XFT: Practical Fault Tolerance Beyond Crashes

Shengyun Liu
NUDT[*]

Paolo Viotti
EURECOM

Christian Cachin
IBM Research - Zurich

Vivien Quéma
Grenoble INP

Marko Vukolić
IBM Research - Zurich

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 to 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 cost 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. 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

Tolerating 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., [12, 7]) increase the number of *nines of reliability*[1] 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 [36, 9].

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 [13, 3].

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 yet. In particular, asynchronous Byzantine fault-tolerance (BFT) that promises to resolve this problem [8] has not lived up to this expectation, largely due to its extra cost compared to CFT. Namely, asynchronous (that is, "eventually synchronous" [17]) BFT SMR must use at least $3t + 1$ replicas to tolerate t non-crash faults [6] instead of only $2t + 1$ replicas for CFT, as used by Paxos [27] or Raft [34], 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 *en-*

[*]Work done while PhD student at EURECOM.

[1]As 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.

tire 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 for building resilient systems that:

- do not use extra resources (replicas) compared to asynchronous CFT;
- preserve *all* reliability guarantees of asynchronous CFT (that is, in absence of Byzantine faults); and
- provide correct service (i.e., safety and liveness [1]) even when Byzantine faults do occur, so 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 due to a network fault).

In particular, we envision XFT for wide-area or *geo-replicated* systems [12], 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 good 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 in 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 the 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 rest of this paper is organized as follows. In Section 2, we first define the system model, which is then followed by the definition of the XFT reliability 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. For space limitations, the full correctness proof of XPaxos is given in [31].

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 be Byzantine *faulty*: we distinguish between *crash* faults, where a machine simply stops any 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. Notice 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 [17].

Note that we model 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, long local processing time is never an issue on correct machines compared to network delays.

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

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 are more than one subset with the maximum size, only one of them is recognized as the largest subset. For example in Fig 1, the number of partitioned replicas is 3, counting either group of p_1, p_4 and p_5 or group 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.

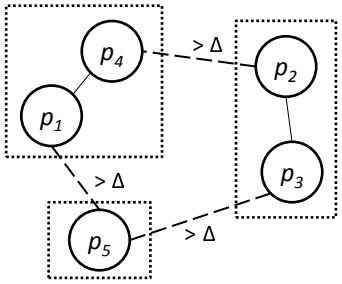


Figure 1: An illustration of partitioned replicas: $\{p_1, p_4, p_5\}$ or $\{p_2, p_3, p_5\}$ are partitioned based on Definition 1.

Problem. In this paper we focus on the state-machine replication problem (SMR) [36]. 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 to the client an application-level reply.

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

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 (that includes *any number* of network faults). Hence, previous work can be classified into four categories: synchronous CFT [15, 36], asynchronous CFT [36, 27, 33], synchronous BFT [29, 16, 5], and asynchronous BFT [8, 2].

XFT, in contrast, redefines the boundaries between machine and network fault dimensions: XFT allows designing 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:

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 and liveness properties [1].

Definition 3 (XFT protocols)**.** *Protocol P is an XFT protocol if: (a) P satisfies safety in all executions in which the system is never in anarchy, and (b) P satisfies liveness eventually provided a majority of replicas is correct and synchronous.*

		Maximum number of each type of replica faults		
		non-crash faults	crash faults	partitioned replicas
Asynchronous CFT (e.g., Paxos [28])	consistency	0	n	$n-1$
	availability	0	$\lfloor \frac{n-1}{2} \rfloor$ (combined)	
Asynchronous BFT (e.g., PBFT [8])	consistency	$\lfloor \frac{n-1}{3} \rfloor$	n	$n-1$
	availability	$\lfloor \frac{n-1}{3} \rfloor$ (combined)		
(Authenticated) Synchronous BFT (e.g., [29])	consistency	$n-1$	n	0
	availability	$n-1$ (combined)		0
XFT (e.g., XPaxos)	consistency	0	n	$n-1$
		$\lfloor \frac{n-1}{2} \rfloor$ (combined)		
	availability	$\lfloor \frac{n-1}{2} \rfloor$ (combined)		

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

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, 34] guarantee consistency despite *any* number of crash faulty replicas and despite *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 not partitioned. As soon as a single machine is non-crash faulty, CFT protocols guarantee neither consistency nor availability.

Optimal asynchronous BFT protocols [8, 22, 2] guarantee consistency despite any number of crash faulty or partitioned replicas and 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. Notice that BFT availability might be weaker than that of CFT in absence of non-crash faults — unlike CFT, BFT does not guarantee availability when the sum of crash-faulty and partitioned replicas is in range $[n/3, n/2)$.

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, guarantees 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 asynchronous CFT and synchronous BFT models. This is however misleading, as even with actual non-crash faults, XFT is incomparable to authenticated synchronous BFT. Namely, authenticated synchronous protocols such as the seminal Byzantine Generals protocol [29] may violate consistency in presence of a single partitioned replica. For instance, with $n = 5$ replicas and execution in which three replicas are correct and synchronous, one replica is correct but partitioned and one replica is non-crash faulty, XFT model mandates that the consistency is preserved whereas the Byzantine Generals protocol may violate consistency.[2]

Furthermore, from Table 1, it is immediate 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, 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 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

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

of independent 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. In principle, we envision XFT for use cases in which an adversary cannot easily coordinate enough network partitions and Byzantine faulty machine actions at the same time. Some interesting candidates 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, with which network faults can be assumed to be largely independent from machine 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, so 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 [32], participants may be financially motivated to act maliciously, yet may lack 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 [40].

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 network as the bottleneck, with high link latency and relatively low, heterogenous 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 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* [8]. 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*. Remaining t replicas in a given view are called *passive* replicas; optionally, passive 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 and not only the leader. All $t+1$ active replicas from the new synchronous group sg_{i+1} try to transfer the state from 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, 8]), 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). XPaxos' novel and decentralized view-change scheme guarantees that, even in 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.

Besides, we specially design a fault detection (FD) mechanism (Section 4.4), which can help detect, outside anarchy, non-crash faults that would leave the system in an inconsistent state in anarchy. The FD mechanism serves to minimize the impact of long-lived non-crash faults in the system and help detect them before they coincide with a sufficient number of crash faults and network faults to push the system into anarchy.

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 omit to 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 synchronously communicate 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 enough crash or partitioned machines (i.e., in anarchy) to violate consistency.

In the following, we explain the core of XPaxos for the common-case (Sec. 4.2), view-change (Sec. 4.3) and fault detection (Sec. 4.4) components. We discuss XPaxos optimizations in Sec. 4.5 and give XPaxos correctness arguments in Sec. 4.6. For space limitations, the complete pseudocode and correctness proof are postponed to [31].

4.2 Common case

Figure 2 shows the common-case message patterns of XPaxos for the general case ($t \geq 2$) 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 very relevant in practice (see e.g., Google Spanner [12]). In the following, we first explain XPaxos in general case, and then focus on the $t = 1$ special case.

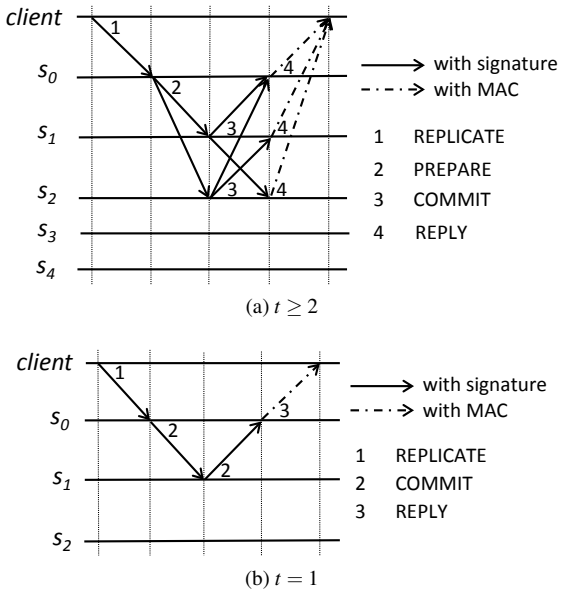


Figure 2: XPaxos common-case message patterns (a) for the general case when $t \geq 2$ and (b) for the special case of $t = 1$. The synchronous groups are (s_0, s_1, s_2) and (s_0, s_1), respectively.

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 ($t \geq 2$)

The common-case message pattern of XPaxos is shown in Fig. 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 clients' timestamp), the primary (say s_0): (1) increments sequence number sn and assigns sn to req, (2) logs $\langle req, prep \rangle$ into its prepare log $PrepareLog_0[sn]$ (we say s_0 *prepares* req), and (3) forwards req to *all other active replicas* (i.e, the t followers) together with the $prep = \langle \text{PREPARE}, D(req), sn, i \rangle_{\sigma_{s_0}}$ message.

Each follower s_j ($1 \leq j \leq t$) verifies the primary's and client's signatures, checks if 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 $\langle \text{COMMIT}, D(req), sn, i \rangle_{\sigma_{s_j}}$ 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 ($0 \leq k \leq t$) logs $prep$ and the t signed COMMIT messages into its commit log $CommitLog_{s_k}[sn]$. We say s_k *commits* req when this oc-

curs. 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 replicas. Active replicas then forward such 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 involve only 2 messages between 2 active replicas (see Fig. 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 if its local sequence number equals $sn - 1$. Then, 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 yet 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 ($CommitLog_{s_1}[sn] = \langle req, m_0, m_1 \rangle$) 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 $R(req)$ to 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 XPaxos view change, consider synchronous groups sg_i and sg_{i+1} of views i and $i+1$, respectively, each containing $t+1$ replicas. Notice that proofs of requests committed in sg_i might have been logged by *only one* correct replica in sg_i. Nevertheless, 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 [8, 22, 11] where the entire view change is led by a single replica, usually the primary of the new view. Namely, 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 $i+1$.

		Synchronous Groups ($i \in \mathbb{N}_0$)		
		sg_i	sg_{i+1}	sg_{i+2}
Active replicas	Primary	s_0	s_0	s_1
	Follower	s_1	s_2	s_2
Passive replica		s_2	s_1	s_0

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

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. Additionally, 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., $t = 1$ nd $t = 2$). For a large number of replicas, the combinatorial number of synchronous groups may be inefficient. To this end, XPaxos may 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 are, however, beyond 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 view change initiation, s_j stops participating in the current view and sends $\langle \text{SUSPECT}, i, s_j \rangle_{\sigma_{s_j}}$ to *all* other replicas.

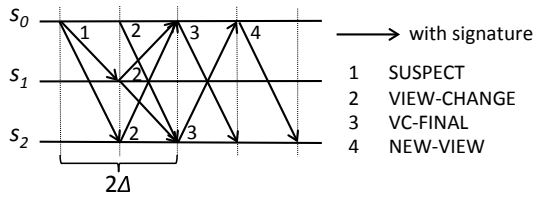


Figure 3: XPaxos view change illustration: synchronous group is changed from (s_0,s_1) to (s_0,s_2).

4.3.3 Performing view-change

Upon receiving a SUSPECT message from an active replica in view i (see the message pattern in Fig. 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 logs might 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 the new primary only. 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 at least 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 in set $VCSet_{s_j}^{i+1}$. Then s_j sends $\langle \text{VC-FINAL}, i+1, s_j, VCSet_{s_j}^{i+1} \rangle_{\sigma_{s_j}}$ 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 array *PrepareLog* contains 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 prepare logs in *PrepareLog* as described in the common case (see Sec. 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 case $t = 1$ (see [31] 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 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 XPaxos view change as follows: in addition to exchanging their commit logs, replicas also exchange their prepare logs. Notice that in 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 common case and view change protocols described above are sufficient to guarantee correctness, we applied several standard performance optimizations to XPaxos. These include checkpointing and lazy replication [26] to passive replicas (to help shorten the state transfer during view change) as well as batching (to improve the throughput). Below, we provide a brief overview of these optimizations — details are postponed

to [31].

Checkpointing. Similarly to other replication protocols, XPaxos includes a checkpointing mechanism that speeds up view changes and allows for garbage collection (by shortening commit logs). To this end, every CHK requests (where CHK is a configurable parameter) XPaxos checkpoints the state within the synchronous group. Then the proof of checkpoint is lazily propagated to passive replicas.

Lazy replication. To speed up the state transfer in view change, the every follower in the synchronous group lazily propagates the commit log to one passive replica. With lazy replication, a new active replica, which might be the passive replica in preceding view, may only need to retrieve the missing state from others during view change.

Batching and pipelining. In order 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. In case there are not enough requests received 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 receives 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 that we postpone to [31].

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 the timer set to 2Δ to expire. Then, replicas in sg_{i+1} exchange this information within VC-FINAL messages. Notice 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 in case the synchronous group contains only correct and synchronous replicas. With eventual synchrony we can assume that, eventually, there will be no network faults. Additionally, 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

In this section, we evaluate the performance of XPaxos and compare it to Zyzzyva [22], PBFT [8] and a WAN-optimized version of Paxos [27], using the Amazon EC2 worldwide cloud platform. We chose a 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 rest of this section, we first present the experimental setup (Section 5.1), and then evaluate the performance (throughput and latency) in the fault-free scenario (Section 5.2), as well as under faults (Section 5.3). Finally, we perform a performance comparison using a real application: the Zookeeper coordination service [19] (Section 5.4), by comparing native Zookeeper to Zookeper 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 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 Sec. 2). Notably, Δ is vital to the XPaxos view-change (Sec. 4.3).

To understand the value of Δ in our geo-replicated context, we ran a 3-month experiment during which we continuously measured round-trip latency across six Amazon EC2 datacenters worldwide using TCP ping

	US West (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 (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% / 99.999% / maximum.

(hping3). We used the least expensive EC2 micro instances, that 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 minutes, our experiment showed that the round-trip latency between *any* two datacenters was less than 2.5 seconds 99.99% of the time. Therefore, we adopted the value of $\Delta = 2.5/2 = 1.25$ seconds.

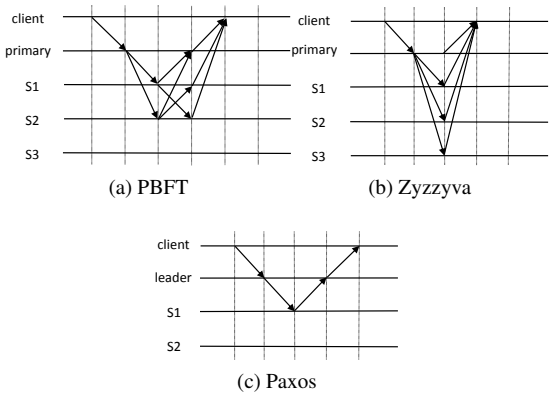
5.1.2 Protocols under test

We compare XPaxos against three protocols whose common case message patterns when $t = 1$ are depicted in Figure 4. The first two are BFT protocols, namely (a speculative variant of) PBFT [8] 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 4a; see also [8]). 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 4b). The third protocol we compare against is a very efficient WAN-optimized variant of crash-tolerant Paxos inspired by [4, 23, 12]. We have chosen the variant of Paxos that exhibits the fastest write pattern (Figure 4c). This variant requires $2t+1$ replicas to tolerate t faults, but involves $t+1$ replicas in the common case, i.e., just like XPaxos.

In order to provide a fair comparison, all protocols rely on the same Java code base and use batching, with batch size set to 20. We rely on HMAC-SHA1 to compute MACs and RSA1024 to sign and verify signatures.

5.1.3 Experimental testbed and benchmarks

We run the experiments on the Amazon EC2 platform that comprises widely distributed datacenters, interconnected by the Internet. Communications between datacenters have a low bandwdith and a high latency. We run the experiments on mid-range virtual machines that contain 8 vCPUs, 15GB of memory, 2 x 80 GB SSD Storage,



(a) PBFT (b) Zyzzyva

(c) Paxos

Figure 4: Communication patterns of the three protocols under test ($t = 1$).

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 analyzed protocol. 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 [37, 12].[3]

To stress the protocols, we run a microbenchmark that is similar to the one used in [8, 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 varying the request size and the reply size. 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 1/0 and 4/0 microbenchmarks. The results are depicted

[3] In practice, modern geo-replicated system, like Spanner [12], use hundreds of CFT SMR instances across different partitions to accommodate for geo-distributed clients.

in Figures 5a and 5b. On each graph, the X-axis shows throughput (in kops/s), and Y-axis shows latency (in ms).

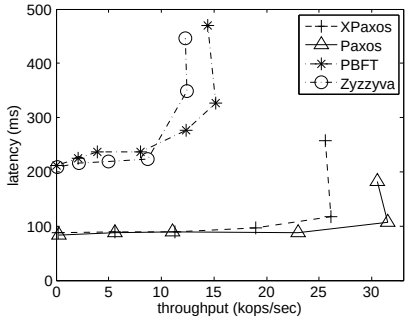
PBFT	Zyzzyva	Paxos	XPaxos	EC2 Region
Primary	Primary	Primary	Primary	US West (CA)
Active	Active	Active	Follower	US East (VA)
		Passive	Passive	Tokyo (JP)
Passive		-	-	Europe (EU)

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

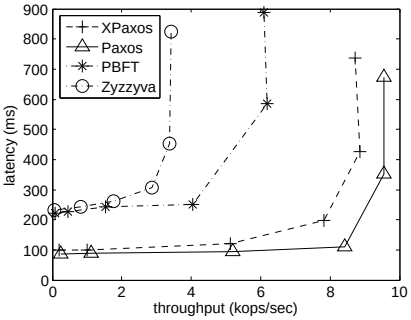
As we can see, in both benchmarks, XPaxos achieves significantly better performance than PBFT and Zyzzyva. This comes from the fact that in a worldwide cloud environment, the network is the bottleneck and the message patterns of BFT protocols, namely PBFT and Zyzzyva, tend to be expensive. Compared to PBFT, XPaxos simpler message pattern allows better throughput, whereas compared to 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 in Zyzzyva). Moreover, XPaxos performance 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 at the first $t + 1$ datacenters, and their passive replicas to 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.

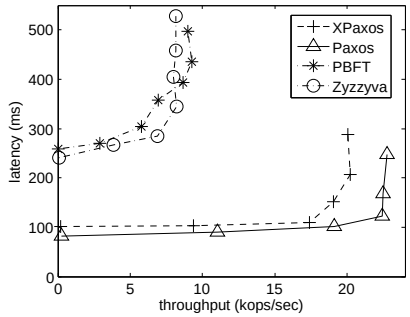
We observe that XPaxos again clearly outperforms PBFT and Zyzzyva and achieves performance very close to that of Paxos. Moreover, unlike PBFT and Zyzzyva, Paxos and XPaxos only suffer a moderate performance decrease with respect to the $t = 1$ case.



(a) 1/0 benchmark, $t = 1$



(b) 4/0 benchmark, $t = 1$



(c) 1/0 benchmark, $t = 2$

Figure 5: Fault-free performance

5.3 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 180s, we crash the follower, VA. At time 300s, we crash the CA replica. At time 420s, we crash the third replica, JP. Each replica recovers 20s after having crashed. Moreover, the timeout 2Δ (used during state transfer in view change, Sec-

tion 4.3) is set to 2.5s (see Sec. 5.1.1). We show the throughput of XPaxos in function of time in Figure 6, 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 10s, 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 latency between the primary and the follower, and between the primary and clients, varies from view to view.

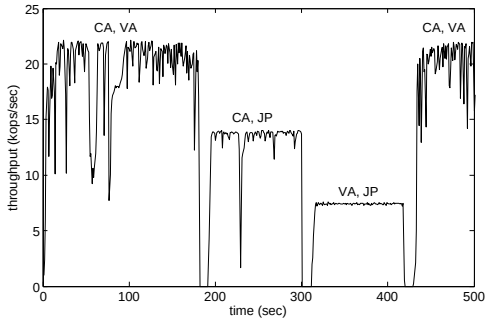


Figure 6: XPaxos under faults.

5.4 Macro-benchmark: ZooKeeper

In order 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 has been 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 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 tested configuration tolerates one fault ($t = 1$). ZooKeeper clients were located in the same region as the primary (CA). Each client invokes 1kB write operations in a closed loop.

Figure 7 depicts the results. The X-axis represents the throughput in kops/sec. The Y-axis represents the latency in ms. As for macro-benchmarks, we observe that Paxos and XPaxos clearly outperform BFT protocols and XPaxos achieves 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, bottleneck in the WAN setting is the bandwidth at the leader, yet the leader in Zab sends requests to all other $2t$ replicas whereas the XPaxos leader sends requests only to t followers, which yields higher peak throughput for XPaxos.

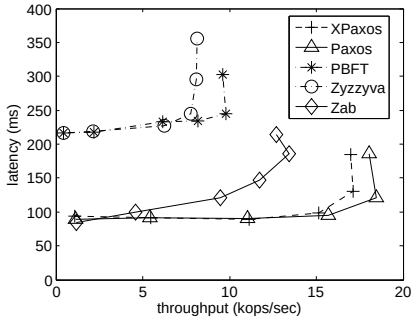


Figure 7: Latency vs. throughput for the ZooKeeper application ($t = 1$).

6 Reliability Analysis

In this section, we illustrate the reliability guarantees of XPaxos by analytically comparing them to 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 $p_{non\text{-}crash} = 1 - p_{benign}$. Besides, we assume there is a probability $p_{synchrony}$ that a replica is not partitioned, where $p_{synchrony}$ is a function of Δ, the network, and the system environment. Finally, the probability that a replica is partitioned equals $1 - p_{synchrony}$.

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.

We focus here on *consistency* guarantees, which is less obvious than availability, given that XPaxos clearly guarantees better availability than any asynchronous CFT or BFT protocol (see Table 1). We postpone availability analysis to [31].

6.1 XPaxos vs. CFT

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

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

which gives $9ofC(CFT) \approx 9_{benign} - \lceil \log_{10}(n) \rceil$ for values of p_{benign} close to 1, when p^i_{benign} 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 [12], a CFT system as a whole loses one nine of consistency from the likelihood that a single replica is benign.

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.,

$$P[\text{XPaxos is consistent}] = p^n_{benign} + \sum_{i=1}^{t=\lfloor \frac{n-1}{2} \rfloor} \binom{n}{i} p^i_{non\text{-}crash} \times \sum_{j=0}^{t-i} \binom{n-i}{j} p^j_{crash} \times p^{n-i-j}_{correct} \times \sum_{k=0}^{t-i-j} \binom{n-i-j}{k} \times p^{n-i-j-k}_{synchrony} \times (1-p_{synchrony})^k.$$

To quantify the difference between XPaxos and CFT more tangibly, we calculated $9ofC(\text{XPaxos})$ and $9ofC(CFT)$ for all values of 9_{benign}, $9_{correct}$ and $9_{synchrony}$ ($9_{benign} \geq 9_{correct}$) 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 ($t = 2$ case is postponed to [31]):

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

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 $p_{benign} \geq p_{correct}$ must hold.

Example 1. When $p_{benign} = 0.9999$ and $p_{correct} = p_{synchrony} = 0.999$, we have $p_{non\text{-}crash} = 0.0001$ and $p_{crash} = 0.0009$. In this example, $9 \times p_{non\text{-}crash} = p_{crash}$, 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 $9ofC(\text{XPaxos}_{t=1}) - 9ofC(CFT_{t=1}) = 9_{correct} - 1 = 2$, i.e., $9ofC(\text{XPaxos}_{t=1}) = 5$. 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 $9ofC(\text{XPaxos}_{t=1}) - 9ofC(CFT_{t=1}) = p_{correct} = 3$, i.e., $9ofC(\text{XPaxos}_{t=1}) = 6$. XPaxos adds 3 nines of consistency on top of CFT and achieves 6 nines of consistency in total.

6.2 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,

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

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

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

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

$$p_{correct} \times p_{synchrony} > p^{1.5}_{benign}.$$

Hence, for $t = 1$, XPaxos has *stronger consistency guarantees* than *any* asynchronous BFT protocol whenever the probability that a machine is correct and not partitioned 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 $t = 1$ ($t = 2$ is again postponed to [31]), 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:

$$9ofC(BFT_{t=1}) - 9ofC(\text{XPaxos}_{t=1}) =$$

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

Notice that in cases where XPaxos guarantees better consistency than BFT ($p_{correct} \times p_{synchrony} > p_{benign}^{1.5}$), 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 $9ofC(BFT_{t=1}) - 9ofC(\text{XPaxos}_{t=1}) = 9_{benign} - 9_{synchrony} + 1 = 2$, i.e., $9ofC(\text{XPaxos}_{t=1}) = 5$ 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 $9ofC(BFT_{t=1}) - 9ofC(\text{XPaxos}_{t=1}) = 1$, i.e., $9ofC(\text{XPaxos}_{t=1}) = 6$ and $9ofC(BFT_{t=1}) = 7$. XPaxos has one nine of consistency less than BFT (albeit the only 7th).

7 Related work and concluding remarks

In this paper we introduced XFT, a novel fault-tolerance model that allows designing 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, yet is also capable of tolerating non-crash faults, so 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 to reducing the resource cost of BFT to $2t + 1$ replicas only [14, 30, 21, 39].

XPaxos is also different from PASC [13], 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 [38] 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 [10, 35] appear orthogonal to our XFT approach.

Specifically, Visigoth Fault Tolerance (VFT) [35] 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. In the future, it would be very interesting to explore interactions between the hybrid fault model (including its refinements such as VFT) and XFT.

Going beyond the research directions outlined above, this paper opens more 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 through the XFT prism.

Acknowledgments

We thank Shan Lu and anonymous OSDI reviewers, for their invaluable comments that helped us considerably improve the paper. Shengyun Liu's work was supported in part by the National Key Research and Development Program (2016YFB1000101). This work was also supported in part by the EU H2020 project SUPERCLOUD (grant No. 643964) and Swiss Secretariat for Education, Research and Innovation (contract No. 15.0025).

References

[1] B. Alpern and F. Schneider. Recognizing safety and liveness. *Distributed Computing*, 2(3):117–

126, 1987.

[2] 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.

[3] P. Bailis and K. Kingsbury. The network is reliable: An informal survey of real-world communications failures. *ACM Queue*, 2014.

[4] J. Baker, C. Bond, J. C. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *Proceedings of the Conference on Innovative Data system Research (CIDR)*, pages 223–234, 2011.

[5] 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.

[6] G. Bracha and S. Toueg. Asynchronous consensus and broadcast protocols. *J. ACM*, 32(4):824–840, 1985.

[7] 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, New York, NY, USA, 2011. ACM.

[8] M. Castro and B. Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst.*, 20(4):398–461, Nov. 2002.

[9] 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 '07, pages 398–407, New York, NY, USA, 2007. ACM.

[10] A. Clement, M. Kapritsos, S. Lee, Y. Wang, L. Alvisi, M. Dahlin, and T. Riche. Upright cluster services. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 277–290, New York, NY, USA, 2009. ACM.

[11] A. Clement, E. 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'09, pages 153–168. USENIX Association, 2009.

[12] 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.

[13] M. Correia, D. G. Ferro, F. P. Junqueira, and M. Serafini. Practical hardening of crash-tolerant systems. In *USENIX ATC'12*, 2012.

[14] 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.

[15] 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.

[16] D. Dolev and H. R. Strong. Authenticated algorithms for Byzantine agreement. *SIAM Journal on Computing*, 12(4):656–666, Nov. 1983.

[17] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35, April 1988.

[18] M. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[19] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, USENIX ATC'10, pages 11–11, Berkeley, CA, USA, 2010. USENIX Association.

[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. Liu, P. Viotti, C. Cachin, V. Quéma, and M. Vukolić. XFT: practical fault tolerance beyond crashes. *CoRR*, abs/1502.05831, 2015.

[32] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. May 2009.

[33] 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.

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

[35] 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.

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

[37] 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.

[38] P. M. Thambidurai and Y. Park. Interactive consistency with multiple failure modes. In *Seventh Symposium on Reliable Distributed Systems, SRDS 1988, Columbus, Ohio, USA, October 10-12, 1988, Proceedings*, pages 93–100, 1988.

[39] 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.

[40] 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, Zurich, Switzerland, October 29, 2015, Revised Selected Papers*, pages 112–125, 2015.