ParisTech
INSTITUT DES SCIENCES ET TECHNOLOGIES
PARIS INSTITUTE OF TECHNOLOGY

TELECOM ParisTech

2015-ENST-0065

EDITE - ED 130

Doctorat ParisTech

T H È S E

pour obtenir le grade de docteur délivré par

TELECOM ParisTech

Spécialité « INFORMATIQUE et RESEAUX »

présentée et soutenue publiquement par

Shengyun LIU

le 21 Octobre 2015

Techniques for efficient and fault-tolerant geo-replication

Directeur de thèse : **Marko VUKOLIĆ**

Jury

M. Pietro MICHIARDI, Professeur, EURECOM, France	Président
M. Rüdiger KAPITZA, Professeur, Université Technique de Braunschweig, Allemagne	Rapporteur
M. Luigi LIQUORI, Directeur de Recherche, INRIA, France	Rapporteur
M. Vivien QUÉMA, Professeur, Grenoble INP, France	Examinateur

TELECOM ParisTech
école de l'Institut Télécom - membre de ParisTech

T H È S E

Abstract

Modern internet applications replicate their service across globally deployed data centers, in order to serve their clients efficiently and reliably. When confronted with failures such as machine crashes or data center outages, many applications rely on a State Machine Replication (SMR) protocol to keep their service up and running correctly. To mask failures, a SMR protocols synchronizes a group of machines (a.k.a., replicas) and mimics a centralized server that the application developers are accustomed to.

Existing work typically target two fault models: (1) Crash-Fault Tolerance (CFT), in which machines can stop processing requests; and (2) Byzantine-Fault Tolerance (BFT), in which machines can behave arbitrarily. However, since the performance of network connections over a large geographical distance is bounded and non-uniform, existing SMR protocols may not be well applicable to new environment.

Despite years of intensive research, a *practical* answer to arbitrary or *non-crash* faults of the machines that comprise a distributed system remains elusive. Classical BFT has not lived to expectations due to its resource and operation overhead with respect to its CFT counterparts. This overhead of BFT comes from the assumption of a powerful adversary that can simultaneously control both the Byzantine faulty machines and the *entire* network at will. To many practitioners, however, such strong attacks appear irrelevant.

The first contribution of this thesis introduces XFT ("cross fault tolerance"), a novel approach to building reliable and secure distributed systems. XFT allows for both Byzantine machines and network faults (communication asynchrony), yet considers them independent. This allows the development of XFT systems at the cost of CFT (already paid for in practice), yet with many more nines of reliability than CFT and sometimes even better reliability than BFT itself. In short, beyond the guarantees of CFT, XFT tolerates Byzantine faults so long as a majority of replicas comprising the system are correct and communicate synchronously. We demonstrate our approach with XPaxos, the first XFT state machine replication protocol. Despite its much stronger reliability guarantees at no extra resource cost, XPaxos performance matches that of the state-of-the-art CFT protocols.

Furthermore, SMR protocols in the context of geo-replication have a specific challenge: latency, bounded by the speed of light. In particular, clients of a classical SMR protocol, such as celebrated Paxos, must communicate with a

dedicated replica named *leader*, which must in turn communicate with other replicas: wrong selection of a leader may result in unnecessary round-trips across the globe. To cope with limitations of *single leader* protocols that are unsuitable for serving widely dispersed clients, several *all-leader* and *leaderless* geo-replication alternatives have been proposed recently. Unfortunately, none of them fits all circumstances.

In the second contribution, we argue that the "right" choice of the number of leaders in a geo-replication protocol depends on a given replica configuration and propose Droopy and Dripple, two sister approaches built upon SMR protocols, respectively based on the *multi-leader* approach and request *commutativity*. Droopy dynamically reconfigures the set of leaders, electing anywhere from a single to all replicas. Whereas, Dripple divides the system state into partitions and coordinates them wisely in order to preserve strong consistency. Droopy and Dripple together allow the set of leaders of each partition to be reconfigured separately. Our experimental evaluation on Amazon EC2 shows that, Droopy and Dripple reduce latency under imbalanced or localized workloads, comparing to their native protocol. When most requests are non-commutative, our approaches do not affect the performance of their native protocol and both outperform a state-of-the-art leaderless protocol.

Keywords: geo-replication, state machine replication, Byzantine-fault tolerance, latency.

Résumé

Les applications Internet modernes répliquent leur service au travers de centres de données reparties tout autour du globe afin de servir leurs clients de manière fiable et efficace. Lorsqu'elles sont confrontées à des pannes telles que des erreurs machines ou des interruptions de services des centres de données, de nombreuses applications reposent sur l'utilisation de protocoles de réplication de machines à états (RME) pour maintenir un service fiable.

Cependant, étant donné que les performances des connexions réseau entre les centres de données distants sont limités et non uniformes, les protocoles RME existants ne sont pas bien applicables à ce nouvel environnement. Les travaux en cours visent généralement l'un des deux modèles de fautes : (1) fautes franches, dans lequel les machines peuvent se stopper, à savoir, arrêter le traitement des requêtes ; ou (2) fautes Byzantines, dans lequel les machines peuvent se comporter de façon arbitraire, à savoir, ne pas suivre le protocole fidèlement.

La première contribution de cette thèse introduit XFT, une nouvelle approche qui découple l'espace de pannes entre les dimensions de la machine et le réseau, ce qui autorise les machines Byzantines et les fautes de réseau (par exemple, la communication asynchrone), mais les considère indépendante. Nous démontrons notre approche avec XPaxos, le premier protocole RME en XFT. XPaxos tolère des fautes au-delà des crashs d'une manière efficace et pratique, avec beaucoup plus de neuf de fiabilité que le célèbre protocole tolérant aux fautes Paxos, cela sans impact sur ses coûts d'exploitation tout en maintenant des performances similaires. Malgré son faible coût et ses hautes performances qui correspondent aux meilleurs protocoles de Paxos, nous montrons aussi que XPaxos fournit toujours une meilleure disponibilité, et parfois (selon l'environnement système) offre même des garanties de cohérence strictement plus fortes que les protocoles de réplication de tolérance aux fautes Byzantines (TFB) de l'état de l'art.

Dans la deuxième contribution, nous soutenons que le bon choix du nombre de principal dans un protocole de géo-réplication dépend de la configuration et répartition des machines. Nous proposons Droopy et Dripple, deux approches soeur pour les protocoles SMR, respectivement basées sur l'approche tous-principal et sur la commutativité des requêtes. Nous implémentons Droopy et Dripple à partir de Clock-RSM, un protocole tous-principal de l'état de l'art. Notre évaluation sur Amazon EC2 montre que, lors de charges de travail

déséquilibrées ou commutatives, Clock-RSM étendu par Droopy réduit efficacement la latence par rapport au protocole natif, et présente un temps de latence similaire à un protocole sans principal — EPaxos. En revanche, lors de charges de travail équilibrées et non-commutatives, Clock-RSM étendu par Droopy et Dripple réduit la latence par rapport à EPaxos, et présente une latence similaire au protocole natif.

Mots-clés: géo-réplication, réplication de machines à états, tolérance aux fautes Byzantines, latence.

Acknowledgements

Foremost, I would like to express my sincere gratitude to Marko Vukolić, my supervisor, for the invaluable guidance and brilliant ideas he has provided in these 4 years. In particular, his step-by-step advising started from the very first day of my PhD and goes through my time at Eurecom. He taught me what I should insist for conducting a good scientific research. It has been a great honor to be his first PhD student.

I'm also very grateful to other members of my thesis committee: Pietro Michiardi, who presided over my thesis committee, and Rüdiger Kapitza, Luigi Liquori and Vivien Quéma, for evaluating my manuscript and providing me precious comments. I would like to give special thanks to Vivien Quéma, for the experiment support he provided, and for many interesting discussions we had, which concerns not only general ideas but also many details of implementation aspect.

I would further like to thank Ernst Biersack, who introduced me to Marko and provided constant help and care.

I was very lucky to have been worked with two smart colleagues in office 368 — Paolo Viotti and Yongchao Tian. They have always been open and very patient to a discussion. Paolo helped me a lot for the experiments of integrating with ZooKeeper; Yongchao tested several candidates for our macro-benchmark experiments.

I would like to thank other colleagues in Eurecom, for many interesting conversations and fun events we had. To Alberto Benegiamo, who brightened our office by introducing many interesting topics and stories. To Heng Cui, Xiaolan Sha and Rui Min, who helped me survived in the first year of my PhD. To Romain Favraud, for his help in improving French. To Gwenaëlle Le-Stir, Audrey Ratier and Christine Mangiapan, for the administrative support they warmly provided.

I'm deeply grateful to my parents Yuangui Liu and Jinhui Wang, for raising me up and teaching me how to be a good man. In particular, my father led me to the hall of computer programming; my mother takes care of every aspect of my growth.

Finally, I would like to express my deep gratitude to Jingjing Zhang, for the wonderful two years we have been together in 36 BD Wilson. You taught me how beautiful life can be.

Preface

The work described in this thesis was conducted under the supervision of Dr. Marko Vukolić (former faculty in Eurecom, now research staff member in IBM Research - Zurich), in the Networking and Security department, Eurecom, between 2011 October and October 2015.

This thesis is a composition of one published paper and two papers submitted to publication, as listed below:

S. Liu and M. Vukolić, How Many Planet-Wide Leaders Should There be? In proceedings of the 3rd Workshop on Distributed Cloud Computing, Special Issue of the ACM Performance Evaluation Review (PER), December 2015.

S. Liu, P. Viotti, C. Cachin, V. Quéma and M. Vukolić, XFT: Practical Fault Tolerance Beyond Crashes. CoRR, abs/1502.05831, 2015 (submitted for publication)

S. Liu and M. Vukolić, Leader Set Selection for Low-Latency Geo-Replicated State Machine. (submitted for publication)

Contents

Abstract **i**

Résumé **iii**

Acknowledgements **v**

Preface **vii**

Contents **ix**

List of Figures **xiii**

List of Tables **xv**

1 Introduction **1**
1.1 Context 1
1.2 Challenges 2
1.2.1 Non-Uniform Connections Across Global Data Centers . 2
1.2.2 The Trade-Off Between Strong Assumption and Large Cost 3
1.3 Contributions 4
1.3.1 XFT : Practical Fault Tolerance Beyond Crashes 4
1.3.2 Leader Set Selection for Low-Latency Geo-Replicated State Machine 5
1.4 Organization 6

2 Preliminaries **7**
2.1 System Model 7
2.2 State Machine Replication 7

3 Related work **11**
3.1 Geographically Distributed Storage Systems 11
3.1.1 Weakly Consistent Systems 11
3.1.2 Strongly Consistent Systems 12
3.1.3 Fault-Tolerance Considerations 14
3.2 Fault-Tolerance Protocols 15

3.2.1 Crash-Fault Tolerance (CFT) 15
3.2.2 Byzantine-Fault Tolerance (BFT) 19
3.2.3 Hybrid-Fault Tolerance 23

4 XFT : Practical Fault Tolerance Beyond Crashes 25
4.1 Introduction 25
4.2 Contributions 26
4.3 System Model Refinement 27
4.3.1 Network Faults 27
4.3.2 Machine Faults 29
4.3.3 Anarchy 29
4.4 The XFT Model 29
4.5 Motivation 31
4.5.1 The Rare Anarchy Argument 31
4.5.2 Long-Lived Arbitrary Faults 32
4.6 Protocol 33
4.6.1 Common Case 34
4.6.2 View Change 36
4.6.3 Correctness Arguments 38
4.6.4 Fault Detection 39
4.6.5 XPaxos Example Execution 41
4.6.6 Request Retransmission 42
4.6.7 XPaxos Optimizations 42
4.7 Reliability Analysis 44
4.7.1 Reliability Parameters 44
4.7.2 Consistency 44
4.7.3 Availability 49
4.8 Performance Evaluation 52
4.8.1 Experimental Setup 52
4.8.2 Fault-Free Performance 54
4.8.3 CPU Usage 54
4.8.4 Performance Under Faults 56
4.8.5 Macro-Benchmark: ZooKeeper 57

5 Leader Set Selection for Low-Latency Geo-Replicated State Machine 59
5.1 Introduction 59
5.2 Related Work Revisited 61
5.2.1 All-leader Protocols (Delayed Commit) 61
5.2.2 Leaderless Protocols (With Conflict Resolution) 63
5.3 System Model Refinement 64
5.4 Protocol Overview 65
5.4.1 Droopy 66
5.4.2 Dripple 67
5.4.3 Fault-Tolerance 68
5.5 Protocol Details 68
5.5.1 Droopy 68
5.5.2 Dripple 71
5.5.3 Strict Serializability 73
5.6 Experimental Evaluation 74

5.6.1 Implementation 75
5.6.2 Experiment Setup 75
5.6.3 Imbalanced Workloads 76
5.6.4 Balanced Workloads 80

6 Conclusion **85**

Résumé **87**
1 Introduction 87
1.1 Contexte 87
1.2 Défis 88
1.3 Contributions 90
1.4 Organisation 91
2 Préliminaires 92
2.1 Modèle du Système 92
2.2 Réplication de Machines à États 92
3 XFT: Tolérance aux Pannes Pratique Au-delà de Crashs 94
3.1 Introduction 94
3.2 Contribution 95
3.3 Modèle du Système 96
3.4 Le Modèle XFT 98
3.5 Protocole 99
3.6 Évaluation expérimentale 101
4 Principal Sélection Pour Latence Faible Réplication de Machines à États 103
4.1 Introduction 103
4.2 Présentation du Protocole 105
4.3 Évaluation Expérimentale 108
5 Conclusion 110

Bibliography **113**

A XPaxos Pseudocode **123**
A.1 Common Case 123
A.2 View-Change 127
A.3 Request Retransmission 128
A.4 Fault Detection 128

B XPaxos Formal Proof **131**
B.1 Safety (Consistency) 131
B.2 Liveness (Availability) 136
B.3 Fault Detection (FD) 137

List of Figures

1.1 Message patterns of typical CFT and BFT protocols when $t = 1$. . 3

2.1 Architecture of state machine replication. 8

3.1 Paxos message pattern: process 2 is crash. 16
3.2 PBFT message pattern in common case: process 3 is Byzantine faulty. 20
3.3 PBFT message pattern with view-change: the old primary (process 3) is Byzantine faulty, hence the new primary is selected. 21

4.1 Illustration of partitioned replicas : $\{p_1, p_4, p_5\}$ or $\{p_2, p_3, p_5\}$ are partitioned based on Definition 8. 28
4.2 XPaxos common-case message patterns for $t = 1$ and $t \geq 2$ (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. 35
4.3 XPaxos view change illustration: synchronous group is changed from (s_0,s_1) to (s_0,s_2). 38
4.4 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). 40
4.5 XPaxos examples. 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. . . . 41
4.6 XPaxos checkpointing message pattern : synchronous group is (s_0,s_1). 43
4.7 XPaxos common-case message patterns with lazy replication when $t = 1$ and $t \geq 2$ (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. . . . 43
4.8 Message patterns of the three protocols under test ($t = 1$). 53
4.9 Fault-free performance 55
4.10 CPU usage when running the 1/0 and 4/0 micro-benchmarks. . . . 56
4.11 XPaxos under faults. 56
4.12 Latency vs. throughput for the ZooKeeper application ($t = 1$). . . 57

5.1 Delayed commit problem in state-of-the-art all-leader protocols. . . 63
5.2 Architecture of Droopy and Dripple. 66
5.3 Lease renewal. 66
5.4 Dripple example. 68

5.5 Average round-trip latency (ms) among 6 Amazon EC2 sites used in our experiments: US East (UE), US West (UW), Ireland (IR), Japan (JP), Australia (AU) and Frankfurt (FK). 76
5.6 Latency (in ms, lower is better) of Droopy, Clock-RSM and Paxos under imbalanced workload when $t = 1$. 77
5.7 Latency (in milliseconds) of Droopy, Clock-RSM and Paxos under imbalanced workload when $t = 2$. 78
5.8 Latency distribution (in ms) at UE and AU when $t = 2$. 40 clients in UE and 10 clients in AU issue requests. The leader in Paxos is at UW. The leader set is reconfigured to UE and JP in Droopy. . . 79
5.9 Latency (in milliseconds) of Droopy, Clock-RSM and Paxos under imbalanced workloads when $t = 2$ and $n = 6$. 81
5.10 Average latency (bars) and 95%ile (lines atop bars) of D^{2}Clock-RSM and state-of-the-art protocols under balanced workloads. 82

1 Échange de messages de protocoles CFT et BFT typiques lorsque $t = 1$. 89
2 Illustration de réplicas partitionné: $\{p_1, p_4, p_5\}$ ou $\{p_2, p_3, p_5\}$ sont partitionnés sur la base Definition 8. 97
3 Échange de messages de commune cas de XPaxos quand $t = 1$ et $t \geq 2$. Groupe synchrone illustré sont (s_0,s_1) (quand $t = 1$) and (s_0,s_1,s_2) (quand $t = 2$), respectivement. 100
4 Latence vs débit pour l'application ZooKeeper ($t = 1$). 102
5 Architecture de Droopy et Dripple. 105
6 Renouvellement du bail. 106
7 Example de Dripple. 108
8 les charges de travail localisées. 109

A.1 XPaxos common case: Message fields and local variables. 124

B.1 XPaxos proof notations. 132

List of Tables

1.1 The round-trip latencies and bandwidth (ms/Mbps) between Amazon EC2 data centers measured on April 2014; each machine is deployed on c3.2xlarge instance. 2

4.1 The number of each type of faults tolerated by representative SMR protocols. 29
4.2 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. 31
4.3 Synchronous group combinations ($t = 1$). 37
4.4 $9ofC(CFT_{t=1})$, $9ofC(\mathsf{XPaxos}_{t=1})$ and $9ofC(BFT_{t=1})$ values when $3 \leq 9_{benign} \leq 8$, $2 \leq 9_{synchrony} \leq 6$ and $2 \leq 9_{correct} < 9_{benign}$. 47
4.5 $9ofC(CFT_{t=2})$, $9ofC(\mathsf{XPaxos}_{t=2})$ and $9ofC(BFT_{t=2})$ values when $3 \leq 9_{benign} \leq 8$, $2 \leq 9_{synchrony} \leq 6$ and $2 \leq 9_{correct} < 9_{benign}$. 47
4.6 $9ofA(CFT_{t=1})$, $9ofA(BFT_{t=1})$ and $9ofA(\mathsf{XPaxos}_{t=1})$ values when $2 \leq 9_{available} \leq 6$ and $9_{available} < 9_{benign} \leq 8$. 50
4.7 $9ofA(CFT_{t=2})$, $9ofA(BFT_{t=2})$ and $9ofA(\mathsf{XPaxos}_{t=2})$ values when $2 \leq 9_{available} \leq 6$ and $9_{available} < 9_{benign} \leq 8$. 51
4.8 Configurations of replicas. Greyed replicas are not used in the "common" case. 54

1 Les latences aller-retour et la bande passante (ms/Mbps) dans plateforme Amazon EC2 mesurée sur Avril 2014; chaque machine est déployée sur l'instance c3.2xlarge. 89
2 Le nombre de chaque type de fautes tolérés par les protocoles SMR représentatives. 98
3 Latence aller-retour de TCP ping (*hping3*) dans les centres de données Amazon EC2, recueillies au cours de trois mois. Les latences sont donnés en millisecondes, au format suivant: moyen / 99.99% / 99.999% / maximum. 101

Chapter 1

Introduction

1.1 Context

Modern Internet applications are largely deployed among different data centers (a.k.a., *site*) that span over continents, in order to provide efficient and reliable service across the globe. Such services include, among other things, email [11, 15], social networking [9, 19], file-sharing [6, 10] and online business and payments [4, 1, 7, 18, 2]. The very general model behind is a distributed system that contains a group of processes that communicate and coordinate with each other in order to achieve some common goal [112].

Nowadays, since each site can contain thousands, or even millions of servers with moderate performance, failures become the norm rather than the exception. These failures, along the scale, range from a single machine fault to the misconfiguration or power outage of a whole data center, or even a large-scale disaster [13, 3]. Moreover, the types of failures are diverse, which can be machine crashes, disk or memory corruptions, hardware or software bugs [14, 116], or even organizational and malicious attacks. Most of these failures are recorded publicly that have been encountered by applications and resulted in outage of service [49].

Therefore, tolerating many kinds of service disruptions, i.e., building highly reliable distributed systems, is key for the survival of modern Internet applications. More specifically, modern Internet applications have several requirements : (1) high reliability, which is typically indicated as a percentage of the time the system is expected to be up (a.k.a., available) and running correctly (a.k.a., consistent), despite a broad range of failures; (2) high performance with regard to latency and throughput; and, (3) moderate cost (i.e., resource usage).

In order to tolerate ever sophisticated failures, applications call for synchronous replication approaches [23], which in principle guarantee "zero data loss". Among these approaches, State Machine Replication (SMR) [102] is a seminal method and often used to maintain a critical portion of system metadata in a consistent and available way, so that the resilience of the rest of the system can be based on it. SMR models each process as a deterministic finite

	Oregon	Virginia	Japan	Brazil	Ireland	Australia	Singapore
California	20 / 784	80 / 254	118 / 162	193 / 114	168 / 132	175 / 129	179 / 124
Oregon		75 / 193	114 / 153	199 / 96	155 / 124	216 / 99	208 / 90
Virginia			181 / 112	130 / 143	77 / 232	227 / 82	261 / 86
Japan				351 / 56	268 / 74	128 / 158	83 / 257
Brazil					238 / 89	325 / 51	366 / 56
Ireland						320 / 60	207 / 101
Australia							187 / 106

Table 1.1: The round-trip latencies and bandwidth (ms/Mbps) between Amazon EC2 data centers measured on April 2014; each machine is deployed on c3.2xlarge instance.

automaton and ensures every (correct) process triggers events and makes state-transitions (by executing requests from clients) in the same order. The elegant abstraction of SMR approach makes it widely deployed by many productions [36, 38]. In the following of this thesis, we focus on SMR.

During the past decades, numerous SMR protocols have been proposed, which can be roughly categorised into Crash-Fault Tolerance (CFT), Arbitrary-Fault Tolerance (or Byzantine-Fault Tolerance, BFT) [86], as well as some hybrid models that explore the space in between. CFT model targets immunity of fail-stop failures only, i.e., machines simply stop processing and responding requests. BFT model has no restriction on types of failures and pessimistically presumes that a number of processes as well as the network connections can be controlled by a powerful adversary.

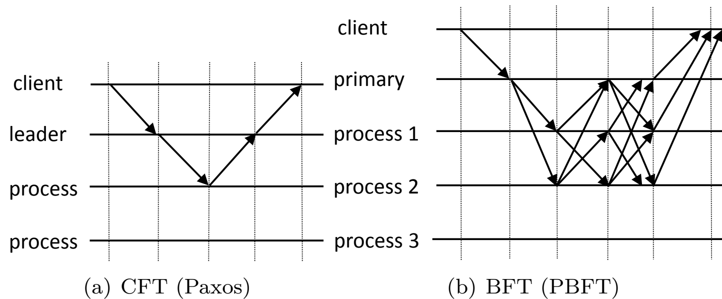
1.2 Challenges

As targeting new context, i.e., deploying reliable service at planetary scale, designers of SMR protocols are facing some new challenges: (1) non-uniform and bounded performance (with regard to latency and bandwidth) of connections among globally distributed data centers, and (2) the trade-off between tolerating ever sophisticated failures and the large cost and complexity introduced (e.g., existing BFT protocols) that badly impact its deployment on geo-scale.

1.2.1 Non-Uniform Connections Across Global Data Centers

Replication becomes challenging at the planetary scale, largely due to the high latency and the low bandwidth of long distance connections. As an illustration, the average round-trip latencies and bandwidth among 8 Amazon EC2 sites are shown in Table 1.1, which are measured by hping [12] and Netperf [16], respectively. It can be observed that, the round-trip latency between remote sites can be as large as several hundred milliseconds (e.g., the one between Japan and Brazil), which practically makes the communication delay within a data center (usually less than 1 ms) neglectable. Classical SMR protocols such as Paxos [80, 81] have a *single leader* that is responsible for sequencing and proposing clients' requests. For clients residing at a remote site with respect to that of the single leader, the much higher latency among different sites implies very costly round-trips across the globe.

Bandwidth can affect the performance as well. As shown in Table 1.1, the bandwidth between two remotely located virtual machines (with moderate



(a) CFT (Paxos) (b) BFT (PBFT)

Figure 1.1: Message patterns of typical CFT and BFT protocols when $t = 1$.

performance) ranges from 50Mb/s to 257Mb/s (except the one connecting California and Oregon). We can directly observe that, (1) the bandwidth among different sites is heterogeneous as well, and (2) the absolute value is much smaller than that of the modern networking devices, such as 1Gb/s or 10Gb/s adapters, switches and routers. For the systems that are designed based on symmetric network (e.g., within a cluster), the throughput of the whole system is probably decided by the worst connection.

Hence, considering the performance limitations, researchers and developers need to rethink the design of SMR protocols in the new environment, in order to deliver their service as fast as possible.

1.2.2 The Trade-Off Between Strong Assumption and Large Cost

Considering a broad range of non-crash faults that have happened recently [49], application developers may have adequate motivation (in near future) to replace de facto standard for SMR, i.e., Paxos [81, 80], by a more reliable candidate. Besides, since each physical machine in public cloud is typically shared by several virtual machines, applications deployed upon are treated naturally as untrustworthy and more prone to suffering from faults beyond crashes.

So far, the well-studied candidates are BFT solutions. However, BFT protocols have been hardly deployed in modern commercial productions (within or among data centers), mainly due to its large cost and complex message patterns comparing to CFT solutions (e.g., BFT requires $3t+1$ replicas to tolerate t machine faults, rather than $2t+1$ in CFT model). BFT protocols further require that at each machine, from its hardware to all related software, every component should be administered and implemented independently, so that the fault occurred at one replica can be isolated from others. However, with large cost, such requirement may not be easily achieved.

To replicate a request, BFT protocols involve more replicas (e.g., at least two-third instead of a majority in CFT), and have more complex message pattern compared to CFT protocols, as shown in Fig. 1.1. Since the capacity of connections in Wide Area Network (WAN) is non-uniform (as shown in Table 1.1), BFT protocols may have much larger latency and lower throughput than its CFT counterparts in geo-replication context, which makes BFT proto-

cols even harder to be deployed. This large cost and complexity are introduced by the very strong assumption that a powerful adversary can control a number of replicas (i.e., at most t replicas) and the entire network at will.

On the one hand, to ease the development and maintenance of modern applications, people badly need a highly reliable replication protocol that can handle many kinds of faults that have occurred in real systems. On the other hand, the more powerful and general the assumption that replication protocol has, the more resource and complexity it requires, which reversely makes the replication protocols harder to be adopted.

1.3 Contributions

The goal of this thesis is to advance state machine replication design in geo-replication context. The contribution includes XFT, a novel approach to building efficient SMR protocols, which treats machine faults and network faults as independent events; and, two sister approaches - Droopy and Dripple, which reduce latency in geo-replicated state machine by dynamically reconfiguring the leader set and by state partitioning, respectively.

1.3.1 XFT : Practical Fault Tolerance Beyond Crashes

The first contribution introduces XFT ("cross fault tolerance"), a novel approach to building reliable and secure distributed systems. Existing BFT protocols presume an extremely bad circumstance, in which a powerful adversary can *simultaneously* control a number of machines as well as the network that connects other correct machines. This strong assumption we believe is the main reason that makes BFT protocols hardly adopted by industry. However, in line with observations by practitioners [77], the Byzantine, non-crash faults experienced today occur independently of network faults (and often also independently of each other). Considering the trade-off between the strong assumption and the large cost introduced in SMR protocols (as discussed in Sec. 1.2.2), XFT allows for both Byzantine machines and network faults, yet considers them independent.

We demonstrate our approach with XPaxos, the first XFT state machine replication protocol. XPaxos makes use of totally $2t + 1$ replicas (instead of $3t + 1$ in BFT model) to tolerate up to t arbitrarily faulty replicas, as long as the sum of machine and network faults does not exceed t. In common case, i.e., there is no machine fault or network fault, XPaxos synchronously replicates requests across specific $t + 1$ replicas, which are organized as a group named *synchronous group*. A synchronous group is led by a distinguished replica called primary. In case these $t + 1$ replicas can not make progress in a timely manner, a novel view-change sub-protocol is executed in order to select a new synchronous group.

Besides common case and view change sub-protocol of XPaxos, we further introduce two important mechanisms: (1) a fault detection mechanism that can detect fatal non-crash faults if there are not enough machine and network faults co-existing; and, (2) a cooperative lazy replication mechanism that explores bandwidth heterogeneity (as we discussed in Sec. 1.2.1), in order to transfer the most recent state efficiently to replicas out of synchronous group.

Our reliability analysis shows that, XPaxos is significantly stronger than CFT protocols regarding consistency and availability, and is always stronger than BFT protocols regarding availability. In some specific scenario, XPaxos can achieve even stronger consistent than existing BFT protocols. By practical evaluation on Amazon EC2 we show that, XPaxos maintains the similar performance as that of Paxos, and significantly outperforms two representative BFT protocols.

1.3.2 Leader Set Selection for Low-Latency Geo-Replicated State Machine

The second contribution targets improvement of perceived latency in geo-replication protocols. Existing geo-replication protocols can be classified into two categories: *all-leader* protocols and *leaderless* protocols. In both methods, every replica (or even client) can act as a leader and propose requests. All-leader protocols are based on a conservative assumption that all requests conflict with each other, hence a total order (e.g., by giving each request a logical sequence number) among requests is required. Whereas, leaderless protocols are based on an aggressive assumption that most requests are commutative and can be executed out of order, hence they order requests in a decentralized way. However, neither of above methods fits all circumstances. All-leader protocols suffer from the delayed commit problem[1], which is introduced by the requirement of coordination among all replicas (in order to maintain a global order), even if sometimes this coordination is unnecessary. In leaderless protocols, if concurrent requests proposed by different replicas are non-commutative, another one or more round-trip message exchanges are required to resolve the conflict, which dramatically increases the perceived latency.

Ideally, those conflicting requests should be ordered "in advance" so as to avoid paying too much for dynamic conflict resolution (in leaderless protocols); and, those commutative requests should be ordered independently, so as to mitigate the delayed commit problem (in all-leader protocols).

On the basis of the above considerations, we propose two sister approaches: Droopy and Dripple. Droopy explores the space between single-leader (e.g., Paxos) and all-leader protocols by dynamically re-configuring the leader set. Each set of leaders is selected based on previous workload and network condition. Dripple in turn divides the state into partitions and coordinates them wisely, in order to decouple unnecessary dependencies among requests that access distinct partitions. With Droopy and Dripple enabled, the set of leaders of each partition can be maintained and re-configured separately.

Our evaluation on Amazon EC2 shows that, under typical imbalanced or localized workloads, Droopy and Dripple enabled protocol efficiently reduces latency compared to their native protocol, and they maintain the similar latency as a state-of-the-art leaderless protocol. In contrast, under balanced and non-commutative workloads, Droopy and Dripple enabled protocol maintains the similar performance as their native protocol, and they both outperform a state-of-the-art leaderless protocol.

[1] That is, the far and lagged replica may slow down the whole system.

1.4 Organization

This thesis is organized as follows. In Chapter 2, we introduce the basic model and concepts that we use in the thesis. In Chapter 3 we discuss some related work regarding distributed storage systems and fault-tolerance protocols. Chapter 4 presents our first contribution XFT, which contains the description of XFT model, the detail of our XPaxos protocol, its reliability analysis and experimental evaluation. In Chapter 5 we describe our second contribution — Droopy and Dripple, and their evaluation on Amazon EC2. Finally, in Chapter 6 we conclude the thesis and discuss some future work. A summary in French is given after. We postpone the complete pseudocode of XPaxos and its formal proof in Appdendix A and B, respectively.

Chapter 2

Preliminaries

2.1 System Model

We first introduce some general notions and assumptions of distributed systems that we use in the thesis.

Machines. We consider a message-passing distributed system containing a set *replicas* of $n = |replicas|$ *machines*, also called *processes* or *replicas*. Among which, we assume at most t machines can be faulty. A network $\mathcal{N}$ connects the replicas, where each pair of replicas can send messages reliably over a point-to-point bi-directional communication channel. Additionally, there is a separate set C of clients, where each client can communicate with any replica and export operations by making requests.

Fault model. We assume any machine may be crash faulty or non-crash (Byzantine) faulty. We distinguish between *crash* faults, where a machine simply stops any computation, and *non-crash* faults, where a machine acts in an arbitrary way so long as it does not break cryptographic primitives, a.k.a., we assume an authenticated Byzantine-fault model. A machine that is not faulty is called *correct*. We say that a replica is *benign* whenever it is correct or crash faulty.

Network. We target an *asynchronous* distributed system in the thesis. The system is 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 precisely define *network fault* as the inability of *correct* replicas to communicate with each other in a timely manner, that is, when a message exchanged between *any* two correct replicas cannot be delivered and processed within delay Δ, known to all replicas.

2.2 State Machine Replication

In this thesis we specifically focus on the State Machine Replication (SMR) problem [102]. In short, SMR ensures *linearizability* [66], or *consistency*, of a replicated and deterministic service, by enforcing a total order across client's

requests using an unbounded number of *consensus* instances, returning to the client an application-level reply. We say that the SMR protocol is *available* if a correct client can eventually deliver its requests.

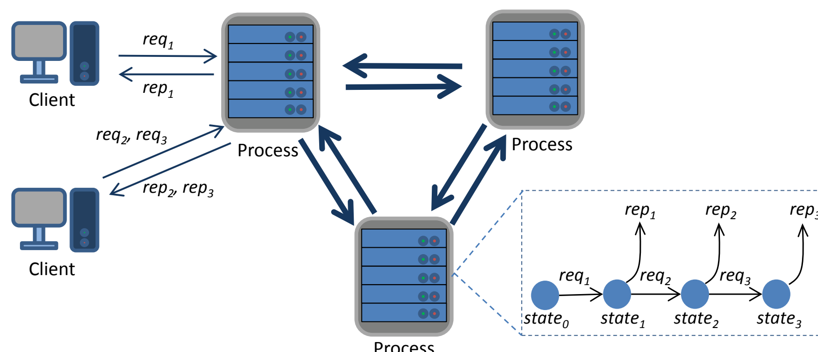


Figure 2.1: Architecture of state machine replication.

A typical SMR architecture is shown in Fig. 2.1. In general, SMR protocol mimics a centralized server that will never fail or behave unfaithfully. Each process or replica $p_i \in replicas$ is modelled as a finite-state automata (i.e., state machine), which maintains a current state $state_i \in \mathcal{S}$. Every request $req_i \in \mathcal{R}$ issued by clients is modelled as an input of state machine and every reply $rep_i \in \mathcal{O}$ is modelled as an output of state machine. That is, there is a deterministic output generation function ω, which takes a specific state $state_i \in \mathcal{S}$ and a request $req_i \in \mathcal{R}$ as its parameters,, i.e., $\omega : \mathcal{S} \times \mathcal{R} \rightarrow \mathcal{O}$. Moreover, there is a deterministic state transition function δ which takes the same parameters, i.e., $\delta : \mathcal{S} \times \mathcal{R} \rightarrow \mathcal{S}$.

For example, if upper application is a key-value store, then each state contains all key-value pairs, and each request can be PUT or GET operation. PUT operation modifies the current state and outputs a simple commitment (or replies nothing). GET operation does not change the state but should return the value required.

To achieve this abstraction, every process starts with the same initial state (i.e., $state_0$) and executes the same set of deterministic requests. It is important to guarantee that, *if there is no request issued further, then every (correct) process should end up with the same state after executing all requests.*

Logically, there are two components in SMR protocols: (1) a consensus algorithm which ensures reliability for each single request despite failures; and, (2) an ordering component which provides a global order among all consensus instances (i.e., requests).

For example, in classical Paxos [80, 81], consensus component ensures that at a given instance, committed request should not be modified or rolled backed despite machine crashes (but it can not survive if some replica exhibits non-crash fault). Ordering component in Paxos gives each consensus instance a logical sequence number, by which requests can be totally ordered.

Consensus. The purpose of consensus is reaching agreement (i.e., making consistent decision) on a common value by a group of processes or *replicas*.

In SMR protocols, making agreement is to executing the same request at a given consensus instance. Roughly speaking, a consensus algorithm ensures that every correct process will decide the same value, which must be proposed by some process. More specifically, consensus algorithm guarantees [37] :

- (*validity*) if a value is decided by a process, then that value should be proposed by some process before;
- (*termination*) every correct process decides a value eventually;
- (*agreement*) if value v is decided by process p, and value v' is decided by process p', then $v = v'$; and,
- (*integrity*) no process decides twice.

However, considering arbitrary (Byzantine) fault model, we need to modify the definitions of validity and agreement, since we cannot restrict arbitrarily faulty processes to follow the protocol faithfully.

- (*BFT validity*) if a value is decided by a *benign* process, then that value has been proposed by some process before;
- (*BFT agreement*) if value v is decided by *benign* process p, and value v' is decided by *benign* process p', then $v = v'$.

To satisfy the above properties in an asynchronous environment, in which network and request processing delay are not bounded, CFT model requires $n = 2t + 1$ processes in order to tolerate t crash faults. Whereas, BFT model requires $n = 3t + 1$ processes in order to tolerate t non-crash faults.

Ordering. In SMR protocol, upon a request is decided at a consensus instance, this request will be executed eventually based on an order defined by ordering component. For example, in Paxos [80, 81], committed requests are executed sequentially based on logical sequence numbers. Whereas, some protocols, such as the ones in [102, 55], have requests ordered by physical clocks. Furthermore, some leaderless protocols execute commutative requests (e.g., two requests accessing distinct state) out of order, hence no total order is pre-defined in this case. Instead, serializability [31] is typically ensured.

Our first contribution (i.e., XFT) targets improvement of the consensus component. Whereas, the second contribution (i.e., Droopy and Dripple) is tailored for optimization of the ordering component.

Chapter 3

Related work

In this chapter we discuss previous work relevant to our contributions, specifically in the context of geographically distributed storage systems and their design choices (in Sec. 3.1), along with fault-tolerance protocols (in Sec. 3.2).

3.1 Geographically Distributed Storage Systems

In order to cater for geographically dispersed clients, as well as provide reliable service despite large scale failures, modern distributed storage systems are widely deployed across distant geographical locations, or, *sites*. Coordination among sites is necessary and challenging, be it synchronously or asynchronously.

There is a fundamental trade-off between consistency and availability (or, between consistency and low-latency), which is exhibited by CAP theorem [60]. In short, CAP theorem states that, a distributed system can not ensure both strong consistency (i.e., one-copy semantics) and availability (i.e., responsiveness), with existence of network partitions. Intuitively, an operation should be acknowledged by some coordinator, in order to confirm a global, one copy order. In case the coordinator is in a remotely located site, the coordination may not be achieved (in a timely manner) as long as network partitions exist.

Hence, either a system gives up strong consistency by providing available service even with network partitions; or, the system sacrifices availability and insist of strong consistency.

3.1.1 Weakly Consistent Systems

On the one hand, numerous systems, such as COPS [90], Eiger [91], Dynamo [51], Bayou [108], Coda [72] and some configurations in Cassandra [79], are tailored for providing high availability and minimizing latency. Their replication message pattern is typically simplified to as few as one round-trip of message exchange between clients and a nearby process or site. After that, committed operations can be lazily propagated to remote sites. Hence, network partitions

or failures of remote sites should not affect performance or availability of local operations. Lazy propagation however, does not maintain one-copy semantics and leads to relaxed consistency models such as causal consistency [62] or eventual consistency [35].

For instance, COPS [90] is a key-value store that delivers causal+ consistency, a consistency level that maintains causality across sites, plus convergent conflict handling, i.e., updates to the same key expose versions in order and no permanent divergence. Read and write operations in COPS can always be committed locally by any site, then lazily propagated to other sites with all causally dependent operations piggybacked with. Other sites, upon receiving committed operations, can commit these operations locally only if their dependencies have been committed already. Eiger [91] is designed based on COPS, which provides casual+ consistency among distant sites as well. Apart from that, Eiger enriches data model in COPS by supporting complex column-family data model, and adds semantics of write-only transactions. Dynamo [51] is a key-value store developed by Amazon, which targets service that is always on. Hence, Dynamo prioritizes high availability rather than strong consistency. Namely, Dynamo ensures that eventually each object is converged at every replica (i.e., eventual consistency).

3.1.2 Strongly Consistent Systems

On the other hand, a lot of systems, such as Spanner [47], MDCC [74], Megastore [28] and Walter [105], insist of stronger consistency despite performance and availability penalties that can be introduced. This is largely because strong consistency gives application developers an illusion of a sequential centralized service, that the developers are accustomed to.

Linearizabiliy and Serializability. At the heart of many strongly consistent systems directly lie crash-fault tolerant state machine replication (SMR) protocols, many of which are variants of Paxos [80, 81] (we discuss how Paxos works in Sec. 3.2.1), which ensures linearizability [66] regarding a single state. Among them, Spanner [47] is a globally distributed database system developed by Google. Spanner uses Paxos to synchronously replicate transactional operations (e.g., to prepare or commit a transaction) at each data set called *tablet*, and across different zones (i.e., locations within or across data centers). For transactions accessing more than one Paxos groups, Spanner makes use of two-phase commit [31] to guarantee serializability [31]. Apart from that, Spanner massively uses hardware-based clock synchronization to assign timestamp for transactions and maintain a leader lease. This is in order to enhance performance without sacrificing strong consistency. Megastore [28] is a globally consistent transactional system developed prior to Spanner, which also relies on Paxos to synchronously replicate user data across different sites. In order to improve performance of read-only operations, Megastore grands read lease to all replicas. MDCC [74] is another transactional system which provides strong consistency ranging from read-committed to serializability. To efficiently achieve that, MDCC makes use of generalized Paxos [82], a variant and optimization of celebrated Paxos, to implement an optimistic concurrency control and replicate operations.

To break the trade-off between strong consistency (i.e., serializability in transactional semantics) and high availability, Lynx [118] leverages a priori static analysis and a transaction decomposition mechanism to achieve both low latency and serializability. Lynx decomposes a transaction into a sequence of hops (or, a sequence of individual sub-transaction) called transaction chain, so that each hop can be executed at one server. Prior to running application, Lynx statically analyses if a transaction chain can be executed piecewise, while at the same time preserving serializability. If so, Lynx can reply to applications directly after the execution at the first hop.

Snapshot Isolation and its variants. Many transactional systems implement variants of classical Snapshot Isolation (SI) [29]. SI is an isolation level provided by several commercial database systems such as Oracle. Roughly speaking, SI ensures that, (1) there exists a global timeline (unknown to processes or sites), and the start and commit time of every transaction is totally ordered at this timeline; (2) transactions read from a consistent snapshot, which is obtained at the start time; and (3) if two concurrent transactions[1] write the same object, then one of them must be aborted. However, SI is not preferable for geo-replicated systems since it usually relies on a total order primitive to form a monotonic sequence of snapshots, updated at the commit time of each transaction. Hence, those systems that implement SI prefer strong consistency instead of high availability.

Clock-SI [54] is a globally distributed data store that implements classical SI. In a novel manner, Clock-SI makes use of loose synchronized clocks to obtain timestamps locally rather than by a centralized way, and maintains the global order of start and commit events based on physical time as well.

Walter [105] is a transactional storage system that implements a novel consistency named Parallel Snapshot Isolation (PSI). Rather than providing sequential snapshots required by SI, PSI maintains an individual snapshot at each process or site. Causal consistency are maintained across different sites, while at the same time PSI ensures that only one of two concurrent transactions[2] can be committed if they accessed the same object. To implement PSI, Walter divides the key space into partitions, so that each partition is managed by a certain site in order to resolve conflicts. A transaction can start locally at any site by obtaining the site's local snapshot, commit locally by successfully obtaining locks from the sites that are in charge of the objects that the transaction wrote (otherwise abort), and be propagated to remote sites asynchronously.

Jessy [24] is another transactional system which implements a consistency level called Non-Monotonic Snapshot Isolation (NMSI). NMSI further improves scalability property based on PSI. Although Walter and Jessy ensure only causal consistency (across sites in Walter and across objects in Jessy), they still need to resolve write conflicts during commit time of each transaction, e.g., by coordination with a certain site. Hence, they prefer strong consistency rather than high availability.

[1] In SI, two transactions are concurrent if start time of one transaction is between start time and commit time of the other transaction.

[2] In PSI, two transactions are concurrent if no transaction causally precedes the other.

Exploring commutativity. Some systems, such as Walter, Gemini [88] MDCC and Lynx, further explore the commutativity [103] of concurrent operations, i.e., if two operations or transactions can be executed out of order, even if they access some common object. Gemini [88] implements RedBlue consistency, which categorizes operations into two types: red operations and blue operations. Red operations are strongly consistent but have to be committed with coordination of remote sites. Whereas, blue operations are eventually consistent and can be committed locally by any site. Gemini further decomposes an operation into a generator operation and a shadow operation. A generator operation has no side-effect hence can be executed locally. A shadow operation is further marked as red or blue, and executed accordingly.

Tunable consistency. Some systems, such as Cassandra [79], SPANStore [115], Pileus [107] and Tuba [25], select a consistency level for each data set or operation based on the requirement provided by applications. For example, Cassandra is a scalable database system that provides a tunable consistency, ranging from writing to any replica (weakest consistency but highest availability) to writing to all replicas (strongest consistency but weakest availability). SPANStore is a key-value store that spans over several geographically located data centers. Rather than improving performance, SPANStore targets the cost reduction for latency-sensitive applications, at the same time satisfying responsiveness (i.e., low latency) and consistency (strong or eventual) goals required by applications. The cause of cost in SPANStore contains storage service, data transmission as well as computing resources. Pileus is a geo-replicated key-value store which dynamically selects a consistency level for each read operation, based on service level agreements (SLA). Tuba is designed based on Pileus (hence Tuba supports dynamic consistency as well). Besides, Tuba implements a configuration service which selects the set of replicas dynamically and periodically based on application's consistency and performance requirements.

3.1.3 Fault-Tolerance Considerations

Notice that, making a system strong consistent is orthogonal (more or less) to ensuring the system survived under faults, although most fault-tolerance protocols are strong consistent, i.e., providing one-copy semantics. For instance, an eventual consistent protocol, e.g., Zeno [104], can have its write operations committed out of order at different processes, but only if every operation is replicated by enough processes before committing, the system is reliable despite failures. Whereas, a strong consistent system can have an ideal (and maybe unrealistic) assumption that no process will crash or behave unfaithfully. With this assumption, no replication is needed. But even in this case, a strong consistent system (e.g., two-phase commit) has to rely on some coordination, e.g., by a distinguished coordinator, to maintain a global order.

Considering large scale failures, e.g., site failures due to power outage or natural disaster, data replicated by a single site could be lost. For example, COPS will lose changes if the data has not been lazily replicated by enough sites. Windows Azure Storage (WAS) [38] is a cloud storage system developed by Microsoft. WAS has two types of replication engine: (1) synchronous replication across different fault domains within a cluster; and (2) asynchronous

replication from primary cluster to secondary clusters. In the event of site failures, recent changes that have not been replicated by secondary clusters might be lost as well. Whereas, some other systems discussed above, such as Walter and Gemini, rely on a fault-tolerance protocol, e.g., Paxos, to synchronously replicate data or operations across different sites. However, in this case, the benefit for latency and availability improvement is sacrificed more or less.[3]

Besides protocols discussed above, some systems target providing even more robust service, such as protocols that tolerate Byzantine faults. Depsky [32] is a cloud storage service tailored for providing secure and durable service for critical applications (e.g., medical record database). Depsky leverages several techniques, including data encryption, Byzantine quorum protocols deployed across distinct cloud providers, and secret sharing scheme in order to ensure durability and confidentiality. Depot [92] is a cloud key-value store which can tolerate any number of non-crash faulty clients and replicas (i.e., not bounded by $\lfloor \frac{n-1}{3} \rfloor$), as it ensures a consistency level named Fork-Join-Causal consistency (FJC), which is a bit weaker than traditional causal consistency. Byzantium [59] is a BFT database replication middleware which makes use of off-the-shelf database system and BFT library (such as PBFT [40], we discuss how PBFT works in Sec. 3.2.2) to ensure Snapshot Isolation. Augustus [99] targets strong consistent and *scalable* storage system. With this in mind, Augustus splits the storage entities into partitions, while each partition is managed by a BFT instance. Singe-partition and read-only transactions are specially optimized so that committing these transactions involve one message commitment in BFT protocol. However, committing multi-partition read-write transactions involve additional one round-trip of message exchange between clients and every involved BFT instance.

To summarize, existing distributed storage systems either (1) relax consistency so as to improve performance and availability, or (2) insist of strong consistency in order to ease the development of applications. In either case, they need to take advantage of fault-tolerance protocols, such as SMR protocols, to provide a reliable service despite many kinds of faults.

3.2 Fault-Tolerance Protocols

In this section we introduce fault-tolerance protocols by classifying them into three categories as discussed in Sec.1.1 : (1) protocols that tolerate crash faults only, i.e., CFT; (2) protocols that tolerate arbitrary (Byzantine) faults, i.e., BFT; and, (3) protocols that target the fault model in between.

3.2.1 Crash-Fault Tolerance (CFT)

Paxos. In CFT model, Paxos is the most well-known SMR protocol which has been implemented and deployed in a number of commercial products. In general, Paxos replicates state and requests at a group of $2t + 1$ processes, among which t processes can crash. Paxos gives each request a logical *sequence number*, based on which requests are executed sequentially at each process. In

[3]That is, although clients of a weakly consistent system can have their operations committed and executed locally, these operations have to be replicated by enough sites before committing so as to provide reliability.

order to guarantee that at any given sequence number, every (correct) process agrees on the same request, there is a Consensus primitive named Synod [80] located at the heart of Paxos. Each sequence number is managed by one consensus instance.

Generally, there are three process' roles in Synod: *proposers*, *acceptors* and *learners*. Proposers are responsible for *proposing* values. After that, each acceptor chooses a value proposed and notifies all learners. Upon a learner confirms that a value is chosen by a majority of acceptors, that value is *decided*. Any majority of processes in Paxos is called a *quorum*. In practical circumstances, a process can behave as several roles (i.e., proposer, acceptor or learner). In *common case*, i.e., there is no crash fault or network asynchrony, a dedicated process is elected as a *leader*, which acts as a distinguished proposer.

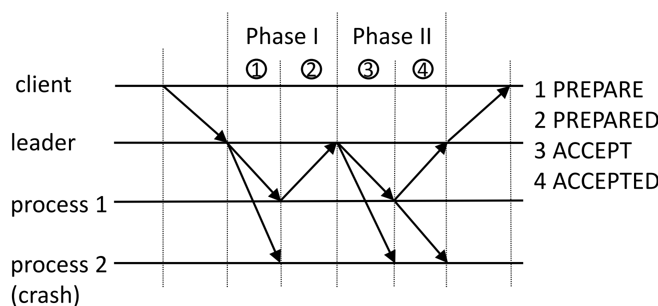


Figure 3.1: Paxos message pattern: process 2 is crash.

The message pattern of Paxos for one Synod instance is shown in Fig. 3.1. There are explicitly two phases: the fist phase is in order to learn any proposed and possibly decided (or will be decided) request at a given consensus instance (i.e., sequence number); the second phase is in order to propose a (new) request at the given instance.

1. Upon receiving a request from client, the leader first chooses an sequence number for the request and broadcasts a PREPARE message to all processes. Note that every message in Paxos is piggybacked with a sequence number and a *ballot number*, which uniquely indicates the leader.

   Besides, there is a critical rule in Paxos: each process only accepts and responds the messages piggybacked with the *highest* ballot number it has received so far. This rule ensures that some slow or recovered process will not change a decided value by proposing another one.

2. Upon a process receives a PREPARE message, the process responds the PREPARE message by sending back a PREPARED message, which contains the request that has the highest ballot number the process has accepted so far (see the next two steps for "accepted").

3. Upon the leader receives PREPARED messages from a majority of processes (including itself), the leader selects a piggybacked request which has the highest ballot number, then proposes the selected request by sending an ACCEPT message to all processes. From this step the second phase starts. If there is no request attached with any PREPARED message, the

leader can propose a new request it has received from clients at the given sequence number.

4. Upon a process receives an ACCEPT message (for sure, with the highest ballot number), the process accepts the requests attached and responds the ACCEPT message by sending an ACCEPTED message to all processes.

5. Upon any process receives ACCEPTED messages from a majority of processes with the same request, sequence number and ballot number, the process decides the request at the given sequence number. All decided requests are executed sequentially based on sequence numbers. After that, the executed request can be replied to its client by the leader or any replica.

In practical circumstances, one can expect a stable leader which proposes requests at a number of instances in parallel. Other processes monitor the behavior of the leader and try to become the new leader by updating the ballot number if, e.g., the current leader can not make progress in a timely manner. A new leader can be elected eventually by leader election module, as soon as the network is synchronous (i.e., eventual synchrony) [58]. Hence, Paxos requires eventual synchrony to guarantee availability (or liveness) but ensures strong consistency (or safety) anyway (as long as no process is non-crash faulty).

Viewstamped Replication. Viewstamped Replication (VR) [97, 89] is a primary-backup replication protocol that actually describes the similar idea as that of Paxos. The first publication of VR protocol [97] is even prior to Paxos. In general, VR makes progress through a sequence of *views*, which can be understand as ballot numbers in Paxos. Given a view number, there is a dedicated replica named *primary* (i.e., the leader in Paxos), which is responsible for sequencing requests issued by clients. Other processes are *backups*. Backups are monitoring status of the primary. If the primary is suspiciously crash or partitioned, a new primary is selected by view-change sub-protocol. The general idea and message pattern of common case and view-change sub-protocol is nearly the same as Phase II and Phase I of Paxos, respectively.

Raft. More recently, Raft [98] is proposed for the purpose of making consensus algorithm easily understandable and straightforward to implement. Raft in principle is very similar to VR and more or less, Paxos, but describes the problem and solutions in a modular way.

Other variants. Besides, numerous variants of Paxos have been proposed recently, e.g., ones in [85, 83, 82]. Among them, Cheap Paxos [85] requires only $t+1$ main processes up and running in common case. Other t auxiliary processes are idle until some main process fails or be partitioned. In order to reduce message exchanges in common case, Fast Paxos [83] allows clients directly proposing their requests to processes. However, if a collision is detected by acceptors, that means, there are two or more requests proposed concurrently and no request can be chosen by enough acceptors, then Fast Paxos relies on classical Paxos and another one or more rounds of message exchange to resolve the collision.

Based on Fast Paxos, Generalized Paxos [82] further explores the commutativity of concurrent requests. Concurrent requests can be executed out of order at different processes if they access disjoint parts of state. For both Fast Paxos and Generalized Paxos, as they de-centralize function of the distinguished leader, they both require a quorum larger than a majority (a.k.a, *fast-quorum*). A fast-quorum contains any $\lceil \frac{3}{2}t \rceil + 1$ processes if there are $2t+1$ processes in total.

Chubby [36] is a lock service developed based on Paxos and used for storing configuration information or name service. ZooKeeper [67] is a coordination service among a group of processes and provides an API similar to file systems. At the heart of ZooKeeper locates a total order primitive named Zab [69]. Chandra et al. [42] discuss, from the engineering perspective, how to build a reliable database system by using Paxos.

Geo-replication. In order to tolerate site outages or large-scale disasters, as well as improve read efficiency for pervasive clients, several SMR protocols have been proposed recently [93, 55, 95, 96] tailored for geo-replication. They can be further classified into two categories: all-leader protocols and leaderless protocols.

In all-leader protocols, every replica can act as a leader and propose requests. Coordination among all processes is required in order to provide one-copy semantics as claimed by SMR protocols.

Mencius [93] is designed base on classical Paxos. Mencius facilitates multiple leaders by evenly pre-partitioning sequence number space across *all* replicas (modulo number of replicas), so that each replica can propose requests (e.g., with 3 replicas, replica 1 sequences requests 1,4,7..., replica 2 sequences requests 2,5,8,..., and replica 3 sequences requests 3,6,9,...). To avoid execution delays, if a replica lags behind it may skip some sequence numbers by assigning empty *no-op* requests to skipped sequence numbers, in order to catch up with other replicas. A skipping replica must however let its peers know which sequence numbers it skips.

Clock-RSM [55] replaces pre-divided sequence numbers by loosely synchronized clocks. In Clock-RSM, each replica proposes requests piggybacked with its physical clock timestamp, which is used to order requests. Replicas synchronize their physical clocks periodically (e.g., every 5 ms in the implementation). With the assumption of FIFO channels, Clock-RSM ensures that requests received and executed at each replica are following the orders of physical clocks.

Alternatively, leaderless protocols presume that there are few conflicts among concurrent requests and sequence requests in a somewhat more decentralized and less ordered fashion. Namely, in leaderless protocols, each request is directly proposed by the client or by any replica. A proposed request is first sent to all replicas. Upon receiving a request, each replica individually and tentatively assigns a sequence number to the request, possibly adding the information about other concurrent requests, and replies to the sender. In case a *fast-quorum* of replicas have given a request the same sequence number, the request is committed. Otherwise, another one or more round-trip delays are introduced in order to arbitrate conflicts. For example, Fast Paxos and Generalized Paxos are cases of leaderless protocols.

A step further, EPaxos [95] reduces the number of replicas in the fast-quorum by exactly one replica compared to Generalized Paxos. Notice that,

EPaxos can achieve this since it introduces one more communication step (i.e., the third hop) from the client to a nearby replica, say command leader, at the first step (hence the $\lceil \frac{3}{2}t \rceil + 1$ fast-quorum lower bound does not apply in this case). The command leader then binds a sequence number (which is not totally ordered among replicas) and a set of conflicting (non-commutative) requests known by the command leader to the request. Upon receiving a proposal from any command leader, each replica updates the conflict set of the received request and re-calculates the sequence number based on the new dependency information, then replies to the command leader. Upon the command leader receives the same sequence number and conflict set from a fast quorum, the request is committed and can be executed. Otherwise, the command leader relies on another round-trip communication among a majority to confirm the order eventually.

In order to balance the performance of read and write operations in geo-replicated SMR, quorum leases [96] is proposed which explores the space between master lease (i.e., leader lease) [42] and all-node lease [28]. For each object, read lease is dynamically grant to replicas (from one to all) that have most frequently served the read operations, while at the same time considering the trade-off of write performance.

3.2.2 Byzantine-Fault Tolerance (BFT)

As targeting fault tolerance of arbitrary failures, applications call for more robust replication protocols that have stronger assumption on process' behavior.

The concept of Byzantine fault is initially proposed by Lamport et al. [86]. Since then, numerous BFT protocols have been proposed over the past decades. In BFT model, each process can exhibit arbitrary behaviors (may except subverting cryptographic techniques), including malicious and organizational attacks. In order to tolerate t Byzantine faults, an asynchronous BFT protocol requires a total of $3t + 1$ processes. Note that this number is the lower bound for standard BFT model with no further assumption (e.g., additionally assume trusted hardware, as we discuss below).

PBFT. Practical Byzantine Fault Tolerance (PBFT) [40] is one of the very first asynchronous protocols that make BFT protocols feasible to implement and deploy. PBFT can be roughly understand as a Byzantine version Viewstamped Replication protocol (also as they are both proposed by Liskov et al.). Similarly to Viewstamped Replication, PBFT has a dedicated process (a.k.a, *replica* in PBFT) named primary, which is responsible for sequencing requests. Every message in PBFT is also piggybacked with a view number which indicates the primary. However, the main difference is, BFT model assumes that some replicas (especially the primary) can be Byzantine faulty, and most dangerously, can equivocate to other replicas.

We first briefly describe how PBFT proceeds in common case, i.e, there is no machine fault or network partition. The message pattern is shown in Fig. 3.2.

1. Upon the primary receives a request issued and attested by an authorized client, the primary first gives it a sequence number, then proposes the request by sending a PRE-PREPARE message to other replicas.

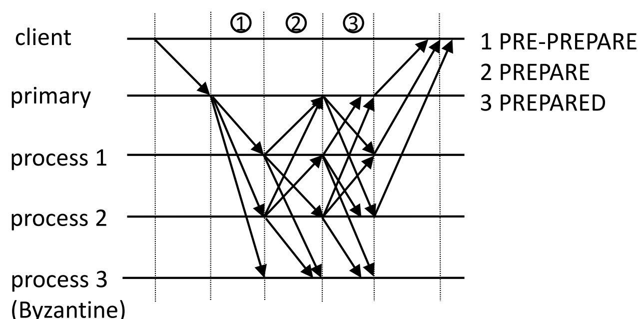


Figure 3.2: PBFT message pattern in common case: process 3 is Byzantine faulty.

For simplicity reason, we assume that every message in PBFT is digitally signed by the original sender[4] and can be correctly verified by other replicas.

2. Upon receiving PRE-PREPARE message from the primary, each replica accepts the request proposed and sends a PREPARE to all replicas.

3. Upon a replica receives $2t+1$ matching PREPARE messages (i.e., with the same request, sequence number and view number), the replica confirms that the request is accepted by $2t+1$ replicas. Then, the replica sends a PREPARED message to all.

   At this moment, the replica has the proof (i.e., digitally signed messages) that $2t+1$ replicas have accepted the request at a given sequence number. This proof serves to provide the evidence for committed request when selecting a new primary.

4. Upon receiving $2t+1$ matching PREPARED messages, each replica knows that $2t+1$ replicas have the proof. Then, the replica can execute the request based on sequence numbers, and replies to the client.

5. Upon the client receives $t+1$ matching replies, it confirms that the request is executed correctly and delivers the reply to application.

PBFT can reduce one message exchange by asking each replica tentatively executing requests and replying to client, upon the replica receives $2t+1$ matching PREPARE messages (i.e., after step 3). In this case, only upon a client receives matching tentative replies from $2t+1$ replicas, the client can accept the reply.

If the primary is possibly faulty, that is, a client can not deliver its request in a timely manner, the client transmits its request to all other replicas (as shown in Fig. 3.3), by which the request is propagated to the primary. If $2t+1$ replicas notice that the request can not be committed locally in a timely manner, they will trigger the view-change sub-protocol, which serves to select a new primary. In view change, the new primary collects the most recent state

[4] In common case, PBFT can be further optimized by replacing asymmetric cryptography with symmetric-key algorithms, such as MAC [39].

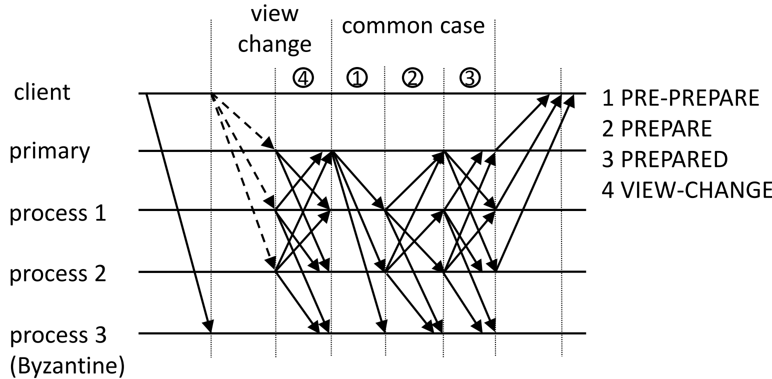


Figure 3.3: PBFT message pattern with view-change: the old primary (process 3) is Byzantine faulty, hence the new primary is selected.

and its proof from enough replicas (i.e., at least $2t + 1$) by transferring VIEW-CHANGE messages. Then the new primary re-proposes these collected requests in the new view as in common case. Note that a Byzantine new primary cannot ignore or modify some committed request on purpose, since the new primary has to forward the signed VIEW-CHANGE messages it has collected to other replicas, which should contain all committed requests.

PBFT and VR (or Paxos) have the very similar ideas at proposing requests (i.e., common case in VR, or Phase II in Paxos) and replacing faulty primary (i.e., view-change in VR, or Phase I in Paxos). However, as they target different fault model, there are several differences in PBFT message pattern compared to VR (or, Paxos):

- every message is attested by the original sender (e.g., with digital signature);
- there are a total of $3t + 1$ replicas and the size of quorum to commit a request is $2t + 1$;
- there is one more message exchange (i.e., PREPARED message) in common case; this message exchange is for the purpose of confirming that $2t + 1$ replicas (among which at least $t+1$ are correct) have the proof that $2t+1$ replicas have accepted the request at a given sequence number; this proof acts as an important evidence during view-change;
- clients expect $t + 1$ ($2t + 1$ in tentative case) matching replies;
- only $2t + 1$ replicas together can trigger view-change sub-protocol.

Zyzzyva. In order to optimize the three-phase commit in PBFT, Zyzzyva [73] leverages speculation mechanism to execute requests tentatively. Similarly to PBFT, there is a replica designated as the primary in Zyzzyva, whereas others replicas are backups. The message pattern in common case is simplified as : (1) a request is issued by client and sent to the primary; (2) the primary broadcasts an ORDER-REQ message, which contains the request, a sequence number, a view

number and the digest of *request history* to all replicas; and, (3) all replicas execute the request tentatively and reply to the client (additionally with the digest of request history as well). If every replica executes the same request based on *the same history*, then the client accepts the matching replies from $3t+1$ replicas. Otherwise, if any replica except the primary is faulty, Zyzzyva instead degrades to slow commit mode, which contains another round-trip of message exchange from the client to $2t+1$ replicas. Finally, if the primary is faulty, $2t+1$ replicas will trigger the view-change sub-protocol in order to select a new primary. The basic idea of view-change in Zyzzyva is similar to that of PBFT. Apart from that, if the new primary confirms that a request has been speculatively executed by $t+1$ replicas, the new primary re-proposes the request in new view as well.

Bft-SMaRt. Bft-SMaRt [33, 5] is a modular BFT SMR library implemented in Java and well maintained since 2007. Besides following the ideas of existing BFT protocols such as PBFT, Bft-SMaRt has implemented reconfiguration and state transfer mechanisms in order to remove faulty replicas and integrate recovered replicas.

Asymmetric cryptography. As an important optimization, during the last fifteen years, BFT researchers have successfully replaced the asymmetric cryptography (e.g., public-key cryptography) by symmetric one (e.g., message authentication code) in common case sub-protocol [40, 39, 114, 104][5]. Asymmetric cryptography is considered as too expensive for CPU cycles, hence in principle it should be avoided as much as possible. Contradictably, Clement et al. [46] argue that in order to maintain a sustainable performance under realistic Byzantine faults, designers should not easily give up asymmetric cryptography. Otherwise, the performance can degrade dramatically under some well-planned attacks.

Trusted hardware. To reduce the number of processes totally required (i.e., $3t+1$ for BFT), as well as simplify the message pattern, some BFT protocols [44, 87, 101, 70] additionally make use of trusted component, so that faulty processes can not equivocate (i.e., sending inconsistent messages to different processes or clients) or ignore committed requests on purpose. A2M [44] is designed as a trusted and append-only log. By enabling A2M, the number of processes required by PBFT is reduced to $2t+1$. Even simpler, Trinc [87] is designed as a trusted and non-decreasing counter. By using Trinc, for example, the Byzantine primary in PBFT can not propose different requests at the same sequence number, and no Byzantine process can provide inconsistent information in view change. With this in mind, a total of $2t+1$ processes are sufficient to ensure strong consistency. CheapBFT [70] makes use of a FPGA-based trusted component to implement a non-decreasing counter. Different from previous protocols, Cheap-BFT requires only $t+1$ active processes in common case for both agreement and execution stages. Another t processes are sleeping until a fault occurs, at which time Cheap-BFT switches to MinBFT [111] for a limited period of time for providing liveness, then switches back to more efficient common case sub-protocol (i.e., $t+1$ active processes only).

[5] They however still require asymmetric cryptography for view-change sub-protocol.

MinBFT is another trusted component based protocol which requires $2t + 1$ processes in all cases.

Geo-replication. Some BFT protocols [22, 21, 101] are proposed targeting replication in WAN. Zeno [104] relaxes consistency level from linearisability to eventual consistency. By which, a client in Zeno can make its requests committed as long as any $t+1$ processes (a total of $3t+1$) are contactable, instead of two-third replicas. Steward [22, 21] leverages a hierarchical architecture for replication within and across sites. Within each site (or data center), Steward makes use of a BFT protocol (e.g., PBFT) to get requests accepted by $2t + 1$ processes. Across sites, Steward leverages a Paxos-like message pattern to make requests replicated by a majority. Hence, Steward requires $(2f + 1) \times (3t + 1)$ processes in total to tolerate any f failed or partitioned sites, and at most t Byzantine processes at each site. EBAWA [101] is designed based on Spinning [109], both rotate the primary (by changing the view number constantly) after each request is committed. Apart from that, EBAWA allows primaries in different views running concurrently so that requests at different sites can make progress in parallel. This idea is similar to Mencius which evenly partitions the sequence space to each process. Besides, EBAWA relies on a trusted counter named USIG [110], by which the total number of processes is reduced.

Other variants. Yin et al. [117] separate the traditional architecture of BFT protocols into two logical parts: one is in charge of sequencing requests (a.k.a., agreement stage) and the other is responsible for executing requests (a.k.a., execution stage). The two logical parts can be deployed at distinct machines. The authors further argue that, although $3t + 1$ processes is the lower bound for making requests sequenced, the number of processes required for executing requests is not necessarily $3t+1$. Then, they propose a new architecture which requires $2t + 1$ processes to execute requests instead. A step further, ZZ [114] requires only $t + 1$ active processes for execution stage in common case. When any process is faulty, ZZ activates some sleeping process and transfers the state on-demand.

Abstract [61, 26] is proposed as an abstraction method which eases the development of new BFT protocols. Abstract implements a new BFT protocol by composing a group of instances. Each instance is designed independently as a special BFT protocol, which always guarantees safety but can sometimes abort clients' requests (i.e., sacrificing liveness). By eventually switching current instance to a full-fledged BFT protocol, Abstract can ensure liveness as well when the network is eventually synchronous. Another advantage of this design is that, each instance can target a specific scenario (e.g., reducing latency if there is no contention, or making progress under Byzantine faults). Instances are switched from one to another in case the condition is not satisfied (e.g., there exist concurrent requests). As a demonstration, the authors design Aliph, a new BFT protocol which uses a quorum-based protocol that improves latency when there is no contention, and a chain protocol that targets high throughput when there are contentions but no failure/asynchrony.

3.2.3 Hybrid-Fault Tolerance

On the one hand, targeting CFT model people can design a simple and efficient protocol, but can not make the protocol survived under non-crash faults. On

the other hand, targeting BFT model people can provide robust service despite arbitrary faults, but can hardly keep the protocol as efficient (regarding cost and performance) and simple as CFT counterparts. Considering this trade-off, some work [45, 49, 100, 52] explores the fault space in between.

Echtle and Masum [57] propose a new Byzantine fault model named non-cooperative Byzantine faults, which is a bit weaker than classical BFT model. Non-cooperative fault model covers most crash and non-crash faults except the cooperative operations among two or more faulty processes. A cooperative operation is, for example, a faulty sender broadcasts two inconsistent messages to different processes, whereas some other faulty process accepts and forwards both messages.

Clement et al. [45] re-define faults as two categories: omission faults (i.e., process simply crashes) and commission faults (i.e., process behaves unfaithfully). In order to tolerate r omission faults and u commission faults, they design a new architecture for BFT protocols named UpRight, which requires a total of $2u + r + 1$ processes for agreement stage and $u + r + 1$ processes for execution stage.

Correia et al. [49] propose a new fault model named Arbitrary State Corruption (ASC), which is defined based on summarizing realistic errors reported in production systems. ASC model assumes that a process can crash, arbitrarily change some variable, or jump to another instruction. In order to tolerate ASC faults, they further propose an ASC hardening technique, which in general ensures that an ASC-faulty process can not propagate erroneous state to others. To guarantee this, each ASC-hardened process replicates its original state locally to a replica state. Besides, each event handler is executed locally twice: one accesses original state and the other accesses replica state. Correspondingly, every message contains an original information generated from the original state and a verification code generated from the replica state. Upon a process receives a message, it verifies if the message is correct with the help of verification code. ASC hardening technique can be applied to any distributed system (e.g., SMR protocols) and tolerate most faults reported.

Hybris [52] is a cloud key-value store which leverages both private and public cloud service. Hybris assumes that private cloud is trusted, in which processes can only crash, but its storage capacity is limited; whereas public cloud is not trusted, in which processes can exhibit arbitrary behaviors, although its storage capacity is nearly unlimited. Hybris stores metadata in private cloud (maintained by ZooKeeper coordination service) and replicates actual data in $2t + 1$ clouds, among which t clouds can behave arbitrarily.

More recently, Visigoth Fault Tolerance (VFT) [100] refines the space (1) between synchrony and asynchrony by quantifying the number of network faults tolerated, and (2) between crash and organizational attack[6] by quantifying the number of crash and organizational faults. Based on VFT model, a SMR protocol named VFT-SMaRt is designed, which has a general message pattern for all combinations of fault cases. Its users can target one specific scenario by simply setting the number of each type of fault that they expect to tolerate.

[6]Organizational attack is recognized as the most dangerous fault in BFT model.

Chapter 4

XFT : Practical Fault Tolerance Beyond Crashes

4.1 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 at cloud-scale and has direct implications on the business behind them [75].

Modern production systems today (e.g., Google Spanner [47] or Microsoft Azure [38]) 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 [102, 42].

These systems cannot deal with *non-crash* (or *Byzantine* [86]) faults, which, besides malicious behavior, include soft errors in the hardware, stale or corrupted data from storage systems, memory errors caused by physical effects, sleeping bugs in software, hardware faults due to ever smaller and faster circuits and human mistakes that cause state corruptions and data loss. Each of those faults has a public record of taking down major production systems and corrupting their service, including those of Amazon, Google, Facebook and others [49, 27]. This is hardly surprising, since the probability that a "very rare" machine fault (say, one that happens with probability $p = 10^{-5}$) affects some machine becomes significant when one considers the ever larger clusters of today, comprising $n > 100,000$ machines (in our example, at least $1 - (1 - p)^n \geq 0.63$).

---

[1] As an illustration, a 5-nines reliability of a system means that the system is up and correctly running at least 99.999% of the time. This corresponds to a system that, on average, malfunctions during one hour every 10 years.

Despite more than 30 years of intensive research in distributed computing since the seminal work of Lamport, Shostak and Pease [86], no *practical* answer to tolerating non-crash faults has emerged yet. In particular, asynchronous Byzantine fault-tolerance (BFT) that promises to resolve this problem [41] has not lived up to this expectation, due to its extra cost compared to CFT. For example, in the context of asynchronous (that is, eventually synchronous [56]) BFT SMR, this implies using at least $3t + 1$ replicas to tolerate t non-crash faults, instead of $2t + 1$ in the case of asynchronous CFT.

The overhead of asynchronous BFT is due to the extraordinary power given to the adversary, which may control both the faulty machines *and* the *entire network* in a coordinated way. In line with observations by practitioners [77], we claim that this adversary model is actually too strong for the phenomena observed in deployed systems: the Byzantine, non-crash faults experienced today occur independently of network faults (and often also independently of each other), just like a network switch failing due to a soft error has no correlation with a server that was misconfigured by an operator. 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. Surprisingly, though, the reliability models that decouple network faults (i.e., asynchrony, or untimely communication) from machine faults (yet allow for both classes of faults) have not been adequately studied.

4.2 Contributions

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

In a nutshell, XFT models crash faults, non-crash faults, and network faults as independent events, which is very reasonable in practice. In particular, XFT is ideal for systems connected by a WAN and for *geo-replicated* systems [47], as well as for similar systems in which network and machine faults are not correlated.

In short, XFT allows building reliable systems that:

- do not use extra resources (replicas) compared to asynchronous CFT;
- preserve *all* reliability guarantees of asynchronous CFT; and
- provide reliable service 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, a minority of the replicas are faulty or partitioned due to a network fault).

This allows for the development of reliable systems that have the cost of asynchronous CFT (already paid for in practice), yet with decisively better reliability than CFT and sometimes even better reliability than fully asynchronous BFT.

As the 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 SMR protocols in CFT model, such as Paxos [80]. Going beyond the guarantees also provided by Paxos and other CFT protocols, *XPaxos tolerates up to t non-crash faults whenever the sum of non-crash faults and network faults (which we quantify as the number of partitioned replicas) at any instant of time does not exceed t.* This comes without resource and performance overhead, as XPaxos uses $2t + 1$ replicas, while maintaining roughly the same performance (common-case communication complexity among replicas) as Paxos. Surprisingly, when assuming that machine and network faults are uncorrelated and all failure events are independent and identically distributed random variables, XPaxos in some cases (depending on the system environment) offers even *strictly stronger* reliability guarantees than state-of-the-art BFT replication protocols. Aligned with the industry practice, we express the reliability guarantees and coverage of fault scenarios through *nines of reliability*. More specifically, we distinguish *nines of availability* (for liveness) and *nines of consistency* (for safety) [20] and use these measures to compare different fault models.

To validate the performance of XPaxos, we deployed it in the increasingly relevant geo-replicated setting, across Amazon EC2 datacenters worldwide. In particular, we deployed XPaxos within the Apache ZooKeeper coordination service, a widely popular consistency service for cloud systems. Our evaluation shows that XPaxos performs almost as good as state-of-the-art CFT protocols on geo-scale, significantly outperforming existing state-of-the-art BFT protocols.

In the rest of the chapter, we first introduce the model we assume in Sec. 4.3. We then introduce XFT and compare it with CFT and BFT models in Sec. 4.4. In sec. 4.5 we motivate the hypothesis we assume. We present XPaxos (Sec. 4.6), analyze its reliability guarantees (Sec. 4.7) and evaluate it in the geo-replicated context. The full pseudocode and correctness proof of XPaxos is given in Appendix A and B.

4.3 System Model Refinement

Besides the assumptions and the concepts of SMR that we discussed in Chapter 2, in this section we refine the model specifically for XFT.

4.3.1 Network Faults

We assume that XFT model is subject to asynchrony, in that clients, replicas and communication do not need to be timely. In other words, *network faults* are possible; we define network faults as inability of *correct* replicas to communicate in a timely manner. More specifically, we declare a network fault if a message exchanged between any two correct replicas cannot be delivered within delay Δ, known to all replicas.

However, we assume that network faults are independent from machine faults. Note that we consider excessive processing delay at machine 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.

Clearly, the choice of delay Δ in connection with the network behavior and the system environment determines whether network faults actually occur.

To quantify the number of network faults, we first give the definition below.

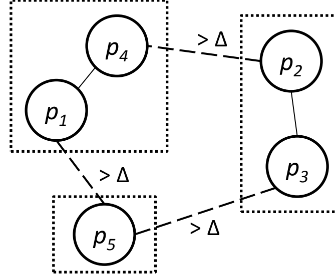


Figure 4.1: Illustration of partitioned replicas : $\{p_1, p_4, p_5\}$ or $\{p_2, p_3, p_5\}$ are partitioned based on Definition 8.

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

We say replicas p is *synchronous* if p is not partitioned. Note that if there are more than one subset of maximum size, then only one of them is recognized as the largest subset. For example in Fig 4.1, p_1 and p_4 can communicate between each other within delay Δ, but cannot make their messages delivered by other replicas in time, so do p_2 and p_3. p_5 is partitioned from others. The number of partitioned replicas in this case is 3, counting either group of p_1, p_4 and p_5 or group of p_2, p_3 and p_5. Note that in the worst case scenario, the number of partitioned replicas can be $n-1$, which means, none of replica can communicate with any other replica within delay Δ.

Then we can quantify the number of network faults.

Definition 2. *The number of network faults at a given moment s is defined as the number of correct but partitioned replicas at s, i.e., $t_p(s)$ ($0 \leq t_p(s) \leq n-1$).*

For several technical reasons, we quantify the network faults as the number of partitioned replicas, i.e., parameter $t_p(s)$. Firstly, the observation and experience by Stonebraker [8] show that "the most likely WAN failure is to separate a small portion of the network from the majority". Secondly, our 3-month hping experiment (we describe it in Sec. 4.5) among globally deployed Amazon EC2 machines show that, all long-time network partitions (usually dozens of seconds) we observed isolate one replica from others. Finally, DDoS attacks typically have one or a few targets (machines or network branches), which are becoming unavailable (i.e., partitioned) to indented users when under attacks [76].

4.3.2 Machine Faults

We assume that non-crash faulty replicas can be coordinated by the adversary; however, the adversary cannot easily impact communication between correct replicas nor break cryptographic primitives, such as digital signatures, digests and MACs. 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 process p and m, and $\langle m \rangle_{\mu_{p,q}}$ denotes message m together with $D(m)$ signed with a secret key shared by processes p and q.

To refine the reliability guarantee of XPaxos, we quantify the number of crash faults and non-crash faults at an instant of time by two parameters below.

- $t_c(s)$: the number of crash faulty replicas at a given moment s ($0 \leq t_c(s) \leq n$); and,
- $t_{nc}(s)$: the number of non-crash faulty replicas at a given moment s ($0 \leq t_{nc}(s) \leq n$).

Combining with network fault model in Sec 4.3.1, we define that:

Definition 3. *a replica is* ***available*** *if the replica is correct and not partitioned.*

4.3.3 Anarchy

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

Definition 4 (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.

4.4 The XFT Model

<table>
<tr><th rowspan="2"></th><th rowspan="2"></th><th colspan="3">Maximum number of each type of replica faults</th></tr>
<tr><th>non-crash faults</th><th>crash faults</th><th>partitioned replicas</th></tr>
<tr><td rowspan="2">Asynchronous CFT (e.g., Paxos [81])</td><td>consistency</td><td>0</td><td>n</td><td>$n-1$</td></tr>
<tr><td>availability</td><td>0</td><td colspan="2">$\lfloor \frac{n-1}{2} \rfloor$ (combined)</td></tr>
<tr><td rowspan="2">Asynchronous BFT (e.g., PBFT [41])</td><td>consistency</td><td>$\lfloor \frac{n-1}{3} \rfloor$</td><td>$n$</td><td>$n-1$</td></tr>
<tr><td>availability</td><td colspan="3">$\lfloor \frac{n-1}{3} \rfloor$ (combined)</td></tr>
<tr><td rowspan="3">XFT (e.g., XPaxos)</td><td rowspan="2">consistency</td><td>0</td><td>n</td><td>$n-1$</td></tr>
<tr><td colspan="3">$\lfloor \frac{n-1}{2} \rfloor$ (combined)</td></tr>
<tr><td>availability</td><td colspan="3">$\lfloor \frac{n-1}{2} \rfloor$ (combined)</td></tr>
</table>

Table 4.1: The number of each type of faults tolerated by representative SMR protocols.

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

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 [50, 102], asynchronous CFT [102, 80, 97], synchronous BFT [86, 53, 30], and asynchronous BFT [41, 26].

XFT, in contrast, redefines the boundaries between machine and network fault dimensions: XFT allows designing reliable protocols that tolerate one class of machine faults (e.g., crash faults) regardless of the network faults (asynchrony) and that, at the same time, tolerate another type of machine faults (e.g., non-crash faults) only when the network is "synchronous enough", i.e., when the sum of machine and network faults is within a threshold. More precisely, XFT provides reliability guarantees with *all* machine and network fault patterns, except those that put the system into anarchy (see Def. 4).

In a nutshell, XFT assumes that machine and network faults in a distributed system arise from different and independent causes. This greatly improves the flexibility in the choice of practically relevant fault scenarios. The intuition behind XFT starts from the observation that "extremely bad" system conditions, such as anarchy, are very rare and might not be worth paying the premium. With XFT, for example, we can design a practical SMR protocol (XPaxos, Sec. 4.6) that uses only $n = 2t + 1$ replicas, yet tolerates up to t *non-crash* faulty replicas, so long as the system remains out of anarchy.

We illustrate specific differences between XFT and asynchronous CFT and BFT in Table 4.1 in terms of their reliability (i.e., consistency and availability) guarantees for SMR.

State-of-the-art asynchronous CFT protocols [81, 98] 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 synchronous (not partitioned). As soon as a single machine is non-crash faulty, CFT protocols are neither consistent nor available.

Modern asynchronous BFT protocols [41, 73, 26] 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 synchronous.

In contrast, our XPaxos guarantees consistency in two modes: *(i)* without non-crash faults, despite any number of crash faulty and partitioned machines (i.e., just like CFT), and *(ii)* with non-crash faults, whenever a majority of replicas are correct and synchronous, 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 synchronous.

From Table 4.1, it is immediate that XFT offers strictly stronger availability guarantees than either CFT and BFT, as well as strictly stronger consistency guarantees than CFT. The consistency comparison between XFT and BFT is not immediately obvious, hence we postpone it to Sec. 4.7, where we also quantify the reliability comparison between XFT and CFT/BFT. Before that, we first exemplify XFT by presenting our XPaxos state machine replication protocol.

We further quantify the benefits of XFT in Sec. 4.7, where we give the reliability analysis of XPaxos.

4.5 Motivation

In this section we justify our hypothesis that motivates XFT, i.e., anarchy is rare in practical systems.

4.5.1 The Rare Anarchy Argument

The key observation behind XFT design is that anarchy is rare, i.e., arbitrary faults in practical systems rarely occur simultaneously with "enough" network faults among correct replicas. We support this observation in two ways: (1) we argue that arbitrary faults that occur in production are independent from network faults, and (2) we experimentally show that even network faults alone are rare.

Within a datacenter, developers can easily leverage resource isolation techniques, such as separate network interface controllers (NICs) [46], to ensure that messages among correct replicas are delivered timely. Hence, in this section we focus on the argument in a geo-replicated setting.

Non-crash faults vs. network faults. Historical examples of non-crash faults, including data losses, data corruptions, bugs and hardware faults [49], neither induce nor are induced by network faults among correct replicas in a geo-distributed setting. Namely, all practically relevant non-crash faults that we know of were constrained to local networks and datacenters and did not affect communication between machines on remote sites. Conversely, we do not know any example of a network fault between two geographically remote sites causing a (non-crash) fault at a third site.

Network faults are rare. Obviously, even if we accept that practically relevant arbitrary faults do not cause and are not caused by network faults among correct replica sites, anarchy might still arise if network faults occur relatively often.

	California	Ireland	Tokyo	Sydney	Sao Paulo
Virginia	88 /1097 /82190 /166390	92 /1112 /85649 /169749	179 /1226 /81177 /165277	268 /1372 /95074 /179174	146 /1214 /85434 /169534
California		174 /1184 /1974 /15467	120 /1133 /1180 /6210	186 /1209 /6354 /51646	207 /1252 /90980 /169080
Ireland			287 /1310 /1397 /4798	342 /1375 /3154 /11052	233 /1257 /1382 /9188
Tokyo				137 /1149 /1414 /5228	394 /2496 /11399 /94775
Sydney					392 /1496 /2134 /10983

Table 4.2: 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.

In practice, *network faults across datacenters are rare*. To support this claim, we made 3-month experiments during which we continuously measured round-trip latency across six Amazon EC2 datacenters worldwide using TCP ping (hping3 [12]). We used the least expensive micro instances, that arguably

have the highest probability of experiencing variable latency due to virtualization. Each instance was pinging all other instances every 100ms. The results of the experiment is summarized in Table 4.2.

Our experiment showed that the 99.99% (four nines) latency is relatively stable across all pairs of datacenters (regardless of their physical location), being always between 1.1 seconds and 2.4 seconds.

The results of our TCP ping experiment suggest that, with timeouts at the order of few seconds, untimely communication among correct replicas executing in different sites can be safely assumed to be rare. On the other hand, such timeouts at the order of few seconds might be very acceptable for use to providing timely communication.

This observation informed the design of XPaxos which indeed relies on timeouts at the order of few seconds in its view-change (in Sec. 4.6.2). System administrators using XPaxos can also be extremely conservative and set view change timeouts to at the order of few minutes (effectively masking all but very exceptional partitions), while not impacting common case performance of XPaxos at all.[2]

Even under DoS attack, to improve chances of excluding network faults, XPaxos can rely on existing defense techniques such as Proactive Surge Protection [43] to isolate bandwidth and send critical messages, which contain the information about the most recent state at the high priority, so that they are unlikely to be dropped by routers.

4.5.2 Long-Lived Arbitrary Faults

Even if network faults occur rarely and independently from arbitrary faults, anarchy may still arise in a long-lived system execution. Namely, a number of arbitrary faults might persist long enough to coincide with enough network faults, giving rise to anarchy. To cope with these long-lived faults, standard techniques such as proactive recovery [41] (which consists of periodically rebooting servers to reduce the lifetime of the faults) can be used.

To further minimize the impact of long-lived arbitrary faults, we added an important mechanism to XPaxos, called *fault detection*. XPaxos fault detection is independent from proactive recovery and is inspired by peer accountability [64]. The key property of XPaxos fault detection is the guarantee to detect, outside anarchy, non-crash faults that would leave the system in an inconsistent state in anarchy.

In a sense, with XPaxos fault-detection, even if the supreme attacker can mount the DoS communication attack, the attacker has to succeed *in his first attempt* to synchronize communication DoS attack and the Byzantine corruptions. Otherwise, the attack would be detected by XPaxos fault detection mechanism. Hence, fault-detection minimizes the XPaxos vulnerability window and helps prevent arbitrary faults from persisting long enough to coincide with network faults. (see Sec. 4.6.4 for more details).

[2] XPaxos view change timeouts could be also set adaptively and dynamically, but this is out of the scope of this thesis.

4.6 Protocol

XPaxos is a new state machine replication (SMR) protocol that ensures consistency and availability as depicted in Table 4.1.

More precisely,

Theorem 1. *as long as $\forall$ given moment s: $t_{nc}(s) = 0$ (no non-crash faulty replicas) or $t_c(s) + t_{nc}(s) + t_p(s) \leq t$, XPaxos ensures linearizability (i.e., one-copy semantics).*

We sketch the proof of Theorem 2 in Sec. 4.6.3 and formally prove it in Appendix B. Based on Theorem 2, XPaxos explicitly renounces to provide guarantees under the system condition in which there exists one or more non-crash faults and the sum of machine and network faults exceeds t, a.k.a., anarchy as defined in Def. 4.

In other words, *if XPaxos is never in anarchy then the system is consistent.* Theorem 2 implies that, XPaxos can have a number of crash or arbitrary faulty replicas as well as a number of partitioned but correct replicas. The system is safe as long as the sum of these numbers does not exceed t, or there is no non-crash fault at all.

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; and,
- 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 FD mechanism serves to minimize the impact of long-lived non-crash faults in the system and help detect them before they coincide with network faults and push the system to anarchy.

XPaxos is orchestrated in a sequence of *views* [41]. The central idea in XPaxos is that, in a common-case operation in a given view, XPaxos synchronously replicates the requests from the clients 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 [78]. A view is not changed unless there is a machine or network fault within its synchronous group.

In the common case (Sec. 4.6.1), the clients send digitally signed requests to the primary which are then replicated to the $t + 1$ 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.

In XPaxos view change (Sec. 4.6.2), 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 is in sharp contrast to classical reconfiguration/view-change in CFT and BFT protocols (e.g., [80, 41]), 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., linearizability) across XPaxos views in the presence of non-crash faults (but in the absence of full anarchy), despite replicating only across $t+1$ replicas in the common case. XPaxos novel and decentralized view-change scheme guarantees that, even in presence of non-crash faults, so long as there are not enough crash or partitioned replicas, at least one correct replica from the new view $i+1$ will be able to transfer the correct state from previous views, as it will be able to contact correct replicas from previous views (at least one for each preceding view).

Finally, the main idea behind the FD scheme of XPaxos (Sec. 4.6.4) is the following. In view change, a non-crash faulty replica (of the old synchronous group) might omit to transfer its latest state to a correct replica in the new synchronous group which may violate consistency in anarchy. However, such a fault can be detected and overcome by sending the digitally signed commit log from the correct replicas (of the old synchronous group), provided that there are correct replicas in preceding view and they can communicate. In a sense, with XPaxos FD, a critical non-crash machine fault must occur for the first time *together* with enough crash or partitioned replicas (i.e., in anarchy) to violate consistency. Hence, FD minimizes the XPaxos vulnerability window and helps prevent machine faults from persisting long enough to coincide with network faults.

In the following, we explain the core of XPaxos for the common-case (Sec. 4.6.1) and view-change (Sec. 4.6.2). XPaxos correctness arguments are given in Sec. 4.6.3. We then introduce fault-detection component in Sec. 4.6.4. We give an example of XPaxos execution in Sec. 4.6.5. A request retransmission scheme is presented in Sec. 4.6.6. We then discuss XPaxos performance optimizations (Sec. 4.6.7). The complete pseudocode is given in Appendix A and the full correctness proof is given in Appendix B.

4.6.1 Common Case

In general, as discussed in Sec. 2.2, the agreement property of a standard consensus algorithm requires that, a correct replica can decide a value only if the replica confirms that the same value will be decided by every correct replica eventually. In the context of XPaxos, this guarantee implies a replica can execute *req*, i.e., decide *req* at some consensus instance, only if the replica confirms that every correct replica in the synchronous group has collected the signed proof (i.e., commit log) of *req*. Hence, during view change, at least one correct replica in preceding view can provide and transfer the proof outside anarchy.

Nevertheless, XPaxos can rely on a speculation mechanism to reduce one message exchange within the synchronous group. Speculation has been used by many state-of-the-art BFT protocols such as PBFT [40] and Zyzzyva [73]. For

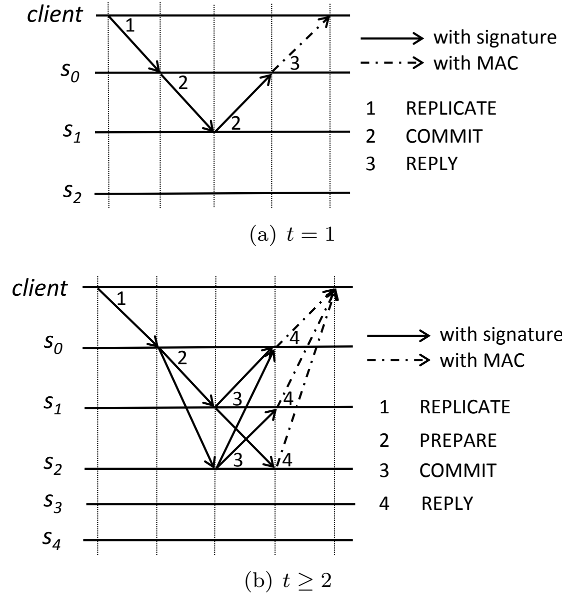


Figure 4.2: XPaxos common-case message patterns for $t = 1$ and $t \geq 2$ (here $t = 2$). Synchronous group illustrated are (s_0,s_1) (when $t = 1$) and (s_0,s_1,s_2) (when $t = 2$), respectively.

efficiency and simplicity, we describe our protocol below directly with speculation. More specifically, a replica in XPaxos can execute a request speculatively only if the replica has collected the signed proof from every active replica (i.e., commit log). Upon the client receives $t + 1$ speculative but matching replies, the client can deliver the reply as well.

A request executed speculatively at some correct replica may abort in view change if, during which this replica is partitioned. In this case, the replica needs to revert its state, e.g., practically to the most recent checkpoint. With our fault-detection mechanism, XPaxos ensures that *a replica may revert its state only if the replica is partitioned in view change and one or more non-crash faults exist simultaneously* (but may still outside anarchy). When $t = 1$, with fault detection, a request speculatively executed at some correct replica will never be aborted outside anarchy.

Fig. 4.2 gives XPaxos common-case message patterns in the special case when $t = 1$ and in the general case when $t \geq 2$. XPaxos is specifically optimized for the case where $t = 1$, as in this case, there are only two active replicas in each view. The special case $t = 1$ is also very relevant in practice (see e.g., Google Spanner [47]). In the following, 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.

Tolerating a single fault ($t = 1$)

For $t = 1$ (see Fig. 3(a)), the XPaxos common case involves only 2 messages between 2 active replicas. Upon receiving a signed request $req = \langle$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) 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 ($PrepareLog_{s_0}[sn] = \langle req, m_0\rangle$) (we say s_0 *prepares* req), and sends the message $\langle req, m_0\rangle$ to the follower, say 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 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.

General case ($t \geq 2$)

In case $t \geq 2$, the common-case message pattern of XPaxos (see Fig. 3(b)) contains an explicit PREPARE phase. More specifically, the primary (s_0) assigns a sequence number sn to a client's signed REPLICATE request req and forwards it to *all other active replicas* (i.e, the t followers) within $\langle req, prep = \langle \text{PREPARE}, D(req), sn, i\rangle_{\sigma_{s_0}}\rangle$. Each follower 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 $PrepareLog_*[sn]$. Then, a follower updates its local sequence number, *signs* the digest of the request, the view number and the sequence number, and forwards it to all active replicas within a COMMIT message. Upon receiving t signed COMMIT messages — one from each follower — (with a matching entry in the prepare log), an active replica logs $prep$ and the t signed COMMIT messages into its commit log $CommitLog_*[sn]$. Finally, each active replica executes the request and sends the 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.

In both cases ($t = 1$ or $t \geq 2$), a client that timeouts 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.

4.6.2 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. Proofs of requests committed in sg_i might be logged by as few as a *single* 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 [41, 73, 46] where the entire view change is led by a single

		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 4.3: Synchronous group combinations ($t = 1$).

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, at least one correct replica from sg_{i+1} will contact (at least one) correct replica from sg_i and transfer the latest correct commit log to the new view $i + 1$.

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.

Choosing active replicas

To choose active replicas for view i, we 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 4.3).

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 *active* replica. 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.

Performing view-change

Upon receiving SUSPECT message from active replica in view i (see the message pattern in Fig. 4.3), 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 $t + 1$ replicas in 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). Moreover, s_j sends the SUSPECT message to all replicas to accelerate the view change.

Note that XPaxos requires all active replicas in 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 purposely omit the VIEW-CHANGE message which contains the most recent state, even outside anarchy. Active replica s_j in view $i + 1$ waits for at least $n - t$ VIEW-CHANGE messages

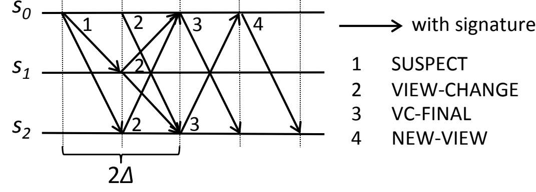


Figure 4.3: XPaxos view change illustration: synchronous group is changed from (s_0,s_1) to (s_0,s_2).

from all, but also waits for 2Δ time trying to collect messages as many as possible.

Upon completion of the above protocol, $s_j \in sg_{i+1}$ inserts all VIEW-CHANGE messages it has received in set $VCSet_{s_j}^{i+1}$. Then s_j sends $\langle$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.

Then, 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 enriches $VCSet_{s_j}^{i+1}$ by combining all $VCSet_{*}^{i+1}$ sets piggybacked in VC-FINAL messages. Then, for each sequence number *sn*, active replicas select the commit log with the highest view number in all VIEW-CHANGE messages, to confirm the committed request at *sn* (might be *null*).

Afterwards, to prepare and commit selected requests in view $i+1$, the new primary ps_{i+1} sends $\langle$NEW-VIEW$, i+1, PrepareLog\rangle_{\sigma_{ps_{i+1}}}$ to all active replicas in sg_{i+1}, where array *PrepareLog* contains prepare logs generated in view $i+1$ for each selected request. Upon receiving NEW-VIEW message, every active replica $s_j \in sg_{i+1}$ processes each prepare log in *PrepareLog* as described in the common case (see Sec. 4.6.1).

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

4.6.3 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 Appendix B.

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 log transfer 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 synchronous and correct replica $s_x \in sg_i$ (such a replica exists in sg_i as at most t replicas in sg_i are 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. Availability in XPaxos is guaranteed in case the synchronous group contains only correct replicas and there are no network faults within the synchronous group. With eventual synchrony we can assume that, eventually, there will be no network faults. Additionally, with all combinations of $t+1$ replicas (out of $2t+1$) rotating in the role of active replicas, XPaxos guarantees that, eventually, view change in XPaxos will complete with $t+1$ *correct* active replicas.

4.6.4 Fault Detection

Unlike BFT protocols, XPaxos does not provide consistency in anarchy[3]. Hence, non-crash faults could violate XPaxos consistency in the long run, if they persist long enough to eventually coincide with network faults.

To cope with long-lived faults, we propose a *Fault Detection (FD)* mechanism. Roughly speaking, FD guarantees the following property: *if a replica p fails arbitrarily 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 faults that occur when there are not enough crash and network faults, would be detected by XPaxos FD.

To ease the understanding of our approach, we first sketch how FD works in case $t = 1$, 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, such data loss faults are dangerous as they cannot be prevented by the use of digital signatures.

In general, 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 Sec. 4.6.1). In the new view, the primary will prepare and the follower will commit transferred requests both in commit logs and in prepare logs.

With the above modification, to violate consistency, a faulty primary (of preceding view i) would need not only to exhibit a data loss fault in its commit log, but also in its prepare log. However, such a data loss fault in the primary's prepare log would be detected, outside anarchy, because (i) the (cor-

[3]Neither do asynchronous BFT protocols when the number of faulty replicas exceeds t.

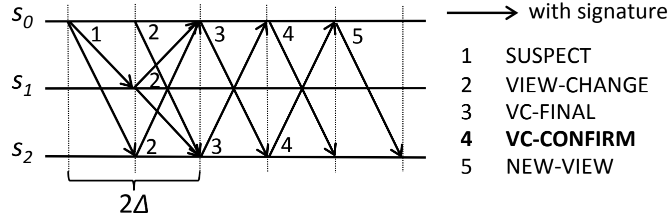
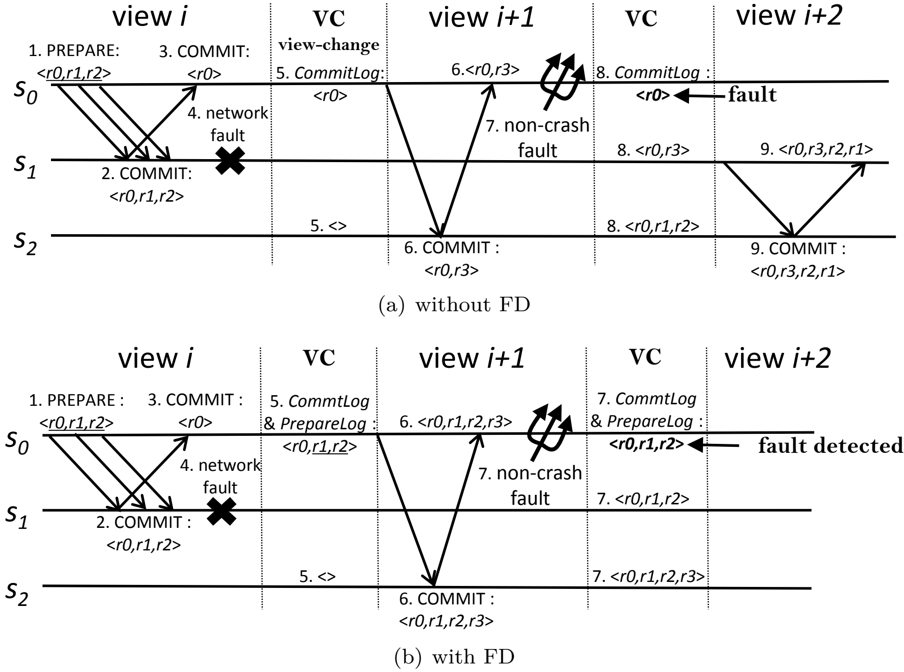


Figure 4.4: Message pattern of XPaxos view-change with fault detection: VC-CONFIRM phase is added; synchronous group is changed from (s_0,s_1) to (s_0,s_2).

rect) 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 (in case $t = 1$) simply detected outside anarchy by verifying the signatures in the commit log of the primary of view i.

We detail our approach as follows. 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 roughly 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 Fig. 4.3 and Fig. 4.4 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 $VCSet^{i+1}_{s_j}$ and adds the faulty replica to set $FSet$; (2) removes faulty messages from $VCSet^{i+1}_{s_j}$; and, (3) signs and sends $\langle \text{VC-CONFIRM}, i + 1, D(VCSet^{i+1}_{s_j})\rangle_{\sigma_{s_j}}$ 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 $VCSet^{i+1}_{s_j}$. $FinalProof_{s_j}[i + 1]$ 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.



(a) without FD

(b) with FD

Figure 4.5: XPaxos examples. 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.

The complete pseudocode is shown in Algorithm 9 and Algorithm 10 in Appendix A. The proof that XPaxos FD is strong completeness and strong accuracy outside anarchy is given in Appendix B.3.

4.6.5 XPaxos Example Execution

Fig. 4.5 gives an example of XPaxos execution when $t = 1$. The role of each replica in each view is shown in Table 4.3.

In Fig. 4.5(a), view change phase proceeds without fault detection. Upon the primary s_0 receives requests $r0$, $r1$, and $r2$ from clients, s_0 prepares these requests locally and sends COMMIT messages to the follower s_1. Then, s_1 commits $r0$, $r1$, and $r2$ locally and sends COMMIT messages to s_0. Because of a network fault, s_0 only receives COMMIT message of $r0$ 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 $i+1$, $r3$ 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 $r0$. Outside anarchy, requests $r0$ and $r3$ are committed in new view $i+2$ by receiving the VIEW-CHANGE message from s_2. Request r_3 is also committed by receiving the VIEW-CHANGE message from s_1. In view $i+2$, $r1$ is finally committed by every active replica.

In example of Fig. 4.5(b), XPaxos fault detection is enabled. In view i, the execution is the same as in Fig. 4.5(a). 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.

4.6.6 Request Retransmission

In order to ensure 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 8 in Appendix A), 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 $\langle$RE-SEND,$req_c\rangle$. Any active replica $s_j \in sg_i$, upon receiving $\langle$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 is to guarantee that the view-change can actually happen at every correct replica. Then client c forwards req_c to the primary of view $i+1$.

4.6.7 XPaxos Optimizations

Although, XPaxos as described above is sufficient to guarantee correctness, several standard performance optimizations can be optionally applied to XPaxos. These include checkpointing and lazy replication [78] to passive replicas (to help shorten the state transfer during view change) as well as batching (to improve the throughput).

Checkpointing. Similarly to other replication protocols, XPaxos can include a checkpointing mechanism that allows for garbage collection. To this end, every CHK requests (where CHK is a configurable parameter) XPaxos checkpoints the state within the synchronous group. Note that active replicas generate a checkpoint proof only upon the replicas confirm that every request implied by this checkpoint has been committed by every active replica.

More specifically, (refer to message pattern in Fig. 4.6) upon an active replica $s_j \in sg_i$ commits and executes a CHK request, s_j sends $\langle$PRECHK$, CHK, i, D(st_{s_j}^{CHK}), s_j\rangle_{\mu_{s_j,s_k}}$ to every active replica s_k, where $D(st_{s_j}^{CHK})$ is the digest of the state at s_j after executing request CHK. Then, upon receiving $t+1$ matching PRECHK messages, each active replica s_j generates the checkpoint

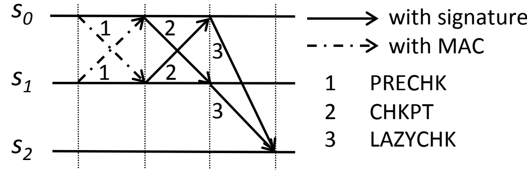
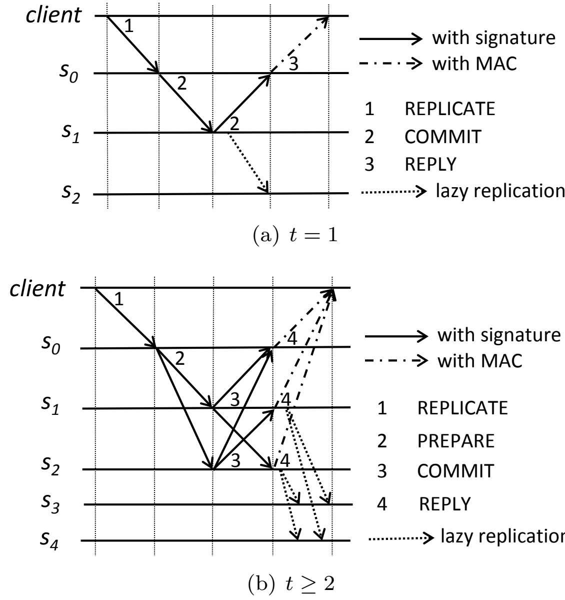


Figure 4.6: XPaxos checkpointing message pattern : synchronous group is (s_0,s_1).



(a) $t = 1$

(b) $t \geq 2$

Figure 4.7: XPaxos common-case message patterns with lazy replication when $t = 1$ and $t \geq 2$ (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.

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

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. 4.7) in case $t = 1$, upon committing request req, the follower sends commit log of req to the passive replica. In case $t \geq 2$, each of t followers 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.

[4]

Similarly, XPaxos propagates checkpoint proofs to all passive replicas by $\langle$LAZYCHK, $chkProof\rangle$,where $chkProof$ contains $t+1$ CHKPT messages (see Fig. 4.6).

Batching. In order to improve the throughput of cryptographic operations, the primary batches several requests when preparing. The primary waits as soon as B requests has received, 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.

4.7 Reliability Analysis

In this section, we compare the guarantees of XPaxos to state-of-the-art asynchronous CFT and BFT protocols, since asynchronous ones have been more widely deployed and studied recently. We discuss below the guarantees of consistency and availability in separate sections.

4.7.1 Reliability Parameters

In order to estimate the *coverage* of a fault model from basic assumptions about machines that we discussed before, 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 by $p_{correct}$, and the probability of a replica being crash faulty by p_{crash}. The probability that a replica is benign equals $p_{benign} = p_{correct} + p_{crash}$. Finally, a replica is non-crash faulty with probability $p_{non\text{-}crash} = 1 - p_{benign}$.

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

Based on our assumption that network faults and machine faults occur independently, for each replica, p_{benign} and $p_{correct}$ are independent from $p_{synchrony}$.

It is straightforward to reason that the probability that a replica is available equals $p_{available} = p_{correct} \times p_{synchrony}$.

For notational convenience, 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 the other probabilities of interest defined before.

4.7.2 Consistency

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

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

[4] Observe that, whereas 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.

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 [47], a CFT system as a whole loses one nine of consistency from the likelihood that an individual replica be benign.

XPaxos vs. CFT

We now quantify the advantage of XPaxos over one in the CFT model. From Table 4.1, 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.,

$$9ofC(\textsf{XPaxos}) =$$

$$\Big\lfloor -\log_{10}\Big(1 - p^n_{benign} - \sum_{i=1}^{t=\lfloor\frac{n-1}{2}\rfloor} \binom{n}{i} p^i_{non\text{-}crash} \times \sum_{j=0}^{t-i} \binom{n-i}{j} p^j_{crash} \times p^{n-i-j}_{correct} \times$$

$$\sum_{k=0}^{t-i-j} \binom{n-i-j}{k} p^{n-i-j-k}_{synchrony} \times (1 - p_{synchrony})^k \Big) \Big\rfloor,$$

where p^n_{benign} is the probability that no replica is non-crash faulty, counter i enumerates the number of non-crash faulty replicas from 1 to t, counter j indicates the number of crash faulty replicas (from 0 to $t-i$), and counter k is the number of network faults allowed, ranging from 0 to $t-i-j$.

To quantify the difference between XPaxos and CFT more tangibly, we calculated $9ofC(\textsf{XPaxos}_t)$, $9ofC(CFT_t)$ and $9ofC(\textsf{XPaxos}_t) - 9ofC(CFT_t)$ 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 $t = 2$ (hence $3 = n = 2t + 1$ or $5 = n = 2t + 1$). Note that $t = 1$ and $t = 2$ cases are most relevant to practice. We observed the following relation.

When $t = 1$,

$$9ofC(\textsf{XPaxos}_{t=1}) - 9ofC(CFT_{t=1}) =$$

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

When $t = 2$,

$$9ofC(\textsf{XPaxos}_{t=2}) - 9ofC(CFT_{t=2}) =$$

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

When $t = 2$, the condition that $9_{benign} = 9_{correct}$ (line 2) is only theoretically possible, which implies that $9_{crash} = 0$, i.e., if a machine is faulty then it must be non-crash faulty.

In Table 4.4 and 4.5 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 $9_{correct}$.

From $t = 1$ and $t = 2$ calculation we observed that, by using XPaxos, the number of nines of consistency added to CFT is proportional to the nines of consistency for correct or synchronous machines. The added nines is not directly related to p_{benign}, although $p_{benign} \geq p_{correct}$ must be satisfied.

Example 1. When $p_{benign} = 0.9999$ and $p_{correct} = p_{synchrony} = 0.999$, we have $p_{non\text{-}crash} = 0.0001$ and $p_{crash} = 0.0009$. $9 \times p_{non\text{-}crash} = p_{crash}$, i.e., $9_{benign} = 9_{correct} + 1$ implies that if a machine has suffered from machine faults for 10 times, then one of them is non-crash fault and the rest are crash faults.

- ($t = 1$.) $9ofC(CFT_{t=1}) = 9_{benign} - 1 = 3$, 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.

- ($t = 2$.) $9ofC(CFT_{t=2}) = 9_{benign} - 1 = 3$, whereas $9ofC(\text{XPaxos}_{t=2}) - 9ofC(CFT_{t=2}) = 2 \times 9_{correct} - 2 = 4$, i.e., $9ofC(\text{XPaxos}_{t=2}) = 7$. XPaxos adds 4 nines of consistency on top of CFT and achieves 7 nines of consistency in total.

Example 2. In a slightly different example, we assume $p_{benign} = p_{synchrony} = 0.9999$ and $p_{correct} = 0.999$, i.e., the network behaves better than Example 1.

- ($t = 1$.) $9ofC(CFT_{t=1}) = 9_{benign} - 1 = 3$, 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.

- ($t = 2$.) $9ofC(CFT_{t=2}) = 9_{benign} - 1 = 3$, whereas $9ofC(\text{XPaxos}_{t=2}) - 9ofC(CFT_{t=2}) = 2 \times 9_{correct} - 1 = 5$, i.e., $9ofC(\text{XPaxos}_{t=2}) = 8$. XPaxos adds 5 nines of consistency on top of CFT and achieves 8 nines of consistency in total.

XPaxos vs. BFT

Recalling that (see Table 4.1) one in asynchronous BFT model is consistent whenever no more than t machines are non-crash faulty. Hence,

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

where counter i enumerates the number of non-crash faults from 0 to t.

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:

		$9ofC(\mathsf{XPaxos}_{t=1})$						
9_{benign}	$9ofC(CFT_{t=1})$	$9_{correct}$	$9_{synchrony}$					$9ofC(BFT_{t=1})$
			2	*3*	*4*	*5*	*6*	
3	2	*2*	3	4	4	4	4	5
4	3	*2*	4	5	5	5	5	7
		3	5	5	6	6	6	
5	4	*2*	5	6	6	6	6	9
		3	6	6	7	7	7	
		4	6	7	7	8	8	
6	5	*2*	6	7	7	7	7	11
		3	7	7	8	8	8	
		4	7	8	8	9	9	
		5	7	8	9	9	10	
7	6	*2*	7	8	8	8	8	13
		3	8	8	9	9	9	
		4	8	9	9	10	10	
		5	8	9	10	10	11	
		6	8	9	10	11	11	
8	7	*2*	8	9	9	9	9	15
		3	9	9	10	10	10	
		4	9	10	10	11	11	
		5	9	10	11	11	12	
		6	9	10	11	12	12	
		7	9	10	11	12	13	

Table 4.4: $9ofC(CFT_{t=1})$, $9ofC(\mathsf{XPaxos}_{t=1})$ and $9ofC(BFT_{t=1})$ values when $3 \leq 9_{benign} \leq 8$, $2 \leq 9_{synchrony} \leq 6$ and $2 \leq 9_{correct} < 9_{benign}$.

		$9ofC(\mathsf{XPaxos}_{t=2})$						
9_{benign}	$9ofC(CFT_{t=2})$	$9_{correct}$	$9_{synchrony}$					$9ofC(BFT_{t=2})$
			2	*3*	*4*	*5*	*6*	
3	2	*2*	4	5	5	5	5	7
4	3	*2*	5	6	6	6	6	10
		3	6	7	8	8	8	
5	4	*2*	6	7	7	7	7	13
		3	7	8	9	9	9	
		4	7	9	10	11	11	
6	5	*2*	7	8	8	8	8	16
		3	8	9	10	10	10	
		4	8	10	11	12	12	
		5	8	10	12	13	14	
7	6	*2*	8	9	9	9	9	19
		3	9	19	11	11	11	
		4	9	11	12	13	13	
		5	9	11	13	14	15	
		6	9	11	13	15	16	
8	7	*2*	9	10	10	10	10	22
		3	10	11	12	12	12	
		4	10	12	13	14	14	
		5	10	12	13	15	16	
		6	10	12	14	16	17	
		7	10	12	14	16	18	

Table 4.5: $9ofC(CFT_{t=2})$, $9ofC(\mathsf{XPaxos}_{t=2})$ and $9ofC(BFT_{t=2})$ values when $3 \leq 9_{benign} \leq 8$, $2 \leq 9_{synchrony} \leq 6$ and $2 \leq 9_{correct} < 9_{benign}$.

$$p_{benign}^{2t+1} + \sum_{i=1}^{t} \binom{2t+1}{i} p_{non\text{-}crash}^{i} \times \sum_{j=0}^{t-i} \binom{2t+1-i}{j} p_{crash}^{j} \times p_{correct}^{2t+1-i-j} \times$$

$$\sum_{k=0}^{t-i-j} \binom{2t+1-i-j}{k} p_{synchrony}^{2t+1-i-j-k} \times (1 - p_{synchrony})^{k} >$$

$$\sum_{i=0}^{t} \binom{3t+1}{i} p_{benign}^{3t+1-i} (1 - p_{benign})^{i},$$

where $p_{non\text{-}crash} = 1 - p_{benign}$.

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

$$p_{benign}^{3} + \binom{3}{1} p_{non\text{-}crash} \times \binom{2}{0} p_{crash}^{0} \times p_{correct}^{2} \times \binom{2}{0} p_{synchrony}^{2} \times (1 - p_{synchrony})^{0} >$$

$$p_{benign}^{4} + \binom{4}{1} p_{benign}^{3} \times (1 - p_{benign}) \Rightarrow$$

$$p_{benign}^{3} + 3p_{non\text{-}crash} \times p_{correct}^{2} \times p_{synchrony}^{2} > p_{benign}^{4} + 4 \times (p_{benign}^{3} - p_{benign}^{4}) \Rightarrow$$

$$3(1 - p_{benign}) \times p_{correct}^{2} \times p_{synchrony}^{2} > 3(1 - p_{benign}) p_{benign}^{3}$$

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

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

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.

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

When $t = 1$,

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

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

Notice 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. In any case, BFT remains more expensive than XPaxos as $t = 1$ implies 4 replicas for BFT and only 3 for XPaxos.

When $t = 2$ we observed that:

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

From the above two functions we inferred that, when $t = 1$ or $t = 2$, only if $9_{benign} = 9_{correct}$ (which is not realistic), the nines of consistency of XPaxos can be the same or larger than BFT. Besides, by using BFT, the added nines to XPaxos is proportional to the nines of benign machines subtracted by the nines of correct or synchrony machines.

Example 1 (cont.). Building upon our example, $p_{benign} = 0.9999$ and $p_{synchrony} = p_{correct} = 0.999$, we have:

- ($t = 1$.) $9ofC(BFT_{t=1}) - 9ofC(\mathsf{XPaxos}_{t=1}) = 9_{benign} - 9_{synchrony} + 1 = 2$, i.e., $9ofC(\mathsf{XPaxos}_{t=1}) = 5$ and $9ofC(BFT_{t=1}) = 7$. BFT brings 2 nines of consistency on top of XPaxos.
- ($t = 2$.) $9ofC(BFT_{t=2}) - 9ofC(\mathsf{XPaxos}_{t=2}) = 2 \times (9_{benign} - 9_{correct}) + 1 = 3$, i.e., $9ofC(\mathsf{XPaxos}_{t=2}) = 7$ and $9ofC(BFT_{t=2}) = 10$. BFT brings 3 nines of consistency on top of XPaxos.

Example 2 (cont.). When $p_{benign} = p_{synchrony} = 0.9999$ and $p_{correct} = 0.999$, we have:

- ($t = 1$.) $9ofC(BFT_{t=1}) - 9ofC(\mathsf{XPaxos}_{t=1}) = 1$, i.e., $9ofC(\mathsf{XPaxos}_{t=1}) = 6$ and $9ofC(BFT_{t=1}) = 7$. XPaxos has one nine of consistency less than BFT (albeit the only 7th).
- ($t = 2$.) $9ofC(BFT_{t=2}) - 9ofC(\mathsf{XPaxos}_{t=2}) = 2 \times (9_{benign} - 9_{correct}) = 2$, i.e., $9ofC(\mathsf{XPaxos}_{t=2}) = 8$ and $9ofC(BFT_{t=2}) = 10$. XPaxos has 2 nines of consistency less than BFT (albeit the only 9th and 10th).

4.7.3 Availability

In this section, we compare the availability guarantees of XPaxos to asynchronous CFT and BFT protocols.

Similarly, we first introduce the number of *nines of availability* for protocol X, denoted by $9ofC(X) = 9of(P[\text{X is consistent}])$.

Recalling that whenever $\lfloor \frac{n-1}{2} \rfloor + 1$ active replicas in synchronous group are correct and not partitioned, XPaxos can make progress despite backups are benign or not, partitioned or not (see Table 4.1). Thus, we have:

$$9ofA(\mathsf{XPaxos}) = \left\lfloor -\log_{10}\left(1 - \sum_{i=\lfloor \frac{n-1}{2} \rfloor + 1}^{n} \binom{n}{i} p_{available}^{i} \times (1 - p_{available})^{n-i}\right)\right\rfloor,$$

where counter i indicates the number of available replicas (from $\lfloor \frac{n-1}{2} \rfloor + 1$ to n).

XPaxos vs. CFT

Considering CFT model (see Table 4.1), CFT is available whenever $n - \lfloor \frac{n-1}{2} \rfloor$ machines are correct and synchronous, plus other machines are benign. Hence,

$$9ofA(CFT) = \left\lfloor -\log_{10}\left(1 - \sum_{i=n-\lfloor \frac{n-1}{2} \rfloor}^{n} \binom{n}{i} p_{available}^{i} \times (p_{benign} - p_{available})^{n-i}\right) \right\rfloor,$$

where counter i indicates the number of available replicas (from $n - \lfloor \frac{n-1}{2} \rfloor$ to n), and $p_{benign} - p_{available}$ is the probability that a machine is benign but not available, i.e., the machine is crash or partitioned, or both.

Similarly to consistency analysis, we calculated $9ofA(CFT)$, $9ofA(\textsf{XPaxos})$ and $9ofA(\textsf{XPaxos}) - 9ofA(CFT)$ for all values of $9_{available}$ and 9_{benign} between 1 and 20 in the cases where $t = 1$ and $t = 2$. Although $p_{available}$ is not independent from p_{benign}, we only need to restrict that $p_{available} < p_{benign}$ is always true, i.e., $9_{available} < 9_{benign}$. We observed the following relation.

When $t = 1$,

$$9ofA(\textsf{XPaxos}) = 2 \times 9_{available} - 1,$$

$$9ofA(\textsf{XPaxos}) - 9ofA(CFT) = \max(2 \times 9_{available} - 9_{benign}, 0).$$

When $t = 2$,

$$9ofA(\textsf{XPaxos}) = 3 \times 9_{available} - 1,$$

$$9ofA(\textsf{XPaxos}) - 9ofA(CFT) =$$

$$\begin{cases} 3 \times 9_{available} - 9_{benign}, & 9_{benign} < 3 \times 9_{available}, \\ 1, & 3 \times 9_{available} \leq 9_{benign} < 4 \times 9_{available}, \\ 0, & 9_{benign} \geq 4 \times 9_{available}. \end{cases}$$

$9_{available}$	$9ofA(CFT_{t=1})$, 9_{benign} = 3	4	5	6	7	8	$9ofA(BFT_{t=1})$	$9ofA(\textsf{XPaxos}_{t=1})$
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 4.6: $9ofA(CFT_{t=1})$, $9ofA(BFT_{t=1})$ and $9ofA(\textsf{XPaxos}_{t=1})$ values when $2 \leq 9_{available} \leq 6$ and $9_{available} < 9_{benign} \leq 8$.

In Table 4.6 and 4.7 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}.

Example 1. When $p_{available} = 0.999$ and $p_{benign} = 0.9999$, we have:

$9_{available}$	$9ofA(CFT_{t=2})$						$9ofA(BFT_{t=2})$	$9ofA(\text{XPaxos}_{t=2})$
	9_{benign}							
	3	4	5	6	7	8		
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 4.7: $9ofA(CFT_{t=2})$, $9ofA(BFT_{t=2})$ and $9ofA(\text{XPaxos}_{t=2})$ values when $2 \leq 9_{available} \leq 6$ and $9_{available} < 9_{benign} \leq 8$.

- $(t = 1.)$ $9ofA(\text{XPaxos}_{t=1}) - 9ofA(CFT_{t=1}) = 2$, i.e., $9ofA(\text{XPaxos}_{t=1}) = 5$ and $9ofA(CFT_{t=1}) = 3$. XPaxos adds 2 nines of availability on top of CFT and achieves 5 nines of availability in total. Besides, XPaxos adds 2 nines of availability on top of an individual machine.

- $(t = 2.)$ $9ofA(\text{XPaxos}_{t=1}) - 9ofA(CFT_{t=1}) = 5$, i.e., $9ofA(\text{XPaxos}_{t=1}) = 8$ and $9ofA(CFT_{t=1}) = 3$. XPaxos adds 5 nines of availability on top of CFT and achieves 8 nines of availability in total. Besides, XPaxos adds 5 nines of availability on top of an individual machine.

From Example 1 we also observed that, when p_{benign} is close to $p_{available}$, e.g., $9_{benign} = 9_{available} + 1$[5], CFT protocols do not gain more nines of availability than an individual machine.

Example 2. When $p_{available} = 0.999$ and $p_{benign} = 0.999999$, we have:

- $9ofA(\text{XPaxos}_{t=1}) - 9ofA(CFT_{t=1}) = 0$, i.e., $9ofA(\text{XPaxos}_{t=1}) = 5$ and $9ofA(CFT_{t=1}) = 5$. In this example, XPaxos has no more nine of availability than CFT and they both achieve 2 nines of availability more than an individual machine.

- $9ofA(\text{XPaxos}_{t=2}) - 9ofA(CFT_{t=2}) = 3$, i.e., $9ofA(\text{XPaxos}_{t=2}) = 8$ and $9ofA(CFT_{t=2}) = 5$. XPaxos bring 3 nines of availability on top of CFT and achieves 8 nines of availability in total. CFT achieves 2 nines of availability more than an individual machine.

XPaxos vs. BFT

From Table 4.1, an asynchronous BFT protocol is available whenever $n - \lfloor \frac{n-1}{3} \rfloor$ machines are available, despite other machines are benign or not, partitioned or not. Thus,

$$9ofA(BFT) = \Big\lfloor -\log_{10}\Big(1 - \sum_{i=n-\lfloor \frac{n-1}{3}}^{n} \rfloor \binom{n}{i} p_{available}^{i} \times (1 - p_{available})^{n-i}\Big)\Big\rfloor.$$

we calculated $9ofA(\text{XPaxos})$ and $9ofA(BFT)$ for all values of $9_{available}$ between 1 and 20 in the cases when $t = 1$ and $t = 2$. 9_{benign} in this comparison

[5] $9_{benign} = 9_{available} + 1$ implies that if a machine has experienced 10 times any types of faults (crash, non-crash or network fault), then one of them is non-crash fault.

does not matter.

When $t = 1$,

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

When $t = 2$,

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

Hence, no matter what value $9_{available}$ is, when $t = 1$, XPaxos has the same number of nines of availability as BFT; when $t = 2$, XPaxos adds 1 nine of availability on top of BFT.

4.8 Performance Evaluation

In this section, we evaluate the performance of XPaxos and compare it to Zyzzyva [73], PBFT [41] and a WAN-optimized version of Paxos [80], using the Amazon EC2 worldwide cloud platform. We start by presenting the experimental setup (Sec. 4.8.1). We then evaluate the performance (throughput and latency) in the fault-free scenario (Sec. 4.8.2). We then analyze the behavior of XPaxos under faults (Sec. 4.8.4). Finally, we perform a performance comparison using a real application (Sec. 4.8.5): we replicate the ZooKeeper [67] coordination service using the four above mentioned protocols and compare its performance to the one achieved by the natively replicated ZooKeeper.

4.8.1 Experimental Setup

Synchrony and XPaxos

In the actual deployment of XPaxos, a critical parameter is the value of the timeout Δ, i.e., the upper bound on communication delay between any two *correct* machines. If the communication between two correct machines takes more than Δ, we declare a network fault. Notably, this timeout is vital to the XPaxos view-change (Sec. 4.6.2).

To understand the value of Δ in our geo-replicated context, we ran the 3-month experiment during which we continuously measured round-trip latency across six Amazon EC2 datacenters worldwide using TCP ping (hping3), as explained in Sec. 4.5.

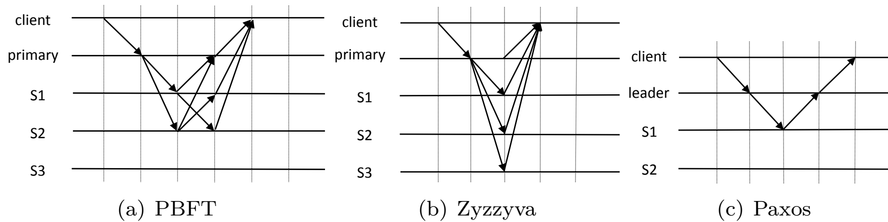
The results of this experiment are summarized in Table 4.2. While we detected network faults lasting up to 3 minutes, our experiment showed that the round-trip latency between *any* two datacenters was less than 2.5 seconds 99.99% of the time. Therefore, we adopted the value of $\Delta = 2.5/2 = 1.25$ seconds, yielding $p_{synchrony} = 0.9999$ (i.e., $9_{synchrony} = 4$).

Protocols under test

We compare XPaxos against three protocols whose common case message patterns when $t = 1$ are depicted in Fig. 4.8. The first two are BFT protocols, namely PBFT [41] and Zyzzyva [73] 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 (Fig. 4.8(a); see also [41]). 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 (Fig. 4.8(b)). The third protocol we compare against is a very efficient WAN-optimized variant of crash-tolerant Paxos inspired by [28, 74, 47]. We have chosen the variant of Paxos that exhibits the fastest write pattern (Fig. 4.8(c)). 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. We rely on HMAC-SHA1 to compute message authentication codes and RSA to sign and verify signatures.



(a) PBFT (b) Zyzzyva (c) Paxos

Figure 4.8: Message patterns of the three protocols under test ($t = 1$).

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, 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 4.8 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 replica to better emulate what is done in modern geo-replicated systems where clients are served by the closest datacenter [105, 47][6].

To stress the protocols, we run a microbenchmark that is similar to the one used in [41, 73]. 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. We 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.

[6]Modern geo-replicated system, like Spanner [47], use hundreds of instances of Paxos across different partitions to accommodate for geo-distributed clients.

<table>
<tr><th>PBFT</th><th>Zyzzyva</th><th>Paxos</th><th>XPaxos</th><th>EC2 Region</th></tr>
<tr><td>Primary</td><td>Primary</td><td>Primary</td><td>Primary</td><td>US West (CA)</td></tr>
<tr><td rowspan="2">Backups</td><td rowspan="3">Backups</td><td>Backup</td><td>Follower</td><td>US East (VA)</td></tr>
<tr><td>Backup</td><td>Passive</td><td>Tokyo (JP)</td></tr>
<tr><td>Backup</td><td>-</td><td>-</td><td>Europe (EU)</td></tr>
</table>

Table 4.8: Configurations of replicas. Greyed replicas are not used in the "common" case.

4.8.2 Fault-Free Performance

We first compare the performance of protocols when $t = 1$. We run both 1/0 and 4/0 microbenchmarks. The results are depicted in Fig. 4.9(a) and 4.9(b). On each graph, the X-axis represents the throughput (in kops/s), whereas the Y-axis represents the latency (in ms).

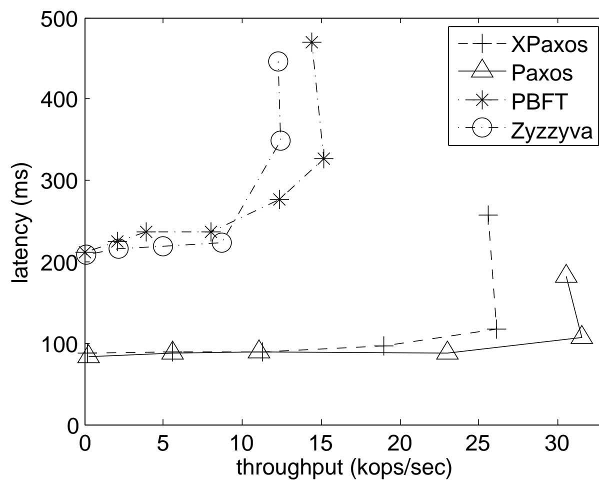
As we can see, in both benchmarks, XPaxos achieves significantly better performance than PBFT and Zyzzyva. Moreover, its performance are very close to that of Paxos. This comes from the fact that in a worldwide cloud environment, network is the bottleneck and message patterns of BFT protocols, namely PBFT and Zyzzyva, reveal costly. Paxos and XPaxos implement a round-trip across two replicas, 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). Paxos and XPaxos use the first $t + 1$ replicas as active replicas and the next t replicas as passive replicas. PBFT uses the first $2t+1$ replicas as active replicas and the last t replicas as passive replicas. Finally, Zyzzyva uses all replicas as active replicas.

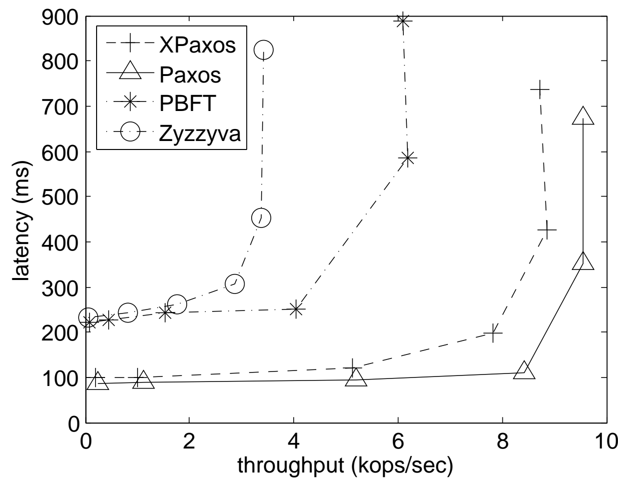
We can make observe that XPaxos again clearly outperforms PBFT and Zyzzyva. It also achieves very close performance to that of Paxos. Moreover, unlike PBFT and Zyzzyva, Paxos and XPaxos only suffer a moderate performance decrease with respect to the $t = 1$ case.

4.8.3 CPU Usage

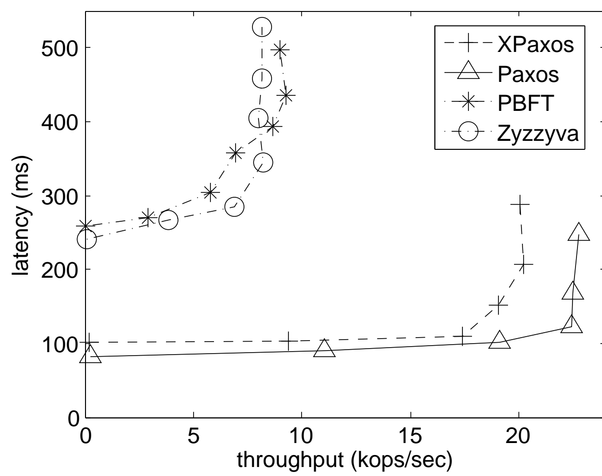
To assess the cost of using signatures in XPaxos, we extracted the CPU usage during the experiments presented in Sec. 4.8.2 with 1/0 and 4/0 micro-benchmarks when $f = 1$. During experiments, we periodically sampled CPU usage at the most loaded node (the primary in every protocol) with the *top* Linux monitoring tool. The results are depicted in Fig. 4.10 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 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



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



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



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

Figure 4.9: Fault-free performance

protocols (PBFT and Zyzzyva), while CPU usage of XPaxos is higher, XPaxos also sustains a significantly higher throughput.

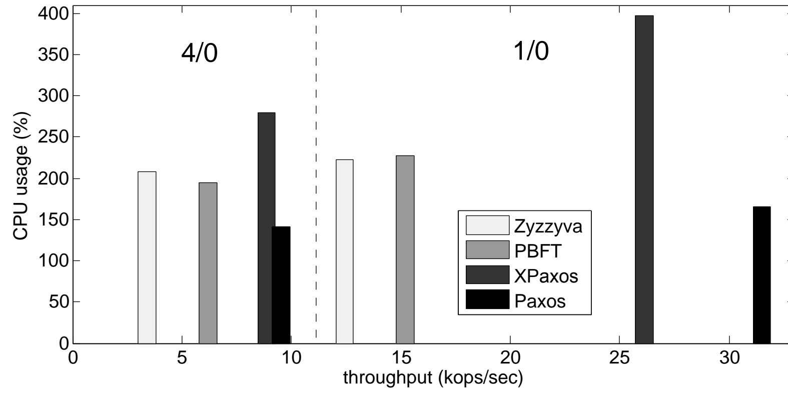


Figure 4.10: CPU usage when running the 1/0 and 4/0 micro-benchmarks.

4.8.4 Performance Under Faults

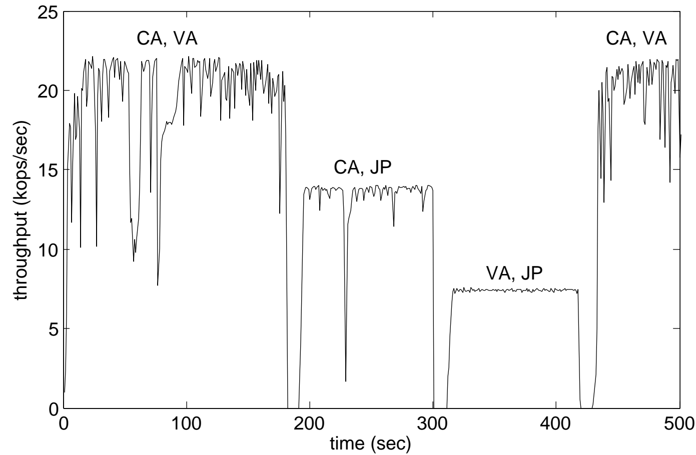


Figure 4.11: XPaxos under faults.

In this section, we analyze the behavior of XPaxos under faults. We execute the 1/0 micro-benchmark on three replicas (CA, VA, JP) to tolerate one fault (see also Table 4.8). 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 to send suspect message and again during state transfer in view change, Sec. 4.6.2) is set to 2.5s (see Sec. 4.8.1). We show the throughput of XPaxos in function of time in Fig. 4.11. We observe that after each crash, the system performs a view change (active replicas in each view are indicated in Fig. 4.11)

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 XPaxos lazy replication that keeps the passive replicas updated. We also observe that XPaxos sustains a different throughput in the different views. This is because the latency between the primary and the follower, and between the primary and clients, varies from view to view.

4.8.5 Macro-Benchmark: ZooKeeper

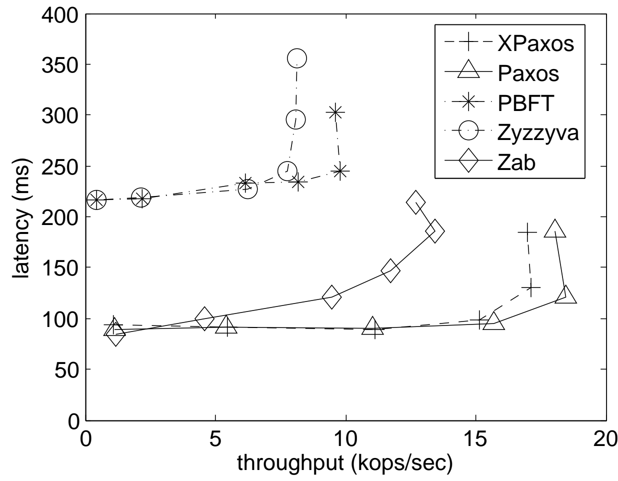


Figure 4.12: Latency vs. throughput for the ZooKeeper application ($t = 1$).

In order to assess the impact of our work on real-life applications, we measured the performance achieved when replicating the ZooKeeper coordination service [67] 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 a built-in replication protocol, called Zab [69]. 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 cancel out the overhead of durable storage that is not very efficient in virtualized environments, we store ZooKeeper data and log directories on a volatile *tmpfs* file system.[7] The tested configuration tolerates one fault ($t = 1$). ZooKeeper clients were located in the same region as the primary replica (CA). Each client invokes 1kB write operations in a closed-loop.

Fig. 4.12 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 achieve performance close to that of Paxos. More surprisingly, we can see that

[7]We used non-persistent storage in ZooKeeper in order to remove possible I/O overhead with the goal of focusing on comparing replication protocol performance, rather than hitting other system bottlenecks.

XPaxos is more efficient than the built-in Zab protocol, although the latter only tolerates crash faults.

Chapter 5

Leader Set Selection for Low-Latency Geo-Replicated State Machine

5.1 Introduction

As we discussed in Sec. 1.1, internet applications [47, 74, 67] make use of a State Machine Replication (SMR) protocol as a basic synchronization primitive to provide reliable service. Replication however becomes challenging at the planetary scale due to latencies that cannot be masked, being bounded by the speed of light.

To illustrate the problem, consider classical SMR protocols such as Paxos that have a *single leader* responsible for sequencing and proposing clients' requests. These proposed requests are then replicated across at least a majority of replicas, executed in order of their logical sequence numbers, with application-level replies eventually sent to the clients. For clients that are remote from the leader, this may imply costly round-trips across the globe.

Many latency-optimized SMR protocols tailored to geo-replication have been proposed recently [82, 106, 95, 93, 55]. These protocols are similar in that they rely on a (variant of) consensus algorithm which ensures reliability despite different types of faults. Their differences have to do with the way they ensure linearizability [66] or strict serializability [31], i.e., a total order among all clients' requests. In general, these protocols can be classified into two categories: *all-leader* protocols and *leaderless* protocols.

In *all-leader* SMR protocols, the total order is pre-established based on, e.g., logical sequence numbers (e.g., Mencius [93]) or physical clocks (e.g., Clock-RSM [55]). A request can be proposed and sequenced by *any* replica, where every replica can act as a leader, typically by partitioning the *sequence index* space. In these protocols, a client submits its request to a nearby replica, avoiding the communication with a single (and possibly remote) leader. Just like classical single-leader Paxos, all-leader SMR protocols work with the con-

servative assumption that clients' requests do not commute — hence all-leader protocols still require replica coordination to maintain a total order across all requests.

A challenge for all-leader SMR protocols is the coordination with distant or slow replicas, which can be the bottleneck, or even block the progress entirely. In some sense, the performance is determined by the "slowest" replica: this causes what is known as a "delayed commit" problem [93]. Roughly speaking, the delayed commit problem (that we detail in Sec. 5.2.1) arises from the need to confirm that all requests with an earlier sequence number are not "missed". For the most typical, imbalanced workloads, e.g., if most requests originate from clients that gravitate to a given site S, this incurs communication with *all* replicas including remote and slow ones. In this case, all-leader SMR may have worse performance than single-leader SMR, in which replication involves only S and the majority of sites closest to S.

On the other hand, *leaderless* protocols (e.g., Generalized Paxos [82] or EPaxos [95]) exploit the possible *commutativity* of concurrent requests and execute such requests (e.g., requests accessing distinct parts of state) out of order at different replicas. In a leaderless protocol, a request is typically proposed directly by a client or by any replica, and committed with a round-trip latency to replicas belonging to a *fast-quorum*.[1] Every replica in a fast-quorum checks the potential conflict among concurrent requests. Unfortunately, if (1) the conflict is detected, *or* (2) available replicas are fewer than a fast-quorum, one or more additional round-trip delay to a majority of replicas is introduced to resolve conflict. Notice that leaderless protocols do not suffer from the delayed commit problem since no order is pre-defined.

In summary, there is no protocol that fits all situations and the choice of the "best" protocol largely depends on the specific replica configuration and the workload. Namely, existing protocols are based on one of the following assumptions: (1) requests come mostly from within the vicinity of a single site (favoring classical single-leader SMR); (2) requests are evenly distributed among all sites and most concurrent requests do not commute (favoring all-leader SMR); or, (3) most concurrent requests commute (favoring leaderless SMR). More than often, none of these assumptions is always true in practice. For instance, due to time zone differences, clients located at a given site may have different access patterns at different times of a day, dynamically changing the "popularity" of sites and possibly also the workload balance.

In this chapter, we present Droopy and Dripple, two sister approaches that explore the way to mitigate issues of all-leader and leaderless SMR protocols. The design considerations behind Droopy and Dripple are straightforward:

1. removing unnecessary coordination and dependencies to avoid delayed commit (in all-leader protocols); and,

2. pre-ordering non-commutative requests to avoid conflict resolution (in leaderless protocols).

Based on these two guidelines, Droopy is designed to dynamically reconfigure the set of leaders which can contain anything from a single to all replicas.

[1] A fast-quorum is larger than a majority [113] - we postpone a more detailed discussion to Sec. 5.2.2.

The selection of leaders is based on previous workload and network condition. Dripple in turn is a state partitioning and coordination algorithm inspired by leaderless protocols. Dripple allows Droopy to reconfigure the set of leaders of each partition separately, at the same time ensuring to provide strong consistency, i.e., strict serializability [31] as a whole.

Although Droopy and Dripple can be applied to any SMR protocol, we implemented our approaches on top of a state-of-the-art all-leader SMR protocol — Clock-RSM [55] (we call our variant D^2Clock-RSM). On Amazon EC2 platform we evaluate D^2Clock-RSM under imbalanced workloads and three representative types of balanced workloads. For a more comprehensive evaluation, we also implemented several state-of-the-art SMR protocols, i.e., Paxos, native Clock-RSM, Mencius and EPaxos [95] by making use of the same code base. Our experimental evaluation shows that, under typical imbalanced workloads, Droopy enabled Clock-RSM efficiently reduces latency compared to its native protocol. With one spare replica (i.e., a total of $n = 2t + 2$ replicas to tolerate t crash faults), Droopy reduces latency by at most 56% (comparing to the native Clock-RSM). Besides, under balanced but commutative workloads, where requests issued by each site access distinct part of state, D^2Clock-RSM still significantly reduces latency compared to the native Clock-RSM, Paxos and Mencius, and it has the performance similar to that of EPaxos. In contrast, under balanced but non-commutative workloads, D^2Clock-RSM has the similar latency as that of the native Clock-RSM, and both achieve the lower latency than EPaxos.

The rest of this chapter is organized as follows. In Sec. 5.2 we discuss, in the context of related work, how delayed commit problem and non-commutative requests affect latency. In Sec. 5.3 we describe the system model and assumptions. In Sec. 5.4 we give the overview and in Sec. 5.5 the details of Droopy and Dripple. We sketch the proof of Dripple in Sec. 5.5.3. Sec. 5.6 depicts the way we implemented and evaluated D^2Clock-RSM.

5.2 Related Work Revisited

Since Lamports' seminal Paxos protocol [80, 81], numerous variants have been designed more recently [83, 82, 93, 106, 95, 55, 109, 101] in order to mitigate the latency problem for geo-replicated state machine. In general, these protocols can be further classified into two categories: *all-leader protocols*, that allow every replica to act as the leader and propose requests piggybacked with a totally ordered sequence index; and *leaderless protocols*, that allow every client or replica to propose requests without pre-imposing total order, but detect conflicts dynamically.

In this section, we argue in the context of related work that neither of these two categories is suitable for all circumstances. In Sec. 5.2.1 we discuss the delayed commit problem in the context of two all-leader protocols. Then, in Sec. 5.2.2 we introduce some leaderless protocols and how their performance is affected by non-commutative requests.

5.2.1 All-leader Protocols (Delayed Commit)

In order to reduce perceived latency for clients scattered across the planet, every replica in all-leader SMR protocols can act as a leader and assign a pre-

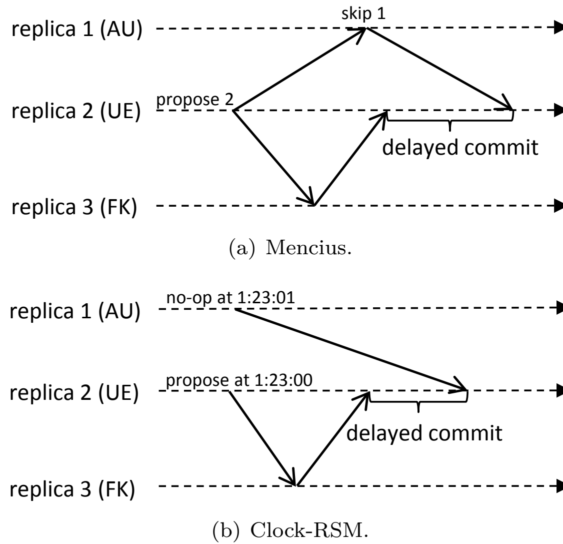
ordered *sequence index* (e.g., logical sequence number or physical clocks) to a request.

Mencius [93] facilitates multiple leaders by evenly pre-partitioning sequence number space across *all* replicas (modulo number of replicas), so that each replica can propose requests. For example, with 3 replicas, replica 1 sequences requests at sequence numbers 1,4,7..., replica 2 sequences requests at 2,5,8,..., and replica 3 sequences requests at 3,6,9,.... To avoid execution delays, if a replica lags behind, it skips some sequence numbers by assigning *no-op* requests (or *empty* requests) to skipped sequence numbers, in order to catch up with other replicas. Such a *skipping* replica must however let its peers know which sequence numbers it skips — leading to what is known as the "delayed commit" problem. Intuitively, the delayed commit problem arises when the workload or latency across replicas is not uniform, which is most often actually the case.

We illustrate the delayed commit problem in Mencius by an example with $n = 3$ replicas deployed on Amazon EC2. The round-trip latency among 6 sites on Amazon EC2 is shown in Fig. 5.5 (Sec. 5.6). The example is shown at Fig. 5.1(a). Assume that replica in UE (replica 2 in Fig. 5.1(a)) proposes request R_1 at sequence number 2, whereas replica at AU (replica 1 in Fig. 5.1(a)) is responsible for proposing a request at sequence number 1. Because of imbalanced workload, replica AU has not proposed any request when it receives proposal of R_1 from replica UE, at which time replica AU skips sequence number 1. Only upon replica UE has received the skip message for sequence number 1, it can execute R_1 locally. Hence, a round-trip latency from UE to AU is introduced (231 ms), which is much larger than the round-trip latency from replica 2 to a majority (including IR besides UE itself) that a solution based on a single UE leader would require (88 ms). We formalize this example by the following definition.

Definition 5. *(Delayed commit) If replica s_2 is proposing request req_2 at sequence index sn_2, and replica s_1 is responsible for proposing request at sequence index sn_1 ($sn_1 < sn_2$), then s_1 may delay the commit process at replica s_2 for req_2, if s_1 takes longer time than a round-trip latency from a majority at s_2 to either (1) notify s_2 that no request will be proposed at sn_1, or (2) make the request proposed at sn_1 committed at s_2.*

More recently, Clock-RSM [55] was proposed with the goal of mitigating the delayed commit problem in Mencius by using loosely synchronized clocks. In Clock-RSM, each replica proposes requests piggybacked with its physical clock timestamp, which is used instead of logical sequence numbers to order requests. In a similar way to Mencius, before executing a request, each replica is obliged to confirm that all previous requests have been executed locally. This requirement implicitly implies that no request with an earlier clock will be proposed later by any replica. In order to achieve this requirement efficiently, especially under imbalanced workloads, replicas in Clock-RSM exchange their physical clocks periodically (e.g., every 5 ms in the implementation) to notify other replicas that requests with an earlier clock timestamp have been sent already. Note that this is guaranteed by assuming a FIFO channel between replicas. Whenever clocks at different replicas are synchronized, Clock-RSM reduces one-way latency for the delayed commit problem in Mencius. That is to say, each replica frequently notifies others that "no more request" will be proposed before the given timestamp.



(a) Mencius.

(b) Clock-RSM.

Figure 5.1: Delayed commit problem in state-of-the-art all-leader protocols.

However, delayed commit problem still exists in Clock-RSM, albeit being less pronounced. In example in Fig. 5.1(b), just after replica UE proposes R_1, replica AU sends its current clock time to all replicas. Upon UE receives the clock message from AU, it confirms that no request with an earlier timestamp will be proposed by replica AU. Nevertheless, one-way latency from AU to UE is still larger than the round-trip latency from UE to a majority.

To sum up, all-leader protocols implicitly assume that (1) concurrent requests are not commutative, and (2) that the workload is balanced, i.e., that every replica receives and proposes requests at approximately the same rate. Hence, these protocols (1) need to maintain a global order among *all* requests and (2) due to the heterogeneity of wide area network and workloads, all-leader protocols suffer from the delayed commit problem.

5.2.2 Leaderless Protocols (With Conflict Resolution)

In order to remove unnecessary dependencies among concurrent but commutative requests, unlike single-leader Paxos or its all-leader variants, leaderless protocols [83, 106, 95] reduce latency by sequencing requests in a somewhat more decentralized and less ordered fashion. Namely, in leaderless protocols, each request is directly proposed by the client or by any replica. A proposed request is first sent to all replicas. Upon receiving the request, each replica individually (and tentatively) assigns a sequence number to the request, possibly adding the information about other concurrent requests, and replies to the sender. In case a *fast-quorum* of replicas have given a request the same sequence number, the request is committed. Otherwise, another one or more round-trip delay is introduced in order to arbitrate conflicts. This design is motivated by an aggressive assumption that most concurrent requests (e.g., more than 98% [95]) are commutative.

The number of replicas in a fast-quorum is larger than a majority [113]. For instance, it was proved [83] that, for a system using the minimal number of $n = 2t+1$ replicas, if a request is committed after a round-trip communication directly from the client to a fast-quorum (which yields only two hops in solving consensus, and hence, SMR), then the number of replicas in a fast-quorum is at least $\lceil \frac{3}{2}t \rceil + 1$.

Fast Paxos [83] allows clients to send requests directly to all replicas and matches the $\lceil \frac{3}{2}t \rceil + 1$ fast-quorum lower bound. In case a collision is detected, i.e., there are not enough replicas giving the same sequence number, Fast Paxos relies on a single leader to re-orders the requests.

Generalized Paxos [82] reduces the rate of conflicts in Fast Paxos by exploiting *commutativity* among concurrent requests. In case two requests commute (e.g., request 1 writes object A, request 2 writes object B), these two requests can be executed out of order at different replicas. Generalized Paxos still relies on a single leader sub-protocol to resolve the order of non-commutative requests.

EPaxos [95] reduces the number of replicas in the fast-quorum by exactly one replica compared to Generalized Paxos. Notice that, EPaxos can achieve this since it introduces one more communication step (i.e., the third hop) from the client to a nearby replica, say command leader, at the first step (hence the $\lceil \frac{3}{2}t \rceil + 1$ fast-quorum lower bound does not apply in this case). The command leader then binds a sequence number (which is not totally ordered among replicas) and a set of conflicting (non-commutative) requests known by the command leader to the request. Upon receiving a proposal from any command leader, each replica updates the conflict set of the received request and re-calculates the sequence number based on the new dependency information, then replies to the command leader. Upon the command leader receives the same sequence number and conflict set from a fast-quorum, the request is committed and can be executed orderly based on conflict information. Otherwise, the command leader relies on another round-trip communication among a majority to confirm the order eventually.

In summary, the performance of leaderless protocols is driven by the number of non-commutative requests as well as the size of fast-quorums. For instance, in a system with 5 replicas in Fig. 5.5, if replica at UE proposes a request which is conflicting with a concurrent request proposed by JP (Japan), then at least another round-trip latency among a majority at each site (80 ms and 107 ms, respectively) is introduced.

5.3 System Model Refinement

We assume there are a total of $n \geq 2t+1$ replicas (or sites), $s_1, s_2, ..., s_n$, among which t can crash (but not behave arbitrarily). We also assume there are an unbounded number of clients which can issue requests and fail by crashing.

Network. We allow for asynchrony in that communication time between any two replicas is not bounded — however, to circumvent the FLP impossibility [58], we assume the system to be eventually synchronous. We further assume a FIFO channel between any two replicas, i.e., messages from on replica to another are delivered by the destination in order. For simplicity reason, we also assume that the communication channel is symmetric (since measuring

one-way latency in WAN is usually non-trivial), which means the one-way latency from one replica to another equals the value of the opposite direction.
Clocks. When referring to clock-based systems (e.g., Clock-RSM), we assume there is a physical clock equipped at each replica. Clocks at different replicas are loosely synchronized using a clock synchronization protocol such as Network Time Protocol (NTP) [17].
Terminologies. Following the terminology of existing SMR protocols, we say replica s_i *proposes* request req at sequence index sn if : (1) s_i is one of the leaders; (2) s_i gives req a sequence index sn; and (3) s_i sends a PROPOSE message which contains req and sn to at least $t+1$ replicas. Similarly, we say replica s_i *commits* request req at sequence index sn if s_i confirms that (1) $t+1$ replicas have replicated req at sequence index sn, and (2) all preceding requests, i.e., the requests with earlier sequence indices than sn, are committed at s_i.
Linearizability versus (strict) serializability. Since Droopy is based on single-leader or all-leader protocols, it provides the same consistency guarantee as that of native protocols, i.e., linearizability [66]. Linearizability abstracts the application state as a indivisible object or register, and each request can read and modify this object.

In contrast, Dripple explores request commutativity by dividing the state into m disjoint partitions $p_1, p_2, .., p_m$ by, e.g., range partitioning. A partition can be a single object or a group of objects . Each request can access (i.e., read or write) several objects among several different partitions, specified at the time when the request is received by any replica. Hence, we further assume that applications using Dripple can provide the information regarding which part of state, i.e., which partitions each request will potentially access.

Since linearizability targets single object scenario, by leveraging state partitioning, Dripple instead ensures strict serializability [31], which is widely adopted in transactional semantics such as database systems. Strict serializability ensures that the execution order of requests over several partitions is serializable, which is equivalent to a sequential execution and this sequential execution respects real-time order. Real-time order implies that, if request R_1 is executed at replica s_i before another request R_2 is issued by any client, then the sequential execution should reflects the order that R_1 is executed before R_2.

5.4 Protocol Overview

The architecture of Droopy and Dripple is shown in Fig. 5.2. Our approaches are built upon existing single-leader or all-leader protocols and provide the same interfaces [2] to upper applications. If Droopy is enabled alone, i.e., the path ②→⑤ in Fig. 5.2, then the protocol explores the space between single-leader and all-leader protocols, i.e., selecting the set of leaders dynamically from one to all. If Dripple is enable alone, i.e., the path ①→④ in Fig. 5.2, then the protocol is state-partitioned, with the leader(s) of each partition statically located at one or all replicas. Finally, if both Droopy and Dripple are enabled, i.e., the path ①→③→⑤, then the protocol is state-partitioned, with the set of leaders of each partition dynamically reconfigured.

[2] Dripple additionally requires information about the accessed partitions.

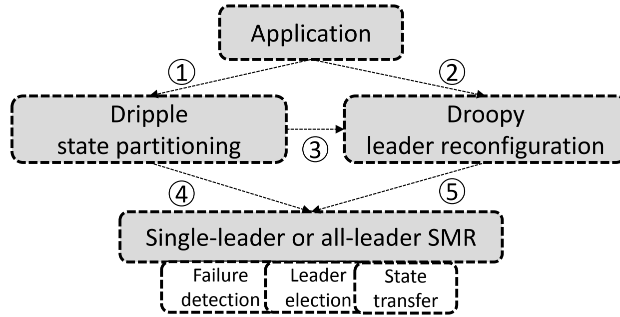


Figure 5.2: Architecture of Droopy and Dripple.

5.4.1 Droopy

Droopy is designed to dynamically reconfigure the set of leaders. Each set of leaders is called a *config*. Generally speaking, Droopy splits the space of sequence *indices* (e.g., logical sequence numbers or physical timestamps) into ranges (equal to e.g., δ sequence numbers or seconds). Each range of indices is mapped to a config and maintained as *a lease*. It is important to notice that, although a lease can reflect a physical time range in clock-based systems, time anomalies such as clock drifts, do not affect the correctness of Droopy, but only its performance.

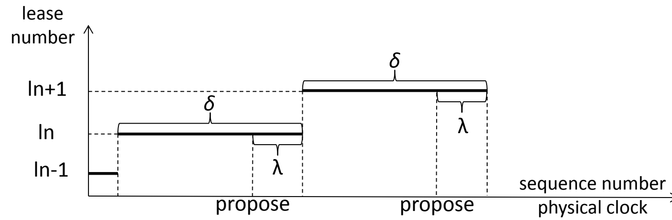


Figure 5.3: Lease renewal.

In a nutshell, a lease in Droopy is a commitment of a subset of replicas to a range of sequence indices, such that only those specific replicas, i.e., the leaders, can propose requests. Leases are proposed by replicas and maintained by a total order primitive, which can be implemented by a classical single-leader Paxos (we call this *Lease-Paxos* or simply *L-Paxos*). In a sense, Droopy follows the approach to state machine reconfiguration proposed in [84].

When current lease ln is about to expire (refer to Fig. 5.3), i.e., less than λ sequence indices are available, at least $t + 1$ replicas propose a new leader configuration for the next lease $ln + 1$, which is selected based on previous workload and network condition. In this article we assume that the goal for Droopy is to minimize the average latency across all sites. Droopy can easily support other criteria such as minimizing the latency for some given sites. At least $t + 1$ replicas should propose a new lease so that the crash of any minority of replicas will not stop the lease renewal process. However, in order to guarantee that all leases are consistent among all replicas, Droopy requires

that the first delivered config for each lease is the one agreed by all. This is guaranteed by the total order primitive within L-Paxos.

A client submits its request by contacting the nearby replica, that we call the *source replica* of the request. Upon reception of the request, the source replica proposes the request if it is one of the leaders in current lease. Otherwise, the source replica propagates the request to one of the leaders that acts as a *proxy*. For each non-leader replica, its proxy is the leader that is expected to introduce the minimum commit latency, with respect to the non-leader replica. Upon a request is committed, the source replica sends a reply to the client. Besides, each replica updates its *frequency array* monitor, that records the number of requests from each source replica. Frequency array is served to help the decision of next leader set. Each replica also periodically measures and shares the round-trip latency from itself to others.

5.4.2 Dripple

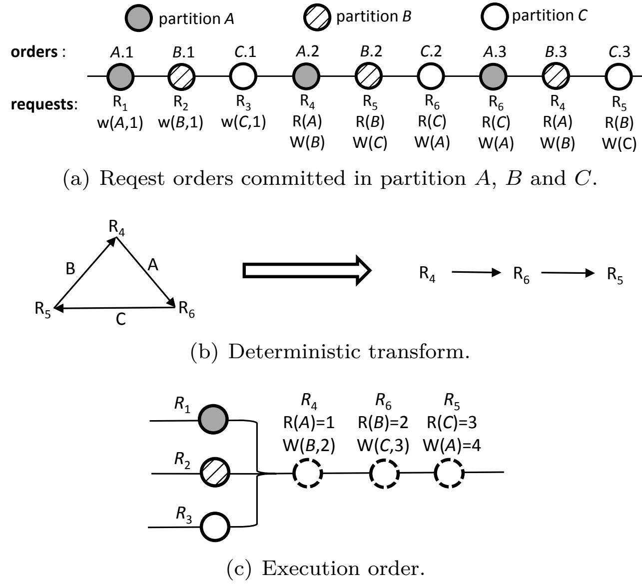
Dripple divides the system state (a.k.a., objects) into multiple partitions and coordinate partitions wisely. Different from the state-partitioning schemes discussed in existing work [65, 34], Dripple is not designed particularly for providing scalability, although existing methods are orthogonal to Dripple and can be easily applied with. Instead, each partition in Dripple is replicated at every replica.

In order to preserve strict serializability across partitions, Dripple dynamically constructs and analyzes a *dependency graph*, which is generated based on orders committed in every accessed partition (see an example below). In general, it is guaranteed that all requests are executed at every replica based on the same logical order. Nevertheless, independent, commutative requests can be executed out of order.

When Dripple is enabled, a client submits its request by contacting the source replica just like in Droopy. The source replica treats Droopy enabled SMR as a black-box and proposes the request in each partition that the request is going to access (i.e., read or write).

Upon a request is committed in every involved partition, each replica tries to execute the request based on topological orders defined in dependency graph. If there is a circular dependency, each replica deterministically executes requests sequentially in the loop.

An illustrative example is shown in Fig. 5.4. Assume there are 3 objects A, B and C, each in a different partition. The order in each partition is given by the format $partitionName.sequenceIndex$ (e.g., in partition A, $A.1 < A.2 < A.3$). The committed orders of 6 requests in these 3 partitions are presented in Fig. 5.4(a). For example, request R_1 is committed at $A.1$, which is prior to request R_4 committed at $A.2$. Since requests R_1, R_2 and R_3 access single and distinct partitions, they can be executed independently upon being committed at any replica. In contrast, requests R_4, R_5 and R_6 form a loop in dependency graph as shown in Fig. 5.4(b). More specifically, R_4 is committed before R_6 since they both access partition A and $A.2 < A.3$. Dependencies introduced by partition B and C are similar. As shown in Fig. 5.4(b), the deterministic order is $R_4 < R_6 < R_5$. Eventually, as shown in Fig. 5.4(c), R_1, R_2 and R_3 are executed in parallel, whereas R_4, R_6 and R_5 are executed sequentially



(a) Reqest orders committed in partition A, B and C.

(b) Deterministic transform.

(c) Execution order.

Figure 5.4: Dripple example.

afterwards at each replica. The execution order at each replica is equivalent to a sequential order, e.g., $R_1 < R_2 < R_3 < R_4 < R_6 < R_5$.

5.4.3 Fault-Tolerance

Since Droopy and Dripple are built upon existing single-leader or all-leader SMR protocols, they can perfectly rely on underlying protocols to tolerate failures such as machine crash. More specifically, Paxos and Mencius use a consensus algorithm [80] to guarantee that any request committed at a given sequence number will never be modified despite machine crash. Clock-RSM makes use of a reconfiguration protocol to remove faulty replicas from current membership. In some sense, our approaches improve the way that requests are ordered, not the way that makes requests reliable.

5.5 Protocol Details

In this section we detail our approaches. We first describe the way we design Droopy, i.e., how we dynamically re-configure the set of leaders. Then, we describe Dripple coordination algorithm in Sec. 5.5.2. Finally, in Sec. 5.5.3 we sketch the proof that Dripple ensures strict serializability.

5.5.1 Droopy

Object definitions given in Alg. 1 are used in Droopy pseudocode, which is given in Alg. 2.

Proposing. To issue request req, client c sends req to the closest replica s_i. Upon receiving req from c (line 1, Alg. 2), s_i becomes the source replica of

Algorithm 1 object definitions.

s_i, s_j, s_k : replicas
req : request from client c
src : source replica
sn : sequence number
ln : current lease number
LE_{ln} : the end index of lease ln
$config_{ln}$: the set of leaders in lease number ln
$replicas$: the set of n replicas in the system
$clock$: physical clock at replica s_i
$d_{*,*}$: latency table (from all to all replicas)
$freq_*$: the number of requests received by each replica
$latest_*$: the most recent sequence indices updated locally

Algorithm 2 Droopy: Dynamic reconfiguration at replica s_i.

abstract function GETORDER()
abstract function GETNEWCONFIG($freq, d$)

1: **upon** receive $\langle$REQUEST$, req, [s_j]\rangle$ from client c or s_k **do**
2: $src \leftarrow s_j = null\ ?\ s_i : s_j$ /* source replica */
3: $sn \leftarrow$ GETORDER()
4: **while** $sn \geq LE_{ln}$ **do** /* update lease */
5: $ln \leftarrow ln + 1$
6: **if** $s_i \in config_{ln}$ **then** /* leader */
7: PROPOSE(req, sn, src) in underlying SMR protocol
8: **else** /* non-leader */
9: sends $\langle$REQUEST$, req, src\rangle$ to $p_i \in config_{ln}$

10: **upon** DECIDE(req, sn, src) in underlying SMR protocol, in which UPDATED(sn) is applied and $true$ (see lines 23-24) **do**
11: $rep \leftarrow$ execute req
12: $freq_{src} \leftarrow freq_{src} + 1$
13: **if** $s_i = src$ **then** /* source replica replies to client */
14: send $\langle$REPLY$, rep\rangle$ to client c
15: **if** $LE_{ln} - \lambda \leq sn$ /* time to propose a new lease */
16: $config \leftarrow$ GETNEWCONFIG($freq, d$)
17: PROPOSE($ln + 1$,$config$,$LE_{ln} + \delta$) in L-Paxos
18: reset $freq_*$

19: **upon** DECIDE(ln',$config$,LE) in L-Paxos **do**
20: **if** $config_{ln'} = null$ **then** /* 1st decision for lease ln' */
21: $config_{ln'} = config$
22: $LE_{ln'} = LE$

23: **function** UPDATED(sn) /* returns boolean */
24: **return** $\forall s_k$ and ln : **if** $s_k \in config_{ln}$ **then** $latest[s_k] \geq LE_{ln}$ or $sn \leq latest[s_k]$

req (line 2). Then, s_i obtains and updates the sequence index by invocation GETORDER() provided by the interface to the underlying SMR protocol (see line 3 in Alg. 2, as well as lines 1-2 in Alg. 3). s_i may update the lease number if necessary (lines 4-5). If s_i is one of the leaders in current lease ln,

i.e., $s_i \in config_{ln}$, then s_i proposes req with the sequence index sn obtained; otherwise, s_i propagates req to its proxy $p_i \in config_{ln}$ which is expected to introduce the minimum commit latency with respect to s_i (lines 6-9).

Note that proxy p_i of replica s_i may not be the closest replica that s_i is aware of. For example, when $t = 2$, non-leader replica s_i can have the aggregated latency of message pattern $s_i \to s_1 \to s_2 \to s_i$ less than that of message pattern $s_i \to s'_1 \to s'_2 \to s_i$, even if the latency of $s_i \to s_1$ is larger than $s_i \to s'_1$. In this case, the proxy for replica s_i should be s_1.

Committing. In the commit phase, Droopy needs to modify the underlying SMR protocol at the condition which checks whether there are uncommitted requests preceding sequence index sn. If there is no uncommitted request with smaller index than sn, then the request at sn can be committed and executed. In particular, this check in all-leader protocols involves all replicas (leaders), whereas in Paxos it involves a single leader (which is trivial with FIFO assumption). In Droopy, this condition depends on which replicas are leaders in each lease. The modification is shown as function UPDATED at line 23 in Alg. 2.

Upon req is committed at replica s_i (line 10), i.e., UPDATED(sn) is true, all SMR protocols maintain the invariant that req is replicated by a majority of replicas and all preceding requests are committed locally at s_i. Then, replica s_i (1) executes req and replies response to client c if s_i is the source replica, and (2) updates frequency array f (lines 11-14).

Lease update. (also refer to Fig. 5.3) When current lease ln is about to to expire (in λ sequence indices, lines 15-18, Alg. 2), replica s_i first selects a new leader set $config$ based on frequency array f_* and table $d_{*,*}$; then s_i proposes $config$ piggybacked with lease number $ln + 1$ in L-Paxos and resets the frequency array $freq_*$. Upon $config$ in lease ln' is committed at replica s_i (lines 19-22), if $config$ is the first decision for ln', then s_i updates the leader set $config_{ln'}$ and the end index $LE_{ln'}$.

Latency update. (ignored in Alg. 2) Replica s_i periodically measures the round-trip latency from itself to others (e.g., using ping utility). Then, s_i updates the latency table $d_{i,*}$ and shares $d_{i,*}$ with other replicas by broadcasting message $\langle \text{LATENCY}, i, d_{i,*} \rangle$. Upon receiving $\langle \text{LATENCY}, i, d \rangle$, replica s_j replaces local $d_{i,*}$ by d.

Config selection. To select a suitable configuration for the next lease, replica s_i enumerates all possible combinations of sets of leaders.[3] Given a set of leaders, s_i calculates the estimated latency based on latency table $d_{*,*}$ and frequency array f_*.

The calculation of estimated latency depends on the internals of an underlying SMR protocol proceeds. For instance, to confirm that all preceding requests have been received, one-way latency from the farthest replica matters in Clock-RSM; whereas in Mencius, this latency can be (at most) a round-trip.

More specifically, the latency for req is dominated by three conditions : ① the time it takes for req to be proposed and further replicated by a majority; ② the time it takes for source replica s_i to confirm that no request with a sequence index smaller than that of req will be proposed by any leader (especially the

[3]For typical small values of n, a brute force algorithm is acceptable. Exceptionally with large n, one can make use of a greedy algorithm instead.

farthest leader); and ③ the time it takes for source replica s_i to commit all requests that precede *req*.

Based on these three conditions, calculation for each specific SMR protocol can be designed separately. As an illustration, a simple calculation for Clock-RSM is shown in line 4 in Alg. 3.[4] To commit a request, ① the source replica has to send the request to the proxy (itself or other site) and make the request replicated by a majority. Besides, ② the source replica has to confirm that the fastest replica will not propose new request with an earlier physical clock, by waiting for a clock message from that replica. Finally, ③ the source replica has to commit all previous requests proposed by other replicas.

Algorithm 3 Clock-RSM function.

1: **function** GETORDER()
2:     **return** *clock* /* physical clock */

3: **function** GETNEWCONFIG($freq, d$)
4:     **return** $s \subseteq replicas$ s.t.

$$min(\sum_{i=1}^{n} freq_i \times min(\\ max\begin{cases} d_{i,j} + median(d_{j,k} + d_{k,i} | \forall s_k), ① \\ d_{i,j} + max(d_{k,i} | \forall s_k \in s), ② \\ d_{i,j} + max((median(d_{k,l} + d_{l,i}) | \forall s_l) | \forall s_k \in s), ③ \end{cases}\\ |\forall s_j \in s) | \forall s \subseteq replicas)$$

5.5.2 Dripple

To simplify the exposition, we assume each Dripple partition contains exactly one object. Nevertheless, highly correlated objects should be grouped into one partition in practice. Either application developer knows in advance the approximate correlation among objects, thus the total number of partitions as well as the mapping from each object to it's partition can be pre-defined; or, partitions can be dynamically managed, e.g., by supporting merge and divide operations — the latter mechanism is out of the scope and is an interesting direction for the avenue of future work.

We say request *req* is proposed or committed in partition *p* if *req* is proposed or committed by the replicated state machine that manages partition *p*.

Proposing. Client *c* issues request *req* by sending *req* to source replica s_i (just like in Droopy). Upon receiving *req* (in Alg. 4 lines 1-2), s_i determines which partitions of state *req* will access; these comprise the *access set* of *req*, denoted by *set*(*req*). This step is accomplished by interacting with upper applications, e.g., through an upper-call provided by applications. For example, a transactional system by this interaction returns the set of partitions that contain the keys that *req* will read or write. In order to guarantee strict serializability [31] among all objects, *set*(*req*) should not exclude any objects that will be actually accessed, but can be a superset of them.

[4] This solution may not be the optimal one but just an illustration.

Algorithm 4 Dripple: multi-partition commit at replica s_i.

$set(req)$: the set of objects that req accesses
$CmtReq_{obj}$: the queue of committed requests that access obj
$OrderedReq$: $\{req | \forall obj : obj \in set(req) \rightarrow req \in CmtReq_{obj}\}$
$ExReq$: the set of requests that have been executed

1: **upon** receive $\langle$REQUEST$, req\rangle$ from client c **do**
2: obtain $set(req)$
3: **for** $\forall$ $obj \in set(req)$ **do**
4: PROPOSE(req, s_i) in partition obj

5: **upon** COMMIT(req, s_j) in partition obj **do**
6: append req to $CmtReq_{obj}$
7: **if** ORDERED(req) **then**
8: append req to $OrderedReq$

9: **upon** $\exists$ a set of requests $s \subseteq OrderedReq$ s.t.
(1) $\forall req \in s$ s.t. if $\exists req'$: PRE(req', req), then either $req' \in V$ or $req' \in ExReq$
(2) $\nexists s' \subset s$: s' satisfies (1) **do**
10: execute $\forall req \in s$ based on a deterministic order
11: remove $\forall req \in s$ from $OrderedReq$ and $CmtReq_*$
12: add req to $ExReq$

13: **function** ORDERED(req)
14: **return** $\forall obj \in set(req) : req \in CmtReq_{obj}$

15: **function** PRE(req, req')
16: **return** $\exists obj : req$ precedes req' in $CmtReq_{obj}$

Then, s_i processes req individually in each partition in $set(req)$ as described in Droopy (lines 3-4), i.e., by proposing req in each accessed partition (in parallel).

Committing. Upon req is committed in partition obj at replica s_i (lines 5-6), s_i inserts req into a *committed queue* for obj, i.e., $CmtReq_{obj}$.

Upon req is committed in every partition in $set(req)$ (lines 7-8), i.e., $\forall obj$: $obj \in set(req) \rightarrow req$ in $CmtReq_{obj}$, s_i inserts req into an ordered set $OrderedReq$. We say req is *ordered* in this case. Notice that req can be executed only if req is ordered.

Executing. In order to guarantee strict serializability, execution of ordered requests should follow logical orders defined by sequence indices in every accessed partition. Without across-partition coordination, execution of a request that accesses multiple partitions may violate serializability. For instance, if $req = \{write(A, 1), write(B, 2)\}$ (i.e., write object A to 1 and B to 2) precedes $req' = \{write(A, 2), write(B, 1)\}$ in partition A whereas req' precedes req in partition B, then two replicas may execute req and req' in opposite direction. We further define a relation between two requests in the following.

Definition 6. *Request req' **precedes** req, or* PRE*(req', req), if $\exists obj$ s.t. req' precedes req in partition obj, i.e., $\exists k : CmtReq_{obj}[k] = req'$ and $CmtReq_{obj}[k+1] = req$.*

We also say req depends on req' or req' is a preceding request of req. To execute ordered requests (line 9), replica s_i selects a set $s \subseteq OrderedReq$, which is named *minimum execution set* and satisfies the following conditions.

Definition 7. *A* ***minimum execution set*** *s is a set of ordered requests which satisfies*

(1) $\forall req \in s$ s.t. if $\exists req'$: PRE*(req', req), then either $req' \in s$ or $req' \in ExReq$;*

(2) $\nexists s' \subset s$: s' satisfies (1).

The first condition guarantees that each of req's preceding requests is either in s or in $ExReq$ (i.e., executed already). The second condition ensures that every replica will select the same s for each given request req.

Then, replica s_i executes requests in s sequentially based on a deterministic order (line 10). For instance, a simple scheme is executing requests in s based on their unique identifiers ($timestamp.clientId$).

Finally, s_i removes all executed requests from $OrderedReq$ and involved $CmtReq_*$ (line 11), and adds executed requests to the set $ExReq$ (line 12).

Read-only requests. As discussed in existing SMR protocols, read-only requests can be optimized particularly by read leases [42, 96]. With Dripple, a read-only request that involves only one partition can be executed by any replica which holds the read lease. However, if a read-only request involves more than one partition, the request has to be treated and executed as a normal read-write request.

5.5.3 Strict Serializability

In this section we prove the correctness of Dripple, i.e., Dripple ensures strict serializability. More specifically, we prove if eventually two replicas s_i and s_i' have executed all requests[5] and no request is further issued, then the execution orders at these two replicas are equivalent to the same sequential order.

At first, we construct a directed graph $G = (V, E)$ based on minimum execution set s, where $V = s$ and E contains all PRE relations between requests in V. Namely, if PRE(req, req') and $req, req' \in s$, then there is an edge from req to req' in E. Then we prove:

Lemma 2. *$G = (V, E)$ is strongly connected.*

A directed graph $G = (V, E)$ is strongly connected if and only if $\forall u, v \in V$, there is a path or "walk" from u to v, which are composed by edges in E.

We prove it by contradiction. $\forall req, req' \in V$, if there is no path from req to req', then we can assume there exists a subset $V_1 \subset V$ that req can walk to (by edges in E) and there exists another subset $V_2 \subset V$ that req can not walk to. Obviously, $req \in V_1$ and $req' \in V_2$. Furthermore, there is no PRE relation from any request req_1 in V_1 to any request req_2 in V_2, otherwise req_2 should be included in V_1. Hence, it's easy to see that V_2 satisfies Def. 7 (1) as well. Def. 7 (2) is violated.

Then we prove

[5] This assumption is guaranteed by availability of replicated state machine used in each partition.

Lemma 3. *if replica s_i have executed a collection C of k minimum execution sets and replica s'_i have executed a collection C' of k' minimum execution sets, then $C = C'$.*

Assume $\exists s \in C$ and $\exists req, req' \in s$, i.e., by replica s_i req and req' are executed within the same minimum execution set s. Based on Lemma 3, there exists a dependency path from req to req' and vice versa. Obviously, $\exists s' \in C'$ satisfying $req \in s'$. Then based on Definition 7 (1), req' must be in s' (it's impossible that req' is executed prior to s' or after s' since there is a dependency path from req' to req and vice versa, and $req \in s'$).

Although $C = C'$, replicas can execute minimum execution sets out of order. Now we prove by induction that

Lemma 4. *the execution order at each replica incrementally forms a Directed Acyclic Graph (DAG).*

A DAG [48] is a directed graph in which there is no directed cycle, i.e., there is no path that starts from any node $v \in V$ and goes through several edges in E and comes back to v. By the property of linear extension of DAG [48], different execution orders that follow the same DAG are equivalent, even if independent requests are executed out of order.

We consider the state after executing i minimum execution sets (no order required here) as a directed graph $G(V_i, E_i)$ $(i \geq 0)$, in which the set of nodes V_i contains all executed requests and the set of edges E_i contains all PRE relations among requests in V_i.

Each replica starts from a initial state with no request executed. We denote the initial state as $G(V_0, E_0)$, i.e., $V_0 = \emptyset$ and $E_0 = \emptyset$. Obviously, $G(V_0, E_0)$ is a DAG.

We assume $G(V_i, E_i)$ $(i \geq 0)$ is a DAG.

Upon the $(i+1)th$ minimum execution set $g = G(V, E)$ is ready to execute, either $|V| = 1$ (i.e., there is only one request in V), hence all of its preceding requests have been executed already (i.e., in V_i); or $|V| > 1$, then g is linearized deterministically into $G(V, E')$. By combining V and V_i, and by combining E (or E'), E_i and the set of all PRE relations from requests in V to requests in V_i, we form a new graph $G(V_{i+1}, E_{i+1})$. It's easy to observe $G(V_{i+1}, E_{i+1})$ is still a DAG, since we combine one DAG (i.e., $G(V_i, E_i)$) with another DAG (i.e., $G(V, E)$ or $G(V, E')$), plus the added edges are only in one direction, i.e., from V_i to V.

By Lemma 3, replicas s_i and s'_i have executed the same collection of minimum execution sets. PRE relations within a minimum execution set are decided deterministically based on request identifiers. Whereas, PRE relations among different minimum execution sets are determined by replicated state machine in each partition. Then by Lemma 4 and by property of DAG, the execution orders at s_i and s'_i are equivalent to the same sequential order.

5.6 Experimental Evaluation

In this section we evaluate the performance of D^{2}Clock-RSM by comparing it to state-of-the-art protocols using the Amazon EC2 worldwide cloud platform. We

start by presenting the way we implemented Droopy and Dripple in Sec. 5.6.1. Then, the experimental setup is described in Sec. 5.6.2. We then evaluate the performance (latency) of Droopy enabled Clock-RSM and compare it to native Clock-RSM and Paxos under imbalanced workloads (Sec. 5.6.3). Finally, by enabling Dripple as well, we compare D^2Clock-RSM with state-of-the-art protocols under three different types of balanced workloads (in Sec. 5.6.4).

5.6.1 Implementation

We implemented Droopy and Dripple in Java on top of Clock-RSM [55]. Besides, to enable comprehensive and fair comparison, we also implemented Paxos [81], Mencius [93], native Clock-RSM and EPaxos [95] by making use of the same code base for, e.g., communication layer and multi-threaded programming. We used TCP as the transport protocol to guarantee FIFO channels in all protocols.

Physical clocks. In native Clock-RSM and D^2Clock-RSM, replicas run NTP daemon to keep the clock synchronized. The timestamp is obtained by method $System.currentTimeMillis()$, which returns the current time in milliseconds. We use $System.currentTimeMillis()$ since (1) it returns the difference between current time and January 1st, 1970, which eases the maintenance of a global order among all replicas; and (2) it is updated with the help of NTP daemon. However, since $System.currentTimeMillis()$ is non-monotonic, we implemented a wrapper which maintains a monotonic return.

Minimum execution set. By state-partitioning, Dripple can easily exploit the benefits of multi-core architecture, i.e., by executing independent requests in parallel. This topic has been discussed by some existing work such as the ones in [71] and [63].

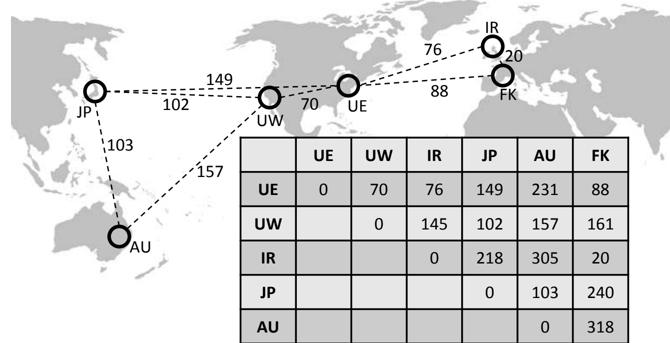
Upon request *req* is committed in partition *obj* (line 5 in Alg. 4), the thread which processes commit event first checks if *req* has been committed in every partition accessed (line 7 in Alg. 4). If this is the case, then the thread further checks if all requests precede *req* have been executed already. If the second condition is satisfied as well, *req* can be executed immediately. Otherwise, if only the first condition is satisfied, the thread (1) inserts *req* into set *OrderedReq*, and (2) checks recursively (by Depth First Search) if *req* and its preceding requests in *OrderedReq* can form a minimum execution set. If yes, then the thread executes all requests in the minimum execution set based on order defined by requests' identifiers (e.g., the format $timestamp.clientId$ which uniquely identifies each request).

Upon *req* is executed, the thread also checks (recursively) if the request that depends on *req* can be executed.

5.6.2 Experiment Setup

Both replicas and clients are deployed in instances on Amazon EC2 platform that comprises widely distributed datacenters, interconnected by the Internet. In each datacenter there is one virtual machine dedicated to a replica and one virtual machine for 40 clients.

We run the experiments on mid-range virtual machines named "c4.large" instances that contain 2 vCPUs, 3.75 GB of memory. The round-trip latency measured by ping utility among 6 EC2 sites is shown in Fig. 5.5.



	UE	UW	IR	JP	AU	FK
UE	0	70	76	149	231	88
UW		0	145	102	157	161
IR			0	218	305	20
JP				0	103	240
AU					0	318

Figure 5.5: Average round-trip latency (ms) among 6 Amazon EC2 sites used in our experiments: US East (UE), US West (UW), Ireland (IR), Japan (JP), Australia (AU) and Frankfurt (FK).

In Droopy, we set δ to 10 seconds and λ to 2 seconds, i.e., the length of each lease is 10 seconds, and 2 seconds before the expiration of current lease, each replica proposes a new config of leader set for the next lease. We believe this two numbers are sufficiently small to demonstrate the usability of Droopy.

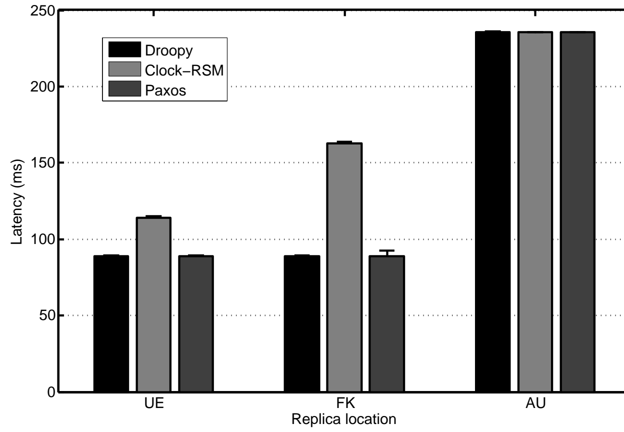
5.6.3 Imbalanced Workloads

In order to emphasize the benefits we can obtain with Droopy's dynamic reconfiguration, we first run Droopy enabled Clock-RSM (or simply Droopy) under *imbalanced* workload and compare it to native Clock-RSM and Paxos. Note that we consider Droopy enabled Clock-RSM as D^2Clock-RSM with only one partition. In this experiment, 40 clients at each specific site issue requests of 64 B to their source replica in closed-loop.

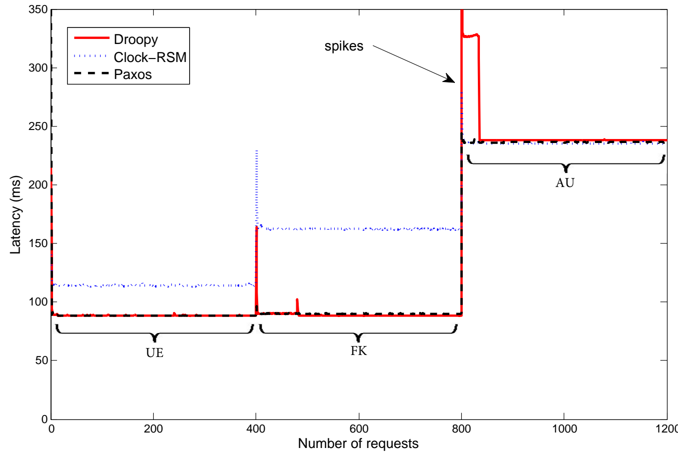
The results for $t = 1$ and $t = 2$ cases are shown in Fig. 5.6 and Fig. 5.7, respectively. To clearly demonstrate the latency in steady state, in Fig. 5.6(a) and Fig. 5.7(a) we manually ensure that the leader is at the same site as the workload (active clients), and we do the same setup and run at each site. Hence, there is no need to reconfigure the leader set in this experiment. In Fig. 5.6(b) and Fig. 5.7(b) we change the location of clients/workload from site to site and show how leader reconfiguration affects the latency. More specifically, upon 40 clients located at one site have executed all requests, clients at another site start issuing their requests.

We change the location of workload from UE (US East) to FK (Frankfurt) and AU (Australia) when $t = 1$, and from UE to UW (US West), FK, JP (Japan) and AU when $t = 2$. For fair comparison, we position the single-leader in Paxos in the (geographical) middle of all replicas. Thus, when $t = 1$ the single-leader is located at UE, whereas when $t = 2$ the single-leader is located at UW.

$t = 1$. From Fig. 5.6(a) we observe that Droopy effectively reduced latency compared to Clock-RSM when the workload is deployed at UE or FK. There are no benefits for workload coming from AU (Australia) as AU is located very far from the other two replicas. Hence, AU introduces the delayed commit problem for FK and UE. We also observe that in Fig. 5.6(a), both Paxos



(a) Average latency (bars) and 95%ile (lines atop bars).

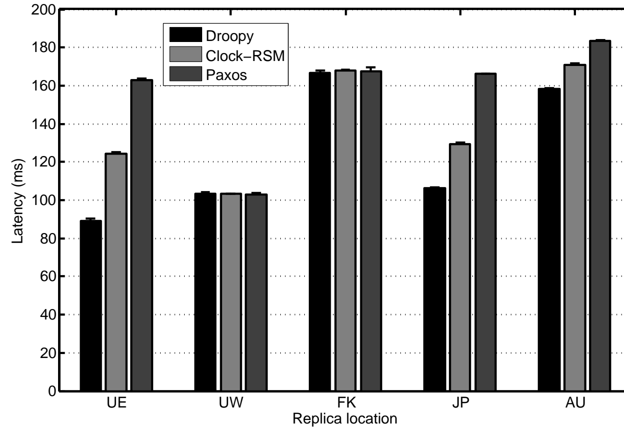


(b) Latency when the workload is moved from one site to another.

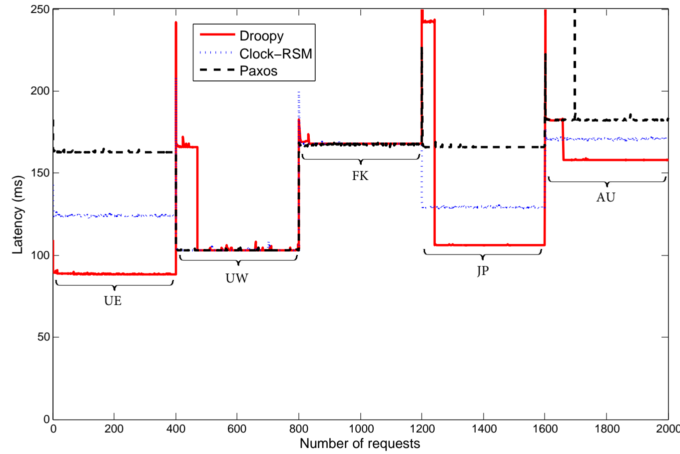
Figure 5.6: Latency (in ms, lower is better) of Droopy, Clock-RSM and Paxos under imbalanced workload when $t = 1$.

and Droopy achieved the optimal latency (i.e., a round-trip latency from the source replica to a majority). Replicas configured at UE, FK and AU can be considered as a special case since UE (which is the single-leader in Paxos) is located between FK and AU, where the round-trip latency from AU to UE plus the round-trip latency from UE to FK is approximately equal to the round-trip latency from AU to FK.

Fig. 5.6(b) demonstrates more clearly how latency changes with the workload movement and leader reconfiguration. More interestingly, as shown in Fig. 5.6(b), latency spikes occur when the workload is moved from FK to AU. This is because the leader set has not yet been adjusted based on the new workload, i.e., the leader is still located at FK. The latency spike did not occur when we moved the workload from UE to FK since sending requests from FK to UE (the old leader) and making them replicated by a majority (UE and FK)



(a) Average latency (bars) and 95%ile (lines atop bars).



(b) Latency when the workload is moved from one site to another.

Figure 5.7: Latency (in milliseconds) of Droopy, Clock-RSM and Paxos under imbalanced workload when $t = 2$.

is equivalent to the round-trip communication from FK to UE.

$t = 2$. In Fig. 5.7(a), when the workload is deployed at UE, JP and AU, we observe that the Droopy latency is reduced comparing to native Clock-RSM. In Clock-RSM, the lantecy at JP and AU is higher because of the delayed commit problem introduced by FK. And, similarly to $t = 1$ experiment, replica AU causes the delayed commit problem for the workload at UE. Paxos has the higher latency compared to other two protocols except when the workload comes from UW and FK.

Fig. 5.6(a) and Fig. 5.7(a) confirm our expectations that (1) neither single-leader nor all-leader protocols can achieve the optimal latency in all cases; and, (2) statically fixing the leaders at one or more replicas cannot guarantee the optimal latency at all times. For example, replica AU should be excluded from the leader set when the workload comes from UE. However, AU should be

included in the leader set when the workload is at AU. These two observations practically motivate Droopy, our leader reconfiguration protocol.

In Fig. 5.7(b), we observe that the latency spikes happened each time when the workload is moved from one site to another (except the movement from UW to FK), due to the delay of leader reconfiguration. However, latency spikes were settled in less than δ (i.e., 10s in our experiment). The spike during the workload movement from FK to JP is much higher than the steady case: this is because the leader was located at FK and the round-trip latency between JP and FK is around 240 ms.

Partially imbalanced workloads. To more comprehensively evaluate Droopy, we further deployed the three protocols (i.e., Droopy, Clock-RSM and Paxos) under partially imbalanced workloads, where 40 clients at UE and 10 clients at AU issuing requests to their source replicas. In Fig. 5.8 we show the latency distribution at two replicas (UE and AU) for each protocol. Very interestingly, under these workloads, Droopy reconfigured the leader set into UE and JP, not exactly the places where the workloads are deployed (i.e., UE and AU). At site UE, Droopy achieved much less latency than Clock-RSM, whereas at site AU Droopy has higher latency than Clock-RSM.

This choice is made consciously. If one of the leaders is located at AU, then extra delay that introduced by AU can affect the latency at UE dramatically. Since 4 times of clients are located at UE rather than AU, replacing AU by JP can mitigate the delayed commit problem introduced by AU, at the same time keeping one of the leaders closer to AU. Besides, we also observe from Fig. 5.8 that Droopy has similar latency as that of Paxos at AU. This is because in Droopy requests from AU are propagated to JP first, then proposed by JP and replicated by UW, and finally committed by AU. This is similar in Paxos, but in the opposite direction, where requests are propagated from AU to UW, then proposed by UW and replicated by JP, and finally committed by AU.

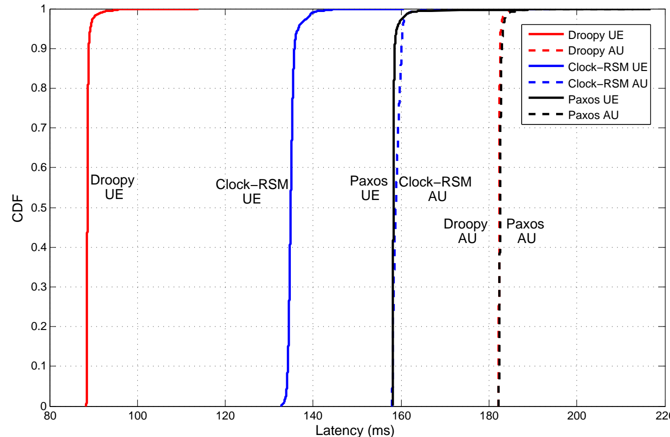


Figure 5.8: Latency distribution (in ms) at UE and AU when $t = 2$. 40 clients in UE and 10 clients in AU issue requests. The leader in Paxos is at UW. The leader set is reconfigured to UE and JP in Droopy.

Spare replica. As discussed in [68], adding spare replicas to geo-replicated

state machine can sometimes (depending on replica locations and workload) dramatically reduce latency, compared with the standard replica configuration where $n = 2t + 1$. We add one more replica (Ireland, IR) to the configuration used in Fig. 5.7, and evaluate how this modification affects the performance of Droopy, Clock-RSM and Paxos.

We deployed the three protocols under imbalanced workloads. The average latency and the latency change with workload movement are shown in Fig. 5.9. In Fig. 5.9(a), at site UE, FK, IR and JP we observe the latency is dramatically reduced by Droopy compared with native Clock-RSM, especially at UE, FK and IR. This is because site IR (the spare replica) eases these replicas to replicate their requests by $t + 1$. When the workload is deployed at IR, the latency is reduced by 56%. Comparing to Fig. 5.7(a), the latency at UE for Paxos is reduced as well. This is because the requests proposed by UE can be replicated by UW and IR more quickly than by UW and FK.

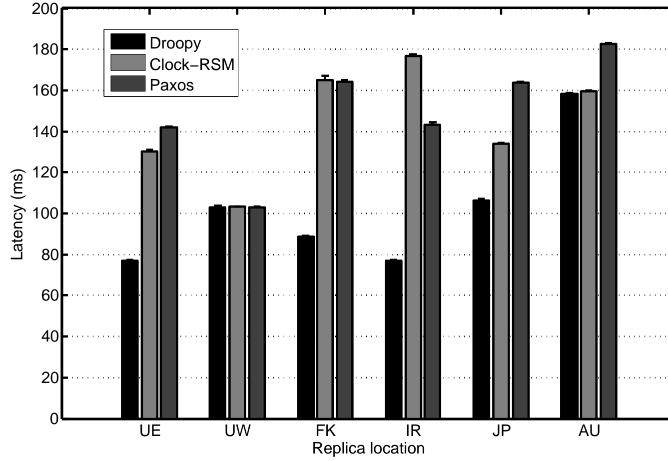
The average latency at IR for Clock-RSM is higher than we expected, where the expected value is based on the measurement by ping utility as shown in Fig. 5.5. More specifically, the round-trip latency from AU to IR is 305 ms, hence coordination cost in Clock-RSM, i.e., one-way latency from AU (the farthest site) to IR is supposed to be around 152 ms (rather than 176 ms in Fig. 5.9(a)). This latency is unexpectedly higher either because (1) the physical clock at IR is not tightly synchronized with other sites; or because (2) the one-way latency from some site (most possibly AU) to IR is not equal to the half of its round-trip value. We can not trivially distinguish these two reasons because we can not confirm if the physical clocks at these replicas are synchronized or not. This exceptional cost further amplifies the needs of Droopy, i.e., removing idle (and remote) leaders from the leader set, so that the long-time coordination caused by clock asynchrony can be avoided.

5.6.4 Balanced Workloads

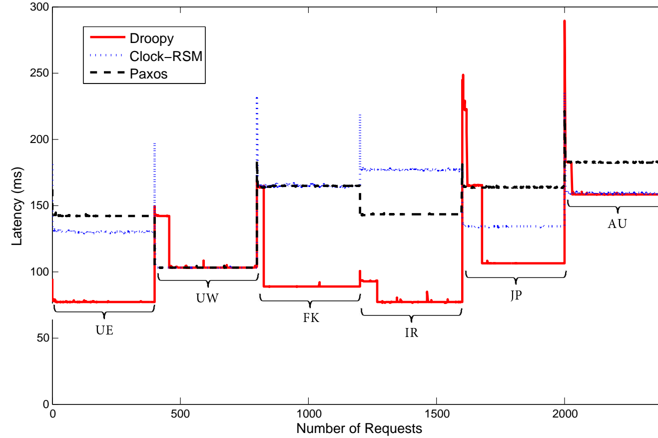
In this section we evaluate Droopy and Dripple enabled Clock-RSM (i.e., D^{2}Clock-RSM) under three types of *balanced* workloads, and compare D^{2}Clock-RSM with state-of-the-art protocols (Paxos, native Clock-RSM, Mencius and EPaxos). In each experiment, 40 clients at each site issue requests 64 B simultaneously to their source replicas in a closed-loop manner.

To explore request commutativity, we further set that requests issued by each client contain 10% multi-partition requests and 90% one-partition requests, where each partition contains 64 B payload. The number of accessed partitions for each multi-partition request obeys uniform distribution, ranging from 2 to 5. We believe these settings are representative since from the perspective of application developers, either each request accesses a very small portion of the whole state (e.g., large-scale key-value store or transactional storage); or the whole state itself is fairly small (e.g., a shared register). In the second case, the state should not be partitioned and Droopy enabled protocol is sufficient.

We illustrate the results when $t = 2$ in Fig 5.10. The leader in Paxos is located at UW. For a fair comparison, we did not enable the optimization in EPaxos that allows replicas replying to clients tentatively, upon requests are replicated by a fast-quorum but not yet executed. This optimization can be applied only if the application does not expect any reply. For example, put



(a) Average latency (bars) and 95%ile (lines atop bars).



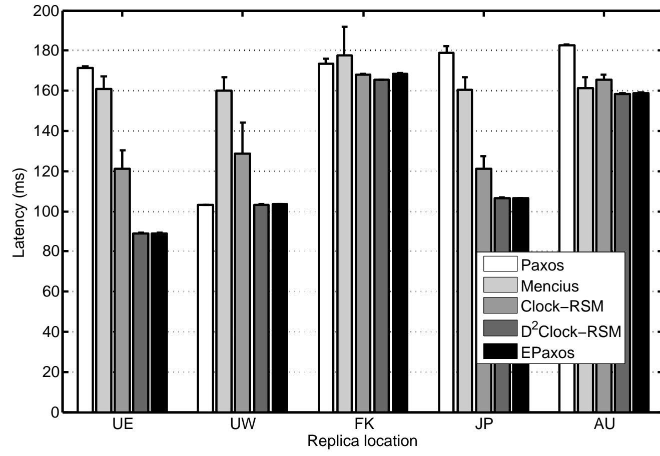
(b) Latency when the workload is moved from one site to another.

Figure 5.9: Latency (in milliseconds) of Droopy, Clock-RSM and Paxos under imbalanced workloads when $t = 2$ and $n = 6$.

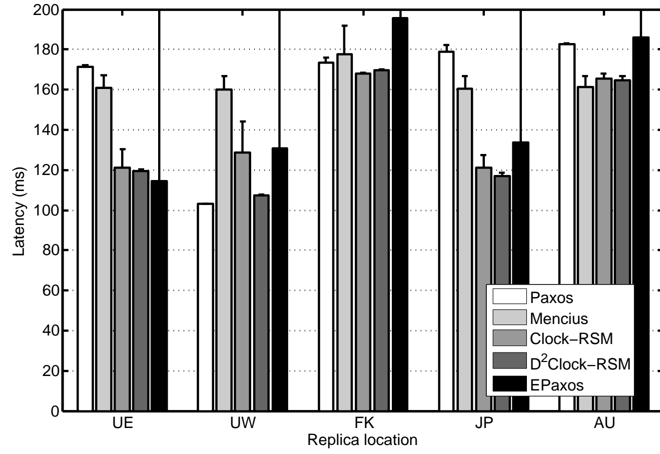
operations in key-value store can be applied with the optimization, but not for read-modify-write operations in transactional system.

Localized workloads. We first run all protocols under localized workloads, i.e., clients located at each site access partitions distinct from other sites. We set the total number of partitions to 1000 in D^{2}Clock-RSM (in EPaxos we set 1000 objects). For example, clients at site UE access partitions from 1 to 200 randomly, clients at UW access partitions from 201 to 400 randomly, and so on. We believe 1000 partitions is sufficient to demonstrate the usability of Dripple — to further provide scalability, applications could make use of existing work such as the one in [34] together with Dripple.

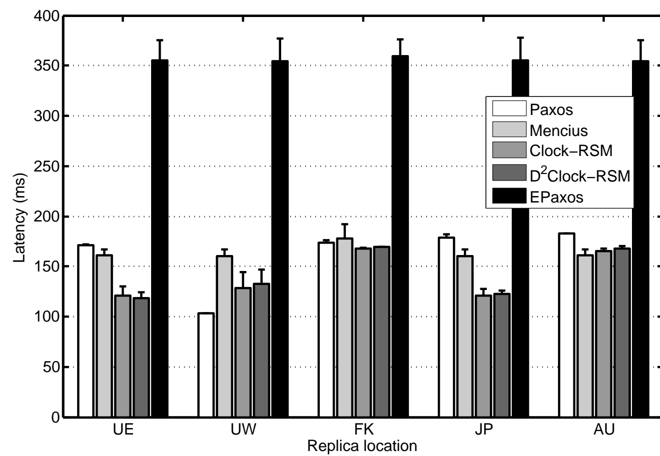
The results for localized workloads are shown in Fig. 5.10(a). D^{2}Clock-RSM and EPaxos have very similar performance and outperform other protocols



(a) Localized workloads.



(b) Uniform workloads.



(c) "Hot spot" workloads.

Figure 5.10: Average latency (bars) and 95%ile (lines atop bars) of D^{2}Clock-RSM and state-of-the-art protocols under balanced workloads.

at all sites. Localized workload can be considered as the best case for both D^{2}Clock-RSM and EPaxos. D^{2}Clock-RSM can reconfigure the leader in each partition to the source. Whereas, in EPaxos requests issued by different sites have no conflict at all, hence every request can be executed as soon as the request is replicated by a fast-quorum (when $t \leq 2$, a majority is equivalent to a fast-quorum in EPaxos). At UE, JP and AU D^{2}Clock-RSM achieves better performance than Clock-RSM as expected from the Droopy experiment we explained in Sec. 5.6.3.

More interestingly, at site UW D^{2}Clock-RSM reduced latency as well compared to native Clock-RSM. This is because dividing state into partitions can remove unnecessary dependencies among concurrent requests, as long as they access distinct partitions. In Paxos the latency is decided by the location of the single-leader, i.e., requests issued by UW have the optimal latency, whereas at other sites clients have to suffer from the additional latency needed to propagate requests to the remote leader.

Uniform workloads. We then deployed all protocols under uniform workloads, where each request randomly accesses one (90% probability) or 2-5 partitions (10% probability). The number of partitions in this workload is still 1000 (and 1000 objects for EPaxos).

The results are shown in Fig. 5.10(b). D^{2}Clock-RSM configures the set of leaders at all replicas, i.e., each partition is managed by a native Clock-RSM. Under uniform workloads, the native Clock-RSM and D^{2}Clock-RSM have the very similar performance except for the replica at UW. The reason for these results is similar to the one under localized workloads. D^{2}Clock-RSM removes unnecessary dependencies among concurrent requests that access distinct partitions. Among 1000 partitions, it's much less likely that requests issued by UW have to wait some concurrent requests issued by other sites to be committed at UW. EPaxos in this case has higher average latency and obviously much higher 95%ile latency than other protocols. This is because EPaxos may rely on another round-trip message exchange among a majority to resolve conflicts, and therefore the latency in the conflicting case is doubled.

"Hot spot" workloads. Finally, we deployed all protocols under the workloads which contain only 5 partitions (5 objects for EPaxos), i.e., the scenario that some small portion of state is accessed frequently by all requests. In this case, EPaxos has its average latency increased dramatically due to (1) the conflict resolution which relies on one more round-trip message exchange, and (2) the strong dependencies among all requests (i.e., each request has to wait a lot of other requests to be executed). These results are the worst case scenario for EPaxos as discussed in the original paper [95]. Clock-RSM and D^{2}Clock-RSM in this case have the similar performance and outperform other protocols (except the Paxos at UW).

Chapter 6

Conclusion

In this thesis we designed and implemented two state machine replication (SMR) protocols tailored for global-scale deployment. SMR is one of the fundermental building blocks for distributed systems, which stores critical state and data in a highly reliable way.

We first introduced XFT, a novel fault-tolerance model that allows 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 state of the art 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 (as discussed in Sec. 3.2.2).

In this thesis, 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 discussed in Sec. 3.2.3, 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 [45, 100] appear orthogonal to our XFT approach.

Visigoth Fault Tolerance (VFT) [100] is another recent extension 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.[1] In addition, VFT protocols have more complex communication patterns than XPaxos. That said, many of the VFT concepts remain orthogonal to XFT. In future, it would be very interesting to explore interactions between the hybrid fault model and its refinements such as VFT with our XFT.

Steward [21] is a rare BFT protocol specifically designed for wide-area networks. Steward uses a hierarchical architecture: it deploys BFT among replicas within a geographic site, and relies on CFT across sites for scalability. In contrast, we deployed XPaxos by applying XFT directly across sites. XFT could be combined with Steward's hierarchical approach — this is an interesting direction for future work.

Beyond some research directions outlined above, XFT opens more avenues for future work. For instance, many important distributed computing problems beyond SMR, such as distributed storage, deserve a novel look through the XFT prism. In addition, many startups such as Stellar [94], look at possibilities of achieving global scale BFT consensus for building "smart contract" (Bitcoin 2.0) blockchains. It would be very interesting to apply XFT and its advantages in the blockchain use case.

In the second contribution, we designed and implemented Droopy and Dripple, two sister approaches tailored for low-latency geo-replicated state machine. Droopy dynamically reconfigures the set of leaders based on previous workload and network condition. Dripple in turn divides the state into partitions and coordinates them wisely, so that the set of leaders of each partition can be reconfigured (by Droopy) separately. Our experimental results on Amazon EC2 show that, Droopy and Dripple practically provide near-optimal latencies under several realistic workloads.

Our contributions discussed in this thesis target the two different components of SMR, i.e., consensus and ordering, and we described them in a modular way. It would be very interesting to integrate XPaxos with Droopy and Dripple. That is, e.g., running several synchronous groups (in XPaxos) concurrently in order to serve clients located at distinct places. The number of concurrently running synchronous groups is dynamically reconfigured (by Droopy) and, furthermore, the state is divided and organized by Dripple.

[1] More specifically, XPaxos can tolerate, with $2t+1$ replicas, t network faults, t non-crash faults and t crash faults, albeit not simultaneously. Specifying such requirements in VFT would yield at least $3t+1$ replicas.

Résumé

1 Introduction

1.1 Contexte

Les applications Internet modernes sont largement déployées entre les centres de données différents qui enjamber sur les continents (alias site), afin de fournir un service efficace et fiable à travers le monde. Ce service comprend, entre autres choses, le courrier électronique [11, 15], le réseau social [9, 19], le partage de fichiers [6, 10], l'affaire en ligne [4, 1, 7] et le paiement sur Internet [18, 2]. Le modèle très général derrière est un système distribué qui contient un ensemble de processus qui communiquent et coordonnent les uns avec les autres pour atteindre un objectif commun [112].

Aujourd'hui, étant donné que chaque site peut contenir des milliers, voire des millions de serveurs avec une performance modérée, les échecs deviennent une norme plutôt que une exception. Ces échecs comprennent le arrêt brutal d'une machine, la corruption de mémoire ou disque, le bug matériel ou logiciel [14, 116], ou même la attaque malveillante et organisationnelle. La plupart de ces défaillances sont enregistrées publiquement qui ont été rencontrés par les applications et entraîné panne de service [49].

Par conséquent, le renforcement des systèmes distribués hautement fiables est la clé pour la survie des applications Internet modernes. Plus précisément, les applications applications Internet modernes ont plusieurs exigences: (1) la haute fiabilité, qui est généralement indiqué en pourcentage du temps, le système devrait être en place (alias disponible) et fonctionne correctement (alias cohérente), malgré la présence de fautes; (2) haute performance à l'égard de latence et le débit; et, (3) un coût modéré (l'utilisation des ressources).

Afin de tolérer fautes sophistiquées, les applications appellent des approches de réplication synchrone [23], qui en principe garantie “zéro de perte de données”. Parmi ces approches, la réplication de machines à états (State Machine Replcation, SMR) [102] est une méthode séminal et utilisée souvent pour maintenir une partie critique des données du système d'une manière cohérente et disponible, de sorte que la résistance du reste du système peut être basé sur elle. SMR modèle chaque processus comme un automate fini déterministe et garantit que chaque processus (correcte) déclenche des événements et fait état-transitions (par l'exécution des requêtes des clients) dans le même or-

dre. L'abstraction élégante d'approche SMR rend largement déployée par de nombreuses productions [36, 38]. Dans cette thèse, nous nous concentrons sur SMR.

Pendant les dernières décennies, de nombreux protocoles SMR ont été proposées, qui peuvent être grossièrement classés en tolérance aux fautes franches (Crash-Fault Tolerance, CFT), tolérance aux fautes Byzantines (Byzantine-Fault Tolerance, BFT) [86], ainsi que des modèles hybrides qui explorent l'espace entre. le modèle CFT objectifs du machines qui simplement arrêter le traitement des requêtes. Considérant que, le modèle BFT n'a pas de restriction sur le type de fautes et suppose pessimiste qu'un certain nombre de machines ainsi que la connexion réseau peuvent être contrôlés par un adversaire puissant.

1.2 Défis

Comme ciblage nouveau contexte, à savoir, le déploiement de service fiable à l'échelle planétaire, les concepteurs de protocoles SMR sont confrontés à de nouveaux défis: (1) la performance non uniforme (en ce qui concerne la latence et de bande passante) de connexions entre les centres de données répartis dans le monde, et (2) le compromis entre tolérer les fautes jamais sophistiqués et le grand coût et la complexité introduite (par exemple, les protocoles BFT existants) qui ont un impact mal son déploiement sur la géo-échelle.

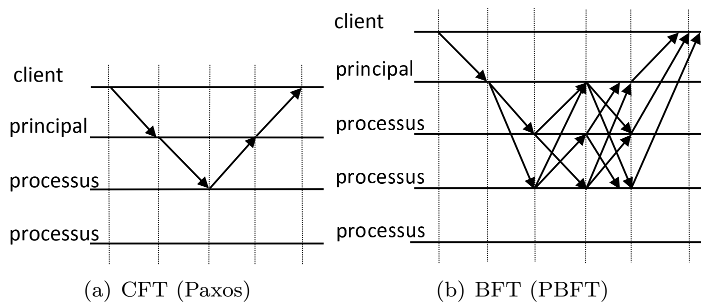
Les Connexions Non-uniformes Dans les Centres de Données Mondiales

Réplication devient contester à l'échelle planétaire, en grande partie en raison de la latence élevée et de la faible bande passante des connexions longue distance. A titre d'illustration, la latence aller-retour et la bande passante moyenne parmi 8 Amazon EC2 sites sont présentés au la Table 1, qui sont évalués par hping [12] et Netperf [16], respectivement. Il peut être observé que, la latence entre les sites distants est non uniforme et peut être aussi grand que plusieurs centaines de millisecondes (par exemple, celui entre le Japon et le Brésil), ce qui rend pratiquement le délai de communication au sein d'un centre de données (généralement moins de 1 ms) négligeable. Protocoles SMR classiques tels que Paxos, ont un seul principal qui est responsable pour le séquençage et de proposer les requêtes des clients. Pour les clients résidant sur un site distant par rapport à celle du principal unique, le temps de latence beaucoup plus élevé chez les différents sites implique des allers-retours très coûteuses.

En outre, la bande passante peut affecter les performances ainsi. Comme montré dans la Table 1.1, la bande passante entre deux machines virtuelles situées à distance (avec une performance modérée) varie de 50 Mb/s à 257Mb/s (exception de la connexion Californie et l'Oregon). Nous pouvons observer directement que, (1) la bande passante entre les différents sites est hétérogène aussi bien, et (2) la valeur absolue est beaucoup plus petite que celle des dispositifs de mise en réseau modernes, tels que les adaptateurs, les commutateurs et les routeurs de 1Gb/s ou 10Gb/s. Pour les systèmes qui sont conçus sur la base de réseaux symétriques (par exemple, à l'intérieur d'une grappe), le débit de l'ensemble du système est déterminée probablement par le maillon faible.

	Oregon	Virginie	Japon	Brésil	Irlande	Australie	Singapour
Californie	20 / 784	80 / 254	118 / 162	193 / 114	168 / 132	175 / 129	179 / 124
Oregon		75 / 193	114 / 153	199 / 96	155 / 124	216 / 99	208 / 90
Virginie			181 / 112	130 / 143	77 / 232	227 / 82	261 / 86
Japon				351 / 56	268 / 74	128 / 158	83 / 257
Brésil					238 / 89	325 / 51	366 / 56
Irlande						320 / 60	207 / 101
Australie							187 / 106

Table 1: Les latences aller-retour et la bande passante (ms/Mbps) dans plateforme Amazon EC2 mesurée sur Avril 2014; chaque machine est déployée sur l'instance c3.2xlarge.



(a) CFT (Paxos) (b) BFT (PBFT)

Figure 1: Échange de messages de protocoles CFT et BFT typiques lorsque $t = 1$.

Par conséquent, compte tenu des limites de performance, les chercheurs et les développeurs ont besoin de repenser la conception de protocoles SMR dans le nouvel environnement, dans le but de livrer leur service aussi vite que possible.

Le Compromis Entre Hypothèse Forte et Grande Coût

Depuis un large éventail d'erreurs qui ont récemment eu lieu [49], les développeurs d'applications peuvent avoir une motivation suffisante (dans un avenir proche) pour remplacer norme de facto pour SMR (Paxos), par un candidat plus fiable. En outre, puisque chaque machine physique en nuage public est partagé par plusieurs machines virtuelles, les applications qui sont déployées sont traités naturellement comme peu fiable et plus enclins à souffrir de fautes Byzantines.

Jusqu'à présent, les candidats bien étudiés sont protocoles dans le modèle BFT. Cependant, les protocoles BFT ont été déployé à peine dans des productions commerciales modernes (à l'intérieur ou entre les centres de données), principalement en raison de son coût élevé et Le complexe d'échange de messages comparant à des solutions CFT (par exemple, BFT nécessite $3t + 1$ réplicas à tolérer t fautes Byzantines simultanées, plutôt que $2t + 1$ dans le modèle CFT, cf. Fig. 1).

D'une part, pour faciliter le développement et la maintenance des applications modernes, les gens ont grandement besoin d'un protocole de réplication extrêmement fiable qui peut traiter de nombreux types de fautes qui ont eu lieu dans les systèmes réels. D'autre part, l'hypothèse puissant qui a un sys-

tème pour l'adversaire, peut avoir un impact mal le déploiement du système, en raison de la grande complexité et coût introduits.

1.3 Contributions

L'objectif de cette thèse est de faire progresser le conception de SMR dans le contexte géo-réplication. La contribution comprend (1) XFT, un nouveau modèle à la construction de protocoles SMR efficaces, qui traite des fautes de la machine et des fautes du réseau comme des événements indépendants; et, (2) deux approches soeur - Droopy et Dripple, qui réduire la latence en machine géo-répliqué à états en reconfigurant dynamiquement l'ensemble de principal et par l'état partitionnement.

XFT: Tolérance aux Pannes Pratique Au-delà Crashs

La première contribution introduit XFT ("cross fault tolerance"), une nouvelle approche pour la construction fiable et sécurisée des systèmes distribués. Protocoles BFT existants supposent une très mauvaise circonstance, dans lequel un puissant adversaire peut contrôler à volonté un certain nombre de machines ainsi que le réseau. Cette hypothèse forte que nous croyons est la principale raison qui fait que les protocoles BFT a peine adoptées par l'industrie. Cependant, en accord avec les observations de praticiens [77], les fautes byzantines expérimentés aujourd'hui se produisent de façon autonome des fautes du réseau (et aussi souvent de façon autonome les uns des autres). Considérant le compromis entre l'hypothèse forte et le coût important dans les protocoles SMR, XFT permet pour les deux fautes byzantines et les fautes du réseau, mais les considère comme indépendant.

Nous démontrons notre approche avec XPaxos, le premier protocole SMR en XFT. XPaxos utilise totalement $2t + 1$ réplicas de tolérer au plus t fautes Byzantines, à condition que la majorité des machines sont disponibles, c'est-à-dire, correcte et non partitionné. En cas commun, à savoir l'absence de faute de la machine et de faute du réseau, XPaxos demandes de réplication synchrone à travers des $t + 1$ réplicas spécifiques, qui sont organisés comme un groupe nommé *groupe synchrone*.

Un groupe synchrone est dirigé par un réplica distingué dite principal. Dans le cas où ces $t + 1$ réplicas ne peuvent pas faire des progrès sur le temps, un protocole de changement de vue est exécuté afin de sélectionner un nouveau groupe synchrone.

Outre les protocoles de cas commun et de changement de vue, nous introduisons deux mécanismes importants: (1) un mécanisme de détection de fautes qui peuvent détecter les fautes mortel si une majorité de réplicas sont disponibles; et (2) un mécanisme de réplication paresseux et coopérative qui explore l'hétérogénéité de la bande passante, afin de transférer l'état hors de groupe synchrone efficacement.

Notre analyse de la fiabilité montre que XPaxos est beaucoup plus fort que les protocoles CFT concernant la cohérence et la disponibilité, et est toujours plus fort que les protocoles BFT concernant la disponibilité. Dans certains scénarios spécifiques, XPaxos peut réaliser une cohérence encore plus fort que les protocoles de BFT existants. Par évaluation pratique dans Amazon EC2

nous montrons que XPaxos maintient la performance similaire à celle de Paxos, et surpasse nettement deux protocoles BFT représentatifs.

Principal Sélection Pour Latence Faible Réplication de Machines à États

La deuxième contribution vise l'amélioration de la latence perçue dans les protocoles géo-réplication. Protocoles géo-réplication existants peuvent être classés en deux catégories: (1) les protocoles tout-principal et (2) les protocoles sans-principal. Dans les protocoles tout-principal, chaque réplica peut agir comme un principal et séquence requêtes. Les protocoles tout-principal sont basés sur une hypothèse prudente qui requêtes sont toujours en conflit avec l'autre, donc un ordre total entre eux est nécessaire. Alors que, les protocoles sans-principal sont basés sur une hypothèse agressif que la plupart des requêtes sont commutative et peuvent être exécutés hors d'ordre, par conséquent, toutes les réplicas (ou clients) proposent requêtes de manière décentralisée. Toutefois, ni de procédés ci-dessus convient à toutes les circonstances. Les protocoles tout-principal souffrent de la problème de *commettre retardée* (Autrement dit, le réplica lent peut ralentir l'ensemble du système.), qui est introduit par l'exigence de coordination entre toutes les réplicas, même si parfois cette coordination n'est pas nécessaire. Alors que, dans les protocoles sans-principal, si les requêtes concurrentes proposées par réplicas différentes sont non-commutative, un ou plus échanges de messages sont plus tenus de résoudre les conflits, ce qui augmente considérablement la latence perçue.

Idéalement, ces requêtes non-commutatives devraient être séquencés à l'avance afin d'éviter de trop payer pour la résolution des conflits; et, ces requêtes commutatives devraient être séquencés indépendamment afin d'atténuer de la problème de commettre retardée.

Sur la base des considérations qui précèdent, nous proposons deux approches sœurs: Droopy et Dripple. Droopy explore l'espace entre les protocoles avec un seul principal (par example, Paxos) et les protocoles tous-principal par reconfigurer dynamiquement l'ensemble de principal. Chaque ensemble de principal est choisi en fonction de la charge de travail précédente et l'état du réseau. Dripple divise à son tour l'état en partitions et les coordonne de façon judicieuse, afin de découpler les dépendances inutiles entre les requêtes qui accèdent à des partitions distinctes. Avec Droopy et Dripple activés, chaque partition peut être maintenu et re-configuré séparément par Droopy.

Notre évaluation sur Amazon EC2 montre que, sous les charges de travail déséquilibrées ou localisées typiques, Droopy et Dripple réduit la latence par rapport à leur protocole natif, et ils maintiennent la latence similaire à celle d'un protocole sans-principal état de l'art. En revanche, sous les charges de travail équilibrées et non-commutatives, Droopy et Dripple maintiennent la latence similaire à celle de leur protocole natif, et ils ont tous deux surperformer un protocole sans-principal état de l'art.

1.4 Organisation

Cette résumé est organisé comme suit. Dans Sec. 2, nous introduisons le modèle de base et les concepts que nous utilisons dans la thèse. Sec. 3 présente notre première contribution XFT. Dans Sec. 4 nous décrivons notre deuxième con-

tribution — Droopy et Dripple. Nous concluons la thèse et discuter de certains travaux futurs dans Sec. 5.

2 Préliminaires

2.1 Modèle du Système

Nous présentons d'abord quelques notions générales et les hypothèses de systèmes distribués que nous utilisons dans la thèse.

Machines. Nous considérons un système distribué de passage de messages contenant un ensemble de n *machines*, aussi appelé *processus* or *réplicas*. Parmi lesquels, on suppose dans la plupart des t machines peut-être défectueux. Un réseau $\mathcal{N}$ relie les réplicas, où chaque paire de répliques peut envoyer des messages de manière fiable sur un canal de communication bidirectionnelle point à point. En outre, il y a un ensemble distinct C de clients, où chaque client peut communiquer avec des réplicas et d'exportation des opérations en faisant des requêtes.

Modèle de faute. Nous supposons une quelconque machine peut être défectueux. Nous distinguons entre les fautes franches, où une machine arrêts simplement tout calcul, et les fautes Byzantines, où une machine agit d'une manière arbitraire pour autant qu'elle ne se brise pas primitives cryptographiques, autrement dit, nous supposons un modèle byzantin authentifié. Une machine qui n'est pas défectueux est appelé correcte. Nous disons que le réplica est bénigne chaque fois qu'il est correct ou défectueux franchement.

Réseau. Nous ciblons un système réparti asynchrone dans la thèse. Le système est asynchrone en ce sens que les machines ne peuvent pas être en mesure d'échanger des messages et obtenir des réponses à leurs demandes en temps. En d'autres termes, les fautes du réseau sont possibles. Nous définissons précisément une *faute de réseau* comme l'incapacité des réplicas de faire communiquer entre eux en temps opportun, qui est, quand un échange de messages entre deux réplicas correcte ne peut pas être livré et traitée dans les délais Δ, connu pour toutes les réplicas.

2.2 Réplication de Machines à États

Dans cette thèse, nous nous concentrons particulièrement sur le problème de réplication de machines à états (State Machine Replcation or SMR) [102]. En bref, SMR assure la linéarisabilité [66] (ou consistance), d'un service déterministe et répliqué, en appliquant un ordre total entre les requêtes du client en utilisant un nombre illimité d'instances de consensus, de retour au client une réponse au niveau de l'application. Nous disons que le protocole SMR est disponible si un bon client peut livrer éventuellement ses requêtes.

En général, le protocole SMR imite un serveur centralisé qui ne manquera jamais ou se comporter de façon arbitraire. Chaque processus ou réplica $p_i \in$ *replicas* est modélisée comme un automate à états finis (machine à état), qui maintient un état actuel $state_i \in \mathcal{S}$. Chaque requête $req_i \in \mathcal{R}$ émise par les clients est modélisée comme une entrée de la machine à état, et chaque réponse $rep_i \in \mathcal{O}$ est modélisée comme une sortie de la machine à état. Autrement dit, il existe une fonction déterministe sortie de génération ω, qui prend un état spécifique $state_i \in \mathcal{S}$ et une requête $req_i \in \mathcal{R}$ que ses paramètres, $\omega : \mathcal{S} \times \mathcal{R} \rightarrow$

$\mathcal{O}$. En outre, il existe une fonction de transition déterministe d'état δ, qui prend les mêmes paramètres, $\delta : \mathcal{S} \times \mathcal{R} \to \mathcal{S}$.

Par exemple, si l'application est un magasin clé-valeur, chaque état contient toutes les paires clé-valeur, et chaque requête peut être opération de PUT ou GET. Opération PUT modifie l'état actuel et délivre un engagement simple (ou répond rien). Opération GET ne change pas l'état, mais doit retourner la valeur requise.

Pour réaliser cette abstraction, chaque processus commence avec le même état initial($state_0$) et exécute le même ensemble de requêtes déterministes. Il est important de garantir que, *si il n'y a pas de requête émise en outre, tous les processus (correcte) devrait se retrouver avec le même état après l'exécution de toutes les requêtes.*

Logiquement, il y a deux composantes dans les protocoles SMR: (1) un algorithme de consensus qui assure la fiabilité pour chaque requête unique malgré les échecs; (2) un composant de séquençage qui fournit un ordre total parmi toutes les instances de consensus.

Consensus. Le but du consensus est parvenir à un accord (la prise de décision cohérente) sur une valeur commune par un groupe de processus ou réplicas. Dans les protocoles SMR, rendant accord est de l'exécution de la même requête à une instance de consensus donné. En général, un algorithme de consensus garantit que chaque processus correct décidera de la même valeur, qui doit être proposé par un certain processus. Plus précisément, un algorithme de consensus garanties [37]:

- (*validité*) si une valeur est décidé par un processus, alors que la valeur devrait être proposée par un certain processus avant;
- (*terminaison*) tous les processus correct décider une valeur finalement;
- (*accord*) si la valeur v est décidé par le processus p et la valeur v' est décidé par le processus p', $v = v'$; et,
- (*intégrité*) aucun processus ne décide deux fois.

Toutefois, compte tenu modèle de fautes arbitraires (Byzantines), nous devons modifier les définitions de la validité et de l'accord, puisque nous ne pouvons pas restreindre le comportement des processus arbitrairement défectueux:

- (*validité BFT*) si une valeur est décidé par un processus *bénigne*, alors que la valeur devrait être proposée par un certain processus avant;
- (*accord BFT*) si la valeur v est décidé par le processus *bénigne* p et la valeur v' est décidé par le processus*bénigne* p', $v = v'$.

Pour satisfaire les propriétés ci-dessus dans un environnement asynchrone, c'est-à-dire, le retard du réseau n'est pas limitées, Modèle CFT exige $n = 2t+1$ processus afin de tolérer t fautes franches. Considérant que, modèle BFT exige $n = 3t + 1$ processus afin de tolérer t fautes Byzantines.

Séquençage. En tout protocole SMR, lorsque la requête est décidé à une instance de consensus, cette requête sera exécutée par la suite sur la base d'un

ordre défini par le composant de séquençage. Par example, dans Paxos, requêtes décidées sont exécutées séquentiellement sur la base des numéros de séquence logique. Certains protocoles, tels que ceux de [102, 55], avoir des requêtes séquencées par des horloges physiques. En outre, certains protocoles sans-principal exécuter concurremment les requêtes commutatives (par exemple, deux requêtes accès état distinct), donc pas d'ordre total est prédéfini dans ce cas. Au lieu de cela, le serializability [31] est assurée généralement.

Notre première contribution (XFT) vise l'amélioration de la composante de consensus. La deuxième contribution (Droopy et Dripple) est adapté pour l'optimisation de le composant de séquençage.

3 XFT: Tolérance aux Pannes Pratique Au-delà de Crashs

3.1 Introduction

Tolérer tout type d'interruption de service, qu'elle soit causée par un faute de matériel ou par une catastrophe de grande échelle, est la clé pour la survie des systèmes modernes distribués et a des implications directes sur l'entreprise derrière eux [75].

Productions modernes (par exemple, Google Spanner [47] ou Microsoft Azure [38]) augmenter le nombre de *neufs de fiabilité* [2] en utilisant les protocoles distribués sophistiqués qui tolèrent les fautes de arrêt brutal aussi bien que les fautes de réseau, qui refléter l'incapacité des machines correctes pour communiquer entre eux en temps opportun. Le cœur de ces systèmes est typiquement un primitif de tolérance aux fautes d'arrêt brutal [102, 42].

Ces systèmes ne peuvent pas tolérer les fautes Byzantines [86]), outre les comportements malveillants, que inclure les erreurs logicielles dans le matériel, les données périmées ou corrompues à partir de systèmes de stockage, les erreurs de mémoire causés par des effets physiques, les bogues dans les logiciels, les pannes matérielles dues aux circuits petits et plus rapides jamais, et les erreurs humaines qui causent des corruptions de l'Etat et la perte de données. Chacun de ces fautes a un dossier public de prendre vers le bas les principaux systèmes de production et de corrompre leur service, y compris ceux d'Amazon, Google, Facebook et d'autres [49, 27]. Cela ne surprend pas, puisque la probabilité qu'un "très rare" faute affecte de la machine devient significative quand on considère les grandes grappes jamais d'aujourd'hui, comprenant $n > 100,000$ des machines.

Malgré plus de 30 années de recherches intensives en calcul distribué depuis les travaux fondateurs de Lamport, Pease et Shostak [86], aucune réponse pratique à tolérer les fautes Byzantines n'a encore émergé. En particulier, tolérance aux fautes Byzantines asynchrone qui promet de résoudre ce problème [41] n'a pas répondu à cette attente, en raison de son surcoût par rapport à la CFT. Par example, dans le contexte de SMR asynchrone, cela implique en utilisant au moins $3t + 1$ réplicas tolérer t fautes Byzantines, à la place de $2t + 1$ dans le cas de CFT asynchrone.

[2] Pour illustrer, un système avec 5 neufs de fiabilité signifie que le système est en cours d'exécution correctement au moins 99.999% du temps. Cela correspond à un système qui, en moyenne, des dysfonctionnements pendant 1 heure tous les 10 ans.

Les frais généraux du asynchrone BFT est en raison de l'extraordinaire pouvoir donné à l'adversaire, qui peut contrôler à la fois les machines défectueuses et l'ensemble du réseau d'une manière coordonnée. En ligne avec les observations par les praticiens [77], nous affirmons que ce modèle d'adversaire est en fait trop forte pour les phénomènes observés dans les systèmes déployés: les fautes Byzantines expérimentés aujourd'hui se produisent indépendamment des fautes du réseau (et souvent aussi indépendamment les uns des autres), tout comme une faute de commutateur de réseau en raison d'une erreur logicielle a pas de corrélation avec un serveur qui a été mal configuré par un opérateur. L'attaquant puissant comme une source commune derrière ces fautes est une simplification populaire et puissant utilisé pour la phase de conception, mais il n'a pas vu la prolifération équivalent dans la pratique.

3.2 Contribution

Dans cette contribution, nous introduisons XFT (court pour "cross fault tolerance"), une nouvelle approche pour la construction des systèmes distribués efficace et fiables qui tolèrent les fautes Byzantines et les fautes du réseau. En un mot, XFT modélise les fautes d'arrêt brutal, les fautes arbitraires, et les fautes de réseau comme des événements indépendants, ce qui est très raisonnable dans la pratique. En particulier, XFT est idéal pour les systèmes reliés par un réseau étendu (c'est-à-dire, Wide Area Network ou WAN) et pour les systèmes de géo-répliqué, ainsi que des systèmes similaires dans lesquels des fautes de réseau et de la machine ne sont pas corrélés.

En bref, XFT permet de construire des systèmes fiables que:

- ne pas utiliser les ressources supplémentaires (réplicas) par rapport à CFT asynchrone;
- préservent toutes les garanties de fiabilité de CFT asynchrone; et,
- fournissent un service fiable, même en cas de fautes byzantines se produisent, aussi longtemps que la majorité des réplicas sont corrects et peut communiquer avec l'autre de manière synchrone (qui est, une minorité des réplicas sont défectueux ou partitionné en raison d'une défaillance du réseau).

Ceci permet le développement de systèmes fiables qui ont le coût de CFT asynchrone (déjà payé dans la pratique), mais avec une meilleure fiabilité décisive que la fiabilité CFT et parfois même mieux que totalement BFT asynchrone.

Comme la vitrine de XFT, nous présentons XPaxos, le premier protocole de réplication de machines à états dans le modèle XFT. XPaxos tolère fautes au-delà d'arrêt brutal d'une manière efficace et pratique, la réalisation de beaucoup plus grande couverture des scénarios de défaillance réalistes que les protocoles SMR état de l'art dans le modèle CFT, comme Paxos. [80]. Au-delà des garanties également fournies par Paxos et d'autres protocoles CFT, XPaxos tolère jusqu'à t fautes Byzantines lorsque la somme des fautes de la machine et des fautes de réseau à tout instant du temps ne dépasse pas t. Cela vient sans ressources et les frais généraux de performance, comme XPaxos utilise $2t + 1$ réplicas, tout en conservant à peu près les mêmes performances comme Paxos.

Étonnamment, quand en supposant que les fautes des machine et les fautes de réseau ne sont pas corrélées et tous les événements de défaillance sont indépendants et identiquement distribués variables aléatoires, XPaxos dans certains cas (en fonction de l'environnement de système), offre encore plus forte strictement garanties de fiabilité que les protocoles de réplication de BFT état de l'art.

Aligné avec la pratique de l'industrie, nous exprimons les garanties de fiabilité et de couverture des scénarios de faute par le nombre de neufs de fiabilité. Plus précisément, nous distinguons le nombre de neufs de disponibilité (pour propriété de vivacité) et le nombre de neufs de cohérence (pour propriété de sûreté) [20] et utilisons ces mesures pour comparer les modèles de faute différentes.

Pour valider la performance de XPaxos, nous déployons lui dans les centres de données Amazon EC2 à travers le monde. En particulier, nous avons déployé XPaxos au sein du service de coordination Apache Zookeeper. Notre évaluation montre que XPaxos effectue presque aussi bon que les protocoles CFT état de l'art sur la géo-échelle, surperformant significativement protocoles BFT état de l'art.

3.3 Modèle du Système

Fautes de réseau

Nous supposons que le modèle XFT est soumise à l'asynchronie, en ce que les clients, les réplicas et la communication peuvent ne pas être en temps opportun. Autrement dit, les fautes du réseau sont possibles. nous définissons les fautes de réseau que l'incapacité des réplicas corrects de communiquer en temps opportun. Plus précisément, nous déclarons une faute de réseau si un échange de messages entre deux réplicas correcte ne peut pas être livré dans les délais Δ, qui est connu pour toutes les réplicas.

Cependant, nous supposons que les fautes du réseau sont indépendants des fautes de la machine. Notez que nous considérons excessive délai de traitement à la machine comme un problème de réseau et non pas comme une question liée à un faute de la machine. Ce choix est fait consciemment, enracinée dans l'expérience que pour la classe générale des protocoles pris en compte dans ce travail, temps de traitement local est jamais un problème sur la machine correcte par rapport aux retards sur le réseau.

Il est clair que le choix d'un retard Δ dans le cadre du comportement du réseau et l'environnement du système détermine si des fautes de réseau se produisent effectivement.

Pour quantifier le nombre des fautes de réseau, nous donnons d'abord la définition ci-dessous.

Definition 8. *Machine ou réplica p est partitionné si p se trouve pas dans le plus grand sous-ensemble de réplicas, dans lequel chaque paire de réplica peut communiquer entre eux à l'intérieur de retard Δ.*

Nous disons réplica p est *synchrone* si n'est-ce pas partitionné. Notez que si il y a plus d'un sous-ensemble de taille maximale, un seul d'entre eux est reconnu comme le plus grand sous-ensemble. Par example (cf. Fig. 2), p_1 et p_2 peuvent communiquer entre eux au sein de retard Δ, mais ne peuvent pas

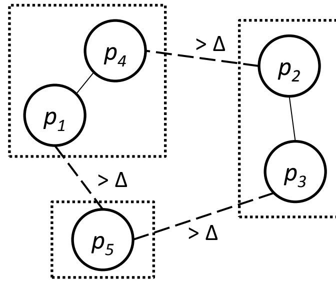


Figure 2: Illustration de réplicas partitionné: $\{p_1, p_4, p_5\}$ ou $\{p_2, p_3, p_5\}$ sont partitionnés sur la base Definition 8.

faire leurs messages livrés par d'autres réplicas dans le temps. Donc faire p_2 et p_3. p_5 est partitionné à partir des autres. Le nombre de réplicas partitionné dans ce cas est de 3, comptant soit groupe de p_1, p_4 et p_5 ou un groupe de p_2, p_3 et p_5. Notez que dans le pire des cas, le nombre de réplicas partitionné peut être $n - 1$, ce qui signifie, aucun de réplica peut communiquer avec tout autre réplica dans du délai Δ.

Ensuite, nous pouvons quantifier le nombre des fautes de réseau.

Definition 9. *Le nombre des fautes du réseau à un moment donné s est défini comme le nombre de réplicas correctes mais partitionné à s, c'est-à-dire, $t_p(s)$ ($0 \leq t_p(s) \leq n - 1$).*

Fautes de machines

Nous supposons que les réplicas Byzantines peuvent être coordonnés par l'adversaire; toutefois, l'adversaire ne peut pas facilement influer sur la communication entre les réplicas correctes ni briser primitives cryptographiques, elles que les signatures numériques, les digestions et le MAC.

Pour affiner la garantie de la fiabilité de XPaxos, nous quantifions le nombre des fautes d'arrêt brutal et les fautes Byzantines à un instant de temps par deux paramètres ci-dessous.

- $t_c(s)$: le nombre des réplicas d'arrêt brutal à un moment donné s ($0 \leq t_c(s) \leq n$); et,
- $t_{nc}(s)$: le nombre des réplicas Byzantins à un moment donné s ($0 \leq t_{nc}(s) \leq n$).

Combinant avec le modèle de faute du réseau, nous définissons que:

Definition 10. *Un réplica est disponible si le réplica est correcte et non partitionné.*

Anarchie

Enfin, nous définissons l'anarchie, que une condition de système très sévère avec réels fautes Byzantines et beaucoup de fautes de différents types, comme:

Definition 11 (Anarchie)**.** *Le système est en anarchie à un instant donné s si et seulement si* $t_{nc}(s) > 0$ *et* $t_c(s) + t_{nc}(s) + t_p(s) > t$.

Ici, t est le seuil de réplicas de fautes, tels que $t \leq \lfloor \frac{n-1}{2} \rfloor$. En d'autres termes, dans l'anarchie, certains réplicas sont Byzantins, et il n'y a pas de majorité synchrone et correcte.

3.4 Le Modèle XFT

		Le nombre maximum de chaque type de fautes		
		fautes Byzantines	fautes d'arrêt	réplicas partitionnés
CFT asynchrone (par example, Paxos [81])	cohérence	0	n	$n-1$
	disponibilité	0	$\lfloor \frac{n-1}{2} \rfloor$ (combiné)	
BFT asynchrone (par example, PBFT [41])	cohérence	$\lfloor \frac{n-1}{3} \rfloor$	n	$n-1$
	disponibilité	$\lfloor \frac{n-1}{3} \rfloor$ (combiné)		
XFT (par example, XPaxos)	cohérence	0	n	$n-1$
		$\lfloor \frac{n-1}{2} \rfloor$ (combiné)		
	disponibilité	$\lfloor \frac{n-1}{2} \rfloor$ (combiné)		

Table 2: Le nombre de chaque type de fautes tolérés par les protocoles SMR représentatives.

Cette section présente le modèle de XFT. CFT et BFT Classique explicitement modèle fautes de processus seulement. Ceux-ci sont ensuite combinés avec un modèle de faute de réseau orthogonal, soit le modèle synchrone (où les fautes de réseau dans notre sens sont exclues), ou le modèle asynchrone (qui comprend un certain nombre de fautes de réseau). Par conséquent, les travaux antérieurs peuvent être classés en quatre catégories: CFT synchrone [50, 102], CFT asynchrone [102, 80, 97], BFT synchrone [86, 53, 30] et BFT asynchrone [41, 26]..

XFT, au contraire, de redéfinir les frontières entre les dimensions de fautes de la machine et du réseau: XFT permet la conception de protocoles fiables qui tolèrent une classe de fautes de machines (faute d'arrêt brutal) quelles que soient les fautes du réseau, et qui, dans le même temps, tolérer un autre type de fautes de machines (par exemple, des fautes Byzantines) que lorsque le réseau est "assez synchrone", lorsque la somme des fautes de machines et des fautes de réseau est à l'intérieur d'un seuil. Plus précisément, XFT fournit des garanties de fiabilité avec tous les modèles de machines et de faute de réseau, sauf ceux qui mettent le système dans l'anarchie (voir Def. 11).

En un mot, XFT suppose que les fautes de la machine et du réseau dans un système distribué proviennent de causes différentes et indépendantes. Ceci améliore grandement la souplesse dans le choix des scénarios d'erreur pratiquement pertinents. L'intuition derrière XFT a commencé à partir de l'observation que "très mauvais" conditions du système, tels que l'anarchie, sont très rares et pourraient ne pas être la peine de payer la prime. Avec XFT, par exemple, nous pouvons concevoir un protocole SMR pratique (XPaxos) qui utilise seulement $n = 2t + 1$ réplicas, mais tolère jusqu'à t réplicas Byzantines, tant que le système reste hors de l'anarchie.

Nous illustrons différences spécifiques entre XFT et CFT asynchrone et BFT asynchrone dans la Table 2 en termes de leur fiabilité (la cohérence et la disponibilité) des garanties pour SMR. Protocoles CFT asynchrone garantir

la cohérence, malgré quelconque nombre de réplicas d'arrêt brutal et malgré tout nombre de réplicas partitionné. Ils garantissent également la disponibilité lorsque la majorité des réplicas ($t \leq \lfloor \frac{n-1}{2} \rfloor$) sont corrects et synchrones (pas partitionné). Dès qu'une seule machine est Byzantine, protocoles CFT sont ni cohérents ni disponibles.

Protocoles BFT asynchrones modernes garantissent la cohérence, malgré quelconque nombre de réplicas d'arrêt brutal ou réplicas partitionnés et avec la plupart des réplicas Byzantins. Ils garantissent également la disponibilité avec jusqu'à $\lfloor \frac{n-1}{3} \rfloor$ fautes combinés, c'est-à-dire, chaque fois que plus de deux tiers des réplicas sont corrects et synchrones.

En revanche, notre XPaxos garantit la cohérence en deux modes: (1) sans fautes Byzantines, malgré quelconque nombre de fautes d'arrêt brutal et partitionnés (c'est-à-dire, tout comme CFT), et (2) avec des fautes Byzantines, lorsque la majorité des réplicas sont corrects et synchrones, c'est-à-dire, à condition que la somme de tous les types de fautes (fautes de machines ou réseau) ne doit pas excéder $\lfloor \frac{n-1}{2} \rfloor$. De même, il garantit également la disponibilité lorsque la majorité des réplicas sont corrects et synchrones.

D'après la Table 2, il est immédiat que XFT offre strictement plus fortes garanties de disponibilité que soit CFT et BFT, ainsi que des garanties de cohérence strictement plus forte que CFT.

3.5 Protocole

XPaxos est un protocole de réplication de machines à états nouvelle qui assure la cohérence et la disponibilité comme illustré dans le Table 2.

Plus précisément,

Theorem 2. *aussi longtemps que* $\forall$ *donnée instant* s*:* $t_{nc}(s) = 0$ *(aucune faute Byzantine) ou* $t_c(s) + t_{nc}(s) + t_p(s) \leq t$*, XPaxos assure la linéarisabilité (en, la sémantique d'une copie).*

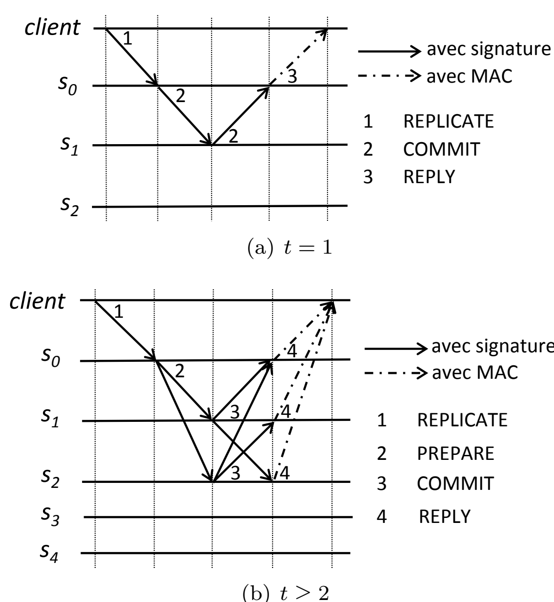
Nous esquissons la démonstration du théorème 2 dans Sec. 4.6.3 et formellement prouver à l'Annexe B. Basé sur le théorème 2, XPaxos renonce explicitement à fournir des garanties en vertu de l'état du système dans lequel il existe un faute Byzantine et la somme du fautes de la machine et du fautes des réseau dépasse t, alias, l'anarchie (tel que défini dans Def. 4).

En un mot, XPaxos compose de trois éléments principaux:

- un protocole commun cas, qui reproduit et totalement ordres demandes à travers des réplicas; ce qui a, en général, le modèle de message et la complexité de la communication entre les réplicas de protocoles CFT état de l'art (par exemple, la phase 2 de Paxos), durcie par l'utilisation de signatures numériques;

- un nouveau protocole de changement de vue, dans lequel les informations sont transférées d'une vue (configuration du système) à un autre de manière décentralisée; et,

- un mécanisme de détection de faute (FD), qui peut aider à détecter, en dehors de l'anarchie, fautes Byzantines qui quittent le système dans un état incohérent dans l'anarchie. Le mécanisme FD sert à minimiser l'impact



(a) $t = 1$

(b) $t \geq 2$

Figure 3: Échange de messages de commune cas de XPaxos quand $t = 1$ et $t \geq 2$. Groupe synchrone illustré sont (s_0,s_1) (quand $t = 1$) and (s_0,s_1,s_2) (quand $t = 2$), respectivement.

de fautes longue durée de vie dans le système et aider à détecter avant qu'ils ne coïncident avec des fautes de réseau et de pousser le système à l'anarchie.

XPaxos est orchestré dans une séquence de *vues* [41]. L'idée centrale dans XPaxos est que, dans une opération commune-cas dans une vue donnée, XPaxos réplique synchrone les requêtes des clients aux seuls $t + 1$ réplicas, qui sont les membres d'un *groupe synchrone* (sur $n = 2t + 1$ réplicas au total). Chaque numéro de vue i détermine de façon unique le groupe synchrone, utilisant un mapping connu pour toutes les réplicas. Chaque groupe synchrone se compose d'un *principal* et des t *adeptes*, qui sont appelés conjointement réplicas actifs. Restant t réplicas dans une vue donnée sont appelés réplicas *passive*; éventuellement, les réplicas passives apprennent l'ordre des réplicas actifs en utilisant l'approche de réplication paresseux. Une vue est pas modifiée, sauf si il y a une faute de machine ou une faute du réseau au sein de son groupe synchrone.

Dans le cas commune (cf. Fig. 3), les clients envoient les requêtes signés au principal, qui sont répliquées sur $t + 1$ réplicas actifs. Ces $t + 1$ réplicas signer numériquement et enregistrent les preuves (messages signés numériquement) pour toutes les requêtes répliquées à leurs registres de commettre localement. Le registre de commettre servent ensuite de base pour le maintien de la cohérence dans les changements de vue.

Pendant le changement de vue, tous les $t + 1$ réplicas actif du nouveau groupe synchrone essayer de transférer l'état de vues précédentes pour la vue $i + 1$. Cette approche décentralisée pour changement de vue est en contraste

frappant avec changement de vue classique dans les protocoles CFT et BFT (par example, [80, 41]), dans lequel un seule réplica (principal) entraîne le changement de vue et transfère l'état de vues précédentes. Cette différence est essentielle pour maintenir la cohérence (c'est-à-dire, la linéarisabilité) à travers XPaxos vues en présence de fautes Byzantines (mais en l'absence de la pleine anarchie), en dépit de répliquer seulement à travers $t + 1$ réplicas dans le cas commun. XPaxos nouveau et décentralisée régime de changement de vue garantit que, même en présence de fautes byzantines, tant qu'il n'y a pas assez réplicas d'arrêt brutal ou partitionnés, au moins une réplica correcte de la nouvelle vue sera en mesure de transférer l'état correct de la précédente vues, car il sera en mesure de communiquer avec des réplicas correctes de vues précédents (au moins un pour chaque vue précédente).

Enfin, l'idée principale derrière le régime de XPaxos FD est la suivante. Compte tenu du changement de vue, un réplica Byzantin (de l'ancien groupe synchrone) pourrait omettre de transférer son dernier état d'un réplica correcte dans le nouveau groupe synchrone qui peut transgresser la cohérence dans l'anarchie. Cependant, un tel faute peut être détecté et surmontée par l'envoi des registres de commettre de réplicas correctes (de l'ancien groupe synchrone), à condition qu'il y a réplicas correctes en vue précédentes et ils peuvent communiquer. Dans un sens, avec XPaxos FD, la faute Byzantine critique doit se produire pour la première fois ensemble avec assez réplicas d'arrêts brutal ou partitionnés (à savoir, dans l'anarchie) de violer la cohérence. Par conséquent, FD minimise la fenêtre de vulnérabilité de XPaxos et aide à prévenir les fautes Byzantines de persister assez longtemps pour coïncider avec les fautes du réseau ou d'arrêt brutal.

3.6 Évaluation expérimentale

Nous évaluons la performance de XPaxos et le comparer à Zyzzyva [73], PBFT [41] et une version optimisée de Paxos pour WAN [80], en utilisant la plateforme Amazon EC2 nuage dans le monde entier.

Installation Expérimentale

	Californie	Irlande	Tokyo	Sydney	Sao Paulo
Virginie	88 /1097 /82190 /166390	92 /1112 /85649 /169749	179 /1226 /81177 /165277	268 /1372 /95074 /179174	146 /1214 /85434 /169534
Californie		174 /1184 /1974 /15467	120 /1133 /1180 /6210	186 /1209 /6354 /51646	207 /1252 /90980 /169080
Irlande			287 /1310 /1397 /4798	342 /1375 /3154 /11052	233 /1257 /1382 /9188
Tokyo				137 /1149 /1414 /5228	394 /2496 /11399 /94775
Sydney					392 /1496 /2134 /10983

Table 3: Latence aller-retour de TCP ping (*hping3*) dans les centres de données Amazon EC2, recueillies au cours de trois mois. Les latences sont donnés en millisecondes, au format suivant: moyen / 99.99% / 99.999% / maximum.

Synchronie et XPaxos. Dans le déploiement de XPaxos, un paramètre critique est la valeur de la limite de temps Δ, c'est-à-dire, la limite supérieure de retard de communication entre deux machines correctes. Si la communication entre deux machines correctes prend plus que Δ, nous déclarons une faute du réseau. Notamment, ce délai est vitale à le changement de vue de XPaxos.

Pour comprendre la valeur de Δ dans notre contexte géo-répliqué, nous avons couru l'expérience de 3-mois au cours de laquelle nous avons mesuré en continu latence à travers six centres de données Amazon EC2 dans le monde entier utilisant le protocole TCP ping (hping3).

Les résultats de cette expérience sont résumés dans la Table 3. Alors que nous avons détecté des fautes de réseau pouvant durer jusqu'à 3 minutes, notre expérience a montré que la latence entre deux centres de données était de moins de 2,5 secondes en 99,99% du temps. Par conséquent, nous avons adopté la valeur de $\Delta = 2,5/2 = 1,25$ secondes, cédant $p_{synchrony} = 0.9999$ (à savoir, $9_{synchrony} = 4$).

Protocoles sous test. Afin de fournir une comparaison équitable, tous les protocoles reposent sur la même base de code Java. Nous comptons sur HMAC-SHA1 pour calculer un message codes d'authentification et RSA 1024 à signer et vérifier les messages.

Banc d'essai. Nous courons les expériences sur la plate-forme Amazon EC2 qui comprend les centres de données largement distribuées, reliés entre eux par Internet. Les communications entre les centres de données ont une bande passante faible et une latence élevée. Nous courons les expériences sur les machines virtuelles avec une performance modérée.

Les clients sont toujours situés dans le même centre de données que le principal pour mieux imiter ce qui se fait dans les systèmes de géo-réplication modernes où les clients sont servis par le centre de données le plus proche [105, 47].

ZooKeeper

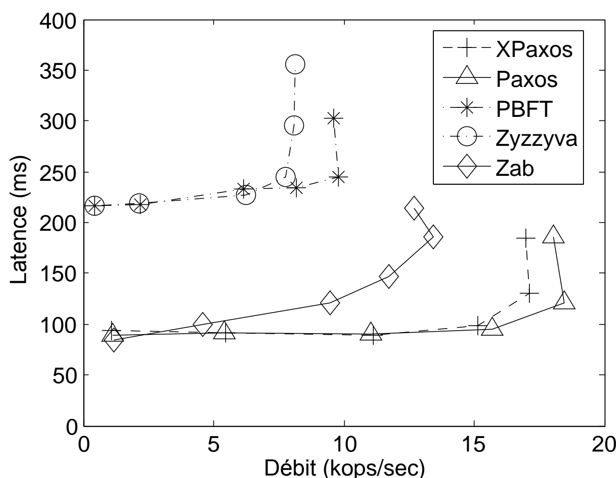


Figure 4: Latence vs débit pour l'application ZooKeeper ($t = 1$).

Afin d'évaluer l'impact de notre travail sur les applications, nous avons mesuré les performances réalisées lors de la réplication du service de coordination de ZooKeeper [67] en utilisant tous les protocoles inclus dans cette étude: Zyzzyva, PBFT, Paxos et XPaxos. Nous compare également la performance avec ZooKeeper native, lorsque le système est reproduit en utilisant un proto-

cole de réplication intégré, appelé Zab [69]. Ce protocole est arrêt-résilient et nécessite $2t + 1$ réplicas à tolérer t fautes.

Nous avons utilisé la base de code ZooKeeper 3.4.6. L'intégration des différents protocoles intérieur ZooKeeper a été réalisée en remplaçant le protocole Zab. La configuration testée tolère un faute ($t = 1$). Clients Zookeeper étaient situés dans la même région que le principal (CA). Chaque client invoque les opérations d'écriture 1kB dans une boucle fermée.

Fig. 4 représente les résultats. L'axe des X représente le débit en kops/sec. L'axe des Y représente la latence en ms. Nous observons que Paxos et XPaxos surpassent considérablement protocoles de BFT et XPaxos atteint une performance proche de celle de Paxos. Plus surprenant, nous pouvons voir que XPaxos est plus efficace que le protocole intégré Zab, bien que ce dernier ne tolère pas la faute byzantine.

4 Principal Sélection Pour Latence Faible Réplication de Machines à États

4.1 Introduction

Comme nous l'avons discuté dans Sec. 1, applications Internet font usage d'un protocole de réplication de machines à états comme une synchronisation de base primitive de fournir un service fiable. Réplication devient toutefois difficile à l'échelle planétaire en raison de latences qui ne peuvent être masqués, étant déterminé par la vitesse de la lumière.

Pour illustrer le problème, envisager des protocoles SMR classiques tels que Paxos qui ont un seul principal chargé de séquencer et proposer les requêtes des clients. Ces requêtes proposées sont ensuite reproduites sur au moins une majorité des réplicas, exécutée dans l'ordre de leurs numéros de séquence logique, avec répond finalement envoyé aux clients. Pour les clients qui sont à distance du principal, cela peut impliquer des allers-retours coûteux à travers le monde.

De nombreux protocoles SMR pour optimisation du temps de latence ont été proposées récemment [82, 106, 95, 93, 55]. Ces protocoles sont similaires en ce qu'elles reposent sur un (variante de) l'algorithme de consensus qui assure la fiabilité, malgré les différents types de fautes. Leurs différences ont à voir avec la façon dont ils assurent la linéarisabilité [66], c'est-à-dire, un ordre total entre les requêtes de tous les clients. En général, ces protocoles peuvent être classés en deux catégories: protocoles tout-principal et protocoles sans-principal.

En protocoles tout-principal, le ordre total de requêtes est pré-établies sur la base, par exemple, des numéros de séquence logique (par exemple, Mencius [93]) ou horloges physiques (par exemple, Clock-RSM [55]). Une requête peut être proposé et séquencé par une réplica, où chaque réplica peut agir comme un chef de file, généralement en cloisonnant l'espace d'index de séquence. Dans ces protocoles, un client soumet sa requête à une réplica à proximité, évitant la communication avec un seule (et éventuellement à distance) principal. Tout comme Paxos, protocoles tout-principal travailler avec l'hypothèse prudente que les requêtes de clients ne commutent pas — par conséquent protocoles tout-principal nécessitent encore la coordination des réplicas de maintenir un ordre total sur l'ensemble des requêtes.

Un défi pour les protocoles tout-principal est la coordination des réplicas lointaines ou lents, qui peut être le goulot d'étranglement, ou même bloquer les progrès entièrement. Dans un certain sens, la performance est déterminée par la réplica lente: ce qui provoque ce qui est connu comme le problème de "commettre retardé" [93]. En général, le problème de commettre retardé découle de la nécessité de confirmer que toutes les requêtes avec un numéro de séquence plus tôt ne sont pas manquées. Par exemple, pour les charges de travail déséquilibrées, la plupart des requêtes proviennent de clients qui gravitent à un site donné S, ceci induit des communications avec toutes les réplicas notamment celles éloignées et lents. Dans ce cas, protocoles tout-principal peut avoir une performance pire que protocoles avec seul principal, dans lequel la réplication implique seulement S et la majorité des sites les plus proches de S.

D'autre part, les protocoles sans-principal (par exemple, Generalized Paxos [82] ou EPaxos [95]) exploitent la commutativité possible de requêtes simultanées et exécutent ces requêtes (par exemple, les requêtes accès parties distinctes de l'etat) pas en ordre à des réplicas différentes. Dans un protocole sans-principal, une requête est proposé directement par un client ou réplica, et commis avec une latence aller-retour aussi réplicas appartenant à un quorum rapide[3]. Chaque réplica dans un quorum rapide vérifie le conflit potentiel entre les requêtes simultanées. Malheureusement, si (1) le conflit est détecté, ou (2) réplicas disponibilité sont moins d'un quorum rapide, un ou retard aller-retour plus supplémentaire pour une majorité de réplicas est introduit pour résoudre les conflits. Notez que les protocoles sans-principal ne souffrent pas de ce problème de commettre retardé depuis aucun ordre est pré-défini.

En résumé, il n'y a pas de protocole qui correspond à toutes les situations et le choix du meilleur protocole dépend en grande partie de la spécifié configuration des réplicas et la charge de travail. Les protocoles existants sont basés sur l'une des hypothèses suivantes: (1) les requêtes proviennent principalement de l'intérieur de la proximité d'un site unique (favorisant Paxos classique); (2) les requêtes sont réparties uniformément sur tous les sites et la plupart des requêtes simultanées ne commutent pas (favorisant protocoles tout-principal); ou (3) la plupart des requêtes simultanées commutent (favorisant protocoles sans-principal). Plus souvent, aucune de ces hypothèses est toujours vrai en pratique. Par exemple, en raison des différences de fuseaux horaires, les clients situés sur un site donné peuvent avoir différents modèles d'accès à différents moments de la journée, changeant dynamiquement la popularité des sites et peut-être aussi la charge de travail.

Dans cette contribution, nous présentons Droopy et Dripple, deux approches sœurs qui explorent la manière pour atténuer les problèmes des protocoles tout-principal et sans-principal. Les considérations de conception derrière Droopy et Dripple sont simples:

1. retirer la coordination et dépendances inutiles pour éviter de commettre retardé (dans protocoles tout-principal); et,

2. pré-commande les requêtes non-commutatives pour éviter la résolution des conflits (dans protocoles sans-principal).

[3]Un quorum rapide est plus grande que la majorité

Basé sur ces deux directives, Droopy est conçu pour reconfigurer dynamiquement l'ensemble de principal, qui peuvent contenir quoi que ce soit à partir d'un unique à toutes les réplicas. La sélection de principal est basée sur la charge de travail précédente et l'état du réseau. Dripple à son tour est un algorithme de état partitionnement et coordination inspiré par des protocoles sans-principal. Dripple permet Droopy pour configurer l'ensemble de principal de chaque partition séparément, dans le même temps assurer de fournir une forte cohérence, c'est-à-dire, le serializability stricte [31] dans son ensemble.

Bien que Droopy et Dripple peuvent être appliquées à tout protocole de SMR, nous avons mis nos approches sur le dessus d'un protocole tout-principal état de l'art — Clock-RSM [55] (nous appelons notre variante D^2Clock-RSM). Sur la plate-forme Amazon EC2 nous évaluons D^2Clock-RSM sous les charges de travail déséquilibrée et les trois types de charges de travail équilibrées. Pour une évaluation plus complète, nous avons également mis en place plusieurs protocoles de l'état de l'art — Paxos, Clock-RSM natif, Mencius et EPaxos [95] en faisant usage de la même base de code. Notre évaluation expérimentale montre que, en vertu de la charge de travail déséquilibrées ou commutative, Droopy permis Clock-RSM réduit efficacement la latence par rapport à son protocole natif et maintient la performance similaire à celle d'un protocole sans-principal — EPaxos. Avec un réplica de réserve (soit un total de $n = 2t + 2$ réplicas de tolérer les fautes t de collision), Droopy réduit la latence d'au plus 56% (comparaison à Clock-RSM indigène). En revanche, dans les charges de travail équilibrées et non-commutatives, D^2Clock-RSM a la latence similaire à celle de l'indigène Clock-RSM, et à la fois à atteindre la latence inférieure à EPaxos.

4.2 Présentation du Protocole

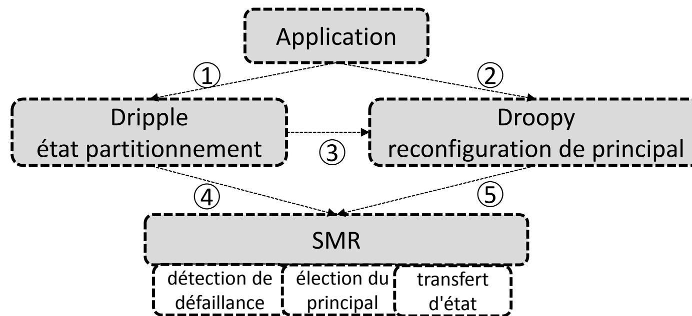


Figure 5: Architecture de Droopy et Dripple.

L'architecture de Droopy et Dripple est montré dans la Fig. 5. Nos approches sont construites sur des protocoles tout-principal existants et fournissent les mêmes interfaces pour des applications supérieures.[4] Si Droopy est activé seul, le chemin ②→⑤ dans Fig. 5, le nouveau protocole explore l'espace entre protocoles avec un seul principal et protocoles tout-principal, c'est-à-dire, sélection de l'ensemble de principal dynamiquement d'un à tous. Si Dripple est activé

[4] Dripple requiert en outre des informations sur les partitions accédé.

seul, le chemin ①→④ dans Fig. 5, le nouveau protocole est l'état partitionné, avec le principal de chaque partition statique situé à un ou toutes les réplicas. Enfin, si les deux Droopy et Dripple sont activés, le chemin ①→③→⑤, le nouveau protocole est l'état partitionné, avec l'ensemble de principal de chaque partition configurée dynamiquement.

Droopy

Droopy est conçu pour reconfigurer dynamiquement l'ensemble de principal. Chaque ensemble de principal est appelé un *config*. D'une manière générale, Droopy scissions l'espace d'indices de séquence (par exemple, des numéros de séquence logiques ou horodateurs physiques) en plages. Chaque gamme d'indices est mappé sur une config et maintenue comme un bail. Il est important de noter que, même si un bail peut refléter une plage de temps physique dans les systèmes basés sur l'horloge, anomalies de temps tels que les dérives d'horloge, ne touchent pas l'exactitude de Droopy, mais seulement sa performance.

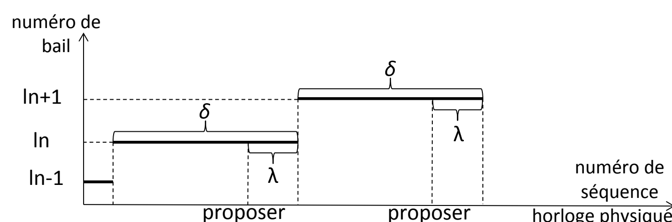


Figure 6: Renouvellement du bail.

En un mot, un bail en Droopy est un engagement d'un sous-ensemble de réplicas à une gamme d'indices de séquence, de telle sorte que seules les réplicas spécifiques, c'est-à-dire le principal, peuvent proposer des requêtes. Les baux sont proposés par des réplicas et entretenus par une primitive de ordre total, qui peut être mis en œuvre par Paxos classique (nous appelons cette *L-Paxos*). Dans un sens, Droopy suit l'approche de reconfiguration de réplication de machines à états proposée dans [84].

Lorsque le bail courant ln arrive à échéance (reportez-vous à la Fig. 6), c'est-à-dire, soit moins de λ indices de séquences sont disponibles, au moins $t + 1$ réplicas proposent une nouvelle configuration de principal pour la bail prochaine $ln + 1$, qui est choisi en fonction de la charge de travail précédente et l'état du réseau. Dans cette contribution, nous supposons que l'objectif de Droopy est de minimiser la latence moyenne sur tous les sites. Droopy peut facilement supporter d'autres critères tels que minimiser la latence pour certains sites donnés. Au moins $t + 1$ réplicas devrait proposer un nouveau bail de sorte que le crash d'une minorité de répliques ne sera pas arrêter le processus de renouvellement de bail. Toutefois, afin de garantir que tous les baux sont compatibles entre toutes les réplicas, Droopy exige que la première configuration livré pour chaque bail est celui accepté par tous. Ceci est garanti par la primitive ordre total au sein de L-Paxos.

Un client soumet sa requête en communiquant avec le réplica à proximité, que nous appelons le réplica de source de la requête. Lors de la réception de la

requête, le réplica de source propose la requête si il est l'un de principal dans le bail actuel. Sinon, le réplica de source propage le requête de l'un de principal qui agit comme un *proxy*. Pour chaque réplica non-principal, son proxy est le principal qui est prévu d'introduire le minimum de latence, à l'égard de le réplica non-principal. Lorsqu'une requête est engagé, le réplica de source envoie une réponse au client. En outre, chaque réplica met à jour son moniteur de fréquence, qui enregistre le nombre de requêtes provenant de chaque réplica de source. Le moniteur de fréquence est servi pour aider à la décision d'ensemble de principal. Chaque réplica mesure également périodiquement et partage la latence à partir de lui-même aux autres.

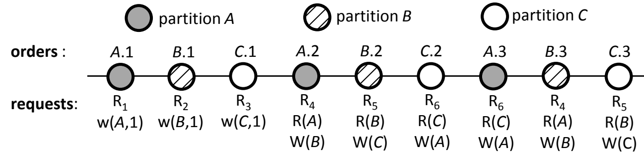
Dripple

Dripple divise l'état du système (alias objets) en plusieurs partitions et de coordonner les partitions à bon escient. Différent des régimes partitionnement étatiques discutés dans les travaux existants [65, 34], Dripple n'a pas été conçu en particulier pour fournir une évolutivité, bien que les méthodes existantes et Dripple sont orthogonales, par conséquent les méthodes existantes peuvent être facilement appliqués. Afin de préserver le serializability stricte entre les partitions, Dripple construit et analyse dynamiquement un graphe de dépendance, qui est généré en fonction des requêtes commis dans chaque partition consultée (voir un exemple ci-dessous). En général, il est garanti que toutes les requêtes sont exécutées à chaque réplica fondée sur le même ordre logique. Néanmoins, les requêtes commutatives peuvent être exécutées pas en ordre.

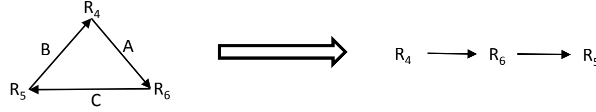
Lorsque Dripple est activée, un client soumet sa requête en communiquant avec la réplica de source comme dans Droopy. Le réplica de source friandises Droopy comme une boîte noire et propose le requête dans chaque partition que le requête va à l'accès (par exemple, lire ou écrire).

Sur requête est engagé dans chaque partition participé, chaque réplica tente d'exécuter la requête en fonction du ordre topologiques définies dans le graphe de dépendance. Si il y a de dépendance circulaire, chaque réplica exécuter séquentiellement les requêtes dans la boucle d'une manière déterministe.

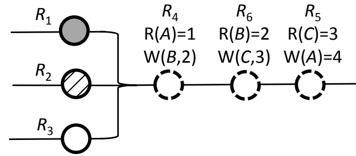
Un exemple illustratif est représenté sur la Fig. 7. Nous supposons il y a trois objets A, B et C, chacun dans une partition différente. L'ordre dans chaque partition est donnée par le format $partitionName.sequenceIndex$ (par exemple, dans la partition A, $A.1 < A.2 < A.3$). Les ordres fermes de six requêtes dans ces trois partitions sont présentés dans la Fig. 7(a). Par exemple, la requête R_1 est engagé au $A.1$, qui est antérieure à requête R_4 commis au $A.2$. Depuis requêtes R_1, R_2 et R_3 accès partitions distincts, elles peuvent être exécutées de façon autonome commis à tout réplica. En revanche, les requêtes R_4, R_5 et R_6 forment une boucle dans le graphe de dépendance comme le montre la Fig. 7(b). Plus précisément, R_4 est engagée avant R_6 puisque les deux accèdent la partition A et $A.2 < A.3$. Dépendances introduites par la partition B et C sont similaires. Comme représenté sur la Fig. 7(b), l'ordre déterminé est $R_4 < R_6 < R_5$. Finalement, comme le montre la Fig. 7(c), R_1, R_2 et R_3 sont exécutées en parallèle, alors que R_4, R_6 et R_5 sont exécutées séquentiellement par la suite à chaque réplica. L'ordre d'exécution à chaque réplica est équivalent à un ordre séquentiel, par exemple, $R_1 < R_2 < R_3 < R_4 < R_6 < R_5$.



(a) Commandes des requêtes commis dans la partition A, B et C.



(b) Transformer déterministe.



(c) Ordre d'exécution.

Figure 7: Example de Dripple.

Tolérance aux Fautes

Depuis Droopy et Dripple sont construits sur des protocoles SMR existants, ils peuvent parfaitement compter sur protocoles sous-jacents à tolérer les fautes. Plus précisément, Paxos et Mencius utilisent un algorithme de consensus [80] pour garantir que toute requêtes commis à un numéro de séquence donnée ne sera jamais modifié, malgré la faute d'arrêt brutal. Clock-RSM fait usage d'un protocole de reconfiguration pour enlever des réplicas défectueux de membres actuels. Dans un certain sens, nos approches améliorent la façon dont les requêtes sont ordonnées, pas la manière qui rend requêtes fiable.

4.3 Évaluation Expérimentale

Nous évaluons la performance de D^2Clock-RSM en la comparant à des protocoles de l'état de l'art à l'aide de la plate-forme Amazon EC2 nuage dans le monde entier.

Installation Expérimentale

Tous les deux réplicas et clients sont déployés dans les cas sur Amazon EC2 plate-forme qui comprend des centres de données largement distribuées, reliés entre eux par Internet. Dans chaque centre de données il y a une machine virtuelle pour un réplica et une machine virtuelle pour 40 clients. Nous courons les expériences sur des machines virtuelles milieu de gamme nommé "c4.large" qui contiennent 2 vCPU, 3,75GB de mémoire.

Nous implémentée nos approches basées sur Clock-RSM. Dans Droopy, nous obtenons δ à dix secondes et λ à deux secondes. Plus précisément, la longueur

de chaque bail est dix secondes; deux secondes avant l'expiration du bail actuel, chaque réplica propose une nouvelle config de principal pour la prochaine bail. Nous croyons que ces deux nombres sont suffisamment petites pour démontrer la facilité d'utilisation de Droopy. D'ailleurs, pour avoir une comparaison équitable, nous avons implémentée Paxos, Mencius et EPaxos en faisant usage de la même base de code.

Les Charges de Travail Localisées

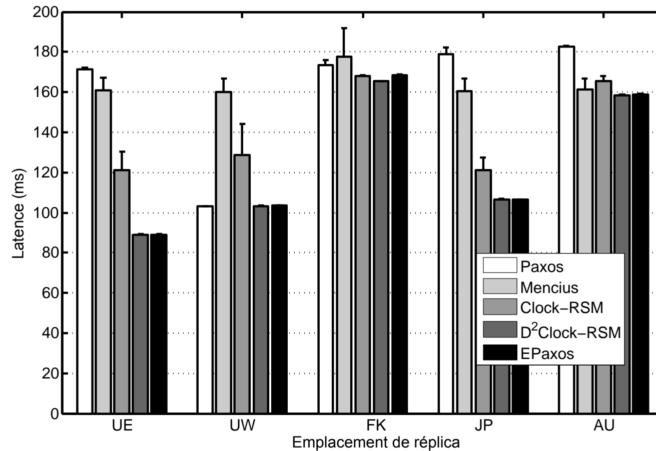


Figure 8: les charges de travail localisées.

Dans cette résumé, nous montrons comment ces protocoles se comporte sous les charges de travail localisées, c'est-à-dire, les clients situés à chaque site accès partitions distinctes à partir d'autres sites. Nous avons fixé le nombre total de partitions à 1000 en D^2Clock-RSM (en EPaxos nous avons fixé 1000 objets). Par exemple, les clients à la site UE accéder aux partitions de 1 à 200 au hasard, les clients à l'UW accéder aux partitions de 201 à 400 au hasard, et ainsi de suite. Nous croyons que 1000 partitions est suffisante pour démontrer la facilité d'utilisation de Dripple— de fournir un complément d'évolutivité, les applications pourraient faire usage de travaux existants, comme celui de [34] avec Dripple.

Les résultats pour les charges de travail localisées sont présentés dans Fig. 8. D^2Clock-RSM et EPaxos ont des performances très similaires et surpassent autres protocoles à tous les sites. La charge de travail localisée peut être considéré comme le meilleur des cas à la fois pour D^2Clock-RSM et EPaxos. D^2Clock-RSM peut configurer le leader dans chaque partition à la réplica de source. Considérant que, dans EPaxos, les requêtes émises par les sites différents n'ont pas de conflit du tout, donc chaque requête peut être exécutée dès que la requête est répliqué par un quorum rapide (quand $t \leq 2$, une majorité est équivalent à un quorum rapide à EPaxos).

Plus intéressant encore, sur le site UW D^2Clock-RSM réduite latence ainsi par rapport à Clock-RSM natif. Ceci est parce divisant d'état en partitions peut supprimer des dépendances inutiles entre les réplicas simultanées, si ils accèdent à des partitions distinctes. Dans Paxos la latence est décidé par

l'emplacement de le principal unique. Par conséquent, les requêtes émises par le site UW (où le principal unique situé) ont le temps de latence optimale, tandis que les clients dans d'autres sites ont à souffrir de la latence supplémentaire nécessaire à la propagation des requêtes au principal distance.

5 Conclusion

Dans cette thèse, nous avons conçu des approches adaptées pour la réplication de machines à états (SMR) à l'échelle mondiale. SMR est l'un des éléments fondamentaux pour les systèmes distribués, qui stocke les données critiques d'une manière très fiable.

Nous avons d'abord présenté XFT, un roman modèle de tolérance de panne qui permet conception de protocoles efficaces qui tolèrent les fautes byzantines. Nous avons démontré XFT travers XPaxos, le premier protocole de réplication de machines à états en XFT qui dispose de nombreux neufs plus de fiabilité que protocoles CFT, avec à peu près le même coût complexité de communication, performance et ressources. C'est-à-dire, XPaxos utilise $2t + 1$ réplicas et offre toutes les garanties de fiabilité de CFT, mais est également capable de tolérer des fautes Byzantines, aussi longtemps que la majorité des réplicas sont corrects et peut communiquer de manière synchrone entre eux.

Comme XFT est entièrement réalisée dans le logiciel, il est fondamentalement différente de l'approche établie qui repose sur du matériel de confiance pour réduire le coût des ressources de BFT à $2t + 1$ réplicas seulement.

Dans cette thèse, nous n'avons pas d'explorer l'impact de la variation du nombre de tolérer des fautes par classe de faute. En bref, cette approche, connue sous le modèle hybride, que distingue le seuil de fautes Byzantines (disons b) malgré quel sécurité doit être assurée, du seuil t de fautes (de toute catégorie) en dépit de disponibilité qui devrait être assurée (étaient souvent $b \leq t$). Le modèle de faute hybride et ses raffinements [45, 100] apparaissent orthogonale à notre approche de XFT.

Visigoth Fault Tolerance (VFT) [100] est une autre extension récente du modèle hybride. En plus d'avoir des seuils différents pour fautes Byzantines et d'arrêt brutal, VFT définit également l'espace entre la synchronie et l'asynchronie de réseau en définissant le seuil de fautes du réseau qui un protocole de VFT peut tolérer. VFT est cependant différente de XFT, en ce qu'elle fixe des seuils de faute distinctes pour les fautes Byzantines et de réseau. Cette différence est fondamentale, comme XFT ne peut pas être exprimée en choisissant des valeurs spécifiques de seuils en VFT. En outre, les protocoles VFT ont des modèles de communication plus complexes que XPaxos. Cela dit, la plupart des concepts VFT restent orthogonale à XFT. A l'avenir, il serait très intéressant d'explorer les interactions entre le modèle hybride et ses raffinements tels que VFT avec notre XFT.

Steward [21] est un protocole BFT spécialement conçu pour les réseaux étendus. Steward utilise une architecture hiérarchique: il déploie BFT entre les réplicas dans un site géographique, et repose sur CFT à travers les sites. En revanche, nous avons déployé XPaxos en appliquant XFT directement à travers les sites. XFT pourrait être combinée avec l'approche hiérarchique de Steward — ce qui est une direction intéressante pour les travaux futurs.

Au-delà de quelques directions de recherche décrites ci-dessus, XFT ouvre plusieurs pistes de travail. Par exemple, de nombreux problèmes importants d'informatique répartie au-delà de SMR, tels que le stockage distribué, méritent un regard nouveau à travers le prisme de XFT.

Dans la deuxième contribution, nous avons conçu Droopy et Dripple, deux approches sœurs adaptées à latence faible geo-réplication de machines à états. Droopy configure dynamiquement l'ensemble de principal en fonction de la charge de travail précédente et l'état du réseau. Dripple divise à son tour l'état en partitions et coordonne eux intelligemment, de sorte que l'ensemble de principal de chaque partition peut être configuré (par Droopy) séparément. Nos résultats expérimentaux sur Amazon EC2 montrent que, Droopy et Dripple fournissent pratiquement latence quasi-optimale sous plusieurs charges de travail réalistes.

Nos contributions abordés dans cette thèse ciblent les différentes composantes de SMR — le consensus et l'ordre. Nous les décrit de façon modulaire. Il serait très intéressant d'intégrer XPaxos avec Droopy et Dripple. Par exemple, l'exécution de plusieurs groupes synchrones simultanément afin de servir les clients situés à des endroits distincts.

Bibliography

[1] Alibaba. https://www.alibaba.com.

[2] Alipay. https://www.alipay.com.

[3] Alipay suffers temporary breakdown. http://europe.chinadaily.com.cn/business/2015-05/28/content_20845886.htm.

[4] Amazon. https://www.amazon.com.

[5] Bft-smart. http://bft-smart.github.io/library.

[6] Dropbox. https://www.dropbox.com.

[7] ebay. https://www.ebay.com.

[8] Errors in database systems, eventual consistency, and the cap theorem. http://http://cacm.acm.org/blogs/blog-cacm/83396-errors-in-database-systems-eventual-consistency-and-the-cap-theorem/fulltext.

[9] Facebook. https://www.facebook.com.

[10] Google drive. https://mail.google.com.

[11] Google mail. https://mail.google.com.

[12] hping. http://www.hping.org/.

[13] Hurricane sandy takes data centers offline with flooding, power outages. http://arstechnica.com/information-technology/2012/10/hurricane-sandy-takes-data-centers-offline-with-flooding-power-outages/.

[14] Lax computer glitch strands 20,000 for 14 hours. http://laist.com/2007/08/12/lax_computer_gl.php.

[15] Microsoft hotmail. https://www.hotmail.com.

[16] Netperf. http://www.netperf.org/.

[17] Network time protocol. http://www.ntp.org/.

[18] Paypal. https://www.paypal.com.

[19] Twitter. https://www.twitter.com.

[20] Alpern, B., and Schneider, F. Recognizing safety and liveness. *Distributed Computing 2*, 3 (1987), 117–126.

[21] Amir, Y., Danilov, C., Dolev, D., Kirsch, J., Lane, J., Nita-Rotaru, C., Olsen, J., and Zage, D. Steward: Scaling byzantine fault-tolerant replication to wide area networks. *Dependable and Secure Computing, IEEE Transactions on 7*, 1 (Jan 2010), 80–93.

[22] Amir, Y., Danilov, C., Kirsch, J., Lane, J., Dolev, D., Nita-Rotaru, C., Olsen, J., and Zage, D. Scaling byzantine fault-tolerant replication towide area networks. In *Dependable Systems and Networks, 2006. DSN 2006. International Conference on* (June 2006), pp. 105–114.

[23] Amir, Y., Danilov, C., Miskin-Amir, M., Stanton, J., and Tutu, C. On the performance of wide-area synchronous database replication. Tech. rep., Citeseer, 2002.

[24] Ardekani, M. S., Sutra, P., and Shapiro, M. Non-monotonic snapshot isolation: Scalable and strong consistency for geo-replicated transactional systems. In *Proceedings of the 2013 IEEE 32Nd International Symposium on Reliable Distributed Systems* (Washington, DC, USA, 2013), SRDS '13, IEEE Computer Society, pp. 163–172.

[25] Ardekani, M. S., and Terry, D. B. A self-configurable geo-replicated cloud storage system. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2014), OSDI'14, USENIX Association, pp. 367–381.

[26] Aublin, P.-L., Guerraoui, R., Knežević, N., Quéma, V., and Vukolić, M. The next 700 bft protocols. *ACM Trans. Comput. Syst. 32*, 4 (Jan. 2015), 12:1–12:45.

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

[28] Baker, J., Bond, C., Corbett, J. C., Furman, J., Khorlin, A., Larson, J., Léon, J.-M., Li, Y., Lloyd, A., and Yushprakh, V. Megastore: Providing scalable, highly available storage for interactive services. In *Proc. of CIDR* (2011), pp. 223–234.

[29] Berenson, H., Bernstein, P., Gray, J., Melton, J., O'Neil, E., and O'Neil, P. A critique of ansi sql isolation levels. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 1995), SIGMOD '95, ACM, pp. 1–10.

[30] Berman, P., Garay, J. A., and Perry, K. J. Towards optimal distributed consensus. In *Proc. 30th IEEE Symposium on Foundations of Computer Science (FOCS)* (1989), pp. 410–415.

[31] BERNSTEIN, P. A., HADZILACOS, V., AND GOODMAN, N. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1987.

[32] BESSANI, A., CORREIA, M., QUARESMA, B., ANDRÉ, F., AND SOUSA, P. Depsky: Dependable and secure storage in a cloud-of-clouds. *Trans. Storage 9*, 4 (Nov. 2013), 12:1–12:33.

[33] BESSANI, A., SOUSA, J., AND ALCHIERI, E. State machine replication for the masses with bft-smart. In *Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP International Conference on* (June 2014), pp. 355–362.

[34] BEZERRA, C., PEDONE, F., AND VAN RENESSE, R. Scalable state-machine replication. In *Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP International Conference on* (June 2014), pp. 331–342.

[35] BURCKHARDT, S. *Principles of Eventual Consistency*, vol. 1 of *Foundations and Trends in Programming Languages.* now publishers, October 2014.

[36] BURROWS, M. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th symposium on Operating systems design and implementation* (Berkeley, CA, USA, 2006), OSDI '06, USENIX Association, pp. 335–350.

[37] CACHIN, C., GUERRAOUI, R., AND RODRIGUES, L. *Introduction to Reliable and Secure Distributed Programming*, 2nd ed. Springer Publishing Company, Incorporated, 2011.

[38] CALDER, B., WANG, J., OGUS, A., NILAKANTAN, N., SKJOLSVOLD, A., MCKELVIE, S., 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* (New York, NY, USA, 2011), SOSP '11, ACM, pp. 143–157.

[39] CASTRO, M., AND LISKOV, B. Authenticated byzantine fault tolerance without public-key cryptography. Tech. rep., Technical Memo MIT/LCS/TM-589, MIT Laboratory for Computer Science, 1999.

[40] CASTRO, M., AND LISKOV, B. Practical byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation* (Berkeley, CA, USA, 1999), OSDI '99, USENIX Association, pp. 173–186.

[41] CASTRO, M., AND LISKOV, B. Practical Byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst. 20*, 4 (Nov. 2002), 398–461.

[42] CHANDRA, T. D., GRIESEMER, R., AND REDSTONE, J. Paxos made live: an engineering perspective. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing* (New York, NY, USA, 2007), PODC '07, ACM, pp. 398–407.

[43] Chou, J.-Y., Lin, B., Sen, S., and Spatscheck, O. Proactive surge protection: A defense mechanism for bandwidth-based attacks. *Networking, IEEE/ACM Transactions on 17*, 6 (Dec 2009), 1711–1723.

[44] Chun, B.-G., Maniatis, P., Shenker, S., and Kubiatowicz, J. Attested append-only memory: Making adversaries stick to their word. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles* (New York, NY, USA, 2007), SOSP '07, ACM, pp. 189–204.

[45] Clement, A., Kapritsos, M., Lee, S., Wang, Y., Alvisi, L., Dahlin, M., and Riche, T. Upright cluster services. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)* (2009), ACM.

[46] Clement, A., Wong, E., Alvisi, L., Dahlin, M., and Marchetti, M. Making byzantine fault tolerant systems tolerate byzantine faults. In *Proceedings of the 6th USENIX symposium on Networked systems design and implementation* (Berkeley, CA, USA, 2009), NSDI'09, USENIX Association, pp. 153–168.

[47] Corbett, J. C., Dean, J., Epstein, M., Fikes, A., Frost, C., Furman, J. J., et al. Spanner: Google's globally-distributed database. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2012), OSDI'12, USENIX Association, pp. 251–264.

[48] Cormen, T. H., Stein, C., Rivest, R. L., and Leiserson, C. E. *Introduction to Algorithms*, 2nd ed. McGraw-Hill Higher Education, 2001.

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

[50] Cristian, F., Aghili, H., Strong, R., and Dolev, D. Atomic broadcast: From simple message diffusion to Byzantine agreement. *Information and Computation 118*, 1 (1995), 158–179.

[51] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W. Dynamo: amazon's highly available key-value store. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles* (New York, NY, USA, 2007), SOSP '07, ACM, pp. 205–220.

[52] Dobre, D., Viotti, P., and Vukolić, M. Hybris: Robust hybrid cloud storage. In *Proceedings of the ACM Symposium on Cloud Computing* (New York, NY, USA, 2014), SOCC '14, ACM, pp. 12:1–12:14.

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

[54] Du, J., Elnikety, S., and Zwaenepoel, W. Clock-si: Snapshot isolation for partitioned data stores using loosely synchronized clocks. In *Proceedings of the 2013 IEEE 32Nd International Symposium on Reliable*

Distributed Systems (Washington, DC, USA, 2013), SRDS '13, IEEE Computer Society, pp. 173–184.

[55] Du, J., Sciascia, D., Elnikety, S., Zwaenepoel, W., and Pedone, F. Clock-rsm: Low-latency inter-datacenter state machine replication using loosely synchronized physical clocks. In *The 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)* (2014), no. EPFL-CONF-198282.

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

[57] Echtle, K., and Masum, A. A multiple bus broadcast protocol resilient to non-cooperative byzantine faults. In *Proceedings of the The Twenty-Sixth Annual International Symposium on Fault-Tolerant Computing (FTCS '96)* (Washington, DC, USA, 1996), FTCS '96, IEEE Computer Society, pp. 158–.

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

[59] Garcia, R., Rodrigues, R., and Preguiça, N. Efficient middleware for byzantine fault tolerant database replication. In *Proceedings of the sixth conference on Computer systems* (New York, NY, USA, 2011), EuroSys '11, ACM, pp. 107–122.

[60] Gilbert, S., and Lynch, N. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News 33*, 2 (June 2002), 51–59.

[61] Guerraoui, R., Knežević, N., Quéma, V., and Vukolić, M. The next 700 bft protocols. In *Proceedings of the 5th European conference on Computer systems* (New York, NY, USA, 2010), EuroSys '10, ACM, pp. 363–376.

[62] Guerraoui, R., and Vukolić, M. Refined quorum systems. *Distributed Computing 23*, 1 (2010), 1–42.

[63] Guo, Z., Hong, C., Yang, M., Zhou, D., Zhou, L., and Zhuang, L. Rex: Replication at the speed of multi-core. In *Proceedings of the Ninth European Conference on Computer Systems* (New York, NY, USA, 2014), EuroSys '14, ACM, pp. 11:1–11:14.

[64] Haeberlen, A., Kouznetsov, P., and Druschel, P. PeerReview: practical accountability for distributed systems. In *SOSP* (2007), pp. 175–188.

[65] Halalai, R., Sutra, P., Riviere, E., and Felber, P. Zoofence: Principled service partitioning and application to the zookeeper coordination service. In *Reliable Distributed Systems (SRDS), 2014 IEEE 33rd International Symposium on* (Oct 2014), pp. 67–78.

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

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

[68] Jo o, S., and Alysson, B. Separating the wheat from the chaff: An empirical design for geo-replicated state machines. In *to appear in 2015 IEEE 34th International Symposium on Reliable Distributed Systems* (Montreal, Quebec, Canada, 2015), SRDS '15, IEEE Computer Society.

[69] Junqueira, F. P., Reed, B. C., and Serafini, M. Zab: High-performance broadcast for primary-backup systems. In *Proceedings of the Conference on Dependable Systems and Networks (DSN)* (2011), pp. 245–256.

[70] Kapitza, R., Behl, J., Cachin, C., Distler, T., Kuhnle, S., Mohammadi, S. V., Schröder-Preikschat, W., and Stengel, K. Cheapbft: Resource-efficient byzantine fault tolerance. In *Proceedings of the 7th ACM European Conference on Computer Systems* (New York, NY, USA, 2012), EuroSys '12, ACM, pp. 295–308.

[71] Kapritsos, M., Wang, Y., Quema, V., Clement, A., Alvisi, L., and Dahlin, M. All about Eve: execute-verify replication for multi-core servers. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2012), OSDI'12, USENIX Association, pp. 237–250.

[72] Kistler, J. J., and Satyanarayanan, M. Disconnected operation in the coda file system. *ACM Trans. Comput. Syst. 10*, 1 (Feb. 1992), 3–25.

[73] Kotla, R., Alvisi, L., Dahlin, M., Clement, A., and Wong, E. Zyzzyva: Speculative Byzantine fault tolerance. *ACM Trans. Comput. Syst. 27*, 4 (Jan. 2010), 7:1–7:39.

[74] Kraska, T., Pang, G., Franklin, M. J., Madden, S., and Fekete, A. Mdcc: Multi-data center consistency. In *Proceedings of the 8th ACM European Conference on Computer Systems* (New York, NY, USA, 2013), EuroSys '13, ACM, pp. 113–126.

[75] Krishnan, K. Weathering the unexpected. *Commun. ACM 55*, 11 (Nov. 2012), 48–52.

[76] Kumar, S., Gomez, O., et al. Denial of service due to direct and indirect arp storm attacks in lan environment. *Journal of Information Security 1*, 02 (2010), 88.

[77] Kuznetsov, P., and Rodrigues, R. BFTW3: Why? When? Where? Workshop on the theory and practice of Byzantine fault tolerance. *SIGACT News 40*, 4 (Jan. 2010), 82–86.

[78] LADIN, R., LISKOV, B., SHRIRA, L., AND GHEMAWAT, S. Providing high availability using lazy replication. *ACM Trans. Comput. Syst. 10*, 4 (Nov. 1992), 360–391.

[79] LAKSHMAN, A., AND MALIK, P. Cassandra: a decentralized structured storage system. *SIGOPS Oper. Syst. Rev. 44*, 2 (Apr. 2010), 35–40.

[80] LAMPORT, L. The part-time parliament. *ACM Trans. Comput. Syst. 16* (May 1998), 133–169.

[81] LAMPORT, L. Paxos made simple. *ACM SIGACT News 32*, 4 (2001), 18–25.

[82] LAMPORT, L. Generalized consensus and paxos. Tech. Rep. MSR-TR-2005-33, Microsoft Research, March 2005.

[83] LAMPORT, L. Fast paxos. *Distributed Computing 19* (2006), 79–103. 10.1007/s00446-006-0005-x.

[84] LAMPORT, L., MALKHI, D., AND ZHOU, L. Vertical paxos and primary-backup replication. In *Proceedings of the 28th ACM Symposium on Principles of Distributed Computing* (New York, NY, USA, 2009), PODC '09, ACM, pp. 312–313.

[85] LAMPORT, L., AND MASSA, M. Cheap paxos. In *Dependable Systems and Networks, 2004 International Conference on* (june-1 july 2004), pp. 307 – 314.

[86] LAMPORT, L., SHOSTAK, R., AND PEASE, M. The Byzantine generals problem. *ACM Trans. Program. Lang. Syst. 4* (July 1982), 382–401.

[87] LEVIN, D., DOUCEUR, J. R., LORCH, J. R., AND MOSCIBRODA, T. Trinc: Small trusted hardware for large distributed systems. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2009), NSDI'09, USENIX Association, pp. 1–14.

[88] LI, C., PORTO, D., CLEMENT, A., GEHRKE, J., PREGUIÇA, N., AND RODRIGUES, R. Making geo-replicated systems fast as possible, consistent when necessary. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2012), OSDI'12, USENIX Association, pp. 265–278.

[89] LISKOV, B., AND COWLING, J. Viewstamped replication revisited.

[90] LLOYD, W., FREEDMAN, M. J., KAMINSKY, M., AND ANDERSEN, D. G. Don't settle for eventual: scalable causal consistency for wide-area storage with cops. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2011), SOSP '11, ACM, pp. 401–416.

[91] LLOYD, W., FREEDMAN, M. J., KAMINSKY, M., AND ANDERSEN, D. G. Stronger semantics for low-latency geo-replicated storage. In *Proceedings of the symposium on Networked systems design and implementation* (2013), NSDI'13.

[92] Mahajan, P., Setty, S., Lee, S., Clement, A., Alvisi, L., Dahlin, M., and Walfish, M. Depot: Cloud storage with minimal trust (extended version). Tech. rep., Technical Report TR-10-33, Department of Computer Science, The University of Texas at Austin, 2010.

[93] Mao, Y., Junqueira, F. P., and Marzullo, K. Mencius: Building efficient replicated state machines for wans. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2008), OSDI'08, USENIX Association, pp. 369–384.

[94] Maziéres, D. The Stellar consensus protocol: A federated model for internet-level consensus. https://www.stellar.org/papers/stellar-consensus-protocol.pdf, November 2015.

[95] Moraru, I., Andersen, D. G., and Kaminsky, M. There is more consensus in egalitarian parliaments. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2013), SOSP '13, ACM, pp. 358–372.

[96] Moraru, I., Andersen, D. G., and Kaminsky, M. Paxos quorum leases: Fast reads without sacrificing writes. In *Proceedings of the ACM Symposium on Cloud Computing* (New York, NY, USA, 2014), SOCC '14, ACM, pp. 22:1–22:13.

[97] Oki, B. M., and Liskov, B. H. 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* (New York, NY, USA, 1988), PODC '88, ACM, pp. 8–17.

[98] Ongaro, D., and Ousterhout, J. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference* (Berkeley, CA, USA, 2014), USENIX ATC'14, USENIX Association, pp. 305–320.

[99] Padilha, R., and Pedone, F. Augustus: Scalable and robust storage for cloud applications. In *Proceedings of the eigth conference on Computer systems* (2013), EuroSys '13.

[100] Porto, D., Leitão, J. a., Li, C., Clement, A., Kate, A., Junqueira, F., and Rodrigues, R. Visigoth fault tolerance. In *Proceedings of the Tenth European Conference on Computer Systems* (New York, NY, USA, 2015), EuroSys '15, ACM, pp. 8:1–8:14.

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

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

[103] Shapiro, M., Preguiça, N., Baquero, C., and Zawirski, M. Conflict-free replicated data types. In *Proceedings of the 13th international conference on Stabilization, safety, and security of distributed systems* (Berlin, Heidelberg, 2011), SSS'11, Springer-Verlag, pp. 386–400.

[104] Singh, A., Fonseca, P., Kuznetsov, P., Rodrigues, R., Maniatis, P., et al. Zeno: eventually consistent byzantine-fault tolerance. In *NSDI* (2009), vol. 9, pp. 169–184.

[105] Sovran, Y., Power, R., Aguilera, M. K., and Li, J. Transactional storage for geo-replicated systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2011), SOSP '11, ACM, pp. 385–400.

[106] Sutra, P., and Shapiro, M. Fast genuine generalized consensus. In *Reliable Distributed Systems (SRDS), 2011 30th IEEE Symposium on* (Oct 2011), pp. 255–264.

[107] Terry, D. B., Prabhakaran, V., Kotla, R., Balakrishnan, M., Aguilera, M. K., and Abu-Libdeh, H. Consistency-based service level agreements for cloud storage. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2013), SOSP '13, ACM, pp. 309–324.

[108] Terry, D. B., Theimer, M. M., Petersen, K., Demers, A. J., Spreitzer, M. J., and Hauser, C. H. Managing update conflicts in bayou, a weakly connected replicated storage system. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 1995), SOSP '95, ACM, pp. 172–182.

[109] Veronese, G. S., Correia, M., Bessani, A. N., and Lung, L. C. Spin one's wheels? Byzantine Fault Tolerance with a spinning primary. In *Proceedings of International Symposium on Reliable Distributed Systems (SRDS)* (2009), IEEE Computer Society.

[110] Veronese, G. S., Correia, M., Bessani, A. N., Lung, L. C., and Verissimo, P. Minimal byzantine fault tolerance: Algorithm and evaluation.

[111] Veronese, G. S., Correia, M., Bessani, A. N., Lung, L. C., and Veríssimo, P. Efficient Byzantine fault-tolerance. *IEEE Trans. Computers 62*, 1 (2013), 16–30.

[112] Vukolic, M. *Abstractions for asynchronous distributed computing with malicious players.* PhD thesis, PhD thesis, EPFL, 2008.

[113] Vukolic, M. *Quorum Systems:With Applications to Storage and Consensus.* Morgan & Claypool, 2012.

[114] Wood, T., Singh, R., Venkataramani, A., Shenoy, P., and Cecchet, E. Zz and the art of practical bft execution. In *Proceedings of the sixth conference on Computer systems* (New York, NY, USA, 2011), EuroSys '11, ACM, pp. 123–138.

[115] Wu, Z., Butkiewicz, M., Perkins, D., Katz-Bassett, E., and Madhyastha, H. V. Spanstore: Cost-effective geo-replicated storage spanning multiple cloud services. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2013), SOSP '13, ACM, pp. 292–308.

[116] Yang, J., Twohey, P., Engler, D., and Musuvathi, M. Using model checking to find serious file system errors. *ACM Trans. Comput. Syst. 24*, 4 (Nov. 2006), 393–423.

[117] Yin, J., Martin, J.-P., Venkataramani, A., Alvisi, L., and Dahlin, M. Separating agreement from execution for byzantine fault tolerant services. In *Proceedings of the nineteenth ACM symposium on Operating systems principles* (New York, NY, USA, 2003), SOSP '03, ACM, pp. 253–267.

[118] Zhang, Y., Power, R., Zhou, S., Sovran, Y., Aguilera, M. K., and Li, J. Transaction chains: Achieving serializability with low latency in geo-distributed storage systems. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2013), SOSP '13, ACM, pp. 276–291.

Appendix A

XPaxos Pseudocode

For simplicity reason, we assume that signature/MAC attached to each message always correctly verifies. Fig. A.1 gives the definition of message fields and local variables for all components of XPaxos. Readers can refer to Sec. 4.6 for protocol description.

To simplify the exposition, this section is organized incrementally as follows. Sec. A.1 gives the pseudocode of XPaxos common case. Sec. A.2 gives the pseudocode of the view change mechanism. Sec. A.3 gives pseudocode of clients' request retransmission mechanism that deals with faulty or partitioned active replica. Finally, Sec. A.4 gives pseudocode of modifications to the view change protocol to enable Fault Detection.

A.1 Common Case

In common case, we assume that all replicas are in the same view. Algorithm 5 and Algorithm 6 describe the common case protocol when $t = 1$ and $t \geq 2$, respectively. Fig. 4.2 gives the message pattern.

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
Π - 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 ($ps_i \in sg_i$)
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_j} - sequence number prepared at replica s_j
ex_{s_j} - sequence number executed at replica s_j
$D(m)$ - digest of a message m
$PrepareLog_{s_j}$ - array of prepared proof at replica s_j
$CommitLog_{s_j}$ - array of commit proofs at replica s_j

View change :
$SusSet_{s_j}$ - set of SUSPECT messages cached for view-change at replica s_j
$timer_i^{net}$ - network establishment timer for view i
Δ - 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
$VCSet_{s_j}^i$ - set of VIEW-CHANGE messages collected in view change to i at replica s_j
$CommitLog_{s_j}^i$ - array of most recent commit proofs selected from $VCSet_{s_j}^i$ at replica s_j
$End(log)$ - end index of array log

Fault detection :
$FinalProof_{s_j}$ - 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_j} - the view number in which $PrepareLog_{s_j}$ is generated
$FinalSet_{s_j}^i$ - set of $t+1$ VC-FINAL messages collected in view change to i at replica s_j
$PrepareLog_{s_j}^i$ - array of most recent prepare proof selected from $VCSet_{s_j}^i$ at replica s_j

Figure A.1: XPaxos common case: Message fields and local variables.

Algorithm 5 Common case when $t = 1$.

Initialization:
client : $ts_c \leftarrow 0$; $req_c \leftarrow nil$
replica : $sn_{s_j} \leftarrow 0; ex_{s_j} \leftarrow 0; PrepareLog_{s_j} = []; CommitLog_{s_j} = []$

1: **upon** invocation of $propose(op)$ at client c **do**
2: **inc**(ts_c)
3: send $req_c \leftarrow \langle \text{REPLICATE}, op, ts_c, c\rangle_{\sigma_c}$ to the primary $ps_i \in sg_i$
4: start $timer_c$

5: **upon** reception of $req = \langle \text{REPLICATE}, op, ts, c\rangle_{\sigma_c}$ from client c at ps_i **do** /* primary */
6: **inc**(sn_{ps_i})
7: $m_{ps_i} \leftarrow \langle \text{COMMIT}, D(req), sn_{ps_i}, i\rangle_{\sigma_{ps_i}}$
8: $PrepareLog_{ps_i}[sn_{ps_i}] \leftarrow \langle req, m_{ps_i}\rangle$
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 */
11: **if** $sn = sn_{fs_i} + 1$ **and** $D(req) = d_{req}$ **then**
12: **inc**(sn_{fs_i})
13: $rep \leftarrow$ execute req
14: **inc**(ex_{fs_i})
15: $m_{fs_i} \leftarrow \langle \text{COMMIT}, D(req), sn, i, req.ts_c, D(rep)\rangle_{\sigma_{fs_i}}$
16: $CommitLog_{fs_i}[sn] \leftarrow \langle req, m_{ps_i}, m_{fs_i}\rangle$
17: send m_{fs_i} to the primary ps_i

18: **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**
19: **if** $D(PrepareLog_{ps_i}[sn].req) = d_{req}$ **then**
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: **inc**(ex_{ps_i})
23: $rep \leftarrow$ execute $CommitLog_{ps_i}[ex_{ps_i}].req$
24: **if** $D(rep) = CommitLog_{ps_i}[ex_{ps_i}].m_{fs_i}.d_{rep}$ **then**
25: send $\langle\langle \text{REPLY}, sn, i, ts, rep\rangle_{\mu_{ps_i,c}}, m_{fs_i}\rangle$ to $CommitLog_{ps_i}[ex_{ps_i}].req.c$

26: **upon** reception of $\langle r_{ps_i}, m_{fs_i}\rangle$ from the primary ps_i at client c, where
$r_{ps_i} = \langle \text{REPLY}, sn, i, ts, rep\rangle_{\mu_{ps_i,c}}$
$m_{fs_i} = \langle \text{COMMIT}, d'_{req}, sn', i', ts', d_{rep}\rangle_{\sigma_{fs_i}}$ **do**
27: **if** $sn = sn'$ **and** $i = i'$ **and** $ts = ts' = req.ts_c$ **and** $D(rep) = d_{rep}$ **then**
28: deliver rep
29: stop $timer_c$

Algorithm 6 Common case when $t > 1$.

Initialization:
client : $ts_c \leftarrow 0$; $req_c \leftarrow nil$
replica : $sn_{s_j} \leftarrow 0; ex_{s_j} \leftarrow 0; PrepareLog_{s_j} = []; CommitLog_{s_j} = []$

1: **upon** invocation of $propose(op)$ at client c **do**
2: **inc**(ts_c)
3: send $req_c \leftarrow \langle \text{REPLICATE}, op, ts_c, c\rangle_{\sigma_c}$ to the primary $ps_i \in sg_i$
4: start $timer_c$

5: **upon** reception of $req = \langle \text{REPLICATE}, op, ts, c\rangle_{\sigma_c}$ from client c at ps_i **do** /* primary */
6: **inc**(sn_{ps_i})
7: $m_{ps_i} \leftarrow \langle \text{PREPARE}, D(req), sn_{ps_i}, i\rangle_{\sigma_{ps_i}}$
8: $PrepareLog_{ps_i}[sn] \leftarrow \langle req, m_{ps_i}\rangle$
9: send $\langle req, m_{ps_i}\rangle$ to $fs_i^k \in sg_i$

10: **upon** reception of $\langle req, m_{ps_i} = \langle \text{PREPARE}, d_{req}, sn, i\rangle_{\sigma_{ps_i}}\rangle$ from the primary ps_i at fs_i^k **do** /* follower */
11: **if** $sn = sn_{fs_i^k} + 1$ **and** $D(req) = d_{req}$ **then**
12: **inc**$(sn_{fs_i^k})$
13: $PrepareLog_{fs_i^k}[sn] \leftarrow \langle req, m_{ps_i}\rangle$
14: $m_{fs_i^k} \leftarrow \langle \text{COMMIT}, D(req), sn, i, fs_i^k\rangle_{\sigma_{fs_i^k}}$
15: send $m_{fs_i^k}$ 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**
17: $CommitLog_{s_j}[sn] \leftarrow \langle req, m_{ps_i}, m_{fs_i^1} ... m_{fs_i^f}\rangle$

18: **upon** $CommitLog_{s_j}[ex_{s_j} + 1] \neq nil$ at s_j **do**
19: **inc**(ex_{s_j})
20: $rep \leftarrow$ execute $CommitLog_{s_j}[ex_{s_j}].req$
21: send $\langle \text{REPLY}, sn, i, req.ts_c, rep\rangle_{\mu_{s_j,c}}$ to client c, where $c = CommitLog_{s_j}[ex_{s_j}].req.c)$

22: **upon** reception of $t + 1$ REPLY messages $\langle \text{REPLY}, sn, i, ts, rep\rangle_{\mu_{s_j,c}}$ at client c **do**
23: **if** $t + 1$ REPLY messages are with the same sn, i, ts and rep **and** $ts = req.ts_c$ **then**
24: deliver rep
25: stop $timer_c$

A.2 View-Change

The message pattern of view-change w/o fault detection is given in Fig. 4.3. Algorithm 7 shows the corresponding pseudocode.

Algorithm 7 View change at replica s_j.

Initialization:
$SusSet_{s_j} \leftarrow \emptyset; VCSet^i_{s_j} \leftarrow \emptyset; CommitLog^i_{s_j} \leftarrow []$

1: **upon** suspicion of view i **and** $s_j \in sg_i$ **do** /* activate view-change to $i+1$ */
2: send $\langle \text{SUSPECT}, i, s_j \rangle_{\sigma_{s_j}}$ 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: $SusSet_{s_j} \leftarrow SusSet_{s_j} \cup \{m\}$
5: forward m to $\forall s_k \in \Pi$

6: **upon** $\exists \langle \text{SUSPECT}, i, s_k \rangle_{\sigma_{s_k}} \in SusSet_{s_j}$ **do** /* enter each view in order */
7: **inc**(i) (i.e., ignore any message in preceding view)
8: send $\langle \text{VIEW-CHANGE}, i, s_j, CommitLog_{s_j} \rangle_{\sigma_{s_j}}$ to $\forall s_k \in sg_i$
9: **if** $s_j \in sg_i$ **then**
10: start $timer^{net}_i \leftarrow 2\Delta$

11: **upon** reception of $m = \langle \text{VIEW-CHANGE}, i, s_k, CommitLog \rangle_{\sigma_{s_k}}$ from replica s_k **do**
12: $VCSet^i_{s_j} \leftarrow VCSet^i_{s_j} \cup \{m\}$

13: **upon** $|VCSet^i_{s_j}| = n$ **or** (expiration of $timer^{net}_i$ **and** $|VCSet^i_{s_j}| \geq n - t$) **do**
14: send $\langle \text{VC-FINAL}, i, s_j, VCSet^i_{s_j} \rangle_{\sigma_{s_j}}$ to $\forall s_k \in sg_i$
15: start $timer^{vc}_i$

16: **upon** reception of $m_* = \langle \text{VC-FINAL}, i, s_k, VCSet \rangle_{\sigma_{s_k}}$ from every $s_k \in sg_i$ **do**
17: $VCSet^i_{s_j} \leftarrow VCSet^i_{s_j} \cup \{\forall m : m \in VCSet \text{ in any } m_k\}$
18: **for** $sn : 1..End(\forall CommitLog | \exists m \in VCSet^i_{s_j} : CommitLog \text{ is in } m)$ **do**
19: $CommitLog^i_{s_j}[sn] \leftarrow CommitLog[sn]$ with the *highest* view number
20: **if** $s_j = ps_i$ **then** /* primary */
21: **for** $sn : 1..End(CommitLog^i_{s_j})$ **do**
22: $req \leftarrow CommitLog^i_{s_j}[sn].req$
23: $PrepareLog[sn] \leftarrow \langle req, \langle \text{PREPARE}, D(req), sn, i \rangle_{\sigma_{ps_i}} \rangle$
24: send $\langle \text{NEW-VIEW}, i, PrepareLog \rangle_{\sigma_{ps_i}}$ to $\forall s_k \in sg_i$

25: **upon** reception of $\langle \text{NEW-VIEW}, i, PrepareLog \rangle_{\sigma_{ps_i}}$ from the primary ps_i **do**
26: **if** $PrepareLog$ is matching with $CommitLog^i_{s_j}$ **then**
27: $PrepareLog_{s_j} \leftarrow PrepareLog$
28: reply and process $\forall m \in PrepareLog$ as in common case
29: $sn_{s_j} \leftarrow End(PrepareLog)$
30: $ex_{s_j} \leftarrow End(PrepareLog)$
31: stop $timer^{vc}_i$
32: **else**
33: suspect view i

34: **upon** expiration of $timer^{vc}_i$ **do**
35: suspect view i

A.3 Request Retransmission

Algorithm 8 Client request retransmission.

1: **upon** expiration of $timer_c$ at client c **do**
2: send $\langle$RE-SEND,$req_c\rangle$ to $\forall s_j \in sg_i$

3: **upon** reception of $\langle$RE-SEND,$req_c\rangle$ at $s_j \in sg_i$ **do**
4: **if** $s_j \neq ps_i$ **then**
5: send req_c to $ps_i \in sg_i$
6: start $timer_{req_c}$
7: ask $\forall s_j \in sg_i$ to sign the reply of req_c

8: **upon** expiration of $timer_{req_c}$ at replica $s_j \in sg_i$ **do**
9: suspect view i
10: send $\langle \text{SUSPECT}, i, s_j\rangle_{\sigma_{s_j}}$ to client c

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**
12: enter view $i+1$
13: send m to $\forall s_j \in sg_{i+1}$
14: send req_c to ps_{i+1}
15: start $timer_c$

16: **upon** execution of req_c at s_j[1] **do**
/* sign the reply by each active replica */
17: send $\langle \text{REPLY}, sn, i, req.ts_c, rep\rangle_{\sigma_{s_j}}$ 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 */
19: **if** $m_1, m_2, ..., m_{t+1}$ are with the same sn, i, ts and rep **then**
20: $replies \leftarrow \{m_1, m_2, ..., m_{t+1}\}$
21: send $\langle$SIGNED-REPLY$, replies\rangle$ to client c
22: stop $timer_{req_c}$

A.4 Fault Detection

Algorithm 9 gives the modifications based on Algorithm 7 for XPaxos with fault detection mechanism. Algorithm 10 enumerates all types of faults that can and should be detected by correct active replicas. Fig. 4.4 gives the new message pattern.

Algorithm 9 Modifications of view-change for fault detection at replica s_j.

Initialization:
$FinalProof_{s_j} \leftarrow []; pre_{s_j} \leftarrow 0; FinalSet^i_{s_j} \leftarrow \emptyset; PrepareLog^i_{s_j} \leftarrow []; FSet \leftarrow []$

/* replace line 8 in Algorithm 7 by : */
1: send $m = \langle \text{VIEW-CHANGE}, i, s_j, CommitLog_{s_j}, PrepareLog_{s_j}, FinalProof_{s_j}[pre_{s_j}] \rangle_{\sigma_{s_j}}$ to $\forall s_k \in sg_i$

/* replace line 11 in Algorithm 7 by : */
2: **upon** reception of $m = \langle \text{VIEW-CHANGE}, i, s_k, CommitLog, PrepareLog, FinalProof \rangle_{\sigma_{s_k}}$ from replica s_k **do**

/* replace lines 18∼24 in Algorithm 7 by : */
3: FAULTDETECTION($vcSet^i_{s_j}$) /* refer to Algorithm 10 */
4: **for** $\forall m : m \in vcSet^i_{s_j}$ **and** m from replica $s \in FSet$ **do**
5: remove m from $vcSet^i_{s_j}$
6: send $\langle \text{VC-CONFIRM}, i, D(vcSet^i_{s_j}) \rangle_{\sigma_{s_j}}$ to $\forall s_k \in sg_i$

/* 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**
8: **if** $m_1, m_2, ...m_{f+1}$ are *not* with the same d_{vcSet} **then**
9: suspect view i
10: **return**
11: $FinalProof_{s_j}[i] \leftarrow \{m_1, m_2, ...m_{f+1}\}$
12: **for** $sn : 1..End(\forall CommitLog | \exists m \in VCSet^i_{s_j} : CommitLog$ is in $m)$ **do**
13: $CommitLog^i_{s_j}[sn] \leftarrow CommitLog[sn]$ with the *highest* view number
14: **for** $sn : 1..End(\forall PrepareLog | \exists m \in VCSet^i_{s_j} : PrepareLog$ is in $m)$ **do**
15: $PrepareLog^i_{s_j}[sn] \leftarrow PrepareLog[sn]$ with the *highest* view number[2]
16: **if** $s_j = ps_i$ **then** /* primary */
17: **for** $sn : 1..End(PrepareLog^i_{s_j} | CommitLog^i_{s_j})$ **do**
18: $req \leftarrow CommitLog^i_{s_j}[sn].req$
19: **if** $req = null$ **or** $PrepareLog^i_{s_j}[sn]$ is generated in a higher view than $CommitLog^i_{s_j}[sn]$ **then**
20: $req \leftarrow PrepareLog^i_{s_j}[sn].req$
21: $PrepareLog[sn] \leftarrow \langle req, \langle \text{PREPARE}, D(req), sn, i \rangle_{\sigma_{s_j}} \rangle$
22: send $\langle \text{NEW-VIEW}, i, PrepareLog \rangle_{\sigma_{s_j}}$ to $\forall s_k \in sg_i$

/* replace line 26 in Algorithm 7 by : */
23: **if** $PrepareLog$ is matching with $CommitLog^i_{s_j}$ and $PrepareLog^i_{s_j}$ **then**

/* add this command after line 27 in Algorithm 7 : */
24: $pre_{s_j} \leftarrow i$ /* update the view in which $PrepareLog_{s_j}$ is generated */

Algorithm 10 Fault detection function and event at replica s_j.

1: **function** FAULTDETECTION($VCSet$)
2: $\forall sn$ and $m, m' \in VCSet$ from replicas s_k and $s_{k'}$, respectively,

3: (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 is faulty)
4: send $\langle$STATE-LOSS$, i, s_k, sn, m, m'\rangle$ to $\forall s_{k''} \in \Pi$
5: add s_k to $FSet$

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** $((i'' = i'$ **and** $PrepareLog[sn].req \neq CommitLog'[sn].req)$ **or** $i'' < i)$ **then** (s_k is faulty)
7: send $\langle$FORK-I$, i, s_k, sn, m, m'\rangle$ to $\forall s_{k''} \in \Pi$
8: add s_k to $FSet$

9: (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** $(PrepareLog[sn] = null$ **or** $PrepareLog[sn].req \neq CommitLog'[sn].req)$ **then** (s_k *might be* faulty)
10: send $\langle$FORK-II-QUERY$, i, s_k, sn, m\rangle$ to $\forall s_{k''} \in sg_{i''}$
11: wait for 2Δ time

12: **upon** reception of $\langle$FORK-II-QUERY$, i, s_k, sn, m\rangle$ at s_j, where $finalProof$ in m is generated in view i'' and $s_j \in sg_{i''}$ **do**
13: **if** $PrepareLog[sn]$ in m is not consistent with $VCSet_{s_j}^{i''}$ **then**
14: send $\langle$FORK-II$, i, s_k, sn, m, finalProof_{s_j}[i''], finalSet_{s_j}^{i''}\rangle$ to $\forall s_k \in \Pi$

15: **upon** reception of $\langle$FORK-II$, i, s_k, sn, m, finalProof, finalSet\rangle$ **do**
16: add s_k to $FSet$

17: **upon** reception of STATE-LOSS, FORK-I or FORK-II message m **do**
18: forward m to $\forall s_k \in \Pi$

Appendix B

XPaxos Formal Proof

In this appendix, we first prove safety (consistency) and liveness (availability) properties of XPaxos. To prove safety (Sec. B.1), we show that when XPaxos is outside anarchy, consistency is guaranteed. In liveness section (Sec. B.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 Sec. B.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 Fig. B.1 to facilitate our proof of XPaxos. All predicates in Fig. B.1 are defined with respect to *benign* clients and replicas.

B.1 Safety (Consistency)

Theorem 3. *(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 5 which shows a useful relation between predicates delivered() and accepted().

Lemma 5. *(view exists) delivered(c, req, rep) $\Leftrightarrow \exists$ view i: accepted(c, req, rep, i).*

Proof: By common case protocol Algorithm 5 *lines*:{26-29} and Algorithm 6 *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 6. *(reply is correct) If accepted(c, req, rep, i), then rep is the reply of req executed by correct replica.*

Proof :

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

Figure B.1: XPaxos proof notations.

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 5 *lines*:{26-29} and Algorithm 6 *lines*:{22-25}.

3. Q.E.D.
   Proof: By 1 and 2. □

By Lemma 5 and Lemma 6, we assume $\exists$ view i for req and $\exists i'$ for req', then we instead prove :

Theorem 4. *(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 7. *(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 5 *lines*:{26-29} and Algorithm 6 *lines*:{22-25}.

2. Benign active replica $s_j \in sg_i$ sends REPLY message for req only upon $\exists sn$: executed(req, i, sn, s_j).

Proof: By common case protocol Algorithm 5 *lines*:{21-25} and Algorithm 6 *lines*:{18-21}.

3. Q.E.D.
Proof: By 1 and 2. □

By Lemma 7, we assume $\exists$ sequence number sn for req and $\exists sn'$ for req'. Then we instead prove:

Theorem 5. *(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 5, we first prove several lemmas below (from Lemma 8 to Lemma 14).

Lemma 8 proves that if a request is executed by a benign active replica, then that request has been committed by the same replica.

Lemma 8. *If executed(req, i, sn, s_j), then committed(req, i, sn, s_j).*

Proof: By common case protocol Algorithm 5 *lines*:{10-21} and Algorithm 6 *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 9. *(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 5 *lines*:{8-9} or Algorithm 5 *lines*:{15-17}, or Algorithm 6 *lines*:{7-9} or Algorithm 6 *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 5 *lines*:{18-20} or Algorithm 6 *lines*:{16-17}, and definition of committed().

3. Q.E.D.
Proof: By 2 and 1. □

Lemma 10 locates at the heart of XPaxos safety proof, which is proved by induction. By Lemma 10 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 10. *(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 7 *lines*:{23,26,28}, a benign active replica generates a PREPARE or COMMIT message in view i' only upon the replica has executed Algorithm 7 *lines*:{16} in view i'; then by Algorithm 7 *lines*:{13-15}.

3. $\exists s_{j'} \in sg_{i'} : s_{j'}$ 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 $s_j \in sg_i$.
   Proof: By 2 and 3, view change protocol Algorithm 7 *lines*:{13-15} 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, $s_j \in sg_i$ 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 $i'' \geq i$.
   Proof: By Algorithm 7 *lines*:{6-7}, 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 7 *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 $i''' > i'' \geq i$; then by 1.

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

By Lemma 10 we can easily get Lemma 11.

Lemma 11. *If sg-committed(req, i, sn) and sg-committed(req', i', sn), then $req = req'$.*

Proof: By Lemma 10 and definition of sg-committed(). □

Lemma 12. *If executed(req, i, sn, s_j), then $\forall sn' < sn : \exists req'$ s.t. committed(req', i, sn', s_j).*

Proof: By common case protocol Algorithm 5 *lines*:{21-22} and Algorithm 6 *lines*:{18-19}, correct active replicas execute requests based on order defined by committed sequence number; by executed(req, i, sn, s_j) and $sn' < sn$, executed(req', i, sn', s_j); and, by Lemma 8. □

Lemma 13. *(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 12, $\exists req''$ s.t. *committed*(req'', i, sn, s_j); by Lemma 9, $req'' = req$; by common case protocol Algorithm 5 *lines*:{21-22} and Algorithm 6 *lines*:{18-19}, benign active replicas execute requests based on order defined by committed sequence number sn and sn'; and, by $sn < sn'$. □

Lemma 14. *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 13. □

Now we can prove Theorem 5.
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 8.

   When $i < i'$:

2. sg-committed(req, i', sn).
   Proof: By sg-executed(req', i', sn'), Lemma 12 and $sn < sn'$, $\exists req''$: $sg-committed(req'', i', sn)$; then by Lemma 11, sg-committed(req, i, sn) and $i < i'$, $req'' = req$.

3. $\forall$ benign active replica $s_{j'} \in sg_{i'}$: prefix($req, req', s_{j'}$).
   Proof: By sg-executed(req', i', sn'), 2, $sn < sn'$ and Lemma 14.

   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 14.

   When $i > i'$:

5. $\exists req''$: sg-committed(req'', i', sn).
   Proof: By Lemma 12, sg-executed(req', i', sn') and $sn < sn'$.

6. $req'' = req$.
   Proof: By 5 and Lemma 11.

7. $\forall$ benign active replica $s_{j'} \in sg_{i'}$: prefix(req, req', s_j).
   Proof: By 5, 6, sg-executed(req', i', sn'), $sn < sn'$ and Lemma 14.

8. Q.E.D.
   Proof: By 3, 4 and 7. □

B.2 Liveness (Availability)

Before proving liveness property, we first prove two Lemmas (15 and 16).

Lemma 15. *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 8 *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 8 *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_j has not received $t + 1$ matching signed REPLY messages for req.
   Proof: By 3, 4 and Algorithm 8 *lines*:{18-22}.

   Either,

6. s_j starts $timer_{req_c}$.
   Proof: By 2, 4 and Algorithm 8 *lines*:{3,6}.

7. s_j suspects view i when $timer_{req_c}$ expires.
   Proof: By 4, 5, 6 and Algorithm 8 *lines*:{8-10}.

   or,

8. s_j starts $timer_i^{vc}$ in view change to i.
   Proof: By Algorithm 7 *lines*:{15}.

9. s_j suspects view i when $timer_i^{vc}$ expires.
   Proof: By 2, 8 and Algorithm 7 *lines*:{34-35}.

10. Q.E.D.
    Proof: By 1 and 7, 9. □

Lemma 16. *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 7 *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 15. □

Theorem 6. *(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 5 and Lemma 15.

3. View change is executed for infinite times.
   Proof: By 1 and 2, Algorithm 8 *lines*:{11-15} and Algorithm 8 *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 (within synchronous group).
   Proof: Eventual synchrony assumption.

5. $\exists$ view i': $\forall$ active replica $s_{j'} \in sg_{i'}$ s.t. $s_{j'}$ 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 16 and c is correct.

7. Q.E.D.
   Proof: By 1, 6, Lemma 5 and contradiction. □

B.3 Fault Detection (FD)

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

At first, by Definition 12, we define the type of messages which can violate consistency in anarchy.

Definition 12. *(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 $s_j \in sg_i$;
(ii) $\exists$ view $i' < i$ 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) $\nexists i'''$ ($i''' > i''$ and $i''' > i'$) and $s_{k'''} \in sg_{i'''}$: committed$(req, i''', sn, s_{k'''})$.

Then we can prove:

Lemma 17. *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 12 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 behavior of non-crash faulty replicas, so m should not affect the state of any correct replica; or m is sent to a crash faulty or passive replica, which has stopped processing or ignores m.

2. If Definition 12 property (ii) is not satisfied, then req has not been committed by some correct replica previously in $sg_{i'}$, hence accepted(c, req, rep, i') is not true and no request is delivered by any benign client.

3. If neither Definition 12 property (iii).(1) nor (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 9 *lines*:{11-21}, m facilitates req to be committed in view i; or if $s_k \notin sg_{i'}$, then either (i) $PrepareLog[sn].req$ is generated in $i'' < i'$, then even if $PrepareLog[sn].req \neq req$, based on Algorithm 9 *lines*:{13,14,18} $PrepareLog[sn].req$ cannot be selected during view change to i if no (faulty) replica in i' sends inconsistent message, hence we consider in this case s_k is harmless; or (ii) $i'' \geq i$‘ and $PrepareLog[sn].req = req$, it facilitates correct commitment as well.

4. if Definition 12 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 (faulty) replicas in $sg_{i'''}$ has to send a non-crash faulty message; otherwise, based on Algorithm 9 *lines*:{11-21}, any non-crash faulty message generated in i'' will be ignored, since at least one VIEW-CHANGE message in view $i''' > i''$ and generated by correct replica can be collected during view change to i. □

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 7. *(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 17 it is sufficient to prove: in view change to i, if m is a non-crash faulty message from replica s_k, then correct active replica $s_j \in sg_i$ detects s_k as faulty.

2. By Definition 12 property (ii), every correct replica $s_{k'} \in sg_{i'}$ has commit log of req generated in view equal to or higher than i'.
Proof: By Lemma 10.

Assume that i_0 is the highest view in which commit log of *req* was once generated ($i' \leq i_0 < i$).

If $i_0 = i'$:

3. Every 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 12 and Lemma 10.

4. If m satisfies Definition 12 property (iii).(1), then s_j detects the fault of s_k.
   Proof: By Definition 12 property (iii).(1), prepare log of *req* is not included in m; then by 3 and Algorithm 10 *lines*:{3}, the fault is detected.

5. If m satisfies Definition 12 property (iii).(2), then s_j detects the fault of s_k.
   Proof: By Definition 12 property (iii).(2), the prepare log at sequence number sn is generated in view $i'' < i$, then by 3 and Algorithm 10 *lines*:{6} the fault of s_k is detected.

If $i_0 > i'$:

6. Every correct active replica in view i_0 has prepared *req* in the view equal to or higher than i_0.
   Proof: By 2, $i_0 > i'$ and Algorithm 7 *lines*:{26,28}.

7. In order to modify request committed at sn (i.e., *req*), at least one (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 6, $i_0 > i'$ and Algorithm 9 *lines*:{19}.

8. Correct active replica $s_j \in sg_i$ should receive m' which contains commit log of *req* generated in view i_2 ($i' \leq i_2 \leq i_0 \leq i_1 < i$) from correct active replica $s_{k'} \in sg_{i'}$.
   Proof: By outside anarchy, 2, Definition 12 and Lemma 10.

9. If $i_2 < i_1$, then the fault of $s_{k'''}$ is detected by Algorithm 10 *lines*:{9-16}; if $i_2 = i_1$, then the fault of $s_{k'''}$ is detected by Algorithm 10 *lines*:{3,6}, which is similar to discussion in 4 or 5.

10. Q.E.D.
    Proof: By 3, 4, 5 and 6 and 10. □

Theorem 8. *(Strong accuracy) If a replica s_k is benign (i.e., behaves faithfully), then XPaxos FD will never detect s_k as faulty.*

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 5 *lines*:{9,17}, Algorithm 6 *lines*:{9,15} and view-change Algorithm 9 *lines*:{1}, s_k sends a prepare log at sequence number sn once s_k prepared a request at sn; by Algorithm 7 *lines*:{6-7}, correct replicas process messages in ascending view order, hence $i'' \geq i'$.

3. s_k will not be detected by Algorithm 10 *lines*:{3} due to committed$(req, i', sn, s_{j'})$.
   Proof: By 2 and Algorithm 10 *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 9.

5. s_k will not be detected by Algorithm 10 *lines*:{6} due to committed$(req, i', sn, s_{j'})$.
   Proof: By 4 and Algorithm 10 *lines*:{6}.

6. s_k will not be detected by Algorithm 10 *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 10 *lines*:{9}, Algorithm 9 *lines*:{3-7} and Lemma 10, 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. □

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

Lemma 18. *In view change to i, if a correct active replica $s_j \in sg_i$ detects the fault of s_k, then eventually every correct replica detects the fault of s_k.*

Proof: By Algorithm 10 *lines*:{6-7}. □