XFT: Practical Fault Tolerance Beyond Crashes

Shengyun Liu NUDT* lius@eurecom.fr Paolo Viotti EURECOM paolo.viotti@eurecom.fr Christian Cachin IBM Research - Zurich cca@zurich.ibm.com

Vivien Quéma INP Grenoble vivien.quema@imag.fr Marko Vukolić IBM Research - Zurich mvu@zurich.ibm.com

Abstract

Despite years of intensive research, Byzantine fault-tolerant (BFT) systems have not yet been adopted in practice. This is due to additional cost of BFT in terms of resources, protocol complexity and performance, compared with crash fault-tolerance (CFT). This overhead of BFT comes from the assumption of a powerful adversary that can fully *control* not only the Byzantine faulty machines, but at the same time also the message delivery schedule across the *entire* network, effectively inducing communication asynchrony and partitioning otherwise correct machines at will. To many practitioners, however, such strong attacks appear irrelevant.

In this paper, we introduce cross fault tolerance or XFT, a novel approach to building reliable and secure distributed systems and apply it to the classical state-machine replication (SMR) problem. In short, an XFT SMR protocol provides the reliability guarantees of widely used asynchronous CFT SMR protocols such as Paxos and Raft, but also tolerates Byzantine faults in combination with network asynchrony, as long as a majority of replicas are correct and communicate synchronously. This allows the development of XFT systems at the price of CFT (already paid for in practice), yet with strictly stronger resilience than CFT — sometimes even stronger than BFT itself.

As a showcase for XFT, we present XPaxos, the first XFT SMR protocol, and deploy it in a geo-replicated setting. Although it offers much stronger resilience than CFT SMR at no extra resource cost, the performance of XPaxos matches that of the state-of-the-art CFT protocols.

1 Introduction

Tolerance to any kind of service disruption, whether caused by a simple hardware fault or by a large-scale disaster, is key for the survival of modern distributed systems. Cloud-scale applications must be inherently resilient, as any outage has direct implications on the business behind them [24].

Modern production systems (e.g., [13, 8]) increase the number of nines of reliability¹ by employing sophisticated distributed protocols that tolerate crash machine faults as well as network faults, such as network partitions or asynchrony, which reflect the inability of otherwise correct machines to communicate among each other in a timely manner. At the heart of these systems typically lies a crash fault-tolerant (CFT) consensus-based state-machine replication (SMR) primitive [35, 10].

These systems cannot deal with non-crash (or Byzantine [29]) faults, which include not only malicious, adversarial behavior, but also arise from errors in the hardware, stale or corrupted data from storage systems, memory errors caused by physical effects, bugs in software, hardware faults due to ever smaller circuits, and human mistakes that cause state corruptions and data loss. However, such problems do occur in practice — each of these faults has a public record of taking down major production systems and corrupting their service [14, 4].

^{*}Work done while being a PhD student at EURECOM.

¹As an illustration, five nines reliability means that a system is up and correctly running at least 99.999% of the time. In other words, malfunction is limited to one hour every 10 years on average.

Despite more than 30 years of intensive research since the seminal work of Lamport, Shostak and Pease [29], no practical answer to tolerating non-crash faults has emerged so far. In particular, asynchronous Byzantine fault-tolerance (BFT), which promises to resolve this problem [9], has not lived up to this expectation, largely because of its extra cost compared with CFT. Namely, asynchronous (that is, "eventually synchronous" [18]) BFT SMR must use at least 3t + 1 replicas to tolerate t non-crash faults [7] instead of only 2t + 1 replicas for CFT, as used by Paxos [27] or Raft [33], for example.

The overhead of asynchronous BFT is due to the extraordinary power given to the adversary, which may control both the Byzantine faulty machines and the entire network in a coordinated way. In particular, the classical BFT adversary can partition any number of otherwise correct machines at will. In line with observations by practitioners [25], we claim that this adversary model is actually too strong for the phenomena observed in deployed systems. For instance, accidental non-crash faults usually do not lead to network partitions. Even malicious non-crash faults rarely cause the whole network to break down in wide-area networks and geo-replicated systems. The proverbial all-powerful attacker as a common source behind those faults is a popular and powerful simplification used for the design phase, but it has not seen equivalent proliferation in practice.

In this paper, we introduce XFT (short for cross fault tolerance), a novel approach to building efficient resilient distributed systems that tolerate both non-crash (Byzantine) faults and network faults (asynchrony). In short, XFT allows building resilient systems that

- do not use extra resources (replicas) compared with asynchronous CFT;
- preserve all reliability guarantees of asynchronous CFT (that is, in the absence of Byzantine faults); and
- provide correct service (i.e., safety and liveness [2]) even when Byzantine faults do occur, as long as a majority of the replicas are correct and can communicate with each other synchronously (that is, when a minority of the replicas are Byzantine-faulty or partitioned because of a network fault).

In particular, we envision XFT for wide-area or *geo-replicated* systems [13], as well as for any other deployment where an adversary cannot easily coordinate enough network partitions and Byzantine-faulty machine actions at the same time.

As a showcase for XFT, we present XPaxos, the first state-machine replication protocol in the XFT model. XPaxos tolerates faults beyond crashes in an efficient and practical way, achieving much greater coverage of realistic failure scenarios than the state-of-the-art CFT SMR protocols, such as Paxos or Raft. This comes without resource overhead as XPaxos uses 2t+1 replicas. To validate the performance of XPaxos, we deployed it in a geo-replicated setting across Amazon EC2 datacenters worldwide. In particular, we integrated XPaxos within Apache ZooKeeper, a prominent and widely used coordination service for cloud systems [19]. Our evaluation on EC2 shows that XPaxos performs almost as well in terms of throughput and latency as a WAN-optimized variant of Paxos, and significantly better than the best available BFT protocols. In our evaluation, XPaxos even outperforms the native CFT SMR protocol built into ZooKeeper [20].

Finally, and perhaps surprisingly, we show that XFT can offer *strictly stronger* reliability guarantees than state-of-the-art BFT, for instance under the assumption that machine faults and network faults occur as independent and identically distributed random variables, for certain probabilities. To this end, we calculate the number of nines of consistency (system safety) and availability (system liveness) of resource-optimal CFT, BFT and XFT (e.g., XPaxos) protocols. Whereas XFT *always* provides strictly stronger consistency and availability guarantees than CFT and *always* strictly stronger availability guarantees than BFT, our reliability analysis shows that, in some cases, XFT also provides strictly stronger consistency guarantees than BFT.

The remainder of this paper is organized as follows. In Section 2, we define the system model, which is then followed by the definition of the XFT model in Section 3. In Section 4 and Section 5, we present XPaxos and its evaluation in the geo-replicated context, respectively. Section 6 provides simplified reliability analysis comparing XFT with CFT and BFT. We overview related work and

conclude in Section 7. The full pseudocode and correctness proof of XPaxos is given in Appendix B and C.

2 System model

Machines. We consider a message-passing distributed system containing a set Π of $n = |\Pi|$ machines, also called *replicas*. Additionally, there is a separate set C of *client* machines.

Clients and replicas may suffer from Byzantine faults: we distinguish between crash faults, where a machine simply stops all computation and communication, and non-crash faults, where a machine acts arbitrarily, but cannot break cryptographic primitives we use (cryptographic hashes, MACs, message digests and digital signatures). A machine that is not faulty is called correct. We say a machine is benign if the machine is correct or crash-faulty. We further denote the number of replica faults at a given moment s by

- $t_c(s)$: the number of crash-faulty replicas, and
- $t_{nc}(s)$: the number of non-crash-faulty replicas.

Network. Each pair of replicas is connected with reliable point-to-point bi-directional communication channels. In addition, each client can communicate with any replica.

The system can be asynchronous in the sense that machines may not be able to exchange messages and obtain responses to their requests in time. In other words, network faults are possible; we define a network fault as the inability of some correct replicas to communicate with each other in a timely manner, that is, when a message exchanged between two correct replicas cannot be delivered and processed within delay Δ , known to all replicas. Note that Δ is a deployment specific parameter: we discuss practical choices for Δ in the context of our geo-replicated setting in Section 5. Finally, we assume an eventually synchronous system in which, eventually, network faults do not occur [18].

Note that we model an excessive processing delay as a network problem and *not* as an issue related to a machine fault. This choice is made consciously, rooted in the experience that for the general class of protocols considered in this work, a long local processing time is never an issue on correct machines compared with network delays.

To help quantify the number of network faults, we first give the definition of partitioned replica.

Definition 1 (Partitioned replica). Replica p is partitioned if p is **not** in the largest subset of replicas, in which every pair of replicas can communicate among each other within delay Δ .

If there is more than one subset with the maximum size, only one of them is recognized as the largest subset. For example in Figure 1, the number of partitioned replicas is 3, counting either the group of p_1 , p_4 and p_5 or that of p_2 , p_3 and p_5 . The number of partitioned replicas can be as much as n-1, which means that no two replicas can communicate with each other within delay Δ . We say replica p is synchronous if p is not partitioned. We now quantify network faults at a given moment s as

• $t_p(s)$: the number of correct, but partitioned replicas.

Problem. In this paper, we focus on the *deterministic* state-machine replication problem (SMR) [35]. In short, in SMR clients invoke requests, which are then committed by replicas. SMR ensures

- safety, or consistency, by (a) enforcing total order across committed client's requests across all correct replicas; and by (b) enforcing validity, i.e., that a correct replica commits a request only if it was previously invoked by a client;
- liveness, or availability, by eventually committing a request by a correct client at all correct replicas and returning an application-level reply to the client.

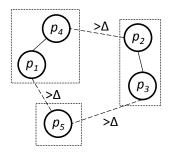


Figure 1: An illustration of partitioned replicas: $\{p_1, p_4, p_5\}$ or $\{p_2, p_3, p_5\}$ are partitioned based on Definition 1.

3 The XFT model

This section introduces the XFT model and relates it to the established crash-fault tolerance (CFT) and Byzantine-fault tolerance (BFT) models.

3.1 XFT in a nutshell

	Maximum number of each type of replica faults				
		non-crash faults	crash faults	partitioned replicas	
Asynchronous CFT (e.g., Paxos [28])	consistency	0 n		n-1	
Asynchronous Of T (e.g., Taxos [20])	availability	$0 \qquad \qquad \lfloor \frac{n-1}{2} \rfloor$		(combined)	
Asynchronous BFT (e.g., PBFT [9])	consistency	$\lfloor \frac{n-1}{3} \rfloor$	n	n-1	
Asylichronous Bi I (c.g., I Bi I [J])	availability		ned)		
(Authenticated) Synchronous BFT (e.g., [29])	consistency	n-1	n	0	
(Authenoicated) Synchronous Bi 1 (e.g., [25])	availability	n-1 (combined)		0	
	consistency	0	n	n-1	
XFT (e.g., XPaxos)	Consistency	$\lfloor \frac{n-1}{2} \rfloor$ (combined)			
	availability	$\lfloor \frac{n-1}{2} \rfloor$ (combined)			

Table 1: The maximum numbers of each type of fault tolerated by representative SMR protocols. Note that XFT provides consistency in two modes, depending on the occurrence of non-crash faults.

Classical CFT and BFT explicitly model machine faults only. These are then combined with an orthogonal network fault model, either the synchronous model (where network faults in our sense are ruled out), or the asynchronous model (which includes *any number* of network faults). Hence, previous work can be classified into four categories: synchronous CFT [16, 35], asynchronous CFT [35, 27, 32], synchronous BFT [29, 17, 6], and asynchronous BFT [9, 3].

XFT, in contrast, redefines the boundaries between machine and network fault dimensions: XFT allows the design of reliable protocols that tolerate crash machine faults regardless of the number of network faults and that, at the same time, tolerate non-crash machine faults when the number of machines that are either faulty or partitioned is within a threshold.

To formalize XFT, we first define *anarchy*, a very severe system condition with actual non-crash machine (replica) faults and plenty of faults of different kinds, as follows:

Definition 2 (Anarchy). The system is in anarchy at a given moment s iff $t_{nc}(s) > 0$ and $t_c(s) + t_{nc}(s) + t_p(s) > t$.

Here, t is the threshold of replica faults, such that $t \leq \lfloor \frac{n-1}{2} \rfloor$. In other words, in anarchy, some replica is non-crash-faulty, and there is no correct and synchronous majority of replicas. Armed with

the definition of anarchy, we can define XFT protocols for an arbitrary distributed computing problem in function of its safety property [2].

Definition 3 (XFT protocol). Protocol P is an XFT protocol if P satisfies safety in all executions in which the system is never in anarchy.

Liveness of an XFT protocol will typically depend on a problem and implementation. For instance, for deterministic SMR we consider in this paper, our XPaxos protocol eventually satisfies liveness, provided a majority of replicas is correct and synchronous. This can be shown optimal.

3.2 XFT vs. CFT/BFT

Table 1 illustrates differences between XFT and CFT/BFT in terms of their consistency and availability guarantees for SMR.

State-of-the-art asynchronous CFT protocols [28, 33] guarantee consistency despite any number of crash-faulty replicas and any number of partitioned replicas. They also guarantee availability whenever a majority of replicas $(t \leq \lfloor \frac{n-1}{2} \rfloor)$ are correct and synchronous. As soon as a single machine is non-crash-faulty, CFT protocols guarantee neither consistency nor availability.

Optimal asynchronous BFT protocols [9, 22, 3] guarantee consistency despite any number of crashfaulty or partitioned replicas, with at most $t = \lfloor \frac{n-1}{3} \rfloor$ non-crash-faulty replicas. They also guarantee availability with up to $\lfloor \frac{n-1}{3} \rfloor$ combined faults, i.e., whenever more than two-thirds of replicas are correct and not partitioned. Note that BFT availability might be weaker than that of CFT in the absence of non-crash faults — unlike CFT, BFT does not guarantee availability when the sum of crash-faulty and partitioned replicas is in the range $\lfloor n/3, n/2 \rfloor$.

Synchronous BFT protocols (e.g., [29]) do not consider the existence of correct, but partitioned replicas. This makes for a very strong assumption — and helps synchronous BFT protocols that use digital signatures for message authentication (so called *authenticated* protocols) to tolerate up to n-1 non-crash-faulty replicas.

In contrast, XFT protocols with optimal resilience, such as our XPaxos, guarantee consistency in two modes: (i) without non-crash faults, despite any number of crash-faulty and partitioned replicas (i.e., just like CFT), and (ii) with non-crash faults, whenever a majority of replicas are correct and not partitioned, i.e., provided the sum of all kinds of faults (machine or network faults) does not exceed $\lfloor \frac{n-1}{2} \rfloor$. Similarly, it also guarantees availability whenever a majority of replicas are correct and not partitioned.

It may be tempting to view XFT as some sort of a combination of the asynchronous CFT and synchronous BFT models. However, this is misleading, as even with actual non-crash faults, XFT is incomparable to authenticated synchronous BFT. Specifically, authenticated synchronous BFT protocols, such as the seminal Byzantine Generals protocol [29], may violate consistency with a single partitioned replica. For instance, with n=5 replicas and an execution in which three replicas are correct and synchronous, one replica is correct but partitioned and one replica is non-crash-faulty, the XFT model mandates that the consistency be preserved, whereas the Byzantine Generals protocol may violate consistency.²

Furthermore, from Table 1, it is evident that XFT offers strictly stronger guarantees than asynchronous CFT, for both availability and consistency. XFT also offers strictly stronger availability guarantees than asynchronous BFT. Finally, the consistency guarantees of XFT are incomparable to those of asynchronous BFT. On the one hand, outside anarchy, XFT is consistent with the number of non-crash faults in the range [n/3, n/2), whereas asynchronous BFT is not. On the other hand, unlike XFT, asynchronous BFT is consistent in anarchy provided the number of non-crash faults is less than n/3. We discuss these points further in Section 6, where we also quantify the reliability comparison between XFT and asynchronous CFT/BFT assuming the special case of independent faults.

 $^{^{2}}$ XFT is not stronger than authenticated synchronous BFT either, as the latter tolerates more machine faults in the complete absence of network faults.

3.3 Where to use XFT?

The intuition behind XFT starts from the assumption that "extremely bad" system conditions, such as anarchy, are very rare, and that providing consistency guarantees in anarchy might not be worth paying the asynchronous BFT premium.

In practice, this assumption is plausible in many deployments. We envision XFT for use cases in which an adversary cannot easily coordinate enough network partitions and non-crash-faulty machine actions at the same time. Some interesting candidate use cases include:

- Tolerating "accidental" non-crash faults. In systems which are not susceptible to malicious behavior and deliberate attacks, XFT can be used to protect against "accidental" non-crash faults, which can be assumed to be largely independent of network faults. In such cases, XFT could be used to harden CFT systems without considerable overhead of BFT.
- Wide-area networks and geo-replicated systems. XFT may reveal useful even in cases where the system is susceptible to malicious non-crash faults, as long as it may be difficult or expensive for an adversary to coordinate an attack to compromise Byzantine machines and partition sufficiently many replicas at the same time. Particularly interesting for XFT are WAN and geo-replicated systems which often enjoy redundant communication paths and typically have a smaller surface for network-level DoS attacks (e.g., no multicast storms and flooding).
- Blockchain. A special case of geo-replicated systems, interesting to XFT, are blockchain systems. In a typical blockchain system, such as Bitcoin [31], participants may be financially motivated to act maliciously, yet may lack the means and capabilities to compromise the communication among (a large number of) correct participants. In this context, XFT is particularly interesting for so-called permissioned blockchains, which are based on state-machine replication rather than on Bitcoin-style proof-of-work [39].

4 XPaxos Protocol

4.1 XPaxos overview

XPaxos is a novel state-machine replication (SMR) protocol designed specifically in the XFT model. XPaxos specifically targets good performance in geo-replicated settings, which are characterized by the network being the bottleneck, with high link latency and relatively low, heterogeneous link bandwidth. In a nutshell, XPaxos consists of three main components:

- A common-case protocol, which replicates and totally orders requests across replicas. This has, roughly speaking, the message pattern and complexity of communication among replicas of state-of-the-art CFT protocols (e.g., Phase 2 of Paxos), hardened by the use of digital signatures.
- A novel view-change protocol, in which the information is transferred from one view (system configuration) to another in a *decentralized*, leaderless fashion.
- A fault detection (FD) mechanism, which can help detect, outside anarchy, non-crash faults that would leave the system in an inconsistent state in anarchy. The goal of the FD mechanism is to minimize the impact of long-lived non-crash faults (in particular "data loss" faults) in the system and to help detect them before they coincide with a sufficient number of crash faults and network faults to push the system into anarchy.

XPaxos is orchestrated in a sequence of views [9]. The central idea in XPaxos is that, during common-case operation in a given view, XPaxos synchronously replicates clients' requests to only t+1 replicas, which are the members of a synchronous group (out of n=2t+1 replicas in total). Each view number i uniquely determines the synchronous group, sg_i , using a mapping known to all replicas. Every synchronous group consists of one primary and t followers, which are jointly called active replicas. The remaining t replicas in a given view are called passive replicas; optionally, passive

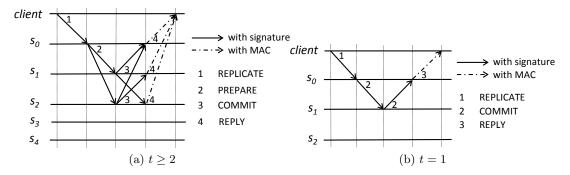


Figure 2: XPaxos common-case message patterns (a) for the general case when $t \ge 2$ and (b) for the special case of t = 1. The synchronous groups are (s_0, s_1, s_2) and (s_0, s_1) , respectively.

replicas learn the order from the active replicas using the *lazy replication* approach [26]. A view is not changed unless there is a machine or network fault within the synchronous group.

In the common case (Section 4.2), the clients send digitally signed requests to the primary, which are then replicated across t + 1 active replicas. These t + 1 replicas digitally sign and locally log the proofs for all replicated requests to their *commit logs*. Commit logs then serve as the basis for maintaining consistency in view changes.

The view change of XPaxos (Section 4.3) reconfigures the entire synchronous group, not only the leader. All t+1 active replicas of the new synchronous group sg_{i+1} try to transfer the state from the preceding views to view i+1. This decentralized approach to view change stands in sharp contrast to the classical reconfiguration/view-change in CFT and BFT protocols (e.g., [27, 9]), in which only a single replica (the primary) leads the view change and transfers the state from previous views. This difference is crucial to maintaining consistency (i.e., total order) across XPaxos views in the presence of non-crash faults (but in the absence of full anarchy). This novel and decentralized view-change scheme of XPaxos guarantees that even in the presence of non-crash faults, but outside anarchy, at least one correct replica from the new synchronous group sg_{i+1} will be able to transfer the correct state from previous views, as it will be able to contact some correct replica from any old synchronous group.

Finally, the main idea behind the FD scheme of XPaxos is the following. In view change, a non-crash-faulty replica (of an old synchronous group) might not transfer its latest state to a correct replica in the new synchronous group. This "data loss" fault is dangerous, as it may violate consistency when the system is in anarchy. However, such a fault can be detected using digital signatures from the commit log of some correct replicas (from an old synchronous group), provided that these correct replicas can communicate synchronously with correct replicas from the new synchronous group. In a sense, with XPaxos FD, a critical non-crash machine fault must occur for the first time together with sufficiently many crash or partitioned machines (i.e., in anarchy) to violate consistency.

In the following, we explain the core of XPaxos for the common case (Section 4.2), view-change (Section 4.3) and fault detection (Section 4.4) components. We discuss XPaxos optimizations in Section 4.5 and give XPaxos correctness arguments in Section 4.6. An example of XPaxos execution is given in Appendix A. The complete pseudocode and correctness proof are included in Appendix B and C.

4.2 Common case

Figure 2 shows the common-case message patterns of XPaxos for the general case $(t \ge 2)$ and for the special case t = 1. XPaxos is specifically optimized for the case where t = 1, as in this case, there are only two active replicas in each view and the protocol is very efficient. The special case t = 1 is also highly relevant in practice (see e.g., Spanner [13]). In the following, we first explain XPaxos in the general case, and then focus on the t = 1 special case.

Notation. We denote the digest of a message m by D(m), whereas $\langle m \rangle_{\sigma_p}$ denotes a message that contains both D(m) signed by the private key of machine p and m. For signature verification, we

assume that all machines have public keys of all other processes.

4.2.1 General case $(t \ge 2)$

The common-case message pattern of XPaxos is shown in Figure 2a. More specifically, upon receiving a signed request $req = \langle \text{REPLICATE}, op, ts_c, c \rangle_{\sigma_c}$ from client c (where op is the client's operation and ts_c is the client's timestamp), the primary (say s_0) (1) increments sequence number sn and assigns sn to req, (2) signs a message $prep = \langle \text{PREPARE}, D(req), sn, i \rangle_{\sigma_{s_0}}$ and logs $\langle req, prep \rangle$ into its prepare log $PrepareLog_0[sn]$ (we say s_0 prepares req), and (3) forwards $\langle req, prep \rangle$ to all other active replicas (i.e, the t followers).

Each follower s_j $(1 \le j \le t)$ verifies the primary's and client's signatures, checks whether its local sequence number equals sn-1, and logs $\langle req, prep \rangle$ into its prepare log $PrepareLog_j[sn]$. Then, s_j updates its local sequence number to sn, signs the digest of the request req, the sequence number sn and the view number i, and sends $\langle COMMIT, D(req), sn, i \rangle_{\sigma_{s_i}}$ to all active replicas.

Upon receiving t signed COMMIT messages — one from each follower — such that a matching entry is in the prepare log, an active replica s_k ($0 \le k \le t$) logs prep and the t signed COMMIT messages into its commit log $CommitLog_{s_k}[sn]$. We say s_k commits req when this occurs. Finally, s_k executes req and sends the authenticated reply to the client (followers may only send the digest of the reply). The client commits the request when it receives matching REPLY messages from all t+1 active replicas.

A client that times out without committing the requests broadcasts the request to all active replicas. Active replicas then forward such a request to the primary and trigger a *retransmission timer*, within which a correct active replica expects the client's request to be committed.

4.2.2 Tolerating a single fault (t = 1).

When t = 1, the XPaxos common case simplifies to involving only 2 messages between 2 active replicas (see Figure 2b).

Upon receiving a signed request $req = \langle \text{REPLICATE}, op, ts_c, c \rangle_{\sigma_c}$ from client c, the primary (s_0) increments the sequence number sn, signs sn along the digest of req and view number i in message $m_0 = \langle \text{COMMIT}, D(req), sn, i \rangle_{\sigma_{s_0}}$, stores $\langle req, m_0 \rangle$ into its prepare $\log (PrepareLog_{s_0}[sn] = \langle req, m_0 \rangle)$, and sends the message $\langle req, m_0 \rangle$ to the follower s_1 .

On receiving $\langle req, m_0 \rangle$, the follower s_1 verifies the client's and primary's signatures, and checks whether its local sequence number equals sn-1. If so, the follower updates its local sequence number to sn, executes the request producing reply R(req), and signs message m_1 ; m_1 is similar to m_0 , but also includes the client's timestamp and the digest of the reply: $m_1 = \langle \text{COMMIT}, \langle D(req), sn, i, req.ts_c, D(R(req)) \rangle_{\sigma_{s_1}}$. The follower then saves the tuple $\langle req, m_0, m_1 \rangle$ to its commit log $\langle CommitLog_{s_1}[sn] = \langle req, m_0, m_1 \rangle$) and sends m_1 to the primary.

The primary, on receiving a valid COMMIT message from the follower (with a matching entry in its prepare log), executes the request, compares the reply R(req) with the follower's digest contained in m_1 , and stores $\langle req, m_0, m_1 \rangle$ in its commit log. Finally, it returns an authenticated reply containing m_1 to c, which commits the request if all digests and the follower's signature match.

4.3 View change

Intuition. The ordered requests in commit logs of correct replicas are the key to enforcing consistency (total order) in XPaxos. To illustrate an XPaxos view change, consider synchronous groups sg_i and sg_{i+1} of views i and i+1, respectively, each containing t+1 replicas. Note that proofs of requests committed in sg_i might have been logged by only one correct replica in sg_i . Nevertheless, the XPaxos view change must ensure that (outside anarchy) these proofs are transferred to the new view i+1. To this end, we had to depart from traditional view change techniques [9, 22, 12] where the entire view-change is led by a single replica, usually the primary of the new view. Instead, in XPaxos view change, every active replica in sg_{i+1} retrieves information about requests committed in preceding views. Intuitively, with correct majority of correct and synchronous replicas, at least one correct and synchronous replica from sg_{i+1} will contact (at least one) correct and synchronous replica from sg_i and transfer the latest correct commit log to the new view i+1.

		Syno	$ \begin{array}{c} \text{chronous} \\ (i \in \mathbb{N}) \end{array} $	Groups
		sg_i	sg_{i+1}	sg_{i+2}
Active replicas	Primary	s_0	s_0	s_1
Active replicas	Follower	s_1	s_2	s_2
Passive rep	olica	s_2	s_1	s_0

Table 2: Synchronous group combinations (t = 1).

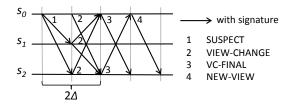


Figure 3: Illustration of XPaxos view change: the synchronous group is changed from (s_0, s_1) to (s_0, s_2) .

In the following, we first describe how we choose active replicas for each view. Then, we explain how view changes are initiated, and, finally, how view changes are performed.

4.3.1 Choosing active replicas

To choose active replicas for view i, we may enumerate all sets containing t+1 replicas (i.e., $\binom{2t+1}{t+1}$ sets) which then alternate as synchronous groups across views in a round-robin fashion. In addition, each synchronous group uniquely determines the primary. We assume that the mapping from view numbers to synchronous groups is known to all replicas (see e.g., Table 2).

The above simple scheme works well for small number of replicas (e.g., t = 1 and t = 2). For a large number of replicas, the combinatorial number of synchronous groups may be inefficient. To this end, XPaxos can be modified to rotate only the leader, which may then resort to deterministic verifiable pseudorandom selection of the set of f followers in each view. The exact details of such a scheme would, however, exceed the scope of this paper.

4.3.2 View-change initiation

If a synchronous group in view i (denoted by sg_i) does not make progress, XPaxos performs a view change. Only an active replica of sg_i may initiate a view change. An active replica $s_j \in sg_i$ initiates a view change if (i) s_j receives a message from another active replica that does not conform to the protocol (e.g., an invalid signature), (ii) the retransmission timer at s_j expires, (iii) s_j does not complete a view change to view i in a timely manner, or (iv) s_j receives a valid SUSPECT message for view i from another replica in sg_i . Upon a view-change initiation, s_j stops participating in the current view and sends $\langle \text{SUSPECT}, i, s_j \rangle_{\sigma_{s_i}}$ to all other replicas.

4.3.3 Performing the view change

Upon receiving a SUSPECT message from an active replica in view i (see the message pattern in Figure 3), replica s_j stops processing messages of view i and sends $m = \langle \text{VIEW-CHANGE}, i+1, s_j, CommitLog_{s_j} \rangle_{\sigma_{s_j}}$ to the t+1 active replicas of sg_{i+1} . A VIEW-CHANGE message contains the commit $\log CommitLog_{s_j}$ of s_j . Commit $\log sight$ be empty (e.g., if s_j was passive).

Note that XPaxos requires all active replicas in the new view to collect the most recent state and its proof (i.e., VIEW-CHANGE messages), rather than only the new primary. Otherwise, a faulty new primary could, even outside anarchy, purposely omit VIEW-CHANGE messages that contain the most recent state. Active replica s_j in view i+1 waits for at least n-t VIEW-CHANGE messages from all, but also waits for 2Δ time, trying to collect as many messages as possible.

Upon completion of the above protocol, each active replica $s_j \in sg_{i+1}$ inserts all VIEW-CHANGE messages it has received into set $VCSet_{s_j}^{i+1}$. Then s_j sends $\langle VC\text{-FINAL}, i+1, s_j, VCSet_{s_j}^{i+1} \rangle_{\sigma_{s_j}}$ to every

active replica in view i + 1. This serves to exchange the received VIEW-CHANGE messages among active replicas.

Every active replica $s_j \in sg_{i+1}$ must receive VC-FINAL messages from all active replicas in sg_{i+1} , after which s_j extends the value $VCSet_{s_j}^{i+1}$ by combining $VCSet_*^{i+1}$ sets piggybacked in VC-FINAL messages. Then, for each sequence number sn, an active replica selects the commit log with the highest view number in all VIEW-CHANGE messages, to confirm the committed request at sn.

Afterwards, to prepare and commit the selected requests in view i+1, the new primary ps_{i+1} sends $\langle \text{NEW-VIEW}, i+1, PrepareLog \rangle_{\sigma_{ps_{i+1}}}$ to every active replica in sg_{i+1} , where the array PrepareLog contains the prepare logs generated in view i+1 for each selected request. Upon receiving a NEW-VIEW message, every active replica $s_j \in sg_{i+1}$ processes the prepare logs in PrepareLog as described in the common case (see Section 4.2).

Finally, every active replica $s_j \in sg_{i+1}$ makes sure that all selected requests in PrepareLog are committed in view i + 1. When this condition is satisfied, XPaxos can start processing new requests.

4.4 Fault detection

XPaxos does not guarantee consistency in anarchy. Hence, non-crash faults could violate XPaxos consistency in the long run, if they persist long enough to eventually coincide with enough crash or network faults. To cope with long-lived faults, we propose (an otherwise optional) Fault Detection (FD) mechanism for XPaxos.

Roughly speaking, FD guarantees the following property: if a machine p suffers a non-crash fault outside anarchy in a way that would cause inconsistency in anarchy, then XPaxos FD detects p as faulty (outside anarchy). In other words, any potentially fatal fault that occurs outside anarchy would be detected by XPaxos FD.

Here, we sketch how FD works in the case t = 1 (see Section B.4 for details), focusing on detecting a specific non-crash fault that may render XPaxos inconsistent in anarchy — a data loss fault by which a non-crash-faulty replica loses some of its commit log prior to a view change. Intuitively, data loss faults are dangerous as they cannot be prevented by the straightforward use of digital signatures.

Our FD mechanism entails modifying the XPaxos view change as follows: in addition to exchanging their commit logs, replicas also exchange their prepare logs. Notice that in the case t = 1 only the primary maintains a prepare log (see Section 4.2). In the new view, the primary prepares and the follower commits all requests contained in transferred commit and prepare logs.

With the above modification, to violate consistency, a faulty primary (of preceding view i) would need to exhibit a data loss fault in both its commit log and its prepare log. However, such a data loss fault in the primary's prepare log would be detected, outside anarchy, because (i) the (correct) follower of view i would reply in the view change and (ii) an entry in the primary's prepare log causally precedes the respective entry in the follower's commit log. By simply verifying the signatures in the follower's commit log, the fault of a primary is detected. Conversely, a data loss fault in the commit log of the follower of view i is detected outside anarchy by verifying the signatures in the commit log of the primary of view i.

4.5 XPaxos optimizations

Although the common-case and view-change protocols described above are sufficient to guarantee correctness, we applied several standard performance optimizations to XPaxos. These include check-pointing and lazy replication [26] to passive replicas (to help shorten the state transfer during view change) as well as batching and pipelining (to improve the throughput).

4.5.1 Checkpointing

Upon active replica $s_j \in sg_i$ commits and executes the request with sequence number $sn = k \times CHK$ (refer to message pattern in Fig. 4), s_j sends $\langle PRECHK, sn, i, D(st_{s_j}^{sn}), s_j \rangle_{\mu_{s_j,s_k}}$ to every active replica s_k , where $D(st_{s_j}^{sn})$ is the digest of the state after executing the request at sn. Upon receiving t+1 matching PRECHK messages, each active replica s_j generates the checkpoint proof message m and

sends it to every active replica $(m = \langle \text{CHKPT}, sn, i, D(st_{s_j}^{sn}), s_j \rangle_{\sigma_{s_j}})$. Upon receiving t+1 matching CHKPT messages, each active replica s_j checkpoints the state and discards previous prepare logs and commit logs.

Besides, each active replica propagates checkpoint proofs to all passive replicas by $\langle LAZYCHK, chkProof \rangle$, where chkProof contains t+1 chkPt messages.

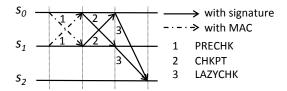


Figure 4: XPaxos checkpointing message pattern: synchronous group is (s_0, s_1) .

4.5.2 Lazy replication

To speed up the state transfer in view change, the followers in synchronous group lazily propagate the commit log to every passive replica. With lazy replication, the new active replica, which might be the passive replica in preceding view, could only retrieve the missing state from others.

More specifically, (refer to message pattern in Fig. 5) in case t = 1, upon committing request req, the follower sends commit log of req to the passive replica. In case $t \geq 2$, either each of t followers sends commit log of req to one passive replica, or each follower sends a fraction of $\frac{1}{t}$ commit logs to every passive replica. Only in case the bandwidth between followers and passive replicas are saturated, the primary is involved in lazy replication. Each passive replica commits and executes requests based on orders defined by commit logs.

Although non-crash faulty replicas can interfere with the lazy replication scheme, this would not impact the correctness of the protocol, but only slow down the view-change.

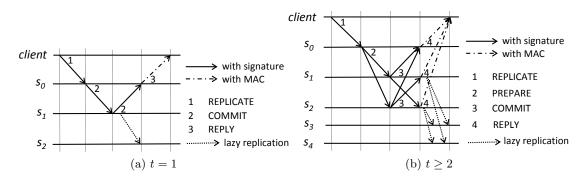


Figure 5: XPaxos common-case message patterns with lazy replication for t = 1 and $t \ge 2$ (here t = 2). Synchronous group illustrated are (s_0, s_1) (when t = 1) and (s_0, s_1, s_2) (when t = 2), respectively.

Batching and pipelining. To improve the throughput of cryptographic operations, the primary batches several requests when preparing. The primary waits for B requests, then signs the batched request and sends it to every follower. If primary receives less than B requests within a time limit, the primary batches all requests it has received.

4.6 Correctness arguments

Consistency (Total Order). XPaxos enforces the following invariant, which is key to total order.

Lemma 1. Outside anarchy, if a benign client c commits a request req with sequence number sn in view i, and a benign replica s_k commits the request req' with sn in view i' > i, then req = req'.

A benign client c commits request req with sequence number sn in view i only after c has received matching replies from t+1 active replicas in sg_i . This implies that every benign replica in sg_i stores

req into its commit log under sequence number sn. In the following, we focus on the special case where: i' = i + 1. This serves as the base step for the proof of Lemma 1 by induction across views which we give in Section C.

Recall that, in view i'=i+1, all (benign) replicas from sg_{i+1} wait for n-t=t+1 VIEW-CHANGE messages containing commit logs transferred from other replicas, as well as for the timer set to 2Δ to expire. Then, replicas in sg_{i+1} exchange this information within VC-FINAL messages. Note that, outside anarchy, there exists at least one correct and synchronous replica in sg_{i+1} , say s_j . Hence, a benign replica s_k that commits req' in view i+1 under sequence number sn must have had received VC-FINAL from s_j . In turn, s_j waited for t+1 VIEW-CHANGE messages (and timer 2Δ), so it received a VIEW-CHANGE message from some correct and synchronous replica $s_x \in sg_i$ (such a replica exists in sg_i as at most t replicas in sg_i are non-crash-faulty or partitioned). As s_x stored req under sn in its commit log in view i, it forwards this information to s_j in a VIEW-CHANGE message, and s_j forwards this information to s_k within a VC-FINAL. Hence req = req' follows.

Availability. XPaxos availability is guaranteed if the synchronous group contains only correct and synchronous replicas. With eventual synchrony, we can assume that, eventually, there will be no network faults. In addition, with all combinations of t+1 replicas rotating in the role of active replicas, XPaxos guarantees that, eventually, view change in XPaxos will complete with t+1 correct and synchronous active replicas.

5 Performance Evaluation

	US West 1 (CA)	Europe (EU)	Tokyo (JP)	Sydney (AU)	Sao Paolo (BR)
US East (VA)	88 /1097 /82190 /166390	92 /1112 /85649 /169749	179 /1226 /81177 /165277	268 /1372 /95074 /179174	146 /1214 /85434 /169534
US West 1 (CA)		174 /1184 /1974 /15467	120 /1133 /1180 /6210	186 /1209 /6354 /51646	207 /1252 /90980 /169080
Europe (EU)			287 /1310 /1397 /4798	342 /1375 /3154 /11052	233 /1257 /1382 /9188
Tokyo (JP)				137 /1149 /1414 /5228	394 /2496 /11399 /94775
Sydney (AU)					392 /1496 /2134 /10983

Table 3: Round-trip latency of TCP ping (hping3) across Amazon EC2 datacenters, collected during three months. The latencies are given in milliseconds, in the format: average / 99.99% / maximum.

In this section, we evaluate the performance of XPaxos and compare it to that of Zyzzyva [22], PBFT [9] and a WAN-optimized version of Paxos [27], using the Amazon EC2 worldwide cloud platform. We chose geo-replicated, WAN settings as we believe that these are a better fit for protocols that tolerate Byzantine faults, including XFT and BFT. Indeed, in WAN settings (i) there is no single point of failure such as a switch interconnecting machines, (ii) there are no correlated failures due to, e.g., a power-outage, a storm, or other natural disasters, and (iii) it is difficult for the adversary to flood the network, correlating network and non-crash faults (the last point is relevant for XFT).

In the remainder of this section, we first present the experimental setup (Section 5.1), and then evaluate the performance (throughput, latency and CPU cost) in the fault-free scenario (Section 5.2 and Section 5.3) as well as under faults (Section 5.4). Finally, we perform a performance comparison using a real application, the ZooKeeper coordination service [19] (Section 5.5), by comparing native ZooKeeper to ZooKeeper variants that use the four replication protocols mentioned above.

5.1 Experimental setup

5.1.1 Synchrony and XPaxos

In a practical deployment of XPaxos, a critical parameter is the value of timeout Δ , i.e., the upper bound on the communication delay between any two *correct* machines. If the round-trip time (RTT) between two correct machines takes more than 2Δ , we declare a network fault (see Section 2). Notably, Δ is vital to the XPaxos view-change (Section 4.3).

To understand the value of Δ in our geo-replicated context, we ran a 3-month experiment during which we continuously measured the round-trip latency across six Amazon EC2 datacenters worldwide using TCP ping (hping3). We used the least expensive EC2 micro instances, which arguably have the

highest probability of experiencing variable latency due to virtualization. Each instance was pinging all other instances every 100 ms. The results of this experiment are summarized in Table 3. While we detected network faults lasting up to 3 min, our experiment showed that the round-trip latency between any two datacenters was less than 2.5 sec 99.99% of the time. Therefore, we adopted the value of $\Delta = 2.5/2 = 1.25$ sec.

5.1.2 Protocols under test

We compare XPaxos with three protocols whose common-case message patterns when t=1 are shown in Figure 6. The first two are BFT protocols, namely (a speculative variant of) PBFT [9] and Zyzzyva [22], and require 3t+1 replicas to tolerate t faults. We chose PBFT because it is possible to derive a speculative variant of the protocol that relies on a 2-phase common-case commit protocol across only 2t+1 replicas (Figure 6a; see also [9]). In this PBFT variant, the remaining t replicas are not involved in the common case, which is more efficient in a geo-replicated settings. We chose Zyzzyva because it is the fastest BFT protocol that involves all replicas in the common case (Figure 6b). The third protocol we compare against is a very efficient WAN-optimized variant of crash-tolerant Paxos inspired by [5, 23, 13]. We have chosen the variant of Paxos that exhibits the fastest write pattern (Figure 6c). This variant requires 2t+1 replicas to tolerate t faults, but involves t+1 replicas in the common case, i.e., just like XPaxos.

To provide a fair comparison, all protocols rely on the same Java code base and use batching, with the batch size set to 20. We rely on HMAC-SHA1 to compute MACs and RSA1024 to sign and verify signatures computed using the Crypto++ [1] library that we interface with the various protocols using JNI.

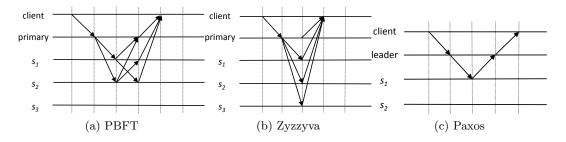


Figure 6: Communication patterns of the three protocols under test (t = 1).

5.1.3 Experimental testbed and benchmarks

We run the experiments on the Amazon EC2 platform which comprises widely distributed datacenters, interconnected by the Internet. Communications between datacenters have a low bandwidth and a high latency. We run the experiments on mid-range virtual machines that contain 8 vCPUs, 15 GB of memory, 2 x 80 GB SSD storage, and run Ubuntu Server 14.04 LTS (PV) with the Linux 3.13.0-24-generic x86_64 kernel.

In the case t = 1, Table 4 gives the deployment of the different replicas at different datacenters, for each protocol analyzed. Clients are always located in the same datacenter as the (initial) primary to better emulate what is done in modern geo-replicated systems where clients are served by the closest datacenter [36, 13].³

To stress the protocols, we run a microbenchmark that is similar to the one used in [9, 22]. In this microbenchmark, each server replicates a *null* service (this means that there is no execution of requests). Moreover, clients issue requests in *closed-loop*: a client waits for a reply to its current request before issuing a new request. The benchmark allows both the request size and the reply size

³In practice, modern geo-replicated system, like Spanner [13], use hundreds of CFT SMR instances across different partitions to accommodate geo-distributed clients.

to be varied. For space limitations, we only report results for two request sizes (1kB, 4kB) and one reply size (0kB). We refer to these microbenchmarks as 1/0 and 4/0 benchmarks, respectively.

5.2 Fault-free performance

We first compare the performance of protocols when t = 1 in replica configurations as shown in Table 4, using the 1/0 and 4/0 microbenchmarks. The results are shown in Figures 7a and 7b. In each graph, the X-axis shows the throughput (in kops/sec), and Y-axis the latency (in ms).

PBFT	Zyzzyva	Paxos	XPaxos	EC2 Region
Primary	Primary	Primary	Primary	US West (CA)
Active		Active	Follower	US East (VA)
Active	Active	Passive	Passive	Tokyo (JP)
Passive		-	-	Europe (EU)

Table 4: Configurations of replicas. Shaded replicas are not used in the common case.

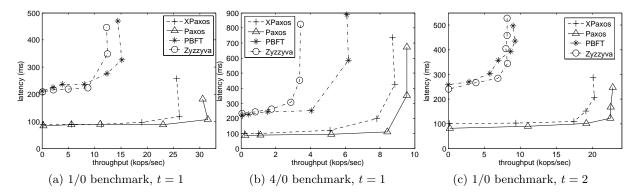


Figure 7: Fault-free performance

As we can see, in both benchmarks, XPaxos achieves a significantly better performance than PBFT and Zyzzyva. This is because, in a worldwide cloud environment, the network is the bottleneck and the message patterns of BFT protocols, namely PBFT and Zyzzyva, tend to be expensive. Compared with PBFT, the simpler message pattern of XPaxos allows better throughput. Compared with Zyzzyva, XPaxos puts less stress on the leader and replicates requests in the common case across 3 times fewer replicas than Zyzzyva (i.e., across t followers vs. across all other 3t replicas). Moreover, the performance of XPaxos is very close to that of Paxos. Both Paxos and XPaxos implement a round-trip across two replicas when t = 1, which renders them very efficient.

Next, to assess the fault scalability of XPaxos, we ran the 1/0 micro-benchmark in configurations that tolerate two faults (t=2). We use the following EC2 datacenters for this experiment: CA (California), OR (Oregon), VA (Virginia), JP (Tokyo), EU (Ireland), AU (Sydney) and SG (Singapore). We place Paxos and XPaxos active replicas in the first t+1 datacenters, and their passive replicas in the next t datacenters. PBFT uses the first 2t+1 datacenters for active replicas and the last t for passive replicas. Finally, Zyzzyva uses all replicas as active replicas.

We observe that XPaxos again clearly outperforms PBFT and Zyzzyva and achieves a performance very close to that of Paxos. Moreover, unlike PBFT and Zyzzyva, Paxos and XPaxos only suffer a moderate performance decrease with respect to the t=1 case.

5.3 CPU cost

To assess the cost of using signatures in XPaxos, we extracted the CPU usage during the experiments presented in Section 5.2 with 1/0 and 4/0 micro-benchmarks when t=1. During experiments, we periodically sampled CPU usage at the most loaded node (the primary in every protocol) with the top Linux monitoring tool. The results are depicted in Figure 8 for both the 1/0 and 4/0 micro-benchmarks. The X-axis represents the peak throughput (in kops/s), whereas the Y-axis represents

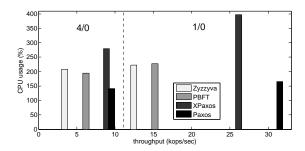


Figure 8: CPU usage when running the 1/0 and 4/0 micro-benchmarks.

the CPU usage (in %). Not surprisingly, we observe that the CPU usage of all protocols is higher with the 1/0 benchmark than with the 4/0 benchmark. This comes from the fact that in the former case, there are more messages to handle per time unit. We also observe that the CPU usage of XPaxos is higher than that of other protocols, due to the use of digital signatures. Nevertheless, this cost remains very reasonable: never more than half of the eight cores available on the experimental machines were used. Note that this cost could probably be significantly reduced by using GPUs, as recently proposed on the EC2 platform. Moreover, compared to BFT protocols (PBFT and Zyzzyva), while CPU usage of XPaxos is higher, XPaxos also sustains a significantly higher throughput.

5.4 Performance under faults

In this section, we analyze the behavior of XPaxos under faults. We run the 1/0 micro-benchmark on three replicas (CA, VA, JP) to tolerate one fault (see also Table 4). The experiment starts with CA and VA as active replicas, and with 2500 clients in CA. At time 180 sec, we crash the follower, VA. At time 300 sec, we crash the CA replica. At time 420 sec, we crash the third replica, JP. Each replica recovers 20 sec after having crashed. Moreover, the timeout 2Δ (used during state transfer in view change, Section 4.3) is set to 2.5 sec (see Section 5.1.1). We show the throughput of XPaxos in function of time in Figure 9, which also indicates the active replicas for each view. We observe that after each crash, the system performs a view change that lasts less than 10 sec, which is very reasonable in a geo-distributed setting. This fast execution of the view-change subprotocol is a consequence of lazy replication in XPaxos that keeps passive replicas updated. We also observe that the throughput of XPaxos changes with the views. This is because the latencies between the primary and the follower and between the primary and clients vary from view to view.

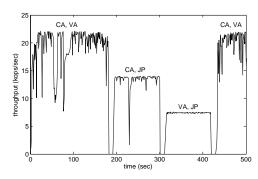


Figure 9: XPaxos under faults.

5.5 Macro-benchmark: ZooKeeper

To assess the impact of our work on real-life applications, we measured the performance achieved when replicating the ZooKeeper coordination service [19] using all protocols considered in this study: Zyzzyva, PBFT, Paxos and XPaxos. We also compare with the native ZooKeeper performance, when the system is replicated using the built-in Zab protocol [20]. This protocol is crash-resilient and requires 2t + 1 replicas to tolerate t faults.

We used the ZooKeeper 3.4.6 codebase. The integration of the various protocols inside ZooKeeper was carried out by replacing the Zab protocol. For fair comparison to native ZooKeeper, we made a minor modification to native ZooKeeper to force it to use (and keep) a given node as primary. To focus the comparison on the performance of replication protocols, and avoid hitting other system bottlenecks (such as storage I/O that is not very efficient in virtualized cloud environments), we store ZooKeeper data and log directories on a volatile tmpfs file system. The configuration tested tolerates one fault (t=1). ZooKeeper clients were located in the same region as the primary (CA). Each client invokes 1 kB write operations in a closed loop.

Figure 10 depicts the results. The X-axis represents the throughput in kops/sec. The Y-axis represents the latency in ms. In this macro-benchmark, we find that Paxos and XPaxos clearly outperform BFT protocols and that XPaxos achieves a performance close to that of Paxos. More surprisingly, we can see that XPaxos is more efficient than the built-in Zab protocol, although the latter only tolerates crash faults. For both protocols, the bottleneck in the WAN setting is the bandwidth at the leader, but the leader in Zab sends requests to all other 2t replicas whereas the XPaxos leader sends requests only to t followers, which yields a higher peak throughput for XPaxos.

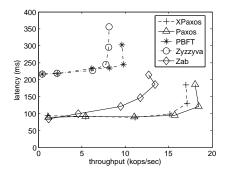


Figure 10: Latency vs. throughput for the ZooKeeper application (t = 1).

6 Reliability Analysis

In this section, we illustrate the reliability guarantees of XPaxos by analytically comparing them with those of the state-of-the-art asynchronous CFT and BFT protocols. For simplicity of the analysis, we consider the fault states of the machines to be independent and identically distributed random variables.

We denote the probability that a replica is correct (resp., crash faulty) by $p_{correct}$ (resp., p_{crash}). The probability that a replica is benign is $p_{benign} = p_{correct} + p_{crash}$. Hence, a replica is non-crash faulty with probability $p_{non-crash} = 1 - p_{benign}$.

Besides, we assume there is a probability $p_{synchrony}$ that a replica is synchronous, where $p_{synchrony}$ is a function of Δ , the network, and the system environment. Therefore, the probability that a replica is partitioned equals $1 - p_{synchrony}$.

Based on the assumption that network faults and machine faults occur independently, it is straightforward to reason for a given machine, p_{benign} and $p_{correct}$ are independent from $p_{synchrony}$. Hence, the probability that a machine is available (i.e., correct and synchronous) is $p_{available} = p_{correct} \times p_{synchrony}$.

Aligned with the industry practice, we measure reliability guarantees and coverage of fault scenarios using nines of reliability. Specifically, we distinguish nines of consistency and nines of availability and use these measures to compare different fault models. We introduce a function 9of(p) that turns a probability p into the corresponding number of "nines", by letting $9of(p) = \lfloor -\log_{10}(1-p)\rfloor$. For example, 9of(0.999) = 3. For brevity, 9_{benign} stands for $9of(p_{benign})$, and so on, for other probabilities of interest. Beyond the analysis and examples that follow, Appendix D contains additional examples of practical values of nines of reliability achieved by XFT, CFT and BFT protocols.

6.1 Consistency

We start with the number of nines of consistency for an asynchronous CFT protocol, denoted by 9ofC(CFT) = 9of(P[CFT is consistent]). As $P[CFT \text{ is consistent}] = p_{benign}^n$, a straightforward calculation yields:

$$9ofC(CFT) = \left[-\log_{10}(1 - p_{benign}) - \log_{10}(\sum_{i=0}^{n-1} p_{benign}^{i}) \right],$$

which gives $9ofC(CFT) \approx 9_{benign} - \lceil \log_{10}(n) \rceil$ for values of p_{benign} close to 1, when p_{benign}^i decreases slowly. As a rule of thumb, for small values of n, i.e., n < 10, we have $9ofC(CFT) \approx 9_{benign} - 1$.

In other words, in typical configurations, where few faults are tolerated [13], a CFT system as a whole loses one nine of consistency from the likelihood that a single replica is benign.

6.1.1 XPaxos vs. CFT

We now quantify the advantage of XPaxos over asynchronous CFT. From Table. 1, if there is no non-crash fault, or there are no more than t faults (machine faults or network faults), XPaxos is consistent, i.e.,

$$\begin{split} P[\mathsf{XPaxos} \text{ is consistent}] &= p_{benign}^n + \sum_{i=1}^{t = \lfloor \frac{n-1}{2} \rfloor} \binom{n}{i} p_{non\text{-}crash}^i \times \\ \sum_{i=0}^{t-i} \binom{n-i}{j} p_{crash}^j \times p_{correct}^{n-i-j} \times \sum_{k=0}^{t-i-j} \binom{n-i-j}{k} p_{synchrony}^{n-i-j-k} \times (1-p_{synchrony})^k. \end{split}$$

To quantify the difference between XPaxos and CFT more tangibly, we calculated 9ofC(XPaxos) and 9ofC(CFT) for all values of 9_{benign} , $9_{correct}$ and $9_{synchrony}$ ($9_{benign} \ge 9_{correct}$) between 1 and 20 in the special cases where t=1 and t=2, which are most relevant in practice. For t=1, we observed the following relation:

$$9ofC(\mathsf{XPaxos}_{t=1}) - 9ofC(CFT_{t=1}) = \\ \begin{cases} 9_{correct} - 1, & 9_{benign} > 9_{synchrony} \text{ and} \\ & 9_{synchrony} = 9_{correct}, \\ min(9_{synchrony}, 9_{correct}), & \text{otherwise.} \end{cases}$$

$$9ofC(\mathsf{XPaxos}_{t=2}) - 9ofC(CFT_{t=2}) = \\ \begin{cases} 2 \times 9_{correct} - 2, & 9_{benign} > 9_{synchrony} \text{ and} \\ & 9_{synchrony} = 9_{correct} > 1, \\ 2 \times 9_{correct}, & 9_{synchrony} > 2 \times 9_{benign} \text{ and} \\ & 9_{benign} = 9_{correct}, \\ 2 \times min(9_{synchrony}, 9_{correct}) - 1, & \text{otherwise.} \end{cases}$$

Hence, for t=1 we observe that the number of nines of consistency XPaxos adds on top of CFT is proportional to the nines of probability for correct or synchronous machine. The added nines are not directly related to p_{benign} , although $p_{benign} \ge p_{correct}$ must hold.

Example 1. When $p_{benign} = 0.9999$ and $p_{correct} = p_{synchrony} = 0.999$, we have $p_{non-crash} = 0.0001$ and $p_{crash} = 0.0009$. In this example, $9 \times p_{non-crash} = p_{crash}$, i.e., if a machine suffers a faults 10 times, then one of these is a non-crash fault and the rest are crash faults. In this case, $9ofC(CFT_{t=1}) = 9_{benign} - 1 = 3$, whereas $9ofC(\mathsf{XPaxos}_{t=1}) - 9ofC(CFT_{t=1}) = 9_{correct} - 1 = 2$, i.e., $9ofC(\mathsf{XPaxos}_{t=1}) = 5$. XPaxos adds 2 nines of consistency on top of CFT and achieves 5 nines of consistency in total.

Example 2. In a slightly different example, let $p_{benign} = p_{synchrony} = 0.9999$ and $p_{correct} = 0.999$, i.e., the network behaves more reliably than in Example 1. $9ofC(CFT_{t=1}) = 9_{benign} - 1 = 3$, whereas $9ofC(\mathsf{XPaxos}_{t=1}) - 9ofC(CFT_{t=1}) = p_{correct} = 3$, i.e., $9ofC(\mathsf{XPaxos}_{t=1}) = 6$. XPaxos adds 3 nines of consistency on top of CFT and achieves 6 nines of consistency in total.

6.1.2 XPaxos vs. BFT

Recall that (see Table 1) SMR in asynchronous BFT model is consistent whenever no more than one-third machines are non-crash faulty. Hence,

$$P[\text{BFT is consistent}] = \sum_{i=0}^{t=\lfloor \frac{n-1}{3} \rfloor} \binom{n}{i} (1 - p_{benign})^i \times p_{benign}^{n-i}.$$

We first examine the conditions under which XPaxos has stronger consistency guarantees than BFT. Fixing the value t of tolerated faults, we observe that P[XPaxos is consistent] > P[BFT is consistent] is equivalent to:

$$\begin{split} p_{benign}^{2t+1} + \sum_{i=1}^{t} \binom{2t+1}{i} p_{non\text{-}crash}^{i} \times \sum_{j=0}^{t-i} \binom{2t+1-i}{j} p_{crash}^{j} \times \\ p_{correct}^{2t+1-i-j} \times \sum_{k=0}^{t-i-j} \binom{2t+1-i-j}{k} p_{synchrony}^{2t+1-i-j-k} \times \\ (1-p_{synchrony})^{k} > \sum_{i=0}^{t} \binom{3t+1}{i} p_{benign}^{3t+1-i} (1-p_{benign})^{i}. \end{split}$$

In the special case when t=1, the above inequality simplifies to

$$p_{available} > p_{benian}^{1.5}$$
.

Hence, for t = 1, XPaxos has stronger consistency guarantees than any asynchronous BFT protocol whenever the probability that a machine is available is larger than 1.5 power of the probability that a machine is benign. This is despite the fact that BFT is more expensive than XPaxos as t = 1 implies 4 replicas for BFT and only 3 for XPaxos.

In terms of nines of consistency, again for t=1 and t=2, we calculated the difference in consistency between XPaxos and BFT SMR, for all values of 9_{benign} , $9_{correct}$ and $9_{synchrony}$ ranging between 1 and 20, and observed the following relation:

$$9ofC(BFT_{t=1}) - 9ofC(\mathsf{XPaxos}_{t=1}) = \\ \begin{cases} 9_{benign} - 9_{correct} + 1, & 9_{benign} > 9_{synchrony} \text{ and} \\ 9_{synchrony} = 9_{correct}, \\ 9_{benign} - min(9_{correct}, 9_{synchrony}), & \text{otherwise.} \end{cases}$$

$$9ofC(BFT_{t=2}) - 9ofC(\mathsf{XPaxos}_{t=2}) = \\ \begin{cases} 2 \times (9_{benign} - 9_{correct}) + 1, & 9_{benign} > 9_{synchrony} \text{ and} \\ 9_{synchrony} = 9_{correct}, \\ -1, & 9_{synchrony} > 2 \times 9_{benign} \\ & \text{and } 9_{benign} = 9_{correct}, \\ 2 \times (9_{benign} - min(9_{correct}, 9_{synchrony})), & \text{otherwise.} \end{cases}$$

Note that in cases where XPaxos guarantees better consistency than BFT $(p_{available} > p_{benign}^{1.5})$, it is only "slightly" better and does not materialize in additional nines.

Example 1 (cont'd.). Building upon our example, $p_{benign} = 0.9999$ and $p_{synchrony} = p_{correct} = 0.999$, we have $9ofC(BFT_{t=1}) - 9ofC(XPaxos_{t=1}) = 9_{benign} - 9_{synchrony} + 1 = 2$, i.e., $9ofC(XPaxos_{t=1}) = 5$ and $9ofC(BFT_{t=1}) = 7$. BFT brings 2 nines of consistency on top of XPaxos.

Example 2 (cont'd.). When $p_{benign} = p_{synchrony} = 0.9999$ and $p_{correct} = 0.999$, we have $9ofC(BFT_{t=1}) - 9ofC(XPaxos_{t=1}) = 1$, i.e., $9ofC(XPaxos_{t=1}) = 6$ and $9ofC(BFT_{t=1}) = 7$. XPaxos has one nine of consistency less than BFT (albeit the only 7th).

6.2 Availability

Then, we quantify the stronger availability guarantees of XPaxos over asynchronous CFT and BFT protocols. We define the number of nines of availability for protocol X, as 9ofA(X) = 9of(P[X is available]).

Recalling that whenever $\lfloor \frac{n-1}{2} \rfloor + 1$ active replicas in synchronous group are available, XPaxos can make progress despite passive replicas are benign or not, partitioned or not (see Table 1). Thus, we have $P[\mathsf{XPaxos} \text{ is available}] = \sum_{i=\lfloor \frac{n-1}{2} \rfloor + 1}^{n} \binom{n}{i} p_{available}^{i} \times (1-p_{available})^{n-i}$.

6.2.1 XPaxos vs. CFT

a CFT protocol (e.g., Paxos) is available whenever $n - \lfloor \frac{n-1}{2} \rfloor$ machines are correct and synchronous, plus other machines are benign (see Table 1). Hence, $P[\text{CFT is available}] = \sum_{i=n-\lfloor \frac{n-1}{2} \rfloor}^{n} \binom{n}{i} p_{available}^{i} \times (p_{benign} - p_{available})^{n-i}$.

Similarly to consistency analysis, we calculated 9ofA(CFT) and 9ofA(XPaxos) for all values of $9_{available}$ and 9_{benign} between 1 and 20 in the cases where t=1 and t=2. Notice that $p_{available} < p_{benign}$ is always true, i.e., $9_{available} < 9_{benign}$. We observed the following relation for t=1:

$$9ofA(\mathsf{XPaxos}_{t=1}) - 9ofA(CFT_{t=1}) = \\ max(2 \times 9_{available} - 9_{benign}, 0).$$

When t = 2, we observed:

$$\begin{split} gofA(\mathsf{XPaxos}_{t=2}) &= 3 \times 9_{available} - 1, \\ gofA(\mathsf{XPaxos}_{t=2}) - gofA(CFT_{t=2}) &= \\ \begin{cases} 3 \times 9_{available} - 9_{benign}, & 9_{benign} < 3 \times 9_{available}, \\ 1, & 3 \times 9_{available} \leq 9_{benign} < 4 \times 9_{available}, \\ 0, & 9_{benign} \geq 4 \times 9_{available}. \end{split}$$

Example. When $p_{available} = 0.999$ and $p_{benign} = 0.99999$, we have $9ofA(\mathsf{XPaxos}_{t=1}) - 9ofA(CFT_{t=1}) = 1$, i.e., $9ofA(\mathsf{XPaxos}_{t=1}) = 5$ and $9ofA(CFT_{t=1}) = 4$. XPaxos adds 1 nine of availability on top of CFT and achieves 5 nines of availability in total. Besides, XPaxos adds 2 nines of availability on top of individual machine availability.

6.2.2 XPaxos vs. BFT

From Table 1, an asynchronous BFT protocol is available when $n - \lfloor \frac{n-1}{3} \rfloor$ machines are available despite faults of other machines. Thus, $P[\text{BFT is available}] = \sum_{i=n-\lfloor \frac{n-1}{3} \rfloor}^{n} \binom{n}{i} p_{available}^{i} \times (1-p_{available})^{n-i}$.

We calculated $9ofA(\mathsf{XPaxos})$ and 9ofA(BFT) for all values of $9_{available}$ between 1 and 20 in the cases when t=1 and t=2. In this comparison 9_{beniqn} does not matter. When t=1,

$$9ofA(\mathsf{XPaxos}_{t=1}) = 9ofA(BFT_{t=1}) = 2 \times 9_{available} - 1.$$

On the other hand, when t=2,

$$9ofA(\mathsf{XPaxos}_{t=2}) = 9ofA(BFT_{t=2}) + 1 = 3 \times 9_{available} - 1.$$

Hence, when t = 1, XPaxos has the same number of nines of availability as BFT. When t = 2, XPaxos adds 1 nine of availability to BFT.

7 Related work and concluding remarks

In this paper, we introduced XFT, a novel fault-tolerance model that allows the design of efficient protocols that tolerate non-crash faults. We demonstrated XFT through XPaxos, a novel state-machine replication protocol that features many more nines of reliability than the best crash-fault-tolerant (CFT) protocols with roughly the same communication complexity, performance and resource cost. Namely, XPaxos uses 2t+1 replicas and provides all the reliability guarantees of CFT, but is also capable of tolerating non-crash faults, as long as a majority of XPaxos replicas are correct and can communicate synchronously among each other.

As XFT is entirely realized in software, it is fundamentally different from an established approach that relies on trusted hardware for reducing the resource cost of BFT to 2t + 1 replicas only [15, 30, 21, 38].

XPaxos is also different from PASC [14], which makes CFT protocols tolerate a subset of Byzantine faults using ASC-hardening. ASC-hardening modifies an application by keeping two copies of the state at each replica. It then tolerates Byzantine faults under the "fault diversity" assumption, i.e., that a fault will not corrupt both copies of the state in the same way. Unlike XPaxos, PASC does not tolerate Byzantine faults that affect the entire replica (e.g., both state copies).

In this paper, we did not explore the impact on varying the number of tolerated faults per fault class. In short, this approach, known as the hybrid fault model and introduced in [37] distinguishes the threshold of non-crash faults (say b) despite which safety should be ensured, from the threshold t of faults (of any class) despite which the availability should be ensured (where often $b \leq t$). The hybrid fault model and its refinements [11, 34] appear orthogonal to our XFT approach.

Specifically, Visigoth Fault Tolerance (VFT) [34] is a recent refinement of the hybrid fault model. Besides having different thresholds for non-crash and crash faults, VFT also refines the space between network synchrony and asynchrony by defining the threshold of network faults that a VFT protocol can tolerate. VFT is, however, different from XFT in that it fixes separate fault thresholds for non-crash and network faults. This difference is fundamental rather than notational, as XFT cannot be expressed by choosing specific values of VFT thresholds. For instance, XPaxos can tolerate, with 2t+1 replicas, t partitioned replicas, t non-crash faults and t crash faults, albeit not simultaneously. Specifying such requirements in VFT would yield at least 3t+1 replicas. In addition, VFT protocols have more complex communication patterns than XPaxos. That said, many of the VFT concepts remain orthogonal to XFT. It would be interesting to explore interactions between the hybrid fault model (including its refinements such as VFT) and XFT in the future.

Going beyond the research directions outlined above, this paper opens also other avenues for future work. For instance, many important distributed computing problems that build on SMR, such as distributed storage and blockchain, deserve a novel look at them through the XFT prism.

References

- [1] Crypto++ library 5.6.2. http://www.cryptopp.com/, 2014.
- [2] B. Alpern and F. Schneider. Recognizing safety and liveness. *Distributed Computing*, 2(3):117–126, 1987.
- [3] P.-L. Aublin, R. Guerraoui, N. Knežević, V. Quéma, and M. Vukolić. The next 700 BFT protocols. ACM Trans. Comput. Syst., 32(4):12:1–12:45, Jan. 2015.
- [4] P. Bailis and K. Kingsbury. The network is reliable. Commun. ACM, 57(9):48–55, 2014.

- [5] J. Baker, C. Bond, J. C. Corbett, J. J. Furman, A. Khorlin, J. Larson, J. Leon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In Fifth Biennial Conference on Innovative Data Systems Research (CIDR), pages 223–234, 2011.
- [6] P. Berman, J. A. Garay, and K. J. Perry. Towards optimal distributed consensus. In Proc. 30th IEEE Symposium on Foundations of Computer Science (FOCS), pages 410–415, 1989.
- [7] G. Bracha and S. Toueg. Asynchronous consensus and broadcast protocols. J. ACM, 32(4):824–840, 1985.
- [8] B. Calder, J. Wang, A. Ogus, et al. Windows Azure storage: A highly available cloud storage service with strong consistency. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 143–157. ACM, 2011.
- [9] M. Castro and B. Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst.*, 20(4):398–461, Nov. 2002.
- [10] T. D. Chandra, R. Griesemer, and J. Redstone. Paxos made live: An engineering perspective. In Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Distributed Computing, PODC 2007, pages 398–407, 2007.
- [11] A. Clement, M. Kapritsos, S. Lee, Y. Wang, L. Alvisi, M. Dahlin, and T. Riche. Upright cluster services. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles, SOSP'09*, pages 277–290, 2009.
- [12] A. Clement, E. L. Wong, L. Alvisi, M. Dahlin, and M. Marchetti. Making byzantine fault tolerant systems tolerate byzantine faults. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, NSDI 2009, pages 153–168, 2009.
- [13] J. C. Corbett, J. Dean, M. Epstein, et al. Spanner: Google's globally-distributed database. In Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation, OSDI'12, pages 251–264, Berkeley, CA, USA, 2012. USENIX Association.
- [14] M. Correia, D. G. Ferro, F. P. Junqueira, and M. Serafini. Practical hardening of crash-tolerant systems. In 2012 USENIX Annual Technical Conference, pages 453–466, 2012.
- [15] M. Correia, N. F. Neves, and P. Verissimo. How to tolerate half less one Byzantine nodes in practical distributed systems. In *Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems*, SRDS '04, pages 174–183. IEEE Computer Society, 2004.
- [16] F. Cristian, H. Aghili, R. Strong, and D. Dolev. Atomic broadcast: From simple message diffusion to Byzantine agreement. *Information and Computation*, 118(1):158–179, 1995.
- [17] D. Dolev and H. R. Strong. Authenticated algorithms for Byzantine agreement. SIAM Journal on Computing, 12(4):656–666, Nov. 1983.
- [18] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35, April 1988.
- [19] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. ZooKeeper: Wait-free coordination for internet-scale systems. In 2010 USENIX Annual Technical Conference, pages 11–11, 2010.
- [20] F. P. Junqueira, B. C. Reed, and M. Serafini. Zab: High-performance broadcast for primary-backup systems. In *Proceedings of the Conference on Dependable Systems and Networks (DSN)*, pages 245–256, 2011.
- [21] R. Kapitza, J. Behl, C. Cachin, T. Distler, S. Kuhnle, S. V. Mohammadi, W. Schröder-Preikschat, and K. Stengel. CheapBFT: Resource-efficient Byzantine fault tolerance. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 295–308, New York, NY, USA, 2012. ACM.

- [22] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: Speculative Byzantine fault tolerance. *ACM Trans. Comput. Syst.*, 27(4):7:1–7:39, Jan. 2010.
- [23] T. Kraska, G. Pang, M. J. Franklin, S. Madden, and A. Fekete. MDCC: multi-data center consistency. In *Eighth Eurosys Conference 2013*, pages 113–126, 2013.
- [24] K. Krishnan. Weathering the unexpected. Commun. ACM, 55:48-52, Nov. 2012.
- [25] P. Kuznetsov and R. Rodrigues. BFTW3: Why? When? Where? Workshop on the theory and practice of Byzantine fault tolerance. SIGACT News, 40(4):82–86, Jan. 2010.
- [26] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat. Providing high availability using lazy replication. *ACM Trans. Comput. Syst.*, 10(4):360–391, Nov. 1992.
- [27] L. Lamport. The part-time parliament. ACM Trans. Comput. Syst., 16:133–169, May 1998.
- [28] L. Lamport. Paxos made simple. ACM SIGACT News, 32(4):18–25, 2001.
- [29] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. ACM Trans. Program. Lang. Syst., 4:382–401, July 1982.
- [30] D. Levin, J. R. Douceur, J. R. Lorch, and T. Moscibroda. TrInc: Small trusted hardware for large distributed systems. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'09, pages 1–14. USENIX Association, 2009.
- [31] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. May 2009.
- [32] B. M. Oki and B. H. Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, PODC '88, pages 8–17, New York, NY, USA, 1988. ACM.
- [33] D. Ongaro and J. K. Ousterhout. In search of an understandable consensus algorithm. In *Proc. USENIX Annual Technical Conference*, pages 305–319, 2014.
- [34] D. Porto, J. a. 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*, EuroSys '15, pages 8:1–8:14, New York, NY, USA, 2015. ACM.
- [35] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. ACM Comput. Surv., 22(4):299–319, 1990.
- [36] Y. Sovran, R. Power, M. K. Aguilera, and J. Li. Transactional storage for geo-replicated systems. In Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles, SOSP '11, pages 385–400, New York, NY, USA, 2011. ACM.
- [37] P. M. Thambidurai and Y. Park. Interactive consistency with multiple failure modes. In *Proceedings of the Seventh Symposium on Reliable Distributed Systems, SRDS*, pages 93–100, 1988.
- [38] G. S. Veronese, M. Correia, A. N. Bessani, L. C. Lung, and P. Veríssimo. Efficient Byzantine fault-tolerance. *IEEE Trans. Computers*, 62(1):16–30, 2013.
- [39] M. Vukolić. The quest for scalable blockchain fabric: Proof-of-work vs. BFT replication. In *Open Problems in Network Security IFIP WG 11.4 International Workshop*, *iNetSec 2015*, pages 112–125, 2015.

Appendix A XPaxos example execution

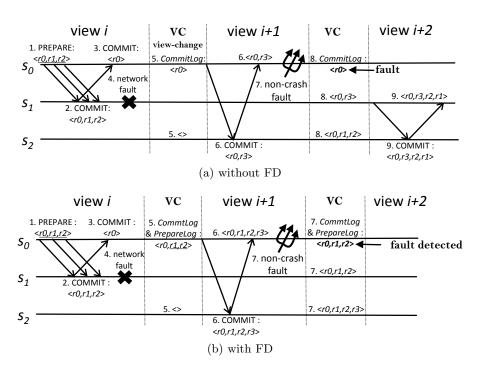


Figure 11: XPaxos example. The view is changed from i to i + 2, due to the network fault of s_1 and the non-crash fault of s_0 , respectively.

In Fig. 11 we give an example of XPaxos execution when t = 1. The role of each replica in each view is shown in Table 2.

In Fig. 11a, view change phase proceeds without fault detection. Upon the primary s_0 receives requests r_0 , r_1 , and r_2 from clients, s_0 prepares these requests locally and sends COMMIT messages to the follower s_1 . Then, s_1 commits r_0 , r_1 , and r_2 locally and sends COMMIT messages to s_0 . Because of a network fault, s_0 only receives COMMIT message of r_0 in a timely manner, thus the view change phase to i+1 is activated by s_0 . During view change to i+1, s_0 sends the VIEW-CHANGE message with commit log of r_0 to all active replicas in view i+1 (i.e., s_0 and s_2). In view i+1, r_0 is further committed by s_0 and s_2 . After that, s_0 is under non-crash fault and the view is changed to i+2. During view change to i+2, s_1 and s_2 provide all their commit logs to new active replicas (i.e., s_1 and s_2), whereas non-crash faulty replica s_0 only reports the commit log of r_0 . Outside anarchy, requests r_0 and r_0 are committed in new view i+2 by receiving the VIEW-CHANGE message from s_1 . In view i+2, r_1 is finally committed by every active replica.

In example of Figure 11b, XPaxos fault detection is enabled. In view i, the execution is the same as in Figure 11a. During view change to i+1, commit log of r0 and prepare logs of r1 and r2 are sent by s_0 , which are committed by s_0 and s_2 in view i+1, as well as the new request r3. The same as before, s_0 is non-crash faulty and the view is changed to i+2. During view change to i+2, commit logs of r0, r1, r2 and r3 are sent by s_2 . At the same time, because of missing prepare log of r3, the fault of s_0 is detected with the help of the VIEW-CHANGE message from s_2 .

Appendix B XPaxos pseudocode

In this appendix we give the pseudocode of XPaxos. For simplicity reason, we assume that signature/MAC attached to each message always correctly verifies. Figure 12 gives the definition of message fields and local variables for all components of XPaxos. Readers can refer to Section 4 for protocol description.

This appendix is organized incrementally as follows. Section B.1 gives the pseudocode of XPaxos common case. Section B.2 gives the pseudocode of the view change mechanism. Section B.3 describes and gives the pseudocode of clients' request retransmission mechanism that deals with faulty primary. Finally, Section B.4 depicts the modification to the view change protocol to enable Fault Detection and gives the pseudocode.

```
Common case:
c, op, ts_c - id of the client, operation, client timestamp
req_c - ongoing request at client c
n - total number of replicas
\Pi - set of n replicas
i - current view number
s_j - replica id
sg_i - set of t+1 replicas in synchronous group in view i
ps_i - the primary in view i (ps_i \in sq_i)
fs_i - the follower in view i for t = 1 (fs_i \in sg_i)
fs_i^k - the followers in view i for t \geq 2 (fs_i^k \in sg_i)
req - client request
rep - reply of client request
sn_{s_i} - sequence number prepared at replica s_j
ex_{s_i} - sequence number executed at replica s_j
D(m) - digest of a message m
PrepareLog_{s_i} - array of prepared proof at replica s_i
CommitLog_{s_i} - array of commit proofs at replica s_j
View change:
SusSet_{s_i} - set of SUSPECT messages cached for view-change at replica s_i
timer_i^{net} - network establishment timer for view i
\Delta - maximum message delay between two correct replicas, beyond which a network fault is declared
timer_i^{vc} - view-change timer in view change to i
VCSet_{s_i}^i - set of VIEW-CHANGE messages collected in view change to i at replica s_j
CommitLog_{s_i}^i - array of most recent commit proofs selected from VCSet_{s_i}^i at replica s_j
End(log) - end index of array log
Fault detection:
Final Proof_{s_i} - array of t+1 VC-CONFIRM messages which prove that \forall s_k \in sg_i collected the same VCSet_{s_k}^i
pre_{s_i} - the view number in which PrepareLog_{s_j} is generated
FinalSet_{s_i}^i - set of t+1 VC-FINAL messages collected in view change to i at replica s_j
PrepareLog_{s_i}^i - array of most recent prepare proof selected from VCSet_{s_i}^i at replica s_j
```

Figure 12: XPaxos common case: Message fields and local variables.

B.1 Common case

In common case, we assume that all replicas are in the same view. Algorithm 1 and Algorithm 2 describe the common case protocol when t = 1 and $t \ge 2$, respectively. Figure 2 gives the message pattern.

Algorithm 1 Common case when t = 1.

```
Initialization:
     client : ts_c \leftarrow 0; req_c \leftarrow nil
     replica: sn_{s_i} \leftarrow 0; ex_{s_i} \leftarrow 0; PrepareLog_{s_i} = []; CommitLog_{s_i} = []
 1: upon invocation of propose(op) at client c do
             \mathbf{inc}(ts_c)
 2:
            send req_c \leftarrow \langle \text{REPLICATE}, op, ts_c, c \rangle_{\sigma_c} to the primary ps_i \in sq_i
 3:
 4:
            start timer_c
                                                                                                                                                /* primary */
 5: upon reception of req = \langle \text{REPLICATE}, op, ts, c \rangle_{\sigma_c} from client c at ps_i do
 6:
            \mathbf{inc}(sn_{ps_i})
             m_{ps_i} \leftarrow \langle \text{COMMIT}, D(req), sn_{ps_i}, i \rangle_{\sigma_{ps_i}}
 7:
             PrepareLog_{ps_i}[sn_{ps_i}] \leftarrow \langle req, m_{ps_i} \rangle
 8:
 9:
            send \langle req, m_{ps_i} \rangle to the follower fs_i
10: upon reception of \langle req, m_{ps_i} = \langle \text{COMMIT}, d_{req}, sn, i \rangle_{\sigma_{ps_i}} \rangle from the primary ps_i at fs_i do /* follower
             if sn = sn_{fs_i} + 1 and D(req) = d_{req} then
11:
12:
                    \mathbf{inc}(sn_{fs_i})
13:
                    rep \leftarrow \text{execute } req
14:
                    \mathbf{inc}(ex_{fs_i})
                    m_{fs_i} \leftarrow \langle \text{COMMIT}, D(req), sn, i, req.ts_c, D(rep) \rangle_{\sigma_{fs_i}}
15:
                    CommitLog_{fs_i}[sn] \leftarrow \langle req, m_{ps_i}, m_{fs_i} \rangle
16:
17:
                    send m_{fs_i} to the primary ps_i
     upon reception of m_{fs_i} = \langle \text{COMMIT}, d_{req}, sn, i, ts, d_{rep} \rangle_{\sigma_{fs_i}} from the follower fs_i at ps_i do
18:
             if D(PrepareLog_{ps_i}[sn].req) = d_{req} then
19:
20:
                    CommitLog_{ps_i}[sn] \leftarrow \langle req, m_{ps_i}, m_{fs_i} \rangle
21: upon CommitLog_{ps_i}[ex_{ps_i} + 1] \neq nil at ps_i do
22:
             \mathbf{inc}(ex_{ps_i})
             rep \leftarrow \text{execute } CommitLog_{ps_i}[ex_{ps_i}].req
23:
24:
             if D(rep) = CommitLog_{ps_i}[ex_{ps_i}].m_{fs_i}.d_{rep} then
                    send \langle\langle \text{REPLY}, sn, i, ts, rep \rangle_{\mu_{ps_i,c}}, m_{fs_i} \rangle to CommitLog_{ps_i}[ex_{ps_i}].req.c
25:
26: upon reception of \langle r_{ps_i}, m_{fs_i} \rangle from the primary ps_i at client c, where
     \begin{array}{l} r_{ps_i} = \langle \text{REPLY}, sn, i, ts, rep \rangle_{\mu_{ps_i,c}} \\ m_{fs_i} = \langle \text{COMMIT}, d'_{req}, sn', i', ts', d_{rep} \rangle_{\sigma_{fs_i}} \ \mathbf{do} \end{array}
            if sn = sn' and i = i' and ts = ts' = req.ts_c and D(rep) = d_{rep} then
27:
                    deliver rep
28:
29:
                    stop timer_c
```

Algorithm 2 Common case when t > 1.

```
Initialization:
     client : ts_c \leftarrow 0; req_c \leftarrow nil
     replica : sn_{s_j} \leftarrow 0; ex_{s_j} \leftarrow 0; PrepareLog_{s_j} = []; CommitLog_{s_i} = []
 1: upon invocation of propose(op) at client c do
 2:
 3:
            send req_c \leftarrow \langle \text{REPLICATE}, op, ts_c, c \rangle_{\sigma_c} to the primary ps_i \in sq_i
 4:
            start timer_c
                                                                                                                                          /* primary */
 5: upon reception of req = \langle \text{REPLICATE}, op, ts, c \rangle_{\sigma_c} from client c at ps_i do
            m_{ps_i} \leftarrow \langle \text{PREPARE}, D(req), sn_{ps_i}, i \rangle_{\sigma_{ps_i}}
 7:
             PrepareLog_{ps_i}[sn] \leftarrow \langle req, m_{ps_i} \rangle
 8:
            send \langle req, m_{ps_i} \rangle to fs_i^k \in sg_i
 9:
                                                                                                                                                            /*
10: upon reception of \langle req, m_{ps_i} = \langle PREPARE, d_{req}, sn, i \rangle_{\sigma_{ps_i}} \rangle from the primary ps_i at fs_i^k do
     follower */
            if sn = sn_{fs^k} + 1 and D(req) = d_{req} then
11:
                   \mathbf{inc}(sn_{fs_{\cdot}^{k}})
12:
                   PrepareLog_{fs_{i}^{k}}[sn] \leftarrow \langle req, m_{ps_{i}} \rangle
13:
                   m_{fs_i^k} \leftarrow \langle \text{COMMIT}, D(req), sn, i, fs_i^k \rangle_{\sigma_{fs^k}}
14:
15:
                   send m_{fs_{\cdot}^{k}} to \forall s_{k} \in sg_{i}
16: upon reception of m_{fs_i^k} = \langle \text{COMMIT}, d_{req}, sn, i, fs_i^k \rangle_{\sigma_{fs_i^k}} from every follower fs_i^k \in sg_i at s_j \in sg_i do
            CommitLog_{s_j}[sn] \leftarrow \langle req, m_{ps_i}, m_{fs_i^1}...m_{fs_i^f} \rangle
17:
18: upon CommitLog_{s_i}[ex_{s_i}+1] \neq nil at s_i do
19:
            \mathbf{inc}(ex_{s_i})
            rep \leftarrow \text{execute } CommitLog_{s_i}[ex_{s_i}].req
20:
            send \langle \text{REPLY}, sn, i, req. ts_c, rep \rangle_{\mu_{s_i,c}} to client c, where c = CommitLog_{s_j}[ex_{s_j}].req.c \rangle
21:
22: upon reception of t+1 REPLY messages \langle \text{REPLY}, sn, i, ts, rep \rangle_{\mu_{s_i,c}} at client c do
            if t+1 REPLY messages are with the same sn, i, ts and rep and ts=req.ts_c then
23:
24:
                   deliver rep
                   stop timer_c
25:
```

B.2 View-change

The message pattern of view-change w/o fault detection is given in Figure 3. Algorithm 3 shows the corresponding pseudocode. The description of view change can be found in Section 4.3.

Algorithm 3 View change at replica s_j .

```
Initialization:
     SusSet_{s_j} \leftarrow \emptyset; VCSet_{s_i}^i \leftarrow \emptyset; CommitLog_{s_i}^i \leftarrow []
 1: upon suspicion of view i and s_j \in sg_i do
           send \langle \text{SUSPECT}, i, s_j \rangle_{\sigma_{s_i}} to \forall s_k \in \Pi
 3: upon reception of m = \langle \text{SUSPECT}, i', s_k \rangle_{\sigma_{s_k}} and s_k \in sg_{i'} do
 4:
            SusSet_{s_j} \leftarrow SusSet_{s_j} \cup \{m\}
            forward m to \forall s_k \in \Pi
 5:
 6: upon \exists \langle \text{SUSPECT}, i, s_k \rangle_{\sigma_{s_k}} \in SusSet_{s_j} \mathbf{do}
                                                                                                          /* enter each view in order */
           \mathbf{inc}(i) (i.e., ignore any message in preceding view)
 7:
           send \langle VIEW-CHANGE, i, s_j, CommitLog_{s_i} \rangle_{\sigma_{s_i}} to \forall s_k \in sg_i
 8:
 9:
           if s_j \in sg_i then
                  start timer_i^{net} \leftarrow 2\Delta
10:
11: upon reception of m = \langle \text{VIEW-CHANGE}, i, s_k, CommitLog \rangle_{\sigma_{s_k}} from replica s_k do
            VCSet_{s_i}^i \leftarrow VCSet_{s_i}^i \cup \{m\}
13: upon |VCSet_{s_i}^i| = n or (expiration of timer_i^{net} and |VCSet_{s_i}^i| \ge n - t) do
14:
           send (VC-FINAL, i, s_j, VCSet_{s_i}^i \rangle_{\sigma_{s_i}} to \forall s_k \in sg_i
           start timer_i^{vc}
15:
16: upon reception of m_* = \langle VC\text{-FINAL}, i, s_k, VCSet \rangle_{\sigma_{s_k}} from every s_k \in sg_i do
            VCSet_{s_i}^i \leftarrow VCSet_{s_i}^i \cup \{ \forall m : m \in VCSet \text{ in any } m_k \}
17:
           for sn: 1..End(\forall CommitLog | \exists m \in VCSet_{s_i}^i : CommitLog \text{ is in } m) do
18:
                  CommitLog_{s_i}^i[sn] \leftarrow CommitLog[sn] with the highest view number
19:
                                                                                                                                 /* primary */
           if s_j = ps_i then
20:
21:
                  for sn: 1..End(CommitLog_{s_i}^i) do
22:
                        req \leftarrow CommitLog_{s_i}^i[sn].req
                         PrepareLog[sn] \leftarrow \langle req, \langle PREPARE, D(req), sn, i \rangle_{\sigma_{ps,i}} \rangle
23:
                  send (NEW-VIEW, i, PrepareLog)_{\sigma_{ps_i}} to \forall s_k \in sg_i
24:
25: upon reception of \langle NEW-VIEW, i, PrepareLog \rangle_{\sigma_{ps_i}} from the primary ps_i do
26:
           if PrepareLog is matching with CommitLog_{s_i}^i then
27:
                  PrepareLog_{s_i} \leftarrow PrepareLog
28:
                  reply and process \forall m \in PrepareLog as in common case
                  sn_{s_i} \leftarrow End(PrepareLog)
29:
30:
                  ex_{s_i} \leftarrow End(PrepareLog)
                  stop timer_i^{vc}
31:
32:
           else
                  suspect view i
33:
34: upon expiration of timer_i^{vc} do
           suspect view i
35:
```

B.3 Request retransmission

In order to provide availability with respect to faulty primary or followers, as well as long-lived network faults within the synchronous group, we propose a request retransmission mechanism which broadcasts the request to all active replicas upon retransmission timer expires at client side. Retransmission mechanism requires every active replica to monitor the progress. In case a request is not executed and replied in a timely manner, the correct active replica in the synchronous group will eventually suspect the view.

More specifically (the pseudocode is given in Algorithm 4), if a client c does not receive the matching replies of request req_c in a timely manner, c re-sends req_c to all active replicas in current view i by $\langle \text{RE-SEND}, req_c \rangle$. Any active replica $s_j \in sg_i$, upon receiving $\langle \text{RE-SEND}, req_c \rangle$ from c, (1) forwards req_c to the primary $ps_i \in sg_i$ if $s_j \neq ps_i$, (2) starts a timer $timer_{req_c}$ locally, and (3) asks each active replica to sign the reply. Upon $timer_{req_c}$ expires and the active replica $s_j \in sg_i$ has not received t+1 signed replies, s_j suspects view i and sends the SUSPECT message to the client c; otherwise, s_j forwards t+1 signed replies to client c.

Upon receiving SUSPECT message m for view i, client c forwards m to every active replica in view i+1. This step serves to guarantee that the view-change can actually happen at all correct replicas. Then client c forwards req_c to the primary of view i+1.

Algorithm 4 Client request retransmission.

```
1: upon expiration of timer_c at client c do
          send \langle \text{RE-SEND}, req_c \rangle to \forall s_j \in sg_i
 3: upon reception of \langle \text{RE-SEND}, req_c \rangle at s_j \in sg_i do
           if s_i \neq ps_i then
                 send req_c to ps_i \in sq_i
 5:
 6:
          start timer_{req_c}
          ask \forall s_j \in sg_i to sign the reply of req_c
 7:
 8: upon expiration of timer_{req_c} at replica s_j \in sg_i do
 9:
          suspect view i
          send \langle \text{SUSPECT}, i, s_j \rangle_{\sigma_{s_i}} to client c
10:
11: upon reception of m = \langle \text{SUSPECT}, i, s_k \rangle_{\sigma_{s_k}} at client c and s_k \in sg_i and c is in view i do
          enter view i+1
12:
          send m to \forall s_j \in sg_{i+1}
13:
14:
           send req_c to ps_{i+1}
15:
          start timer_c
16: upon execution of req_c at s_i^a do
           /* sign the reply by each active replica */
17:
           send \langle \text{REPLY}, sn, i, req.ts_c, rep \rangle_{\sigma_{s_i}} to \forall s_j \in sg_i
18: upon reception of m_k = \langle \text{REPLY}, sn, i, ts, rep \rangle_{\sigma_{s_k}} from every s_k \in sg_i at replica s_j do
                                                                                                 /* collect t+1 signed replies */
           if m_1, m_2, ..., m_{t+1} are with the same sn, i, ts and rep then
19:
20:
                 replies \leftarrow \{m_1, m_2, ..., m_{t+1}\}
                 send \langle SIGNED-REPLY, replies \rangle to client c
21:
22:
                 stop timer_{reg}
   ^aby line 13 or 23 in Algorithm 1 or line 20 in Algorithm 2
```

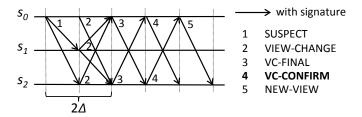


Figure 13: Message pattern of XPaxos view-change with fault detection: VC-CONFIRM phase is added; synchronous group is changed from (s_0, s_1) to (s_0, s_2) .

B.4 Fault detection

In this section we describe XPaxos with Fault Detection (FD). Specifically, in order to detect all the fatal faults that can possibly violate consistency in anarchy, view change to i + 1 with FD includes the following modifications.

- Every replica s_j appends its prepare logs $PrepareLog_{s_j}$ into the VIEW-CHANGE message when replying to active replicas in view i+1. Besides, synchronous group sg_{i+1} prepares and commits requests piggybacked in commit or prepare logs. The selection rule is almost the same as in view change without FD: for each sequence number sn, the request with the highest view number $i' \leq i$ is selected, either in a commit log or in a prepare log.
- XPaxos FD additionally inserts a VC-CONFIRM phase after exchanging VIEW-CHANGE messages among active replicas in view i+1, i.e., after receiving t+1 VC-FINAL messages (see Figure 3 and Figure 13 for the comparison). In VC-CONFIRM phase, every active replica $s_j \in sg_{i+1}$ (1) detects potential faults in the VIEW-CHANGE messages in $VCSet_{s_j}^{i+1}$ and adds the faulty replica to set FSet; (2) removes faulty messages from $VCSet_{s_j}^{i+1}$; and, (3) signs and sends $\langle \text{VC-CONFIRM}, i+1, D(VCSet_{s_j}^{i+1}) \rangle_{\sigma_{s_j}}$ to every active replica in sg_{i+1} . Upon $s_j \in sg_{i+1}$ receives t+1 VC-CONFIRM messages with matching $D(VCSet_*^{i+1})$, s_j (1) inserts the VC-CONFIRM messages into set $FinalProof_{s_j}[i+1]$; and (2) prepares and commits the requests selected based on $VCSet_{s_j}^{i+1}$. $FinalProof_{s_j}[i+1]$ serves to prove that t+1 active replicas in sg_i have agreed on the set of filtered VIEW-CHANGE messages.
- Every replica s_j appends $FinalProof_{s_j}[i']$ into the VIEW-CHANGE message when replying to active replicas in new view, where i' is the view in which $PrepareLog_{s_j}$ is generated. In case a prepare log in $PrepareLog_{s_j}$ is not consistent with some commit log, $FinalProof_{s_j}[i]$ can prove that there exists correct replica $s_j \in sg_{i'}$ which can prove the fault of the prepare log.

Algorithm 5 gives the modifications based on Algorithm 3 for XPaxos with fault detection mechanism. Algorithm 6 enumerates all types of faults that can and must be detected by correct active replicas. Figure 13 gives the new message pattern.

Algorithm 5 Modifications for fault detection at replica s_i .

```
Initialization:
    FinalProof_{s_{j}} \leftarrow []; pre_{s_{j}} \leftarrow 0; FinalSet_{s_{j}}^{i} \leftarrow \emptyset; PrepareLog_{s_{j}}^{i} \leftarrow []; FSet \leftarrow []
     /* replace line 8 in Algorithm 3 by : */
          send m = \langle \text{VIEW-CHANGE}, i, s_j, CommitLog_{s_i}, PrepareLog_{s_i}, FinalProof_{s_i}[pre_{s_i}] \rangle_{\sigma_{s_i}} to
    \forall s_k \in sg_i
     /* replace line 11 in Algorithm 3 by : */
 2: upon reception of m = \langle \text{VIEW-CHANGE}, i, s_k, CommitLog, PrepareLog, FinalProof \rangle_{\sigma_{s_k}} from replica
     s_k \ \mathbf{do}
     /* replace lines 18\sim24 in Algorithm 3 by : */
                                                                                                       /* refer to Algorithm 6 */
           FAULTDETECTION(vcSet_{s_i}^i)
           for \forall m : m \in vcSet_{s_j}^i and m from replica s \in FSet do
 4:
                remove m from vcSet_{s_j}^{\imath}
 5:
          send \langle VC\text{-CONFIRM}, i, D(vcSet_{s_i}^i) \rangle_{\sigma_{s_i}} to \forall s_k \in sg_i
 6:
     /* new event handler */
 7: upon reception of m_* = \langle \text{VC-CONFIRM}, i, d_{vcSet} \rangle_{\sigma_{s_k}} from every s_k \in sg_i do
           if m_1, m_2, ... m_{f+1} are not with the same d_{vcSet} then
 8:
                suspect view i
 9:
10:
                return
           Final Proof_{s_j}[i] \leftarrow \{m_1, m_2, ... m_{f+1}\}
11:
           for sn: 1..End(\forall CommitLog | \exists m \in VCSet_{s_i}^i : CommitLog \text{ is in } m) do
12:
                 CommitLog_{s_i}^i[sn] \leftarrow CommitLog[sn] with the highest view number
13:
           for sn: 1..End(\forall PrepareLog | \exists m \in VCSet_{s_i}^i: PrepareLog \text{ is in } m) do
14:
                 PrepareLog_{s_i}^i[sn] \leftarrow PrepareLog[sn] with the highest view number
15:
                                                                                                                       /* primary */
           if s_j = ps_i then
16:
                for sn: 1..End(PrepareLog_{s_i}^i|CommitLog_{s_i}^i) do
17:
18:
                      req \leftarrow CommitLog_{s_i}^i[sn].req
                       if req = null or PrepareLog^i_{s_i}[sn] is generated in a higher view than CommitLog^i_{s_j}[sn]
19:
     then
                             req \leftarrow PrepareLog_{s_j}^{i}[sn].req
20:
                       PrepareLog[sn] \leftarrow \langle req, \langle PREPARE, D(req), sn, i \rangle_{\sigma_{s_i}} \rangle
21:
22:
                send \langle \text{NEW-VIEW}, i, PrepareLog \rangle_{\sigma_{s_i}} to \forall s_k \in sg_i
     /* replace line 26 in Algorithm 3 by : */
           if PrepareLog is matching with CommitLog_{s_i}^i and PrepareLog_{s_i}^i then
     /* add this command after line 27 in Algorithm 3 : */
24:
                                                            /* update the view in which PrepareLog_{s_i} is generated */
```

```
Algorithm 6 Fault detection function at replica s_i.
```

```
1: function FAULTDETECTION(VCSet)
         \forall sn \text{ and } m, m' \in VCSet \text{ from replicas } s_k \text{ and } s_{k'}, \text{ respectively,}
               (state loss) if s_k, s_{k'} \in sg_{i'} (i' < i) and CommitLog'[sn] in m' is generated in view i' and
    PrepareLog is in m and PrepareLog[sn] = nil then (s_k \text{ is faulty})
                    send \langle \text{STATE-LOSS}, i, s_k, sn, m, m' \rangle to \forall s_{k''} \in \Pi
 4:
                    add s_k to FSet
 5:
 6:
               (fork-I) if s_k, s_{k'} \in sg_{i'} (i' < i) and PrepareLog[sn] in m is generated in view i''
    and CommitLog'[sn] in m' is generated in view i' and ((i'' = i' \text{ and } PrepareLog[sn].req \neq
    CommitLog'[sn].req) or i'' < i) then (s_k \text{ is faulty})
                    send \langle \text{FORK-I}, i, s_k, sn, m, m' \rangle to \forall s_{k''} \in \Pi
 7:
 8:
                    add s_k to FSet
               (fork-II-query) if PrepareLog[sn] in m is generated in view i'' (i'' < i) and CommitLog'[sn]
    in m' is generated in view i' (i' < i'' < i) and (PrepareLog[sn] = null \text{ or } PrepareLog[sn].req \neq
    CommitLog'[sn].req) then (s_k might be faulty)
10:
                    send (FORK-II-QUERY, i, s_k, sn, m) to \forall s_{k''} \in sg_{i''}
11:
                    wait for 2\Delta time
12: upon reception of (FORK-II-QUERY, i, s_k, s_n, m) at s_j, where final Proof in m is generated in view
    i'' and s_i \in sg_{i''} do
         if PrepareLog[sn] in m is not consistent with VCSet_{s_i}^{i''} then
13:
               send \langle \text{FORK-II}, i, s_k, sn, m, finalProof_{s_i}[i''], finalSet_{s_i}^{i''} \rangle to \forall s_k \in \Pi
14:
15: upon reception of \langle FORK-II, i, s_k, sn, m, finalProof, finalSet \rangle do
         add s_k to FSet
17: upon reception of STATE-LOSS, FORK-I or FORK-II message m do
         forward m to \forall s_k \in \Pi
18:
```

Appendix C XPaxos correctness proof

In this appendix, we first prove safety (consistency) and liveness (availability) properties of XPaxos. To prove safety (Section C.1), we show that when XPaxos is outside anarchy, consistency is guaranteed. In liveness section (Section C.2), we show that XPaxos can make progress with at most t faulty replicas and any number of faulty clients, if eventually the system is synchronous (i.e., eventual synchrony).

Then, in Section C.3, we prove that the fault detection mechanism is strong completeness and strong accuracy outside anarchy, with respect to non-crash faults which can violate consistency in anarchy.

We use the notation in Figure 14 to facilitate our proof of XPaxos. All predicates in Figure 14 are defined with respect to *beniqn* clients and replicas.

```
c, req, rep : \mathsf{Client}\ c, \mathsf{request}\ req\ \mathsf{from}\ \mathsf{client}\ \mathsf{and}\ \mathsf{reply}\ rep\ \mathsf{of}\ req. \mathsf{delivered}(c, req, rep) - \mathsf{Client}\ c\ \mathsf{delivers}\ \mathsf{response}\ rep\ \mathsf{for}\ \mathsf{request}\ req. \mathsf{before}(req, req') - \mathsf{Request}\ req\ \mathsf{is}\ \mathsf{executed}\ \mathsf{prior}\ \mathsf{to}\ \mathsf{request}\ req',\ \mathsf{i.e.},\ req'\ \mathsf{is}\ \mathsf{executed}\ \mathsf{based}\ \mathsf{on}\ \mathsf{execution}\ \mathsf{of}\ req. sg_i: \mathsf{the}\ \mathsf{set}\ \mathsf{of}\ \mathsf{replicas}\ \mathsf{in}\ \mathsf{synchronous}\ \mathsf{group}\ i. \mathsf{accepted}(c, req, rep, i) - \mathsf{Client}\ c\ \mathsf{receives}\ t+1\ \mathsf{matching}\ \mathsf{replies}\ \mathsf{of}\ req\ \mathsf{from}\ \mathsf{every}\ \mathsf{active}\ \mathsf{replica}\ \mathsf{in}\ \mathsf{view}\ i. \mathsf{prefix}(req, req', s_j) - \mathsf{Request}\ req'\ \mathsf{is}\ \mathsf{executed}\ \mathsf{after}\ \mathsf{execution}\ \mathsf{of}\ \mathsf{req}\ \mathsf{at}\ \mathsf{replica}\ s_j. \mathsf{committed}(req, i, sn, s_j) - \mathsf{Active}\ \mathsf{replica}\ s_j \in sg_i\ \mathsf{has}\ \mathsf{received}\ f+1\ \mathsf{matching}\ \mathsf{PREPARE}\ \mathsf{or}\ \mathsf{COMMIT}\ \mathsf{messages}. \mathsf{sg-committed}(req, i, sn, s_j) - \mathsf{Active}\ \mathsf{replica}\ s_j \in sg_i\ \mathsf{has}\ \mathsf{executed}(req, i, sn, s_j). \mathsf{executed}(req, i, sn, s_j) - \mathsf{Active}\ \mathsf{replica}\ s_j \in sg_i:\ \mathsf{executed}(req, i, sn, s_j). \mathsf{prepared}(req, i, sn, s_j) - \mathsf{Active}\ \mathsf{replica}\ s_j \in sg_i\ \mathsf{has}\ \mathsf{received}\ \mathsf{PREPARE}\ \mathsf{message}\ \mathsf{at}\ sn\ \mathsf{for}\ req.
```

Figure 14: XPaxos proof notation.

C.1 Safety (Consistency)

Theorem 1. (safety) If delivered(c, req, rep), delivered(c', req', rep'), and req \neq req', then either before(req, req') or before(req', req).

To prove the safety property, we start from Lemma 2 which shows a useful relation between predicates delivered() and accepted().

Lemma 2. (view exists) delivered(c, req, rep) $\Leftrightarrow \exists view i: accepted(c, req, rep, i)$.

Proof: By common case protocol Algorithm 1 lines:{26-29} and Algorithm 2 lines:{22-25}, client c delivers a reply only upon it receives t+1 matching REPLY messages from all active replicas in the same view. Conversely, upon client c receives t+1 matching REPLY messages from active replicas in the same view, it delivers the reply.

Lemma 3. (reply is correct) If accepted(c, req, rep, i), then rep is the reply of req executed by correct replica.

Proof:

- 1. $\exists s_j \in sg_i$: s_j is correct. **Proof**: Assumption of at most t faulty replicas and $|sg_i| = t + 1$.
- 2. Client c expects matching replies from t+1 active replicas in sg_i . Proof: By common case protocol Algorithm 1 lines:{26-29} and Algorithm 2 lines:{22-25}.
- 3. Q.E.D. Proof: By 1 and 2.

By Lemma 2 and Lemma 3, we assume \exists view i for req and $\exists i'$ for req', then we instead prove:

Theorem 2. (safety) If accepted(c, req, rep, i) and accepted(c', req', rep', i'), then before(req, req') or before(req', req).

Now we introduce sequence number.

Lemma 4. (sequence number exists) If accepted(c, req, rep, i), then \exists sequence number sn: sg-executed(req, i, sn).

Proof:

- 1. Client c accepts rep in view i as reply of req upon:
 - (1) c receives REPLY messages with matching ts, rep, sn and i; and,
 - (2) REPLY messages are attested by t+1 active replicas in sg_i .

Proof: By common case protocol Algorithm 1 lines: {26-29} and Algorithm 2 lines: {22-25}.

- 2. Benign active replica $s_j \in sg_i$ sends REPLY message for req only upon $\exists sn : \text{executed}(req, i, sn, s_j)$. **Proof**: By common case protocol Algorithm 1 $lines:\{21-25\}$ and Algorithm 2 $lines:\{18-21\}$.
- 3. Q.E.D.

Proof: By 1 and 2. \Box

By Lemma 4, we assume \exists sequence number sn for reg and \exists sn' for reg'. Then we instead prove:

Theorem 3. (safety) If sg-executed(req, i, sn), sg-executed(req', i', sn') and sn < sn', then \forall benign active replica $s_{j'} \in sg_{i'}$: $prefix(req, req', s_{j'})$.

Towards the proof of Theorem 3, we first prove several lemmas below (from Lemma 5 to Lemma 11). Lemma 5 proves that if a request is executed by a benign active replica, then that request has been committed by the same replica.

Lemma 5. If $executed(req, i, sn, s_i)$, then $committed(req, i, sn, s_i)$.

Proof: By common case protocol Algorithm 1 lines:{10-21} and Algorithm 2 lines:{10-18}, every benign active replica first commits a request by receiving t+1 matching PREPARE or COMMIT messages, then it executes the request based on committed order.

Lemma 6. (committed() is unique) If committed(req, i, sn, s_j) and committed($req', i, sn, s_{j'}$), then req = req'.

Proof: Proved by contradiction.

- 1. We assume \exists requests req and req': committed (req, i, sn, s_j) , committed $(req', i, sn, s_{j'})$ and $req \neq req'$.
 - **Proof**: Contradiction assumption.
- 2. \exists correct active replica $s_k \in sg_i : s_k$ has sent PREPARE or COMMIT message for both req and req' at sn (i.e., s_k has executed common case protocol Algorithm 1 $lines:\{8-9\}$ or Algorithm 1 $lines:\{15-17\}$, or Algorithm 2 $lines:\{7-9\}$ or Algorithm 2 $lines:\{14-15\}$, for both req and req'). Proof: By $|sg_i| = t + 1$, $\exists s_k : s_k$ is correct; then by 1, common case protocol Algorithm 1 $lines:\{18-20\}$ or Algorithm 2 $lines:\{16-17\}$, and definition of committed().
- 3. Q.E.D.

Proof: By 2 and 1. \Box

Lemma 7 locates at the heart of XPaxos safety proof, which is proved by induction. By Lemma 7 we show that, if request req is committed at sn by every (benign) active replica in the same view, and, if request req' is committed by any replica in the preceding view at sn, then req = req'.

Lemma 7. (sg-committed() is durable) If sg-committed(req, i, sn), then $\forall i' > i$: if committed(req', i', sn, $s_{j'}$) then req = req'.

Proof:

1. We assume $\forall i''$ and $s_{j''}$: $i \leq i'' < i'$ and $s_{j''} \in sg_{i''}$, if committed $(req'', i'', sn, s_{j''})$ then req = req''.

Proof: Inductive Hypothesis.

2. \forall benign replica $s_{j'} \in sg_{i'}$: $s_{j'}$ has been waiting for VIEW-CHANGE messages from $\forall s_k \in \Pi$ within 2Δ time.

Proof: By committed $(req', i', sn, s_{j'})$, $s_{k'}$ has generated PREPARE or COMMIT message at sn; by view change protocol Algorithm 3 $lines:\{23,26,28\}$, a benign active replica generates a PREPARE or COMMIT message in view i' only upon the replica has executed Algorithm 3 $lines:\{16\}$ in view i'; then by Algorithm 3 $lines:\{13-15\}$.

3. $\exists s_{j'} \in sg_{i'} : s_{j'} \text{ is correct.}$

Proof: By $|sg_{i'}| = t + 1$ and at most t faulty replicas.

4. During view change to i', $s_{j'}$ has collected VIEW-CHANGE message m from a correct active replica $s_i \in sg_i$.

Proof: By 2 and 3, view change protocol Algorithm 3 lines: $\{13\text{-}15\}$ have been executed at $s_{j'}$; $s_{j'}$ polls all replicas for VIEW-CHANGE messages and waits for response from t+1 replicas as well as the timer set to 2Δ to expire. Assume that $s_{j'}$ has received VIEW-CHANGE messages from $r \geq 1$ replicas in view i. The other t+1-r replicas in view i are either faulty or partitioned based on definitions. Among r replicas which have replied, at most t-(t+1-r)=r-1 are faulty. Hence, at least one replica, say, $s_i \in sq_i$ is correct and has replied with m.

5. m contains t+1 matching PREPARE or COMMIT messages for request req'' at sequence number sn, generated in view $i'' \ge i$.

Proof: By Algorithm 3 lines: $\{6\text{-}7\}$, benign replicas process messages in ascending view order, so that commit log at sn generated in view i will not be replaced by any commit log generated in view i''' < i; then by 4 and sg-committed (req, i, sn).

6. In view i', $\forall s_{k'} \in sg_{i'} : s_{k'}$ can commit req'', or any req''' which is committed in view i''' > i'' at sn.

Proof: By Algorithm 3 lines: $\{19\}$ and 5.

7. req'' = req''' = req.

Proof: By 4 and 5, req'' is committed in i'' and req''' is committed in i''', where $i''' > i'' \ge i$; then by 1.

8. req' = req.

Proof: By 6, 7 and committed $(req', i', sn, s_{i'})$.

By Lemma 7 we can easily get Lemma 8.

Lemma 8. If sq-committed(req, i, sn) and sq-committed(req', i', sn), then req = req'.

Proof: By Lemma 7 and definition of sg-committed().

Lemma 9. If $executed(req, i, sn, s_i)$, then $\forall sn' < sn : \exists req' \ s.t. \ committed(req', i, sn', s_i)$.

Proof: By common case protocol Algorithm 1 lines: $\{21-22\}$ and Algorithm 2 lines: $\{18-19\}$, correct active replicas execute requests based on order defined by committed sequence number; by executed (req, i, sn, s_i) and sn' < sn, executed (req', i, sn', s_i) ; and, by Lemma 5.

Lemma 10. (executed() in order) If committed(req, i, sn, s_j), executed(req', i, sn', s_j) and sn < sn', then prefix(req, req', s_j).

Proof: By Lemma 9, $\exists req''$ s.t. $committed(req'', i, sn, s_j)$; by Lemma 6, req'' = req; by common case protocol Algorithm 1 $lines:\{21-22\}$ and Algorithm 2 $lines:\{18-19\}$, benign active replicas execute requests based on order defined by committed sequence number sn and sn'; and, by sn < sn'. \square

Lemma 11. If sg-committed(req, i, sn), sg-executed(req', i, sn') and sn < sn', then \forall benign active $replica\ s_j: prefix(req, req', s_j)$.

Proof: By Lemma 10. □

Now we can prove Theorem 3.

Proof:

1. sg-committed(req, i, sn) and sg-committed(req', i', sn').

Proof: By sg-executed(req, i, sn), sg-executed(req', i', sn') and Lemma 5.

When i < i':

2. $\operatorname{sg-committed}(req, i', sn)$.

Proof: By sg-executed(req', i', sn'), Lemma 9 and sn < sn', $\exists req'' : sg-committed(req'', i', sn)$; then by Lemma 8, sg-committed(req, i, sn) and i < i', req'' = req.

3. \forall benign active replica $s_{i'} \in sg_{i'}$: prefix $(req, req', s_{i'})$.

Proof: By sg-executed (req', i', sn'), 2, sn < sn' and Lemma 11.

When i = i':

4. \forall benign active replica $s_j \in sg_i$: prefix (req, req', s_j) .

Proof: By 1, sg-executed(req', i', sn'), i = i', sn < sn' and Lemma 11.

When i > i':

5. $\exists req''$: sg-committed(req'', i', sn).

Proof: By Lemma 9, sg-executed (req', i', sn') and sn < sn'.

6. req'' = req.

Proof: By 5 and Lemma 8.

7. \forall benign active replica $s_{i'} \in sg_{i'}$: prefix (req, req', s_i) .

Proof: By 5, 6, sg-executed(req', i', sn'), sn < sn' and Lemma 11.

8. Q.E.D.

Proof: By 3, 4 and 7. \Box

C.2 Liveness (Availability)

Before proving liveness property, we first prove two Lemmas (12 and 13).

Lemma 12. If a correct client c issues a request req in view i, then eventually, either (1) accepted (c, req, rep, i) or (2) XPaxos changes view to i + 1.

Proof:

1. We assume accepted(c, req, rep, i) is false, then we prove that eventually view i is changed to i+1.

Proof: Equivalent.

2. Client c sends req to every active replica upon $timer_c$ expires.

Proof: By 1, c is correct, and Algorithm 4 lines: $\{1-2\}$.

3. No replica in sg_i sent matching SIGNED-REPLY message for req to client c.

Proof: By 1, c is correct and Algorithm 4 lines: $\{18-22\}$.

4. \exists active replica $s_j \in sg_i$: s_j is correct.

Proof: By assumption $|sg_i| = t + 1$ and at most t faulty replicas.

5. s_i has not received t+1 matching signed REPLY messages for req.

Proof: By 3, 4 and Algorithm 4 lines:{18-22}.

Either,

6. s_i starts $timer_{req_c}$.

Proof: By 2, 4 and Algorithm 4 $lines: \{3,6\}$.

7. s_j suspects view i when $timer_{req_c}$ expires.

Proof: By 4, 5, 6 and Algorithm 4 $lines:\{8-10\}$.

or,

8. s_j starts $timer_i^{vc}$ in view change to i.

Proof: By Algorithm 3 $lines:\{15\}$.

9. s_j suspects view i when $timer_i^{vc}$ expires.

Proof: By 2, 8 and Algorithm 3 $lines: \{34-35\}$.

10. Q.E.D.

Proof: By 1 and 7, 9.

Lemma 13. If a correct client c issues a request req in view i, the system is synchronous for a sufficient time and \forall active replica $s_j \in sg_i$: s_j is correct, then eventually accepted (c, req, rep, i).

Proof:

1. All active replicas in sg_i and c follows protocol correctly.

Proof: c is correct and \forall active replica $s_j \in sg_i$: s_j is correct.

2. No timer expires.

Proof: By 1 and the system is synchronous.

3. No view change happens.

Proof: By 1 and Algorithm 3 lines: $\{1-7\}$, no faulty replica in sg_i , and no faulty passive replica in view i can suspect view i deliberately; and by 2, no correct replica in sg_i suspects view i.

4. accepted(c, req, rep, i).

Proof: By 3 and Lemma 12.

Theorem 4. (liveness) If a correct client c issues a request req, then eventually, delivered(c, req, rep).

Proof: Proved by Contradiction.

1. We assume delivered (c, req, rep) is always false.

Proof: Contradiction assumption.

2. If current view is i, then view is eventually changed to i + 1.

Proof: By 1, Lemma 2 and Lemma 12.

3. View change is executed for infinite times.

Proof: By 1 and 2, Algorithm 4 lines: $\{11-15\}$ and Algorithm 4 lines: $\{1-2\}$, correct client c always multicasts SUSPECT message and req to every active replica in new view.

4. Eventually the system is synchronous.

Proof: Eventual synchrony assumption.

5. \exists view i': \forall active replica $s_{i'} \in sg_{i'}$ s.t. $s_{i'}$ is correct.

Proof: View change protocol is rounded among combinations of 2t + 1 replicas, among which there exists one synchronous group containing only correct active replicas.

6. accepted(c, req, rep, i').

Proof: By 3, 4, 5, Lemma 13 and c is correct.

7. Q.E.D.

Proof: By 1, 6, Lemma 2 and contradiction.

C.3 Fault detection (FD)

In this section we prove that the fault detection mechanism is strong completeness and strong accuracy outside anarchy.

At first, in Definition 4 we define the type of messages which can possibly violate consistency in anarchy.

Definition 4. (non-crash faulty message) In view change to i, a VIEW-CHANGE message m from replica s_k is a non-crash faulty message if:

- (i) m is sent to a correct active replica $s_i \in sq_i$;
- (ii) $\exists view i' < i \text{ and request } req : sg committed(req, i', sn);$
- (iii) at least one of two properties below is satisfied:
 - (1) $s_k \in sg_{i'}$ and in m: PrepareLog[sn] is generated in view i'' < i'; or,
 - (2) in m: $PrepareLog[sn].req \neq req$ and PrepareLog[sn] is generated in view $i'' \geq i'$; and,
- $(iv) \ \exists i''' \ (i''' > i'' \ and \ i''' > i') \ and \ s_{k'''} \in sg_{i'''} : committed(req, i''', sn, s_{k'''}).$

Then we can prove:

Lemma 14. If a VIEW-CHANGE message m is not a non-crash faulty message, then m cannot violate consistency in anarchy.

Proof: Proved by Contradiction.

- 1. If Definition 4 property (i) is not satisfied, then either m is sent to a non-crash faulty replica, based on our model we have no assumption on non-crash faulty replicas, so m should not affect the state of any correct replica; or m is sent to a crashed or passive replica, which just stops processing or ignores m.
- 2. If Definition 4 property (ii) is not satisfied, then req has not been committed by some correct replica in $sg_{i'}$, hence accepted(c, req, rep, i') is not true.
- 3. If neither of Definition 4 property (iii).(1) or (2) is satisfied, then either $s_k \in sg_{i'}$ and m contains prepare log of req at sn generated in view $i'' \geq i$, so by Algorithm 5 $lines:\{11-21\}$, m facilitates req to be committed in view i; or, if $s_k \notin sg_{i'}$, then either PrepareLog[sn].req is generated in i'' < i', even if $PrepareLog[sn].req \neq req$, based on Algorithm 5 $lines:\{13,14,18\}$ PrepareLog[sn].req cannot be selected in view change to i if no (faulty) replica in i' sends inconsistent message (e.g., a prepare log generated in view lower than i' by $s_{k'} \in sg_{i'}$), hence we consider in this case s_k is harmless; or $i'' \geq i$ and PrepareLog[sn].req = req, the argument is the same as before.

4. if Definition 4 property (iv) is not satisfied, then $\exists i''' \ (i''' > i'')$ and i''' > i') and $s_{k'''} \in sg_{i'''}$: committed $(req, i''', sn, s_{k'''})$. In this case, to modify req committed at sn, at least one of (faulty) replicas in $sg_{i'''}$ has to send a non-crash faulty message; otherwise, based on Algorithm 5 lines:{11-21}, any non-crash faulty message generated in i'' will be ignored.

Finally, we prove fault detection property: strong completeness and strong accuracy. Roughly speaking, (strong completeness) if a message is a *non-crash faulty* message, then the sender will be detected eventually; otherwise, (strong accuracy) if a replica is correct, then it will never be detected.

Theorem 5. (strong completeness) If a replica s_k fails arbitrarily outside anarchy, in a way that would cause inconsistency in anarchy, then XPaxos FD detects s_k as faulty (outside anarchy).

Proof:

- 1. By Lemma 14, it is equivalent to prove: in view change to i, if m is a non-crash faulty message from replica s_k , then correct active replica $s_i \in sg_i$ detects the fault of s_k .
- 2. By Definition 4 property (ii), every correct replica $s_{k'} \in sg_{i'}$ has commit log of req at sn generated in view equal to or higher than i'. Assume that the highest view in which commit log of req is generated is i_0 ($i' \le i_0 < i$).

Proof: By Lemma 7.

If in 2 $i_0 = i'$:

3. Correct active replica $s_j \in sg_i$ should receive m' which contains commit log of req generated in view i' from correct active replica $s_{k'} \in sg_{i'}$.

Proof: By outside anarchy, 2, Definition 4 and Lemma 7.

- 4. If m satisfies Definition 4 property (iii).(1), then s_j detects the fault of s_k . **Proof**: By Definition 4 property (iii).(1), prepare log of req is not included in m; then by 3 and Algorithm 6 $lines:\{3\}$, the fault is detected.
- 5. If m satisfies Definition 4 property (iii).(2), then s_j detects the fault of s_k . **Proof**: By Definition 4 property (iii).(2), the prepare log at sequence number sn is generated in view i'' < i, then by 3 and Algorithm 6 $lines:\{6\}$ the fault of s_k is detected.
- 6. If m satisfies Definition 4 property (iii).(3), then s_i detects the fault of s_k .

Proof: If in Definition 4 property (iii).(3) i'' = i', then by 3 and Algorithm 6 lines:{6} the fault of s_k is detected; otherwise, if i'' > i', then based on Lemma 7 req must be retrieved by every correct active replica in view i''; hence by outside anarchy and Algorithm 6 lines:{9-14} the fault of s_k is detected.

If in 2 $i_0 > i'$:

7. Every replica (correct or faulty) in view i_0 has retrieved and prepared req in view equal to or higher than i_0 .

Proof: By 2, i''' > i' and Algorithm 3 lines: $\{26,28\}$.

8. In order to modify request committed at sn (i.e., req), at least one of (faulty) replicas, say $s_{k'''}$ (in sg_{i_0} or not), has to send an inconsistent prepare log generated in view $i_1 \geq i_0$. Hence, s_k in this case is harmless.

Proof: By 7, n = 2t + 1, $i_0 > i'$ and Algorithm 5 lines: {19}.

9. Correct active replica $s_j \in sg_i$ should receive m' which contains commit log of req generated in view i_2 ($i' \le i_2 \le i_0 \le i_1 < i$) from correct active replica $s_{k'} \in sg_{i'}$.

Proof: By outside anarchy, 2, Definition 4 and Lemma 7.

10.	If $i_2 < i_1$, then the fault of $s_{k'''}$ is detected by Algorithm 6 lines:{9-16}, which is similar to discussion in 6; if $i_2 = i_1$, then the fault of $s_{k'''}$ is detected by Algorithm 6 lines:{3,6}, which is similar to discussion in 4 or 5.
11.	Q.E.D. Proof : By 3, 4, 5 and 6 and 10.
	orem 6. (Strong accuracy) If a replica s_k is benign (i.e., behaves faithfully), then XPaxos FD never detect s_k as faulty.
F	Proof:
1.	It is equivalent to prove: in view change to i , if s_k is benign and s_k sends a VIEW-CHANGE message m to all active replicas in view i , then no active replica in sg_i can detect s_k as faulty. $Proof$: Equivalent.
	\forall request req , view $i' < i$ and replica $s_{j'}$ s.t. $s_k, s_{j'} \in sg_{i'}$ and committed $(req, i', sn, s_{j'})$:
2.	m contains prepare log of req' at sn generated in view $i'' \geq i'$. Proof : By common case protocol Algorithm 1 $lines:\{9,17\}$, Algorithm 2 $lines:\{9,15\}$ and viewchange Algorithm 5 $lines:\{1\}$, s_k sends a prepare log at sequence number sn once s_k prepared a request at sn ; by Algorithm 3 $lines:\{6-7\}$, correct replicas process messages in ascending view order, hence $i'' \geq i'$.
3.	s_k will not be detected by Algorithm 6 $lines:\{3\}$ due to committed $(req, i', sn, s_{j'})$. Proof : By 2 and Algorithm 6 $lines:\{3\}$.
4.	No other request $req'' \neq req'$ is committed by any replica at sequence number sn in view i' . Proof : By s_k is correct and Lemma 6.
5.	s_k will not be detected by Algorithm 6 lines:{6} due to committed($req, i', sn, s_{j'}$). Proof : By 4 and Algorithm 6 lines:{6}.
6.	s_k will not be detected by Algorithm 6 $lines:\{9\}$ due to committed $(req, i', sn, s_{j'})$. Proof : By s_k is correct, s_k did not generate or accept any incorrect prepare log during view-change to view i'' ; by Algorithm 6 $lines:\{9\}$, Algorithm 5 $lines:\{3-7\}$ and Lemma 7, no conflict $vcSet_{k'}^{i''}$ and $finalProof_{s_{k'}}[i'']$ exists in view i'' at any active replica.
7.	Q.E.D.
	Proof : By 3, 5 and 6. \Box

We can easily prove that if a fault is detected by any correct replica, then the fault is detected by

Lemma 15. In view change to i, if a correct active replica $s_j \in sg_i$ detects the fault of s_k , then

every replica eventually.

eventually every correct replica detects the fault of s_k .

Proof: By Algorithm 6 *lines*:{6-7}.

Appendix D Reliability analysis (examples)

In Table 5 and 6 we show the nines of consistency of each model when t = 1 and t = 2 for some practical values of 9_{benign} , $9_{synchrony}$ and $9_{correct}$; in Table 7 and 8 we show the nines of availability of each model when t = 1 and t = 2 for some practical values of $9_{available}$ and 9_{benign} .

			9ofC								
0.	$9ofC(CFT_{t=1})$	$9_{correct}$		9	synchro	$9ofC(BFT_{t=1})$					
9_{benign}	$gojC(CTTt\equiv 1)$	Jcorrect	2	3	4	5	6	$gojC(DI^{*}I_{t=1})$			
3	2	2	3	4	4	4	4	5			
4	3	2	4	5	5	5	5	7			
4		3	5	5	6	6	6	'			
		2	5	6	6	6	6				
5	4	3	6	6	7	7	7	9			
		4	6	7	7	8	8				
		2	6	7	7	7	7				
6	5	3	7	7	8	8	8	11			
0	9	4	7	8	8	9	9				
		5	7	8	9	9	10				
		2	7	8	8	8	8				
		3	8	8	9	9	9				
7	6	4	8	9	9	10	10	13			
		5	8	9	10	10	11				
		6	8	9	10	11	11				
		2	8	9	9	9	9				
		3	9	9	10	10	10				
8	7	4	9	10	10	11	11	15			
	'	5	9	10	11	11	12	10			
		6	9	10	11	12	12				
		7	9	10	11	12	13				

Table 5: $9ofC(CFT_{t=1})$, $9ofC(\mathsf{XPaxos}_{t=1})$ and $9ofC(BFT_{t=1})$ values when $3 \leq 9_{benign} \leq 8$, $2 \leq 9_{synchrony} \leq 6$ and $2 \leq 9_{correct} < 9_{benign}$.

			9ofC						
0,	$9ofC(CFT_{t=2})$	$9_{correct}$		9_s	ynchro	ny		$9ofC(BFT_{t=2})$	
9_{benign}	30JC(C1 1 t=2)	Jeorrect	2	3	4	5	6	$30JC(DT T_{t=2})$	
3	2	2	4	5	5	5	5	7	
4	3	2	5	6	6	6	6	10	
4		3	6	7	8	8	8	10	
		2	6	7	7	7	7		
5	4	3	7	8	9	9	9	13	
		4	7	9	10	11	11		
		2	7	8	8	8	8		
6	5	3	8	9	10	10	10	16	
		4	8	10	11	12	12		
		5	8	10	12	13	14		
		2	8	9	9	9	9		
		3	9	19	11	11	11		
7	6	4	9	11	12	13	13	19	
		5	9	11	13	14	15		
		6	9	11	13	15	16		
		2	9	10	10	10	10		
		3	10	11	12	12	12		
8	7	4	10	12	13	14	14	22	
	'	5	10	12	13	15	16	22	
		6	10	12	14	16	17		
		7	10	12	14	16	18		

Table 6: $9ofC(CFT_{t=2})$, $9ofC(\mathsf{XPaxos}_{t=2})$ and $9ofC(BFT_{t=2})$ values when $3 \leq 9_{benign} \leq 8$, $2 \leq 9_{synchrony} \leq 6$ and $2 \leq 9_{correct} < 9_{benign}$.

		90	fA(C)	$FT_{t=}$	=1)				
$9_{available}$			9_{be}	nign			$9ofA(BFT_{t=1})$	$9ofA(XPaxos_{t=1})$	
Javailable	3	4	5	6	7	8	$JOJI(DI\ It=1)$	$30jH(\mathcal{M} axos_{t=1})$	
2	2	3	3	3	3	3	3	3	
3		3	4	5	5	5	5	5	
4			4	5	6	7	7	7	
5				5	6	7	9	9	
6					6	7	11	11	

Table 7: $9ofA(CFT_{t=1})$, $9ofA(BFT_{t=1})$ and $9ofA(\mathsf{XPaxos}_{t=1})$ values when $2 \leq 9_{available} \leq 6$ and $9_{available} < 9_{benign} \leq 8$.

		90	fA(C	$FT_{t=}$	=2)				
9_{benign}						$9ofA(BFT_{t=2})$	$9ofA(XPaxos_{t=2})$		
$9_{available}$	3	4	5	6	7	8	30J1(B1 1t=2)	$\mathcal{F}_{0}(\mathcal{M} axos_{t=2})$	
2	2	3	4	4	4	5	4	5	
3		3	4	5	6	7	7	8	
4			4	5	6	7	10	11	
5				5	6	7	13	14	
6					6	7	16	17	

Table 8: $9ofA(CFT_{t=2})$, $9ofA(BFT_{t=2})$ and $9ofA(\mathsf{XPaxos}_{t=2})$ values when $2 \le 9_{available} \le 6$ and $9_{available} < 9_{benign} \le 8$.