cloud is equal or greater than $2c + 1$, and the public cloud does not satisfy the $\alpha < \frac{1}{3}$ constraint, an enterprise might still rent some replicas from the public cloud for load balancing purposes.

Note that Equation 2 can easily be extended to address the situation where the public cloud provides information on the ratio of both malicious and crash nodes, i.e., the ratio of malicious nodes to the size of public cloud ($\alpha = \frac{m}{\mathcal{P}}$) as well as the ratio of crash nodes to the size of public cloud ($\beta = \frac{c}{\mathcal{P}}$) are known. In such a situation, Equation 2 can be rewritten as:

$$P = \lceil \frac{S - (2c+1)}{3\alpha + 2\beta - 1} \rceil \tag{3}$$

This method, which identifies the required number of nodes from a public cloud, assumes a uniform distribution of faulty nodes in the public cloud. However, public clouds might not guarantee a uniform distribution of α and alternatively specify the maximum number of concurrent failures in a cluster of rental nodes explicitly. In such a setting, even if an enterprise rents a portion of that cluster, there is no guarantee that the percentage of the faulty nodes in that portion is equal to the percentage of the faulty nodes in the entire cluster. For example, a public cloud might guarantee that in a cluster of 10 nodes, at most two concurrent failures can occur. Nonetheless, if an enterprise rents only two nodes from that cluster, both of them might fail at the same time. Assuming that the number of concurrent malicious failures in a (cluster of nodes in a) public cloud is given and equal to M, we would want to identify the required number of nodes P to rent from such a public cloud. The total number of nodes in the network is $N = 3m + 2c + 1 = S + P$ and there is no guarantee on a uniform distribution of malicious nodes in the public cloud, thus $m = M$. Hence, the required number of nodes is $P = (3M + 2c + 1) - S$.

Similar to the first method, if the public cloud distinguishes between different types of failures and provides information on the number of both crash and malicious failures, given as C and M, the required number of nodes from the public cloud is $P = (3M + 2C + 2c + 1) - S$ where c, similar as before, is the number of crash failures in the private cloud.

Finally, it should be noted that both methods of identifying the public cloud size can be generalized to multiple public clouds as well. In Such a settings, since different public clouds might have different ratio (number) of faulty nodes, the equation might have multiple solutions.

5. SeeMoRe

In this section we present SeeMoRe, a hybrid fault-tolerant consensus protocol for a public/private cloud environment that tolerates m Byzantine failures in the public and c crash failures in the private cloud.

SeeMoRe is inspired by the known Byzantine fault-tolerant protocol PBFT [32]. In PBFT, as can be seen in Figure 1(d), during a normal case execution, a client sends a request to a (primary) replica, and the primary broadcasts a pre-prepare message to all replicas. Once a replica receives a valid pre-prepare message, it broadcasts a prepare message to all other replicas. Upon collecting $2f$ valid matching prepare messages (including its own message) that are also matched to the pre-prepare message sent by the primary, each replica broadcasts a commit message. In this stage, each replica knows that all non-faulty replicas agree on the contents of the message sent by the primary. Once a replica receives $2f + 1$ valid matching commit messages (including its own message), it executes the request and sends the response back to the client. Finally, the client waits for $f + 1$ valid matching responses from different replicas to make sure at least one correct replica executed its request. PBFT also has a view change routine that provides liveness by allowing the system to make progress when the primary fails.

SeeMoRe consists of *agreement* and *view change* routines where the agreement routine orders requests for execution by the replicas, and the view change routine coordinates the election of a new primary when the current primary is faulty.

The algorithm, similar to most fault-tolerant algorithms, is a form of state machine replication. In such approaches, a service is replicated across a group of servers in a distributed system. Each server maintains a set of state variables, which are modified by a set of "atomic" and "deterministic" operations. Operations are *atomic* if they do not interfere with each other and *deterministic* if the same operation executed in the same initial state generates the same final state. Also, the initial state must be the same in all replicas.

The algorithm has to satisfy two main properties, (1) *safety*: all correct servers execute the same requests in the same order, and (2) *liveness*: all correct client requests are eventually executed. Fischer et al. [29] show that in an asynchronous system, where nodes can fail, consensus has no solution that is both safe and live. Based on that impossibility result, SeeMoRe, similar to most fault-tolerant protocols, ensures the safety property without any synchrony assumption and considers a synchrony assumption to satisfy the liveness property. Indeed, as long as the number of faulty nodes does not exceed the defined threshold, a protocol can produce linearizable executions, independent of whether the network loses, reorders, or arbitrarily delays messages. However, a weak synchrony assumption is needed to satisfy liveness: the delay from the moment when a request is sent by a client for the first time and the moment when it is received by its destination is in some fixed (but potentially unknown) interval.

We identify each replica using an integer in $[0, ..., N-1]$ where replicas in the private cloud, i.e., trusted replicas, have identifiers in $[0, ..., S-1]$ and replicas in the public cloud, i.e., untrusted replicas, are identified using integers in $[S, ..., N-1]$.

In SeeMoRe, the replicas move through a succession of configurations called *views* [27] [28]. In a view, one replica is *the primary* and the others are *backups*. Depending on the mode, some backups are *passive* and do not participate in the agreement. Views are numbered consecutively. All replicas are initially in view 0 and are aware of their current view number at all time.

We explain SeeMoRe in three different modes: *Lion*, *Dog*, and *Peacock* [2]. In the Lion mode, the primary is *always* in the private cloud, thus the primary is non-malicious. The Dog mode is used to reduce the load on the private cloud by assuming that the primary is still in the private cloud, but instead of processing the client requests itself, depends on $3m + 1$ nodes in the public cloud to process the request. This mode reduces the load on the private cloud, because except for the primary, which does a single broadcast of the client's request, other replicas in the private cloud are passive and do not participate in any phases. Finally, in the Peacock mode, an untrusted node is chosen as the primary and the protocol relies completely on the public cloud to process requests. This mode is useful when we intentionally rely completely on the public cloud for two purposes: (1) load balancing when all the nodes in the private cloud are heavily loaded, or (2) reducing the delay when there is a large network distance between the private and the public cloud and the latency of having one more phase of communication within the public cloud is less than the latency of exchanging messages between the two clouds. The agreement routine of the Peacock mode is the same as PBFT [32], however, the view change routine can be more efficient.

In this section, we describe each of these three modes in detail, followed by a technique to dynamically switch between the modes. For each mode, we first present the normal-case operation of the protocol and then show how view changes are carried out when it appears that the primary has failed. Next, we show how SeeMoRe can dynamically switch between these three modes. We use π to show the current mode of the protocol where $\pi \in \{1, 2, 3\}$. At the end of this section, we also present a short discussion on different modes of SeeMoRe and compare it with some known relevant protocols, i.e., the crash fault-tolerant protocol Paxos [36], the Byzantine fault-tolerant protocol PBFT [15], and the hybrid fault-tolerant protocol UpRight [18].

5.1 The Lion Mode: Trusted Primary

Owning a private cloud gives SeeMoRe the chance to choose a trusted node as the primary. When the primary is trusted, all the non-faulty backups receive correct messages from the primary, which eliminates the need to multicast messages by replicas to realize whether all the non-faulty ones receive

---

[2] We call the modes Lion, Dog, and Peacock because seemorq (=SeeMoRe) is composed of these three animals.

the same message or not. Thus, we can reduce one phase of communication and a large number of messages.

In particular, within a view, the normal case operation for SeeMoRe to execute a client request in the Lion mode proceeds as follows. A client sends a request message to the primary, i.e., a trusted replica in the private cloud. The primary assigns a sequence number to the request and multicasts a prepare message including the request to all replicas. Replicas receive a prepare from the primary and send an accept to the primary. The primary upon receiving $2m+c+1$ matching accept messages, sends a commit message to all replicas and a reply to the client. Upon receiving a commit message from the primary, replicas execute the client request. Finally, the client receives a reply message from the primary and marks the request as complete.

Figure 1(a) shows the normal case operation of the Lion mode. Here, replicas 0 and 1 are trusted ($S = 2$) and the four other replicas, 2 to 5, are untrusted ($P = 4$). In addition, one of the trusted replicas (1) is crashed ($c = 1$) and one of the untrusted replicas (5) is malicious ($m = 1$). With a trusted primary, the total number of exchanged messages is $3N$.

The pseudo-code for the Lion mode is presented in Algorithm 1. Although not explicitly mentioned, every sent and received message is logged by the replicas. Each replica is initialized with a set of variables as indicated in lines 1-4 of the algorithm. The primary of view v is a replica p such that $p = (v \mod S)$. A client ς requests a state machine operation op by sending a message $\langle \text{REQUEST}, op, ts_\varsigma, \varsigma \rangle_{\sigma_\varsigma}$ to replica p it believes to be the primary. The client's timestamp ts_ς is used to totally order the requests and to ensure exactly-once semantics. The client also signs the message with signature σ_ς for authentication.

Each replica keeps the state of the service, a message log containing valid messages the replica has received, and two integers denoting the replica's current view and mode numbers. Message logs then serve as the basis for maintaining consistency in view changes.

As indicated in lines 5-8, upon receiving a client request, the primary p first checks if the signature and timestamp in the request are valid and simply discards the message otherwise. The primary assigns a sequence number n to the request and multicasts a signed $\langle \langle \text{PREPARE}, v, n, d \rangle_{\sigma_p}, \mu \rangle$ message to all the replicas where v is the current view, μ is the client's request message, and d is the digest of μ. At the same time, the primary appends the message to its log. The primary signs its message, because it might be used by other replicas later in view changes as a proof of receiving the message.

As shown in lines 9-11 of the algorithm, upon receipt of $\langle \langle \text{PREPARE}, v, n, d \rangle_{\sigma_p}, \mu \rangle$ from primary p, replica r checks if view v is equal to the replica's view. It then logs the prepare message, and responds to the primary with $\langle \text{ACCEPT}, v, n, d, r \rangle$ message. Since accept messages are sent only to the trusted primary and are not used later for any other purposes, there is no need to sign these messages.

Upon collecting $2m+c$ valid accept messages from different replicas (plus itself becomes $2m+c+1$) for the request μ in view v with sequence number n, as seen in lines 12-15, the primary multicasts a commit message $\langle \langle \text{COMMIT}, v, n, d \rangle_{\sigma_p}, \mu \rangle$ to all replicas. The primary attaches the request μ to its commit message, so that if a replica has not received a prepare message for that request, it can still execute the request.

Algorithm 1 The Normal-Case Operation in the *Lion* mode

```
1: init():
2:    r := replicaId
3:    v := viewNumber
4:    if r = (v mod S) then isPrimary := true

5: upon receiving μ=⟨REQUEST, op, ts_ς, ς⟩_{σ_ς} and isPrimary:
6:    if μ is valid then
7:       assign sequence number n
8:       send ⟨⟨PREPARE, v, n, d⟩_{σ_p}, μ⟩ to all replicas

9: upon receiving ⟨⟨PREPARE, v, n, d⟩_{σ_p}, μ⟩ from primary p:
10:    if v is valid then
11:       send ⟨ACCEPT, v, n, d, r⟩ to primary p

12: upon receiving ⟨ACCEPT, v, n, d, r⟩ from 2m+c replicas and is-
    Primary:
13:    send ⟨⟨COMMIT, v, n, d⟩_{σ_p}, μ⟩ to all replicas
14:    execute operation op
15:    send ⟨REPLY, π, v, ts_ς, u⟩_{σ_p} to client ς with result u
```

The primary also executes the operation op and sends a reply message $\langle \text{REPLY}, \pi, v, ts_\varsigma, u \rangle_{\sigma_p}$ to client ς. Mode number π and view number v are sent to clients to enable them to track the current mode and view and hence the current primary. It is important especially when a mode change or view change occurs, replacing the primary.

Once a replica receives a valid commit message with correct view number from the primary, it executes the operation op, if all requests with lower sequence numbers than n has been executed. This ensures that all non-malicious replicas execute requests in the same order as required to provide the safety property. Note that even if the replica has not received a prepare message for that request, as long as the view number is valid and the message comes from the primary, the replica considers the request as committed.

When the client receives a reply message $\langle \text{REPLY}, \pi, v, ts_\varsigma, u \rangle_{\sigma_p}$ with a valid signature from primary p and with the same timestamp as the client's request, it accepts u as the result of the requested operation.

If the client does not receive a reply from the primary after a preset time, the client may suspect a crashed primary. The client then broadcasts the same request to all replicas. A replica, upon receiving the client's request, checks if it has already executed the request; if so, it simply sends the reply message to the client. The client waits for a reply from the private cloud or $m + 1$ matching reply messages from the public cloud before accepting the result. The primary will eventually be suspected to be faulty by enough replicas to trigger a view change.

State Transfer. A fault-tolerant protocol must provide a way to checkpoint the state of different replicas. It is especially required in an asynchronous system where even non-faulty replicas can fall arbitrarily behind. Checkpointing also brings slow replicas up to date so that they may execute more recent requests. Similar to [15], in our protocol, checkpoints are generated periodically when a request sequence number is divisible by some constant (checkpoint period).

Trusted primary p produces a checkpoint and multicasts a $\langle \text{CHECKPOINT}, n, d \rangle_{\sigma_p}$ message to the other replicas, where n is the sequence number of the last executed request and d is the digest of the state. A server considers a checkpoint to be *stable* when it receives a checkpoint message for sequence number n signed by trusted primary p. We call this message
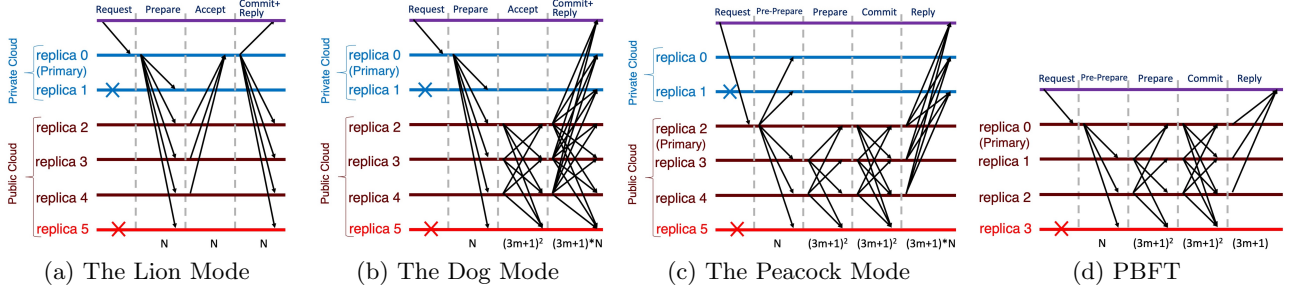
Figure 1: The normal case operation of the three modes of SeeMoRe and PBFT

a *checkpoint certificate*, which proves that the replica's state was correct until that request execution.

Checkpointing not only brings slow replicas up to date, but it can also be used as a garbage collection mechanism. All the messages sent by a replica are kept in a message log in case they have to be re-sent. However, when a checkpoint becomes stable, replicas do not need to keep messages prior to the checkpoint in their log and can simply discard all prepare, accept, and commit messages with sequence numbers less than or equal to the checkpoint's sequence number. They also discard all earlier checkpoints and checkpoint messages.

View Changes. The goal of the view change protocol is to provide liveness by allowing the system to make progress when a primary fails. It prevents replicas from waiting indefinitely for requests to execute. A view change must guarantee that it will not introduce any changes in a history that has been already completed at a correct client. Most view change routines [15,16,22,27,28,34,58] are triggered by timeouts and require enough non-faulty replicas to exchange view change messages. SeeMoRe uses a similar technique in the Lion mode. In such a situation, replicas detect the failure and reach agreement to change the view from v to v'. The primary of new view v' then handles the uncommitted requests, and takes care of the new client requests.

View changes are triggered by timeout. When a replica receives a valid prepare message from the primary, it starts a timer that expires after some defined time τ. When the backup receives a valid commit message, the timer is stopped, but if at that point the backup is waiting for a commit message for some other request, it restarts the timer. If the timer expires, the backup suspects that the primary is faulty and starts a view change.

When a backup suspects that the primary is faulty (its timer for some prepare message expires), it stops accepting prepare and commit messages and multicasts a $\langle$VIEW-CHANGE, $v + 1, n, \xi, \mathcal{P}, \mathcal{C}\rangle$ message to all replicas where n is the sequence number of the last stable checkpoint known to r, ξ is the checkpoint certificate, and $\mathcal{P}$ and $\mathcal{C}$ are two sets of received valid prepare (without the request message μ) and commit messages for requests with a sequence number higher than n.

When primary p' of new view $v + 1$ receives $2m + c$ valid view-change messages from different replicas, it multicasts a $\langle$NEW-VIEW, $v + 1, \mathcal{P}', \mathcal{C}'\rangle_{\sigma_{p'}}$ message to all replicas where $\mathcal{P}'$ and $\mathcal{C}'$ are two sets of prepare and commit messages respectively which are constructed as follows.

Let l be the sequence number of the latest checkpoint, and

h be the highest sequence number of a prepare message in all the received $\mathcal{P}$ sets. For each sequence number n where $l < n \le h$, the primary does the following steps:

1. It first checks all commit messages in set $\mathcal{C}$ of the received view-change messages. If the primary finds a commit message with a valid signature σ_p (p was the primary of view v) for some request μ the primary adds a $\langle\langle$COMMIT, $v + 1, n, d\rangle_{\sigma_{p'}}, \mu\rangle$ to $\mathcal{C}'$

2. If no such commit message is found, the primary checks the prepare messages in $\mathcal{P}$ sets:

   • If the primary finds $2m + c + 1$ valid prepare messages for n, it adds a $\langle\langle$COMMIT, $v + 1, n, d\rangle_{\sigma_{p'}}, \mu\rangle$ to $\mathcal{C}'$.

   • Else, if it receives at least one valid prepare message for n, the primary adds a $\langle\langle$PREPARE, $v+1, n, d\rangle_{\sigma_{p'}}, \mu\rangle$ to $\mathcal{P}'$.

3. If none of the above situations occur, there is no valid request for n, so the primary adds a $\langle$PREPARE, $v+1, n, d\rangle_{\sigma_{p'}}, \mu^{\emptyset}\rangle$ to $\mathcal{P}'$ where $\mu^{\emptyset}$ is a special *no-op* command that is transmitted by the protocol like other requests but leaves the state unchanged. The third situation happens when no replica has received a prepare message from the previous primary.

In contrast to PBFT, since the primary is trusted, it does not need to append all the view-change messages in the new-view message which makes the new-view messages much smaller. The primary inserts all the messages in $\mathcal{P}'$ and $\mathcal{C}'$ to its log. It also checks the log to make sure its log contains the latest stable checkpoint. If not, the primary inserts checkpoint messages for the checkpoint l and discards the earlier information from the log.

Once a replica in view v receives a new-view message from the primary of view $v + 1$, the replica logs all prepare and commit messages, updates its checkpoint in the same way as the primary, and for each prepare message, sends an accept message to the primary. Non-faulty replicas in view v will not accept a prepare message for a new view $v' > v$ without having received a new-view message for v'.

Correctness. Within a view, since the primary is trusted and it assigns sequence numbers to the requests, safety is ensured as long as the primary does not fail. Indeed, for any two committed requests r_1 and r_2 with sequence numbers n_1 and n_2 respectively, if $D(r_1) = D(r_2)$, then $n = n'$.

If the primary fails a view change is executed. To ensure safety across views, the primary waits for $2m + c$ accept

messages (considering itself, a quorum of $2m + c + 1$) from different replicas to ensure that committed requests are totally ordered across views. In fact, for any two committed requests r_1 and r_2 with sequence numbers n_1 and n_2, since a quorum of $2m+c+1$ replicas commits r_1 and a quorum of $2m+c+1$ replicas commits r_2, and these two quorums have at least $m + 1$ overlapping nodes, there should be at least one non-faulty node that commits both r_1 and r_2 but this is not possible because the node is not faulty. As a result, if $D(r_1) = D(r_2)$, then $n = n'$. This guarantees that in the event of primary failure, any new quorum of $2m+c+1$ replicas will have at least $m + 1$ overlapping nodes that received a prepare message (and sent accept) for request μ from the previous primary. Thus, there is at least one non-faulty node in that quorum that helps the protocol to process request μ in the new view.

5.2 The Dog Mode: Trusted Primary, Untrusted Backups

The Dog mode is proposed to reduce the load on the private cloud. In this mode, a *trusted primary* receives a request message, assigns a sequence number, and relies on $3m + 1$ *untrusted nodes* (in the public cloud) to process the request. These $3m + 1$ nodes are called *proxies*. Since a trusted primary assigns the sequence number to the request before broadcasting, this reduces the scope of any malicious behaviour. Whereas in PBFT, when replicas receive a message from the primary, they perform one round of communication to make sure all non-faulty replicas agree on a total order for the requests within a view. However, here, since a trusted primary assigns the sequence numbers, similar to the Lion mode, there is no need for that phase.

Figure 1(b) shows the normal case operation of SeeMoRe with a trusted primary (node 0). As before, two replicas are trusted ($S = 2$), four replicas are untrusted ($P = 4$), $c = 1$, and $m = 1$. Since a trusted primary assigns sequence numbers, the protocol, similar to Paxos, needs two phases to process requests. However, since the protocol tolerates malicious failures, the number of messages in terms of the number of replicas, similar to PBFT, is quadratic. Here, there are totally $N + (3m + 1)^2 + (3m + 1) * N$ messages exchanged where $3m + 1$ is the total number of proxies. In this particular example, since $m = 1$, all replicas in the public cloud are proxies.

Algorithm 2 provides the pseudo-code for the Dog mode. Lines 1-5 indicate the initialization of state variables for each replica, including the primary and the proxies. A replica r in the public cloud is a *proxy* in view v if $r-(v \mod P) \in [S, ..., S+3m]$. Here since replicas are in the public cloud, r is an integer in $[S, ..., N-1]$. The public cloud might have more than $3m+1$ replicas, however, $3m+1$ is enough to reach consensus and any additional replicas may degrade the performance. The trusted primary of view v is chosen in the same way as the first mode, i.e., p is the primary if $p=(v \mod S)$.

As shown in lines 6-9 of the algorithm, the primary, upon receiving request μ, validates the timestamp and signature of μ, assigns a sequence number n, and multicasts signed prepare message $\langle\langle \text{PREPARE}, v, n, d\rangle_{\sigma_p}, \mu\rangle$ to all replicas.

When a proxy receives a prepare message from the primary, as indicated in lines 10-12, it validates the view number, logs the message and sends a signed accept message $\langle \text{ACCEPT}, v, n, d, r\rangle_{\sigma_r}$ to all the other proxies. Here, in con-

Algorithm 2 The Normal-Case Operation in the *Dog* mode

1: **init():**
2:   $r :=$ replicaId
3:   $v :=$ viewNumber
4:   if $r = (v \mod S)$ then isPrimary := true
5:   else if $r - (v \mod P) \in [S, .., S + 3m]$ then isProxy := true

6: **upon receiving** $\mu = \langle \text{REQUEST}, op, ts_\varsigma, \varsigma\rangle_{\sigma_\varsigma}$ and isPrimary:
7:   if μ is *valid* then
8:     **assign** *sequence number* n
9:     **send** $\langle\langle \text{PREPARE}, v, n, d\rangle_{\sigma_p}, \mu\rangle$ to all replicas

10: **upon receiving** $\langle\langle \text{PREPARE}, v, n, d\rangle_{\sigma_p}, \mu\rangle$ from the primary p and isProxy:
11:   if v is *valid* then
12:     **send** $\langle \text{ACCEPT}, v, n, d, r\rangle_{\sigma_r}$ to all proxies

13: **upon receiving** $\langle \text{ACCEPT}, v, n, d, r\rangle$ from **2m+1** proxies:
14:   **send** $\langle \text{COMMIT}, v, n, d, r\rangle_{\sigma_r}$ to all other proxies
15:   **send** $\langle \text{INFORM}, v, n, d, r\rangle_{\sigma_r}$ to all private cloud nodes and non-proxy nodes in public cloud
16:   **execute** operation op
17:   **send** $\langle \text{REPLY}, \pi, v, ts_\varsigma, u\rangle_{\sigma_r}$ to client ς with result u

trast to the first mode, the proxy signs its message as a proof of message reception in case of a view change.

As described in lines 13-17 of the algorithm, upon receiving $2m+1$ matching accept messages (including its own message) with correct signatures, a proxy r multicasts a commit message $\langle \text{COMMIT}, v, n, d, r\rangle_{\sigma_r}$ to the other proxies. Each proxy r also sends a signed inform message $\langle \text{INFORM}, v, n, r, d\rangle_{\sigma_r}$ to all the nodes in the private cloud and all non-proxy nodes in the public cloud. The inform message, including its identifier r and message digest d, to inform them that such a request is committed. Non-proxy nodes wait for $2m + 1$ valid matching inform messages from different proxies which are matched by the prepare message that they received from the primary before executing the request. If the proxy has executed all requests with sequence numbers lower than n, it executes the request n and sends a reply message $\langle \text{REPLY}, \pi, v, ts_\varsigma, u\rangle_{\sigma_r}$ with result u to the client.

Any other replica that receives $m + 1$ matching commit messages from the proxies with valid signatures, correct message digest, and view numbers equal to its view number considers the request as committed, and executes the request. Since all the replicas receive prepare messages from the primary, they have access to the request and can execute it.

The client also waits for $2m + 1$ matching reply messages from different proxies before accepting the result. If the client has not received a valid reply after a preset time, the client multicasts the request to the proxies. The proxies re-send the result if the request has already been processed and the client waits for $m + 1$ matching reply messages from the proxies before accepting the result. Otherwise, similar to the first mode, eventually the primary will be suspected to be faulty by enough replicas and a view change will be triggered.

State Transfer. Checkpointing in the Dog mode works in the same way as the Lion mode. Trusted primary p multicasts a signed checkpoint message to all other replicas with the sequence number of the last executed request and the digest of the state. Upon receiving a checkpoint message from the primary, a server considers that a checkpoint is *stable* and logs the message which is used later as a *checkpoint certificate*.

View Changes. In the Dog mode, view change happens

when the trusted primary is suspected to have crashed. Here, similar to the Lion mode, we rely on the primary of new view to handle the prepared but not yet committed requests. However, since the nodes in the public cloud are processing the requests, they are the ones who send view-change messages. Each node in the public cloud multicasts a view-change message $\langle \text{VIEW-CHANGE}, v+1, n, \xi, \mathcal{P} \rangle$ to all the nodes in the public cloud and the primary of the next view where ξ is the checkpoint certificate for sequence number n, and $\mathcal{P}$ is the set of received valid prepare messages with a sequence number higher than n.

In this mode, in contrast to the Lion mode, nodes do not include the set of commit messages ($\mathcal{C}$) in their view-change messages because in the Dog mode, to show that a request is committed, a nodes needs to include $2m + 1$ valid commit messages for that request, which makes the view-change messages much larger.

Primary p' of the new view waits for $2m + 1$ valid view-change messages from the proxies of the last active view, i.e., the view with a non-faulty primary, before multicasting a new-view message. This is needed to ensure the correctness of the protocol even if there are consecutive crashed primary nodes (inactive views) and the number of nodes in the public cloud are more than the number of proxies (the set of proxies are changed from one view to another).

Upon receiving $2m + 1$ valid view-change messages, primary p' of view $v+1$ multicasts a new-view message $\langle \text{NEW-VIEW}, v+1, \mathcal{P}' \rangle_{\sigma_{p'}}$ to all the replicas where for each sequence number n (between the latest checkpoint and the highest sequence number of a prepare message), if there is any valid prepare message in set $\mathcal{P}$ of the received view-change messages, the primary adds a $\langle \text{PREPARE}, v+1, n, d \rangle_{\sigma_{p'}}$ to $\mathcal{P}'$. Else, there is no valid request for n, so similar to the Lion mode, the primary adds a no-op prepare message $\langle \text{PREPARE}, v+1, n, d \rangle_{\sigma_{p'}}, \mu^{\emptyset} \rangle$ to $\mathcal{P}'$.

Here, again, since the primary is trusted it does not need to include view-change messages in the new-view message. The primary then inserts all the messages in $\mathcal{P}'$ to its log and updates its checkpoint, if needed.

Once a proxy of view $v + 1$ receives a new-view message from the primary of view $v + 1$, the proxy logs all prepare messages, updates its checkpoint, and multicasts an accept message to all the proxies for each prepare message in $\mathcal{P}'$. Other replicas also receive the new-view message to be informed that the view is changed.

Correctness. Within a view, since the primary is trusted and it assigns sequence number to the requests, similar to the Lion mode, safety is ensured as long as the primary does not fail. To ensure safety across views, since $3m + 1$ nodes participate in the protocol, to commit a message, $2m + 1$ matching accept messages are needed. In fact, for any two committed requests r_1 and r_2 with sequence numbers n_1 and n_2, since a quorum of $3m + 1$ replicas commits r_1 and a quorum of $3m + 1$ replicas commits r_2, and these two quorums have at least $m + 1$ overlapping nodes, there is at least one non-faulty node that commits both r_1 and r_2, but this is not possible because the replica is non-faulty. As a result, if $D(r_1) = D(r_2)$, then $n = n'$.

5.3 The Peacock Mode: Untrusted Primary, Untrusted Backups

One characteristic of online services is the ever changing patterns in client requests. While there might be periods of high traffic thus overloading some servers, at other periods, the resources may be underutilized. Also, depending on server placements and communication delays, enterprises may benefit from protocols that allow a subset of the servers, e.g. only the public cloud, to handle certain client requests.

The third mode of the protocol, the Peacock mode, is presented to handle two different situations. First, when the private cloud is heavily loaded and the public cloud can handle the requests by itself for load balancing. Second, when there is a large network distance between the private and the public cloud and the latency due to one more phase is less than the latency of exchanging messages between the two clouds. In both situations, the nodes in the private cloud become passive replicas in the agreement routine and are only informed about the committed messages. However, they still may participate in the view change routine.

In the Peacock mode, SeeMoRe completely relies on $3m + 1$ nodes in the public cloud to process the requests. The untrusted primary of view v in the Peacock mode is replica p where $p = (v \mod P) + S$. Similar to the Dog mode, since there might be more than $3m+1$ replicas in the public cloud, in each view, $3m + 1$ are chosen as proxies. Node i is a proxy in view v if $i - (v \mod P) \in [S, ..., S + 3m]$. This ensures that the primary is always a proxy.

In the Peacock mode, SeeMoRe processes the requests using PBFT [15] with two small changes. First, the primary multicasts signed pre-prepare message along with the request to all the nodes (and not only the $3 + 1$ proxies). Second, when the request is committed, each proxy r sends a signed inform message $\langle \text{INFORM}, v, n, d, r \rangle_{\sigma_r}$ to all the nodes in the private cloud as well as all non-proxy nodes in the public cloud. Other nodes also wait for $m + 1$ valid matching inform messages from different proxies before executing the request.

As indicated in Figure 1(c), similar to PBFT, the Peacock mode processes the requests in three phases: pre-prepare, prepare, and commit. As can be seen, the replicas in the private cloud have no participation in any phases and are only informed about the committed requests. The total number of exchanged messages in the Peacock mode is $N + 2 * (3m + 1)^2 + (1 + S) * (3m + 1)$.

View Changes. In the Peacock mode, we rely on a trusted node in the private cloud, called *transferer*, to change the view. Indeed, instead of the primary of the new view, a transferer changes the view. Replica t in the private cloud is the transferer of view v' (changes the view from v to v') if $t = (v' \mod S)$. Choosing a transferer to change views helps in minimizing the size of new-view messages and more importantly, reduces the delay between the request and its reply. Because even if there are consecutive malicious primary nodes, since the transferer takes care of the uncommitted requests of view v, the protocol does not carry the messages from one view to another. In contrast, in PBFT, it is possible that a valid request in view v be committed in view $v+m$ (when there are m consecutive primaries). Other than the transferer, view change in the Peacock mode is similar to PBFT. Proxies multicast view-change messages to all replicas. When the transferer of new view $v + 1$ receives $2m + 1$ valid view-change messages from different proxies of view v, it multicasts a new-view message to all replicas in both public and private clouds. Once a proxy receives

a valid **new-view** message, it logs all the **prepare** messages, updates its checkpoint, and sends an **accept** message to all other proxies for each **prepare** message. When the transferer has changed the view and the new primary receives the **new-view** message from the transferer, the new primary starts to process new requests in view $v + 1$.

Correctness. In the Peacock mode, the protocol ensures safety and liveness similar to PBFT [15].

5.4 Dynamic Mode Switching

We presented three different modes of SeeMoRe and explained when each mode is useful. Now, we show how to dynamically switch between different modes.

An enterprise might prefer to use the Lion mode of SeeMoRe, because it needs fewer phases (in comparison to the Peacock mode) and less number of message exchanges (in comparison to the Dog or Peacock mode). However, if the private cloud becomes heavily loaded, or at some point, a high percentage of requests are sent by clients that are far from the private cloud and much closer to the public cloud, it might be beneficial to switch to the Dog or Peacock mode. SeeMoRe might also plan to switch back to the Lion mode, e.g., when the load on the private cloud is reduced. To change the mode, the protocol also has to change the view, because the primary and the set of participant replicas might be different in different modes. Therefore, to handle a mode change, the protocol first performs a view change, and then the primary of the new view in the new mode starts to process new requests.

For the switch to happen a *trusted* replica s multicasts a $\langle \text{MODE-CHANGE}, v + 1, \pi' \rangle_{\sigma_s}$ to all the replicas where π' is the new mode of the protocol, i.e., Lion, Dog, or Peacock. When the protocol wants to switch to the Lion or Dog mode, replica s is the primary of view $v + 1$, and when it switches to the Peacock mode, replica s is the transferer of view $v+1$.

5.5 Discussion

In this section, we compare the different modes of SeeMoRe with three well-known protocols: the crash fault-tolerant protocol Paxos [36], the Byzantine fault-tolerant protocol PBFT [15], and the hybrid fault-tolerant protocol UpRight [18].

We consider four parameters in this comparison: (1) the number of communication phases, (2) the number of message exchanges, (3) the receiving network size, and (4) the quorum size. The results are reported in Table 1.

The knowledge of where a crash or a malicious failure may occur and thus choosing a trusted primary simply reduces one phase of communication. In fact, in PBFT, the **prepare** phase is needed only to make sure that non-faulty replicas receive matching **pre-prepare** messages from the primary. In contrast, in the Lion and Dog modes of SeeMoRe, since the primary is a trusted node, replicas receive the same message from the primary, thus there is no need for that phase of communication and the requests, similar to Paxos, are processed in two phases (while in contrast to Paxos malicious failures can occur in the public cloud). In comparison to Upright, although Upright processes the requests in two phases, it utilizes the speculative execution technique introduced by Zyzzyva [34] which becomes costly in the presence of failures. Note that the speculative execution technique can easily be applied to SeeMoRe as well.

Table 1: Comparison of fault-tolerant protocols

Protocol	phases	messages	Receiving Network	Quorum size
Lion	2	$\mathcal{O}(n)$	$3m+2c+1$	$2m+c+1$
Dog	2	$\mathcal{O}(n^2)$	$3m+1$	$2m+1$
Peacock	3	$\mathcal{O}(n^2)$	$3m+1$	$2m+1$
Paxos	2	$\mathcal{O}(n)$	$2f+1$	$f+1$
PBFT	3	$\mathcal{O}(n^2)$	$3f+1$	$2f+1$
UpRight	2	$\mathcal{O}(n^2)$	$3m+2c+1$	$2m+c+1$

The number of message exchanges in the Lion mode is similar to Paxos and is linear in terms of the total number of replicas. In the Dog mode, the number of messages is quadratic, however it is still much less than PBFT (since it has one phase of n-to-n communication instead of two). Upright also has a quadratic number of messages.

The Lion mode, similar to Upright, needs $3m+2c+1$ nodes to receive a client request. In the Dog mode, however, only the trusted primary and $3m+1$ nodes from the public cloud participate in each phase. Since the Peacock mode utilizes PBFT, the number of phases and message exchanges are the same as PBFT. However, since the primary is in the public cloud, communicating with the private cloud has no advantage, thus it proceeds with $3m + 1$ nodes instead of $3m + 2c + 1$ as in the Lion mode and UpRight.

6. PERFORMANCE EVALUATION

This section evaluates the performance of the SeeMoRe protocol. SeeMoRe is implemented by adapting the BFT-SMaRt library [50]. We mainly reuse the communication layer of BFT-SMaRt but implement our agreement and view change routines for the different modes of the protocol.

We first, show how the protocol tolerates different number of failures (both crash and malicious). Next, we evaluate the performance of the protocol in a no failure setting by varying the number of clients (requests) and using different micro-benchmarks, and finally, evaluate the impact of the failure of the primary node (view change) on the performance of SeeMoRe.

In each experiment, we compare different modes of SeeMoRe with an asynchronous crash fault-tolerant (CFT) protocol, an asynchronous Byzantine fault-tolerant (BFT) protocol, and a simplified version of the asynchronous hybrid fault-tolerant protocol UpRight [18] (we call it *S-UpRight*). For both CFT and BFT we use the original BFT-SMaRt codebase (the optimized implementations of Paxos [36] and PBFT [15]). UpRight has two aspects: first, the hybrid model that tolerates both crash and malicious failures (in a network of size $3m + 2c + 1$), and second, the protocol that combines a set of techniques such as speculative execution and separation of ordering and execution. For the S-UpRight protocol, we use the UpRight hybrid model since this part of the UpRight is relevant to SeeMoRe, however, to ensure a fair comparison with other protocols and since all other protocols use the pessimistic approach, we use a PBFT-like protocol (i.e., PBFT protocol with less number of nodes) instead of the UpRight protocol. Note that, as mentioned before, both the speculative execution and separation of ordering from execution techniques can be integrated into SeeMoRe as well.

(a) $f = 2$ ($c = 1$, $m = 1$)
N: SeeMoRe, S-UpRight=6,
CFT=5, BFT=7

(b) $f = 4$ ($c = 2$, $m = 2$)
N: SeeMoRe, S-UpRight=11,
CFT=9, BFT=13

(c) $f = 4$ ($c = 1$, $m = 3$)
N: SeeMoRe, S-UpRight=12,
CFT=9, BFT=13

(d) $f = 4$ ($c = 3$, $m = 1$)
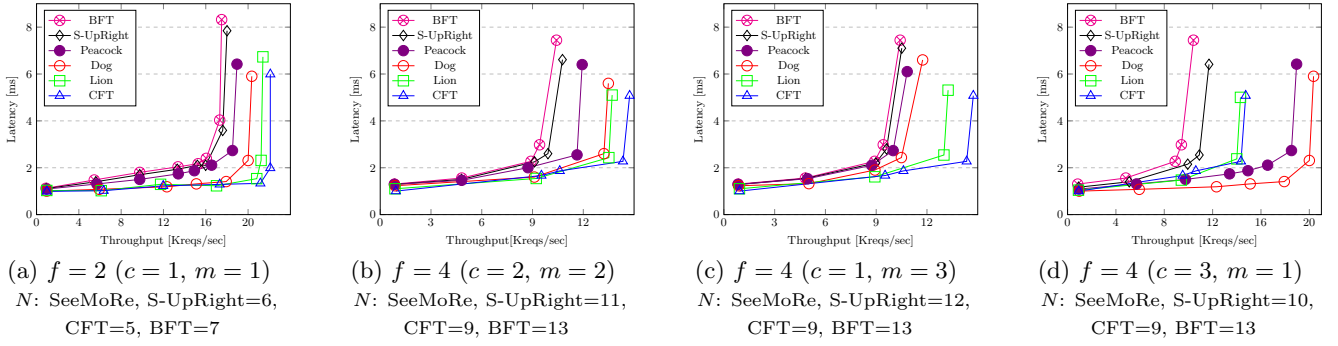N: SeeMoRe, S-UpRight=10,
CFT=9, BFT=13

Figure 2: Throughput/Latency measurement by increasing the number of failures

The experiments were conducted on the Amazon EC2 platform. Each VM is Compute Optimized c4.2xlarge instances with 8 vCPUs and 15GB RAM, Intel Xeon E5-2666 v3 processor clocked at 3.50 GHz. In the experiments, both the public and private clouds are located in the same data center i.e., AWS US West Region.

In each experiment, we vary the number of requests sent by all the clients per second from 10^3 to 10^6 (by increasing the number of clients running on a single VM) and measure the end-to-end throughput (x axis) and latency (y axis) of the system. Each client waits for the reply before sending a subsequent request.

6.1 Fault-Tolerance Scalability

In the first set of experiments, we evaluate the performance of SeeMoRe when configured to tolerate different number of failures (f). We consider the 0/0 micro-benchmark (both request and reply payload sizes are close to 0 KB) and measure the throughput and latency of SeeMoRe, S-UpRight, CFT, and BFT protocols. Since, $f = c + m$, we evaluate CFT and BFT to tolerate $c + m$ failures in each experiment. In all these scenarios and for SeeMoRe, we put $2c$ nodes in the private and $3m+1$ nodes in the public cloud. The results are shown in Fig. 2(a)-(d).

In the first scenario, when $f = 2$ ($c = m = 1$), the network size of the different protocols is close to each other (BFT requires 7, SeeMoRe and S-UpRight require 6, and CFT requires 5 nodes). As a result, as can be seen in Fig. 2(a), the performance of the Lion mode becomes very close to CFT (8% difference in their peak throughput). Similarly, the performances of S-UpRight and BFT are close to each other (4% difference in their peak throughput). Note that the Peacock mode shows better performance than S-UpRight (still worst than the Dog and Lion modes) because in the Peacock mode, SeeMoRe relies only on the public cloud which consists of only 4 nodes. In addition, while in comparison to the Lion mode, both the Peacock and Dog modes need less number of nodes, the Lion mode has better performance because it needs less number of phases and message exchanges.

In the next three scenarios, the network tolerates the same number of failures ($f = 4$), as a result, the performance of BFT and CFT does not change from one scenario to another. However, since the number of crash and malicious failures are varied, the network size of SeeMoRe and S-UpRight changes. Hence, they show different performance in different scenarios.

When both m and c increase to 2 (Fig. 2(b)), The Dog

mode shows similar performance to the Lion mode. This is the result of the trade off between the quorum size and the message complexity; Only 5 nodes ($2m + 1$) participate in the Dog mode which requires $\mathcal{O}(n^2)$ number of messages whereas the quorum size of the Lion mode is 7 ($2m + 1c + 1$) but it requires $\mathcal{O}(n)$ messages (see Table 1). In addition, since SeeMoRe in the Peacock mode communicates with only 7 nodes, it shows much better performance than BFT (24% difference in their peak throughput) and even S-UpRight (18% difference in their peak throughput).

By increasing the number of tolerated malicious failures to 3 while reducing the number of tolerated crash failures back to 1 (Fig. 2(c)), the network size of SeeMoRe becomes closer to the BFT network size. As a result, CFT shows better performance (12% difference in its peak throughput) than the Lion mode and also the performance of SeeMoRe in the Peacock and Dog modes, which communicate with 10 nodes in the public cloud, becomes closer to S-UpRight (with 12 nodes) and BFT (with 13 nodes).

On the other hand, increasing the number of tolerated crash failures to 3 while maintaining the number of malicious failures to 1 (Fig. 2(d)) results in a network size close to CFT. In this setting, the performance of the Dog and Peacock modes become better than both the Lion mode and CFT. This is expected because the Dog mode processes a request in the public cloud which needs only 4 replicas (since $m = 1$) but with the same number of phases as the Lion mode. Similarly, although the Peacock mode processes requests in three phases, since it needs fewer servers to proceed, its performance is better than the Lion mode and CFT. In fact, since the number of malicious failures in this scenario is the same as the first scenario, both the Dog and Peacock modes show the same performance as the first scenario (Fig. 2(a)).

6.2 Changing Payload Size

We now repeat the base case scenario ($c=m=1$) of the previous experiments (Fig. 2(a)) using two micro-benchmarks 0/4, 4/0 to show how request and reply sizes affect the performance of different protocol. Figs. 3(a) and 3(b) show the throughput and latency for 0/4 and 4/0 micro-benchmarks respectively. Since the Lion and Dog modes need less communication phases and message exchanges, their performance is close to CFT, e.g., for latency equal to 4 ms, the throughput of the Lion and Dog modes is 10% and 17% less than CFT respectively. Similarly, the Peacock mode and S-UpRight are close to BFT, e.g., with 4 ms latency, the throughput
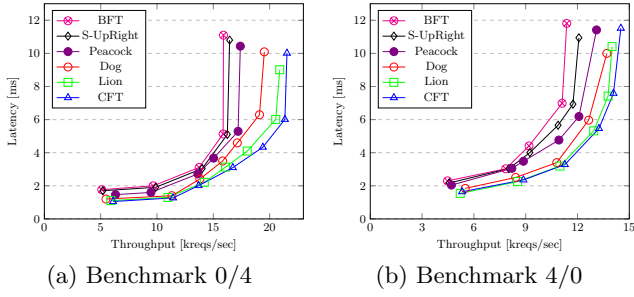
(a) Benchmark 0/4   (b) Benchmark 4/0

Figure 3: Throughput/Latency for $c = 1$ and $m = 1$

Figure 4: Performance during view change

of the Peacock mode is the same as BFT. Note that due to the overhead of request transmission between the replicas, the request size affects the performance of all protocols more than the reply size.

6.3 Performance During View Change

Finally, we evaluate the impact of view change on the performance of SeeMoRe. We trigger a primary failure shortly before the end of a checkpoint period to evaluate the worst-case overhead that can be caused by a failure. We consider the base case scenario ($c = m = 1$) with a total network of $N = 6$ nodes (for SeeMoRe), where 2 nodes are in the private cloud and 4 in the public cloud. The experiment was run with micro-benchmark 0/0 and with a checkpoint period of 10000 request i.e., a checkpoint is taken every 10000 requests. Fig. 4 shows the behavior of SeeMoRe, S-UpRight and BFT where the y-axis is throughput and the x-axis is a timeline with a failure injected around time 30. As can be seen, the protocols behave as expected until the crash is triggered. This crash and the view change routine cause the protocols to be temporarily out of service (in particular, 15 millisecond in the Lion mode, 20 millisecond in the Dog mode, and 24 millisecond in the Peacock mode). However, when the view change is complete, the throughput increases to the original level for each protocol. As can be seen, BFT takes twice as much time as the Lion mode to revive and continue to process the requests. The Peacock mode also recovers faster than S-UpRight and BFT due to its use of transferers.

Overall, the evaluation results for a network that tolerates $f = m + c$ failures where m and c are the number of malicious and crash failures respectively, can be summarized as follow. First, when c is equal or less than m (for small c and m), the performance of SeeMoRe in the Lion mode is very close to the crash fault-tolerant protocol Paxos due to the required number of phases and message exchanges in the Lion mode. In addition, when c is larger than m, SeeMoRe in both the Dog and Peacock modes demonstrates better performance than the Lion mode and even Paxos since in both modes, SeeMoRe relies completely on the public cloud to process the requests. Furthermore, all three modes of SeeMoRe show better performance than the hybrid protocol S-UpRight since SeeMoRe is aware of where the crash faults may occur and where the malicious faults can occur. Finally, all three modes of SeeMoRe have better performance than BFT since they reduce the number of communication phases, messages exchanged and required nodes.
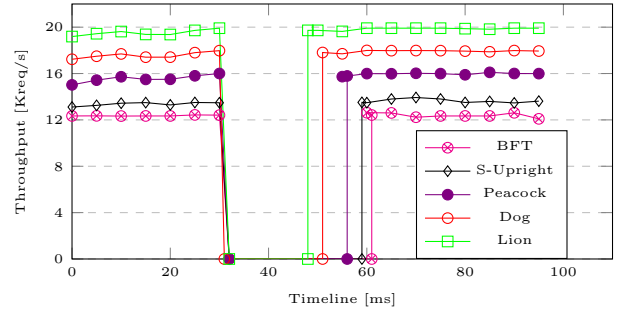
7. CONCLUSIONS

In this paper, we proposed SeeMoRe, a hybrid state machine replication protocol to tolerate both crash and malicious failures in a public/private cloud environment. SeeMoRe is targeted to be used by smaller enterprises that own a small set of servers and intend to rent servers from public cloud providers. Such an enterprise can highly benefit from SeeMoRe, as the protocol distinguishes between crash failures that could occur within the trusted private cloud and malicious failures that could only occur in the public cloud. SeeMoRe can execute in any one of three modes, Lion, Dog, and Peacock, and can dynamically switch among these modes. The Lion and Dog modes of SeeMoRe require less communication phases and message exchanges while the Peacock mode is useful for a heavily loaded private cloud or when there is a large network distance between the two clouds.

Our evaluations show that the performance of Lion and Dog modes is close to Paxos while in contrast to Paxos, which only tolerates crash failures, malicious failures can occur. In the Peacock mode, since the primary is in the public cloud, its performance is similar to PBFT with m failures. However, in comparison to UpRight, which requires quorums of size $2m + c + 1$, Peacock needs quorums of size $2m + 1$, and hence is more efficient.

8. REFERENCES

[1] M. Abd-El-Malek, G. R. Ganger, G. R. Goodson, M. K. Reiter, and J. J. Wylie. Fault-scalable byzantine fault-tolerant services. *ACM SIGOPS Operating Systems Review*, 39(5):59–74, 2005.

[2] M. A. AlZain, B. Soh, and E. Pardede. Mcdb: Using multi-clouds to ensure security in cloud computing. In *IEEE International Conference on Dependable, autonomic and secure computing (DASC)*, pages 784–791. IEEE, 2011.

[3] Y. Amir, B. Coan, J. Kirsch, and J. Lane. Prime: Byzantine replication under attack. *IEEE Transactions on Dependable and Secure Computing*, 8(4):564–577, 2011.

[4] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, et al. Hyperledger fabric: a distributed operating system for permissioned blockchains. In *Proceedings of the Thirteenth EuroSys Conference*, page 30. ACM, 2018.

[5] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, et al. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, 2010.

[6] P. Aublin, S. B. Mokhtar, and V. Quéma. Rbft: Redundant byzantine fault tolerance. In *IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 297–306. IEEE, 2013.

[7] P.-L. Aublin, R. Guerraoui, N. Knežević, V. Quéma, and M. Vukolić. The next 700 bft protocols. *ACM Transactions on Computer Systems (TOCS)*, 32(4):12, 2015.

[8] C. Băsescu, C. Cachin, I. Eyal, R. Haas, A. Sorniotti, M. Vukolić, and I. Zachevsky. Robust data sharing with key-value stores. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 1–12. IEEE, 2012.

[9] A. Bessani, M. Correia, B. Quaresma, F. André, and P. Sousa. Depsky: dependable and secure storage in a cloud-of-clouds. *ACM Transactions on Storage (TOS)*, 9(4):12, 2013.

[10] A. N. Bessani, R. Mendes, T. Oliveira, N. F. Neves, M. Correia, M. Pasin, and P. Verissimo. Scfs: A shared cloud-backed file system. In *USENIX Annual Technical Conference*, pages 169–180, 2014.

[11] G. Bracha and S. Toueg. Asynchronous consensus and broadcast protocols. *Journal of the ACM (JACM)*, 32(4):824–840, 1985.

[12] F. Brasileiro, F. Greve, A. Mostéfaoui, and M. Raynal. Consensus in one communication step. In *International Conference on Parallel Computing Technologies*, pages 42–50. Springer, 2001.

[13] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. Li, et al. {TAO}: Facebooks distributed data store for the social graph. In *Presented as part of the 2013 {USENIX} Annual Technical Conference ({USENIX}{ATC} 13)*, pages 49–60, 2013.

[14] C. Cachin, I. Keidar, and A. Shraer. Trusting the cloud. *Acm Sigact News*, 40(2):81–86, 2009.

[15] M. Castro and B. Liskov. Practical byzantine fault tolerance. In *OSDI*, volume 99, pages 173–186, 1999.

[16] M. Castro, R. Rodrigues, and B. Liskov. Base: Using abstraction to improve fault tolerance. *ACM Transactions on Computer Systems (TOCS)*, 21(3):236–269, 2003.

[17] B.-G. Chun, P. Maniatis, S. Shenker, and J. Kubiatowicz. Attested append-only memory: Making adversaries stick to their word. In *ACM SIGOPS Operating Systems Review*, volume 41-6, pages 189–204. ACM, 2007.

[18] A. Clement, M. Kapritsos, S. Lee, Y. Wang, L. Alvisi, M. Dahlin, and T. Riche. Upright cluster services. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 277–290. ACM, 2009.

[19] A. Clement, E. L. Wong, L. Alvisi, M. Dahlin, and M. Marchetti. Making byzantine fault tolerant systems tolerate byzantine faults. In *NSDI*, volume 9, pages 153–168, 2009.

[20] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, et al. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):8, 2013.

[21] M. Correia, N. F. Neves, and P. Verissimo. How to tolerate half less one byzantine nodes in practical distributed systems. In *IEEE Int. Symposium on Reliable Distributed Systems*, pages 174–183. IEEE, 2004.

[22] J. Cowling, D. Myers, B. Liskov, R. Rodrigues, and L. Shrira. Hq replication: A hybrid quorum protocol for byzantine fault tolerance. In *OSDI*, pages 177–190, 2006.

[23] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. In *ACM SIGOPS operating systems review*, volume 41, pages 205–220. ACM, 2007.

[24] T. Distler, C. Cachin, and R. Kapitza. Resource-efficient byzantine fault tolerance. *IEEE Transactions on Computers*, 65(9):2807–2819, 2016.

[25] T. Distler, I. Popov, W. Schröder-Preikschat, H. P. Reiser, and R. Kapitza. Spare: Replicas on hold. In *NDSS*, 2011.

[26] D. Dobre, P. Viotti, and M. Vukolić. Hybris: Robust hybrid cloud storage. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–14. ACM, 2014.

[27] A. El Abbadi, D. Skeen, and F. Cristian. An efficient, fault-tolerant protocol for replicated data management. In *ACM SIGACT-SIGMOD symp. on Principles of database systems*, pages 215–229. ACM, 1985.

[28] A. El Abbadi and S. Toueg. Availability in partitioned replicated databases. In *ACM SIGACT-SIGMOD symposium on Principles of database systems*, pages 240–251. ACM, 1985.

[29] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, 1985.

[30] G. G. Gueta, I. Abraham, S. Grossman, D. Malkhi, B. Pinkas, M. K. Reiter, D. Seredinschi, O. Tamir, and A. Tomescu. Sbft: a scalable decentralized trust infrastructure for blockchains. *arXiv preprint arXiv:1804.01626*, 2018.

[31] H. S. Gunawi, M. Hao, R. O. Suminto, A. Laksono, A. D. Satria, J. Adityatama, and K. J. Eliazar. Why does the cloud stop computing?: Lessons from hundreds of service outages. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–16. ACM, 2016.

[32] R. Kapitza, J. Behl, C. Cachin, T. Distler, S. Kuhnle, V. Mohammadi, W. Schröder-Preikschat, and K. Stengel. Cheapbft: resource-efficient byzantine fault tolerance. In *ACM European conference on Computer Systems*, pages 295–308. ACM, 2012.

[33] R. M. Kieckhafer and M. H. Azadmanesh. Reaching approximate agreement with mixed-mode faults. *IEEE Transactions on Parallel and Distributed Systems*, 5(1):53–63, 1994.

[34] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and

E. Wong. Zyzzyva: speculative byzantine fault tolerance. *ACM SIGOPS Operating Systems Review*, 41(6):45–58, 2007.

[35] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

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

[37] L. Lamport. Fast paxos. *Distributed Computing*, 19(2):79–103, 2006.

[38] L. Lamport and M. Massa. Cheap paxos. In *International Conference on Dependable Systems and Networks*, pages 307–314. IEEE, 2004.

[39] L. Lamport, R. Shostak, and M. Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382–401, 1982.

[40] S. Liu, P. Viotti, C. Cachin, V. Quéma, and M. Vukolic. Xft: Practical fault tolerance beyond crashes. In *OSDI*, pages 485–500, 2016.

[41] J.-P. Martin and L. Alvisi. Fast byzantine consensus. *IEEE Trans. on Dependable and Secure Computing*, 3(3):202–215, 2006.

[42] F. J. Meyer and D. K. Pradhan. Consensus with dual failure modes. *IEEE Transactions on Parallel & Distributed Systems*, (2):214–222, 1991.

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

[44] D. Porto, J. Leitão, C. Li, A. Clement, A. Kate, F. Junqueira, and R. Rodrigues. Visigoth fault tolerance. In *Proceedings of the Tenth European Conference on Computer Systems*, page 8. ACM, 2015.

[45] H. P. Reiser and R. Kapitza. Hypervisor-based efficient proactive recovery. In *Reliable Distributed Systems, 2007. SRDS 2007. 26th IEEE International Symposium on*, pages 83–92. IEEE, 2007.

[46] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)*, 22(4):299–319, 1990.

[47] M. Serafini, P. Bokor, D. Dobre, M. Majuntke, and N. Suri. Scrooge: Reducing the costs of fast byzantine replication in presence of unresponsive replicas. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 353–362. IEEE, 2010.

[48] H.-S. Siu, Y.-H. Chin, and W.-P. Yang. A note on consensus on dual failure modes. *IEEE Transactions on Parallel and Distributed Systems*, 7(3):225–230, 1996.

[49] Y. J. Song and R. van Renesse. Bosco: One-step byzantine asynchronous consensus. In *International Symposium on Distributed Computing*, pages 438–450. Springer, 2008.

[50] J. Sousa, E. Alchieri, and A. Bessani. State machine replication for the masses with bft-smart. 2013.

[51] P. Thambidurai, Y.-K. Park, et al. Interactive consistency with multiple failure modes. In *Proceedings Seventh Symposium on Reliable Distributed Systems*, pages 93–100. IEEE, 1988.

[52] G. S. Veronese, M. Correia, A. N. Bessani, and L. C. Lung. Spin one's wheels? byzantine fault tolerance with a spinning primary. In *Reliable Distributed Systems, 2009. SRDS'09. 28th IEEE International Symposium on*, pages 135–144. IEEE, 2009.

[53] G. S. Veronese, M. Correia, A. N. Bessani, and L. C. Lung. Ebawa: Efficient byzantine agreement for wide-area networks. In *High-Assurance Systems Engineering (HASE), 2010 IEEE 12th International Symposium on*, pages 10–19. IEEE, 2010.

[54] G. S. Veronese, M. Correia, A. N. Bessani, L. C. Lung, and P. Verissimo. Efficient byzantine fault-tolerance. *IEEE Transactions on Computers*, 62(1):16–30, 2013.

[55] S.-S. Wang, K.-Q. Yan, and S.-C. Wang. Achieving efficient agreement within a dual-failure cloud-computing environment. *Expert Systems with Applications*, 38(1):906–915, 2011.

[56] T. Wood, R. Singh, A. Venkataramani, P. Shenoy, and E. Cecchet. Zz and the art of practical bft execution. In *The conference on Computer systems*, pages 123–138. ACM, 2011.

[57] Z. Wu, M. Butkiewicz, D. Perkins, E. Katz-Bassett, and H. V. Madhyastha. Spanstore: Cost-effective geo-replicated storage spanning multiple cloud services. In *ACM Symposium on Operating Systems Principles*, pages 292–308. ACM, 2013.

[58] J. Yin, J. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin. Separating agreement from execution for byzantine fault tolerant services. *ACM SIGOPS Operating Systems Review*, 37(5):253–267, 2003.

[59] Q. Zhang, L. Cheng, and R. Boutaba. Cloud computing: state-of-the-art and research challenges. *Journal of internet services and applications*, 1(1):7–18, 2010.