



# Towards Performance and Dependability Benchmarking of Distributed Fault Tolerance Protocols

Divya. Gupta

## ► To cite this version:

Divya. Gupta. Towards Performance and Dependability Benchmarking of Distributed Fault Tolerance Protocols. Systems and Control [cs.SY]. Université Grenoble Alpes, 2016. English. NNT : 2016GREAM005 . tel-01376741

**HAL Id: tel-01376741**

**<https://theses.hal.science/tel-01376741>**

Submitted on 5 Oct 2016

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

## THÈSE

Pour obtenir le grade de

**DOCTEUR DE L'UNIVERSITÉ GRENOBLE ALPES**

Spécialité : **Informatique**

Arrêté ministériel : Arrêté *n°*

Présentée par

**Divya GUPTA**

Thèse dirigée par **Pr. Sara Bouchenak**

préparée au sein **Laboratoire d'Informatique de Grenoble**  
et de **École Doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique**

## Performance et fiabilité des protocoles de tolérance aux fautes

Thèse soutenue publiquement le **Mars 18, 2016**,  
devant le jury composé de :

**Pr. Noel De Palma**

Grenoble Université Alpes, LIG, Président

**Asc. Pr. Eddy Caron**

Ecole Normale Supérieure de Lyon, Rapporteur

**Pr. Gilles Grimaud**

Université de Lille 1, Rapporteur

**Asc. Pr. Luciana Arantes**

Université Pierre et Marie Curie, LIP6, Examinatrice

**Pr. Sara Bouchenak**

INSA Lyon, LIRIS, Directrice de thèse



*This thesis is dedicated to my beloved parents and grandparents.  
For their endless love, support and encouragement*

## Acknowledgments

I would like to take this opportunity, to express my sincere gratitude to the people for inspiring me to embark on my PhD candidature. My deepest appreciation to my supervisor who guided me throughout on this vast research area, Professor Sara Bouchenak. I am particularly thankful to her for giving me the opportunity to pursue this research under her guidance. My immeasurable appreciation for her time, emotional support, tremendous encouragement and for sharing an invaluable experience. I thank her for generously guiding me through all the phases of my PhD with enormous patience. I would like to extend my gratitude to the jury members, Eddy Caron and Gilles Grimaud, for accepting to evaluate my work and provide their valuable comments, Luciana Arantes, for accepting to be the examiner, and Noel De Palma, for chairing the jury committee.

I am obliged to thank Vivien Quéma, for occasional conversations on my research topic; his strong theoretical and technical skills greatly helped in structuring the solution for the complex problem of Byzantine. I am also grateful to Vania Marangozova-Martin, for giving her time to review my contributions and manuscript.

I am thankful to Damian Serrano for his profitable discussions and advice on my work, and also guiding me through the initial phases of a PhD student. My sincere thanks to Lucas Perronne, for critiquing dispassionately about various aspects of the problems for understanding them better and improving the research approaches. I am fortunate to have wonderful colleagues in my team, ERODS. They have been very generous friends, supporting and motivating me throughout my work. My very special thanks to Raquel Oliveira, for being there by my side like a guiding light, helping me carve my path with immense patience. I am greatly indebted to her for encouraging and comforting me during all my tough times. I acknowledge the help and support from Ecole Doctorale: Zilora Zouaoui, Pierre Tchounikine, Florence Maraninchi, Pierre Geneves and Brigitte Nonque for kindly helping me with professional and French bureaucratic problems. I am also grateful to the administration staff at LIG: Pascal Poulet, Laurence Schimicci, Muriel Paturel and Amelie Vazquez for assisting me with the complicated, time consuming administrative formalities.

This dissertation would not have been possible without the unconditional love and constant encouragement from my parents, sister, brother and other family members. My grandfather taught me to work hard and value time. My parents always motivated me to achieve the impossible and go beyond the limits. Even though we are miles apart, but their positive thoughts and energy

could be felt all the time. I am lucky to have my sister, Somya, who has been like a teacher, motivating me in the worse times. She stood firm like a pillar, giving me the strength to face all the difficulties. How can I forget, Nimit, my brother, his great discussions, reviews, comments and advice, throughout my PhD career. He made sure that I smiled every day, no matter what. I am fortunate to have two best friends, Shipra and Sushma, for their cheering and sharing my life experiences. I would like to express my warm thanks to all my friends across the globe and in France, for being supportive, encouraging, and believing in me, during my PhD journey.

## Résumé

A l'ère de l'informatique omniprésente et à la demande, où les applications et les services sont déployés sur des infrastructures bien gérées et approvisionnées par des grands groupes de fournisseurs d'informatique en nuage (Cloud Computing), tels Amazon, Google, Microsoft, Oracle, etc., la performance et la fiabilité de ces systèmes sont devenues des objectifs primordiaux. Cette informatique a rendu particulièrement nécessaire la prise en compte des facteurs de la Qualité de Service (QoS pour Quality of Service), telles que la disponibilité, la fiabilité, la vivacité, la sûreté et la sécurité, dans la définition complète d'un système. En effet, les systèmes informatiques doivent être résistants aussi bien aux défaillances qu'aux attaques et ce, afin d'éviter qu'ils ne deviennent inaccessibles, entraînent des coûts de maintenance importants et la perte de parts de marché. L'augmentation de la taille et la complexité des systèmes en nuage rend de plus en plus commun les défauts, augmentant la fréquence des pannes, et n'offrant donc plus la Garantie de Service visée. Les fournisseurs d'informatique en nuage font ainsi face épisodiquement à des fautes arbitraires, dites Byzantines, durant lesquelles les systèmes ont des comportements imprévisibles comme, par exemple, des réponses incorrectes aux requêtes d'un client, l'envoi de messages corrompus, la temporisation intentionnelle dans l'échange de messages, le refus d'honorer des requêtes, etc.

Ce constat a amené les chercheurs à s'intéresser de plus en plus à la tolérance aux fautes byzantines (BFT pour Byzantine Fault Tolerance) et à proposer de nombreux prototypes de protocoles et logiciels. Ces solutions de BFT visent non seulement à fournir des services cohérents et continus malgré des défaillances arbitraires, mais cherchent aussi à réduire le coût et l'impact sur les performances des systèmes sous-jacents. Néanmoins les prototypes BFT ont été évalués le plus souvent dans des contextes ad-hoc, soit dans des conditions idéales, soit en limitant les scénarios de fautes. C'est pourquoi ces protocoles de BFT n'ont pas réussi à convaincre les professionnels des systèmes distribués de les adopter. Tandis que certains considèrent les protocoles de BFT trop coûteux et complexes à mettre en place pour contrer des défaillances arbitraires, d'autres sont tout simplement sceptiques quant à l'utilité de ces techniques. Cette thèse entend répondre à ce problème en proposant un environnement complet de banc d'essai dont le but est de faciliter la création de scénarios d'exécution utilisables pour aussi bien analyser que comparer l'efficacité et la robustesse des propositions BFT existantes.

Dans ce contexte, les contributions de cette thèse sont les suivantes :

- Nous introduisons une architecture générique pour analyser des protocoles distribués. Cette architecture comprend des composants réutilis-

ables permettant la mise en œuvre d’outils de mesure des performances et d’analyse de la fiabilité des protocoles distribués. Cette architecture permet de définir la charge de travail, de défaillance, et l’injection de ces dernières. Elle fournit aussi des statistiques de performance, de fiabilité du système de bas niveau et du réseau. En outre, cette thèse présente les bénéfices d’une architecture générale.

- Nous présentons BFT-Bench, le premier système de banc d’essai de la BFT, pour l’analyse et la comparaison d’un panel de protocoles BFT utilisés dans des situations identiques. BFT-Bench permet aux utilisateurs d’évaluer des implémentations différentes pour lesquels ils définissent des comportements défaillants avec différentes charges de travail. Il permet de déployer automatiquement les protocoles BFT étudiés dans un environnement distribué et offre la possibilité de suivre et de rendre compte des aspects performance et fiabilité. Parmi nos résultats, nous présentons une comparaison de certains protocoles BFT actuels, réalisée avec BFT-Bench, en définissant différentes charges de travail et différents scénarii de fautes. Cette réelle application de BFT-Bench en démontre l’efficacité.

Globalement, cette thèse vise à faciliter l’analyse de performance et de fiabilité de la BFT afin d’en encourager l’utilisation aussi bien par les développeurs des protocoles BFT que ses utilisateurs finaux. Le logiciel BFT-Bench a été conçu en ce sens pour aider les utilisateurs à comparer efficacement différentes implémentations de BFT et apporter des solutions effectives aux lacunes identifiées des prototypes BFT. De plus, cette thèse défend l’idée que les techniques BFT sont nécessaires pour assurer un fonctionnement continu et correct des systèmes distribués confrontés à des situations critiques.

## Abstract

In the modern era of on-demand ubiquitous computing, where applications and services are deployed in well-provisioned, well-managed infrastructures, administered by large groups of cloud providers such as Amazon, Google, Microsoft, Oracle, etc., performance and dependability of the systems have become primary objectives. Cloud computing has evolved from questioning the Quality-of-Service (QoS) making factors such as availability, reliability, liveness, safety and security, extremely necessary in the complete definition of a system. Indeed, computing systems must be resilient in the presence of failures and attacks to prevent their inaccessibility which can lead to expensive maintenance costs and loss of business. With the growing components in cloud systems, faults occur more commonly resulting in frequent cloud outages and failing to guarantee the QoS. Cloud providers have seen episodic incidents of arbitrary (i.e., Byzantine) faults where systems demonstrate unpredictable conducts, which includes incorrect response of a client's request, sending corrupt messages, intentional delaying of messages, disobeying the ordering of the requests, etc.

This has led researchers to extensively study Byzantine Fault Tolerance (BFT) and propose numerous protocols and software prototypes. These BFT solutions not only provide consistent and available services despite arbitrary failures, they also intend to reduce the cost and performance overhead incurred by the underlying systems. However, BFT prototypes have been evaluated in ad-hoc settings, considering either ideal conditions or very limited faulty scenarios. This fails to convince the practitioners for the adoption of BFT protocols in a distributed system. Some argue on the applicability of expensive and complex BFT to tolerate arbitrary faults while others are skeptical on the adeptness of BFT techniques. This thesis precisely addresses this problem and presents a comprehensive benchmarking environment which eases the setup of execution scenarios to analyze and compare the effectiveness and robustness of these existing BFT proposals.

Specifically, contributions of this dissertation are as follows.

- First, we introduce a generic architecture for benchmarking distributed protocols. This architecture comprises reusable components for building a benchmark for performance and dependability analysis of distributed protocols. The architecture allows defining workload and faultload, and their injection. It also produces performance, dependability, and low-level system and network statistics. Furthermore, the thesis presents the benefits of a general architecture.
- Second, we present BFT-Bench, the first BFT benchmark, for analyzing



and comparing representative BFT protocols under identical scenarios. BFT-Bench allows end-users evaluate different BFT implementations under user-defined faulty behaviors and varying workloads. It allows automatic deployment of these BFT protocols in a distributed setting with ability to perform monitoring and reporting of performance and dependability aspects. In our results, we empirically compare some existing state-of-the-art BFT protocols, in various workloads and fault scenarios with BFT-Bench, demonstrating its effectiveness in practice.

Overall, this thesis aims to make BFT benchmarking easy to adopt by developers and end-users of BFT protocols. BFT-Bench framework intends to help users to perform efficient comparisons of competing BFT implementations, and incorporating effective solutions to the detected loopholes in the BFT prototypes. Furthermore, this dissertation strengthens the belief in the need of BFT techniques for ensuring correct and continued progress of distributed systems during critical fault occurrence.

# Contents

<b>List of Figures</b>	<b>xiv</b>
<b>List of Tables</b>	<b>xv</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Background and Motivation . . . . .	2
1.2 Problem Statement and Research Challenges . . . . .	6
1.3 Contribution of the Thesis . . . . .	9
1.4 Organization of the Thesis . . . . .	10
<b>2 Related Work</b>	<b>13</b>
2.1 Distributed System Characterizations . . . . .	15
2.1.1 Interaction Models . . . . .	16
2.1.2 Network . . . . .	17
2.1.3 Verification & Authentication Mechanisms . . . . .	18
2.1.4 Fault Categorization . . . . .	20
2.2 State Machine Replication . . . . .	22
2.2.1 Definitions . . . . .	23
2.2.2 Message Primitives . . . . .	24
2.2.3 Types of Replication . . . . .	25
2.2.4 Problem of Consensus . . . . .	26
2.2.5 System Model . . . . .	27
2.3 Byzantine Fault Tolerance . . . . .	28
2.3.1 Understanding $3f+1$ Bound . . . . .	29
2.3.2 Types of Byzantine Behaviors . . . . .	31
2.4 BFT Protocols at Present . . . . .	33
2.4.1 BFT from Theory to Practice . . . . .	33
2.4.2 Group 1: Performance Enhancements in Fault-Free Conditions . . . . .	33
2.4.3 Group 2: Minimizing Performance Degradation in Faulty Conditions . . . . .	48
2.5 Benchmarking Tools . . . . .	56
2.5.1 Performance Benchmarks . . . . .	56

2.5.2	Dependability Benchmarks . . . . .	58
2.6	Discussion . . . . .	60
<b>3</b>	<b>A General Architecture for Performance and Dependability Benchmarking of BFT Protocols</b>	<b>63</b>
3.1	Overview and Objectives . . . . .	66
3.2	Dependability and Performance Benchmarking Specifications and Validations . . . . .	68
3.2.1	Categorization . . . . .	69
3.2.2	Measures . . . . .	70
3.2.3	Experimental Dimensions . . . . .	70
3.3	General Benchmarking Architecture and Framework for Distributed Protocols . . . . .	73
3.3.1	Benchmarking Steps . . . . .	73
3.3.2	High-level class Diagram of Performance and Dependability Benchmark Architecture . . . . .	75
3.3.3	Overview of Communication Primitives Operation by Orchestrator . . . . .	78
3.4	BFT-Bench: Case Study of Benchmarking BFT Protocols . . . . .	79
3.4.1	Faultload Dimensions . . . . .	80
3.4.2	Workload Dimensions . . . . .	81
3.4.3	Measurement Analysis . . . . .	82
3.4.4	Potential Benchmark Users . . . . .	82
3.5	Benefits of General Architecture . . . . .	83
3.5.1	Reduction in Software Development Cost . . . . .	83
3.5.2	Extensibility . . . . .	83
3.5.3	Reusability . . . . .	84
3.5.4	Testability . . . . .	84
3.6	Summary . . . . .	84
<b>4</b>	<b>BFT-Bench: Performance and Dependability Benchmarking Framework for BFT Protocols</b>	<b>85</b>
4.1	Background . . . . .	88
4.2	Objectives of BFT-Bench . . . . .	89
4.3	Design Principles of BFT-Bench Framework . . . . .	90
4.3.1	BFT Protocols in Consideration . . . . .	90
4.3.2	Fault Types in Consideration . . . . .	93

<b>Contents</b>	<b>xi</b>
4.4 Overview of BFT-Bench . . . . .	94
4.4.1 Cluster Setup . . . . .	95
4.4.2 BFT Protocol Selection . . . . .	96
4.4.3 Faultload . . . . .	96
4.4.4 Workload . . . . .	98
4.4.5 Fault Injection . . . . .	98
4.4.6 Performance and Dependability Analysis in BFT-Bench	102
4.5 Automatic Deployment of Experiments . . . . .	104
4.6 Using BFT-Bench . . . . .	104
4.7 Portability of BFT-Bench . . . . .	105
4.7.1 Portability of Workload Injection . . . . .	105
4.7.2 Portability of Fault Injection . . . . .	106
4.7.3 Portability of Performance and Dependability Analysis	106
4.7.4 Portability of Automatic Experiment Deployer . . . . .	106
4.8 Summary . . . . .	106
<b>5 Experimental Evaluation</b>	<b>109</b>
5.1 Experimental Setup . . . . .	110
5.1.1 Hardware Settings . . . . .	110
5.1.2 Software Settings . . . . .	111
5.2 Comparative Evaluation under Faulty Scenarios . . . . .	112
5.2.1 Presence of Replica Crash . . . . .	114
5.2.2 Presence of Message Delay . . . . .	118
5.2.3 Presence of Network Flooding . . . . .	122
5.2.4 Presence of System Overloading . . . . .	126
5.2.5 Combination of Different Types of Faults . . . . .	130
5.3 Summary . . . . .	132
<b>6 Conclusions and Perspectives</b>	<b>135</b>
6.1 Conclusions . . . . .	136
6.2 Perspectives . . . . .	137
6.3 Publications . . . . .	138
6.4 Acknowledgments . . . . .	138
<b>Bibliography</b>	<b>139</b>



# List of Figures

1.1	Overview of Cloud Computing . . . . .	3
1.2	Service Models of Cloud Computing . . . . .	4
2.1	Inclusion relation of different fault types . . . . .	22
2.2	Request and response message primitives of a client-server model with state machine replication. . . . .	25
2.3	Read and write quorums need $2f + 1$ replicas to intersect in at least one correct replica executing both the operations (read- /write). . . . .	31
2.4	Communication pattern of PBFT . . . . .	34
2.5	Communication pattern of Chain . . . . .	35
2.6	Communication pattern of Ring . . . . .	37
2.7	Communication pattern of Q/U . . . . .	39
2.8	Communication pattern of Quorum . . . . .	40
2.9	Communication pattern of Zyzzyva . . . . .	41
2.10	Communication pattern of OBFT . . . . .	43
2.11	Communication pattern of two sub-protocols of CheapBFT . .	45
2.12	Communication pattern of MinBFT during normal case execu- tions . . . . .	46
2.13	Communication pattern of MinZyzzyva during non-gracious ex- ecution . . . . .	47
2.14	Communication pattern of Prime . . . . .	52
2.15	Communication pattern of RBFT . . . . .	55
3.1	Specifications and Validations of a Performance and Depend- ability Benchmark . . . . .	69
3.2	Various modules of a Generic Performance and Dependability Benchmark . . . . .	71
3.3	Benchmarking steps for performance and dependability bench- marking . . . . .	74
3.4	High-level Class Diagram of Generic Benchmark Architecture .	76
3.5	Communication Primitives Overview at Orchestrator . . . . .	79
3.6	An example of faultload descriptor . . . . .	81
4.1	Overview of BFT-Bench Framework . . . . .	95

4.2	Faultloads for different types of faults considered in Section 4.3.2	97
4.3	Architecture of Faultload Injection . . . . .	99
4.4	Performance and Dependability Benchmarking Architecture of BFT-Bench . . . . .	102
5.1	Performance evaluation of PBFT, Chain, RBFT when primary crashes . . . . .	115
5.2	Dependability analysis of PBFT, Chain, RBFT when primary replica crashes . . . . .	117
5.3	CPU and Network utilization of PBFT in presence of primary replica crash . . . . .	118
5.4	Performance evaluation of PBFT, Chain and RBFT in presence of message delay fault at primary . . . . .	119
5.5	Dependability analysis of PBFT, Chain, RBFT in presence of message delay fault at primary . . . . .	120
5.6	CPU and Network usage in presence of message delay fault at primary with $\#clients = 2$ . . . . .	122
5.7	Performance evaluation of PBFT, Chain and RBFT in presence of network flooding by a non primary replica . . . . .	123
5.8	Dependability analysis of PBFT, Chain, RBFT in presence of network flooding by a non primary replica . . . . .	125
5.9	CPU and Network usage in presence of network flooding by a non primary replica with $\#clients = 10$ . . . . .	126
5.10	Performance evaluation of PBFT, Chain and RBFT when system is overloaded with increasing number of clients at every 200s . . . . .	127
5.11	Dependability analysis of PBFT, Chain, RBFT when system is overloaded with increasing number of clients at every 200s . . . . .	128
5.12	CPU Utilization of PBFT, Chain and RBFT when system is overloaded with increasing number of clients at every 200s . . . . .	129
5.13	Performance evaluation of PBFT, Chain and RBFT in presence of system overloading with message delay fault. At every 200s, number of clients increases in the system . . . . .	130
5.14	Dependability analysis of PBFT, Chain and RBFT in presence of system overloading with message delay fault. At every 200s, number of clients increases in the system . . . . .	131

# List of Tables

2.1	Table presents some results from Dwork [51] showing the minimum number of replicas required for handling crash (or fail-stop) and Byzantine faults in synchronous and partially synchronous environments. It also presents the minimum number of replicas required to execute client request once the request is totally ordered. . . . .	30
2.2	Theoretical Analysis of BFT protocols aiming to enhance the performance in fault free scenarios . . . . .	48
2.3	Performance evaluation of robust state-of-the-art protocols under different types of attacks . . . . .	49
5.1	Hardware Configuration of the Cluster in Grid'5000 . . . . .	111
5.2	List of different faultloads considered for evaluation . . . . .	113
5.3	Comprehensive analysis of fault handling by each BFT protocol	114





CHAPTER 1

# Introduction

---

## Contents

---

1.1	Background and Motivation . . . . .	2
1.2	Problem Statement and Research Challenges . . . . .	6
1.3	Contribution of the Thesis . . . . .	9
1.4	Organization of the Thesis . . . . .	10

---

## 1.1 Background and Motivation

Cloud computing, the internet (cloud) based development using computer technology (computing), dynamically scalable, providing virtualized resources as a service over the network where users have no knowledge, expertise or control over the technology infrastructure in the cloud [2, 85]. Cloud computing enables easy access to resources, anytime, anywhere, and from any platform such as mobile devices or desktop. From an end user's perspective, a cloud is a single entity composed of potentially large numbers of configurable computing resources running multiple service instances. Cloud computing focuses on maximizing the effectiveness of the shared resources and dynamically reallocate them on-demand. It provides users various capabilities to store and process their data in third-party data centers. Figure 1.1, presents the features of a cloud computing domain.

Cloud computing is the result of the evolution and adoption of existing technologies and IT paradigms such as Virtualization, Service Oriented Architecture (SOA), Distributed and Parallel Computing. Definition of Cloud computing in National Institute of Standards and Technology (NIST) is composed of five essential characteristics, three service models, and four deployment models [85].

### **Essential characteristics of cloud systems.**

Cloud computing shares these characteristics with client-server model [9], grid computing, utility computing [10, 33, 42] and peer-to-peer distributed architecture.

1. *On-demand self-service and Cost.* Users can unilaterally provision computing capabilities, such as a number of servers, server time, replication and network storage, as needed. Cost depends on the type of services and infrastructures [4].
2. *Wide network accessibility.* Services are available over the network and accessed through standard mechanisms that promote the use of heterogeneous thin or thick client platforms, for example, mobile phones, tablets, laptops, and workstations.
3. *Resource pooling and Multitenancy.* Provider's computing resources are pooled to serve multiple consumers using a multi-tenant model, with

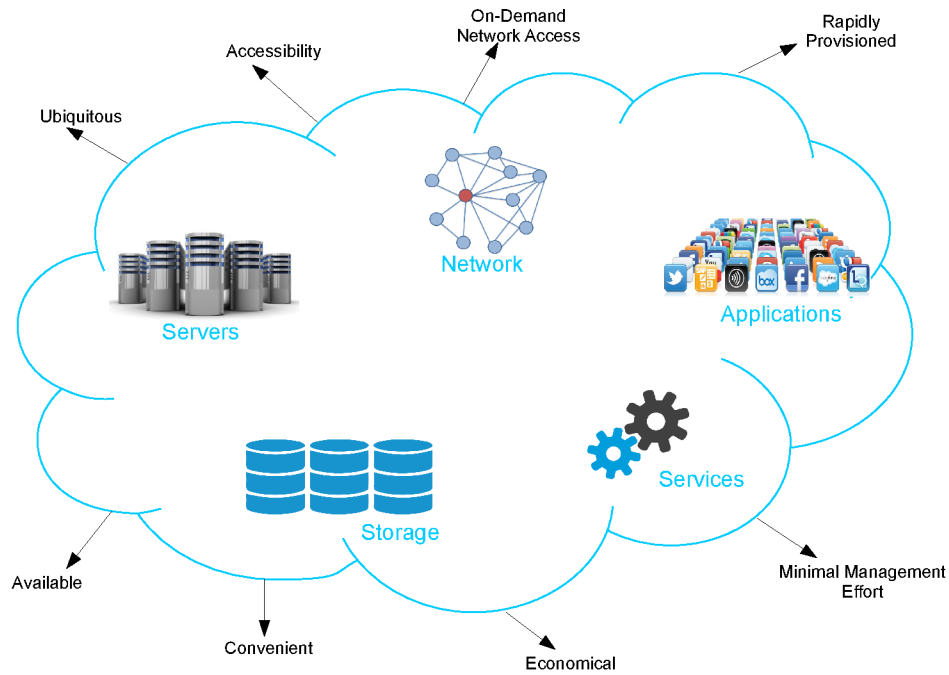


Figure 1.1: Overview of Cloud Computing

different physical and virtual resources dynamically assigned according to consumer's demand.

4. *Rapid elasticity and Scalability.* Capabilities can be elastically provisioned and released, in some cases automatically, to scale rapidly outward and inward commensurate with demand.
5. *Performance and Measured service.* Cloud systems automatically control and optimize resource use by leveraging a metering capability at some level of abstraction appropriate to the type of service (e.g., storage, processing, bandwidth, and active user accounts). Resource usage can be monitored, controlled, and reported, providing transparency for both the provider and consumer of the utilized service.
6. *Reliability.* It is provided by using replication of customers' data/applications over multiple redundant sites, which makes well-designed cloud computing suitable for business continuity and disaster recovery.

### Service Models

Figure 1.2 presents the different service models and the abstraction provided by them in terms of different layers of a system with some examples of such services.

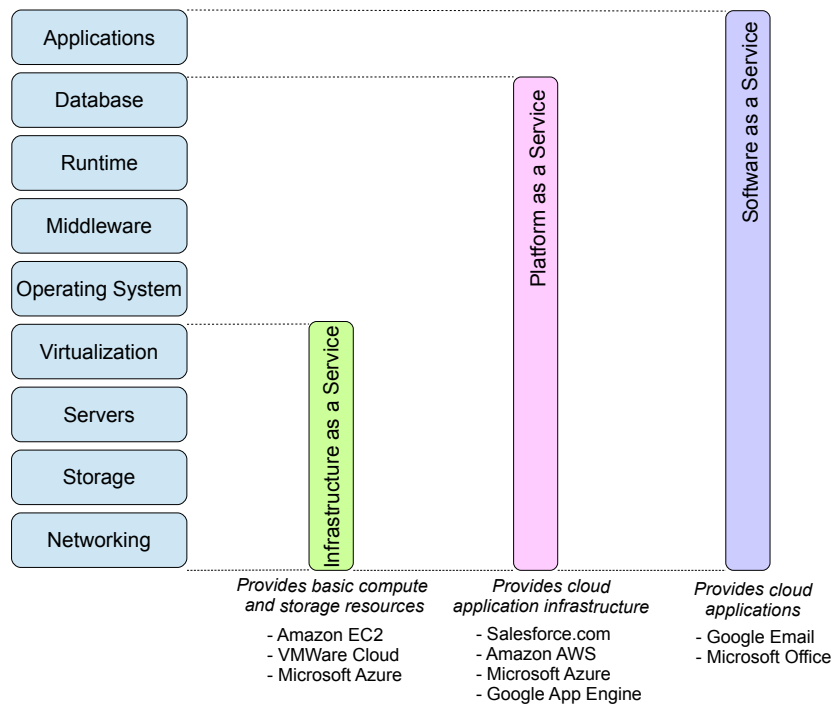


Figure 1.2: Service Models of Cloud Computing

## Deployment Models

1. *Public Cloud.* The public cloud infrastructure is provisioned for open use by the general public where its computing services are delivered over the Internet. The data center is off-premises and uses a “pay as you go” or “metered service” model.
2. *Private Cloud.* The cloud infrastructure is provisioned for exclusive use by a single organization comprising multiple consumers (e.g., business units). It may be owned, managed, and operated by the organization, a third party, or a combination of them, and it may exist on or off premises. It is not shared with another organization.
3. *Community Cloud.* The cloud infrastructure is provisioned for exclusive use by a specific community of consumers from organizations that have shared concerns (e.g., mission, security requirements, policy, and compliance considerations). It may be owned, managed, and operated by one or more of the organizations in the community, a third party, or some combination of them, and it may exist on or off premises.
4. *Hybrid Cloud.* The cloud infrastructure is a composition of two or more distinct cloud infrastructures (private, community or public) that remain

unique entities, but are bound together by standardized or proprietary technology that enables data and application portability (e.g., cloud bursting for load balancing between clouds).

5. There are many more deployment models such as *Distributed cloud* [41], *Inter cloud* [23] and *Multi cloud* [72, 91].

Currently, most of the cloud applications or services are deployed in well-provisioned and well-managed infrastructures administered by large groups of cloud providers such as Amazon, Google, and Microsoft. Cloud computing has evolved from addressing the Quality of Services (QoS) and with its expansion, factors such as availability, reliability, liveness, safety, and security of the systems have become important in the complete description of the system. With the increase in hardware and software components in cloud systems, faults occur more commonly alike exceptions, resulting in failure to meet expected consumer QoS.

Furthermore, the performance of these systems degrades when servers encounter arbitrary (i.e. Byzantine) faults [80]. Cloud providers have seen episodic incidents of such Byzantine faults in practice. For instance, Google Compute Engine (GCE) became unavailable as GCE's virtual network stopped issuing routing information, Apple iCloud cloud-based services went down while some were extremely slow. Rackspace suffers from Distributed Denial-of-Service (DDoS) attack which caused malfunctioning of DNS servers and blocked network. Another example, Amazon Web Services suffered long hauls of unavailability due to increased error rate for DNS. But for none of the cases, it could be proved that the techniques proposed by state-of-the-art BFT protocols are adept in preventing such faults. Some argued on the assimilation of expensive BFT to tolerate faults which perhaps could be handled with simpler approaches such as error detection checksums, voting, etc. Recently, some researchers are also skeptical about the applicability of BFT protocols in cloud systems and questions the integration of very complex and difficult to implement BFT protocols in practice [25, 111]. Alysson et al. proposed to first use BFT protocols for improving the security of critical systems, particularly, critical infrastructures (e.g., power plant distributed control networks), network infrastructure systems (e.g., name services, authentication systems, firewalls) and coordination services for open and dynamic systems (e.g., wireless ad-hoc and P2P networks) [14]. However, Byzantine community stands affirm on the need of BFT for ensuring correct and continued progress of cloud services during critical fault occurrence [16, 25]. They claim that BFT protocols are scalable, capable of being optimized for high load, sustain transaction streams and variant enough to terminate expensive execution when there are

no faults [25]. Thus, to attain non-disrupted availability, BFT services must be employed in cloud-based systems to prevent faulty events like network failures, software glitches, maintenance issues, faulty performance updates and hardware collapses.

Many fault tolerance mechanisms are employed by major cloud providers to prevent unavailability but failed to mask the arbitrarily occurring failure and led to periods of cloud outages. For example, Google App Engine uses automatic replication of data on number of fault tolerant servers, Amazon Web Services employ fail-over alerts and syncs back to the last known consistent state. Another instance, Netsuite uses hot backups to store data on redundant servers.

In the past fifteen years, constant improvements and enhancements have been proposed to state-of-the-art BFT protocols. Some of them focus on improving the performance of the protocols in fault-free scenarios [11, 30, 37, 40, 57, 74, 90, 98, 104]; while others consider the impact on performance in faulty cases [15, 18, 36, 105, 106]. Considerably lesser progress has been made to test the robustness and effectiveness of BFT implementations under real world conditions where nodes can demonstrate arbitrary behaviors [19]. Spinning [105], Aardvark [36], Prime [15] and Redundant BFT (RBFT) [18] performed the evaluation in the face of certain Byzantine behaviors. However, these evaluations were conducted in simple settings.

Recent fundamental trends such as increasing expensive hardware (for replication), frequent occurrence of Byzantine faults and reduction of BFT replication overhead are in line with the 3-way replication (commercial used by many systems such as Google File System (GFS) [54], bridges the gap between acceptance of BFT protocols in practice [35]. Evaluation of different BFT prototypes under various Byzantine behaviors would allow incorporate corrective measures during the designing and implementation of BFT protocols. To convince the practitioners to use BFT implementations in cloud systems, we believe, it is important to have performance and dependability metrics showing the robustness and effectiveness of BFT implementations in practice.

## 1.2 Problem Statement and Research Challenges

Cloud computing models are attracting a lot of users' with alluring benefits such as reliability, scalability, on-demand access, availability, etc. However, these models still face challenges of maintaining the performance, reliability,

security of user data in the presence of arbitrary incidents. Companies such as Amazon, Microsoft, Google, Apple, Facebook, Verizon, etc., have suffered from many incidents of cloud outage [3]. The reasons for such unavailability of these cloud services are not very clear. There are speculations of incorrect integration of fixes (Microsoft), cyber attacks (Facebook), etc. Companies have also experienced severe performance degradation and unreliability issues. For example, eleven cloud services, of Apple went slower, which included storage spaces like iCloud Drive, iCloud Mail, iCloud Backup, etc. and with a huge loss of users data. Reasons for this behavior are still not revealed, but it has affected a lot of cloud users relying on Apple cloud services. Similarly, AVG data center went down abruptly affecting its customer email security services across all regions. The exact cause of this outage is still to be determined, but it is speculated to be a third-party intervention to get rights on storage arrays. There are many other such cases of long duration of unavailability, performance degradation, data security, and unreliability caused by internal issues of cloud services, software faults, hardware failures, outdated network control systems, DNS issues or unknown arbitrary failures. These commonly occurring outages affirm the fact that there is a need to handle arbitrary faults in cloud services to prevent frequent performance and dependability issues.

For any cloud service to 100% available and reliable, it is important for it to be fault tolerant. Considerable work has been done on performance benchmarks for web services, databases, cluster computing, web servers, etc., by both academia and industry but we recognize the absence of scientific approaches for evaluating many fault tolerant protocols, precisely, BFT protocols which can benefit in controlling random cloud outages due to mishandled arbitrary faults [5, 6, 8, 27, 65, 70]. Fault tolerance is a fundamental requirement for reliable cloud services. Companies have matured adopting solutions for *benign faults* [73] such as crash faults using the well-known Paxos algorithm [77, 78] but they still lack fault tolerant solutions for arbitrary failures. BFT protocols replicate services in several replicas to ensure service availability and correctness despite the fault occurrence [51, 53, 76, 94, 95].

BFT problem in itself is very complex to understand and comparing many of such proposed BFT solutions can be extremely hard and time-consuming. However, considerable effort has been made in studying Byzantine faults and plenty of BFT algorithms have been introduced. Traditional approaches to compare these protocols are theoretical. We have recognized that practical evaluations of BFT protocols have been performed in the most ad-hoc settings. There is a lack of efficient approaches to characterize and empirically evaluate the performance and dependability aspects of BFT systems under several real world faulty and fault-free settings. Understanding BFT protocols in itself is



difficult and developing a benchmark solution for the same can be extremely challenging. Here we list a few of the challenging issues encountered while developing our framework:

1. **Understanding complex Byzantine behaviors.** Any arbitrary fault is said to be Byzantine. Byzantine faults are the arbitrary behaviors shown by the servers when they deviate from the correct execution of the protocol specifications. It is not only important but also very necessary to distinguish them from network failures, incorrect routing and switching, DNS issues, Denial-of-Service (DoS) attacks, etc., to prevent from incorrect detection of fault. This might lead to expensive fault detection and recovery mechanism without resolving the impacting fault.
2. **Challenging BFT implementations.** BFT protocols are generally perceived as too complex to understand because the implementations are large and the protocol logic is hidden in low-level implementation details. Although most of the protocols use PBFT as their baseline, but their communication patterns, a number of message exchanges, cryptographic operations performed, etc., makes them a class apart and equally difficult to comprehend. Each protocol has a different way of interaction between the client and server replicas which again adds on to the tricky comparisons and analysis. Most importantly, these variations add new challenges during runtime. Complex BFT implementations add higher complexities as they become vulnerable to a higher number of software bugs, and thus can reduce the overall performance and dependability of the systems.
3. **Quantitative Evaluation in Realistic Conditions.** With a dozen of practical BFT protocols proposed so far, focusing particularly on improving performance in ideal conditions such as zero latency, gigabit bandwidth, zero packet loss, and no faults, makes no sense in today's practical and complex evaluations. There is a lack of an appropriate implementation, simulation, and evaluation environment to understand their performance in realistic conditions such as high network delay, low bandwidth, frequent packet loss, and faults. Additionally, the available implementations of these protocols use different language primitives, crypto libraries, and transport protocols, which makes it difficult to compare these protocols fairly. Correct analysis of the behaviors of BFT implementations using proper performance and dependability metrics is important to demonstrate their robustness and effectiveness in practice; otherwise opportunity for BFT integration in cloud-based systems may be lost.

## 1.3 Contribution of the Thesis

In this thesis, we present a benchmark solution to evaluate and compare the performance and dependability aspects of many BFT implementations. This will make BFT benchmarking easy to adopt by developers and end-users intending to use BFT protocols. We first introduces a generic architecture for benchmarking any distributed protocol, and later we present our novel framework for benchmarking BFT implementations by adapting the components of a generic architecture and accelerating the software development process. Contributions of this thesis are as follows.

1. **General Architecture for Benchmarks for Distributed Protocols:** We introduce a generic architecture for benchmarking distributed protocols used in cloud services. This architecture comprises of various modules which are reusable and adaptable for building any benchmark for performance and dependability analysis of any distributed protocol. This architecture encompasses the generation of various workloads and faultloads, injection of these loads in an application or service running on a cloud system, and production of performance, dependability and network statistics. Furthermore, we demonstrate the benefits of having a general architecture and how it can be re-used for building cost effective benchmarks for distributed protocols used in cloud computing.
2. **BFT-Bench Framework:** We introduce BFT-Bench , a novel framework for measuring the effectiveness and robustness of BFT implementations in practice by analyzing their performance and dependability aspects in the face of real world settings. We tested our framework by evaluating state-of-the-art BFT protocols. The benchmark is extendable to incorporate other BFT code bases using portability attributes of BFT-Bench. BFT-Bench allows users to define and inject various workloads and faultloads. It also includes mechanisms for automatic deployment of experiments in the cluster and cloud environments, as well as performance and dependability monitoring and reporting.
3. **No-One-Guarantee-All Analysis:** Our extensive study of behaviors and properties of BFT protocols and practical analysis of a few of them strengthened our believe that none of the protocols is robust enough to handle all types of Byzantine faults and maintain a constant performance of fault-free scenario. We perform three steps of analysis: (1) theoretical analysis of performance on the basis of a number of message exchanges in communication pattern and number of authentication needed; which is

only limited to fault-free scenario. (2) second analysis considers a number of faults tolerated by each protocol during their evaluation analysis. and (3) final study is experimental which determines the capabilities of BFT protocols in the presence and absence of faults using BFT-Bench.

This thesis aims to BFT benchmarking easy to adopt by developers and end users of BFT protocols. BFT-Bench framework intends to help users to perform efficient comparisons of competing BFT implementations and incorporating effective solutions to the detected loopholes in the BFT prototypes.

## 1.4 Organization of the Thesis

The rest of the document is organized as follows:

### Chapter 2

Chapter 2 gives an overview of different types of faults with a major focus on Byzantine behaviors. It provides a detailed study of Byzantine fault tolerant models and their importance in today's world of cloud-dependent services. We present some BFT concepts and techniques, and give a brief introduction to some of the well-known state-of-the-art protocols (some are them are used in this thesis for evaluation) and their evolution from theory to practice. This chapter motivates and demonstrates the need for a benchmark solution which compares and evaluates dependability and performance measures of competing BFT protocols in the face of Byzantine faults. We also focus on the previous works done in the area of dependability and performance measurements of cloud services. Furthermore, we describe some fundamental outcomes of distributed computing which form the baseline for all the BFT protocols.

### Chapter 3

Chapter 3 introduces the generic architecture for performance and dependability framework for distributed protocols used by cloud services. This architecture is validated in the next chapter with our BFT-Bench-framework. We present the benchmarking design specifications and validations of the generic architecture and demonstrate how it can be re-used to build software frameworks for other distributed protocols. We present various modules that are reusable and adaptable, and can help to reduce the cost of building new software prototypes.

### Chapter 4

The BFT-Bench framework is introduced in Chapter 4. This chapter begins by presenting the overview of the BFT-Bench, the novel benchmark suite for evaluating and comparing BFT implementations in practice by measuring their performance and dependability aspects in the presence of different types

of Byzantine faults. It then presents how various components of generic architecture have been re-used and the various functions performed by them. It defines workloads and faultloads and their corresponding injections in the system for evaluating prototypes of BFT protocols. This chapter also presents the system model considered for testing the proposed benchmark. It defines different fault types and BFT implementations in considered. Further in the chapter, we demonstrate the automatic deployment of experiments in BFT-Bench framework and portability of different components of the benchmark to incorporate other fault models and BFT protocols.

### **Chapter 5**

Chapter 5 explores an interesting experimental performance and dependability evaluation of BFT implementations using BFT-Bench framework, followed by a demonstration of ineffectiveness of BFT protocols under different fault injections. The results expose many loopholes in BFT implementations, which leads to termination of prototypes when a fault is triggered. This chapter affirms that BFT prototypes have always been tested in an ad-hoc manner and why having BFT-Bench framework is important.

### **Chapter 6**

Chapter 6 summarizes the main contribution of this work followed by future research perspectives.



## CHAPTER 2

# Related Work

---

### Contents

---

<b>2.1</b>	<b>Distributed System Characterizations . . . . .</b>	<b>15</b>
2.1.1	Interaction Models . . . . .	16
2.1.1.1	Synchronous Distributed Systems . . . . .	16
2.1.1.2	Asynchronous Distributed Systems . . . . .	16
2.1.1.3	Partial Synchrony . . . . .	17
2.1.2	Network . . . . .	17
2.1.3	Verification & Authentication Mechanisms . . . . .	18
2.1.3.1	Cryptographic Hash Function . . . . .	19
2.1.3.2	Message Authentication Code (MAC) . . . . .	19
2.1.3.3	Digital Signatures . . . . .	20
2.1.4	Fault Categorization . . . . .	20
<b>2.2</b>	<b>State Machine Replication . . . . .</b>	<b>22</b>
2.2.1	Definitions . . . . .	23
2.2.1.1	Safety . . . . .	24
2.2.1.2	Liveness . . . . .	24
2.2.2	Message Primitives . . . . .	24
2.2.3	Types of Replication . . . . .	25
2.2.4	Problem of Consensus . . . . .	26
2.2.5	System Model . . . . .	27
<b>2.3</b>	<b>Byzantine Fault Tolerance . . . . .</b>	<b>28</b>
2.3.1	Understanding $3f+1$ Bound . . . . .	29
2.3.2	Types of Byzantine Behaviors . . . . .	31
<b>2.4</b>	<b>BFT Protocols at Present . . . . .</b>	<b>33</b>
2.4.1	BFT from Theory to Practice . . . . .	33
2.4.2	Group 1: Performance Enhancements in Fault-Free Con- ditions . . . . .	33

---

2.4.2.1	Agreement-Based Protocols . . . . .	34
2.4.2.2	Quorum-Based Protocols . . . . .	38
2.4.2.3	Speculation-Based Protocols . . . . .	41
2.4.2.4	Client-Based Protocols . . . . .	43
2.4.2.5	Trusted Component-Based Protocols . . . . .	44
2.4.3	Group 2: Minimizing Performance Degradation in Faulty Conditions . . . . .	48
2.4.3.1	Robust Protocols . . . . .	50
<b>2.5</b>	<b>Benchmarking Tools . . . . .</b>	<b>56</b>
2.5.1	Performance Benchmarks . . . . .	56
2.5.1.1	a/b Microbenchmarks . . . . .	57
2.5.1.2	Hermes Framework . . . . .	57
2.5.2	Dependability Benchmarks . . . . .	58
<b>2.6</b>	<b>Discussion . . . . .</b>	<b>60</b>

---

This chapter characterizes a distributed system and presents the fundamentals of State Machine Replication (SMR) techniques for building robust and effective fault tolerant services. Our focus is five-fold. First, we present system definitions and assumptions of a distributed system (Section 2.1). Second, we define and discuss the SMR approaches for implementing replication management protocols to tolerate different types of faults (Section 2.2). Third, we discuss Byzantine Fault-Tolerance (BFT) concepts and properties, and describe the motivation of having a BFT service in a distributed system (Section 2.3). These three sections are important to learn for better understanding various BFT protocols discussed in the next section. Fourth, we present a concise summary of research on BFT protocols by briefly discussing some of the prominent state-of-the-art BFT protocols (Section 2.4). This section gives the taxonomy of BFT protocols proposed so far categorized under various type-based protocols. Our benchmark comparatively evaluates some of these BFT protocols to demonstrate the effectiveness and robustness of our benchmark, BFT-Bench. This section also performs a theoretical comparison of performance of the discussed BFT protocols in terms of throughput and latency. Finally, we discuss some of the existing benchmarking tools for evaluating performance and dependability aspects (Section 2.5).

## 2.1 Distributed System Characterizations

A distributed system involves the cooperation of autonomous, heterogeneous computing entities, interacting with each other over a network via different communication protocols and distributed middleware services, which enables them to coordinate their activities and share resources. Examples of such distributed systems include Service Oriented Architecture (SOA) based systems, peer-to-peer applications, distributed databases, network file systems, aircraft control systems, and cloud computing. In this section of the chapter, we present some theoretical aspects and abstractions of a distributed system model. We also discuss some traditional fault behaviors that bring resource conflicts and disruptions in continuous coordination and service availability of a distributed system. Such phenomenon led to the research on State Machine Replication (SMR), quorum replication, agreement and consensus problems, Byzantine fault tolerance and fault recovery mechanisms.

A distributed system is structured as a set of processes, called servers, coordinating together to offer underlying services to its users, called clients. The client-server architecture is usually based on a simple request/response protocol, implemented with send/receive primitives, Remote Procedure Calls



(RPC) or Remote Method Invocation (RMI). The *server* is defined as a computing resource selectively sharing its resources and services to process the client requests. The *client* is a computer or a computer process initiating contact with a server in order to use its resources and services. Each server has its own local memory and the information between servers and clients can be exchanged only by passing messages using communication channels. The network enables data access through client-server and server-to-server communication protocols. In this thesis, we consider server as a single physical machine. We will use terms servers, nodes and processes interchangeable in the entire manuscript.

### 2.1.1 Interaction Models

In a distributed system, it is difficult to set time limits on the time taken to process (service/task) execution, message delivery between two interacting servers or clock drift at a server. This can be achieved by making some assumptions at the system level. Following are three such interaction models.

#### 2.1.1.1 Synchronous Distributed Systems

Servers in synchronous distributed systems are synchronized with an external clock before they start to communicate. A system makes some strong temporal assumptions over process and communication network which include: (i) time to execute a process has known lower and upper bounds, (ii) each message transmitted over a channel is received within a known bounded time, and (iii) drift rates of local clocks from real time has a known bound. Synchronous systems give a notion of global physical time with a relative precision depending on the clock drift rate and predictable behavior in terms of timing which makes it possible and safe to use timeouts in order to detect failures (of a process or communication links). Such systems are easier to handle with lower communication overhead, but determining their realistic bounds can be hard or impossible. These are used for hard real-time applications or critical embedded systems.

#### 2.1.1.2 Asynchronous Distributed Systems

Asynchronous distributed systems are more abstract and general with no bounds on temporal process and communication network: (i) process execution may take any arbitrary time, (ii) a message can have long hauls of transmission delays (message delivery time), and (iii) clock drift rates are random. Servers exchange data intermittently, without any prior clock synchronization.

This exactly models the Internet, in which servers have different computing powers with no intrinsic bounds (on server or network load) and communication channels are unreliable (message drops, loss, delay, etc.). Therefore, servers can take any amount of time to execute a request (task/operation), or transmit a message. There is a notion of a logical clock, but only to account the passage of time and the number of events that have occurred since the system started.

### 2.1.1.3 Partial Synchrony

Partial synchrony [51] refers to eventually synchronous distributed systems where temporal bounds on computation and communications may exist, but knowledge of such bounds is limited. It is an intermediate model between synchronous and asynchronous systems to achieve optimal resiliency. These systems can be intuitively modeled as *somewhat timely* which helps distributed algorithms distinguish between a crashed process from a slow process, circumventing impossibility results of consensus in fault tolerant distributed systems [53] and others in pure asynchrony [52]. Partial synchrony asserts two constants which can vary during executions: an upper bound  $\delta$  on the maximum delay of any message, and an upper bound  $\phi$  on relative execution speeds. Values of  $\delta$  and  $\phi$  are known, but holds after some unknown Global Stabilization Time (GST). It has been generally observed, that messages are somewhat timely synchronous and server speeds do not change randomly except during communication delays or crashes. Partial synchrony can handle such occurrences of uncertain faults like node crashes [93], Byzantine behaviors [80, 89], state corruptions during consensus [48, 79, 89], atomic commit [81], mutual exclusion [46] and clock synchronization [76]. This fundamental contribution also demonstrated that by separating the *Safety* and *Liveness* properties (discussed in Section 2.2) of a protocol, it is possible to solve consensus in a partially synchronous system model. This separation was later exploited by many state-of-the-art BFT protocols [15, 18, 30, 36, 71, 77, 105].

### 2.1.2 Network

In a distributed setting, servers and clients communicate with each other through communication protocols using network links (or communication channels). These communication channels are bi-directional and used by nodes to exchange messages. Each server has two message primitives, sending and receiving:  $send(m, p)$ , where server  $p$  sends a message  $m$  and  $receive(m, q)$  where server  $q$  receives a message  $m$ .

Each message uses a unique identifier for detecting and rejecting duplicate

packets and checksums to detect and reject corrupted packets. Network failures are common and at times difficult to detect. We consider a network to be asynchronous and unreliable where the network itself may fail to deliver messages, duplicate them, delay them out of order, or corrupt them. In case of the message delivery failures, the incoming buffer of a server is full and cannot accept any new messages. This tends to reject some incoming messages which can be prevented by re-transmitting a message until receipt of its acknowledgment. Message duplication and disorder are common faulty network behaviors. These issues can be handled by using unique sequence numbers that define an order to message delivery and nonces to detect duplication of the same message. The network may also intentionally delay some messages which are difficult to distinguish from a malicious server demonstrating the same behavior (we will discuss this fault again in following chapters). This is usually handled using a timeout at receiving servers. Finally flooding and corruption of a message by the network or by an intruder, or by a malicious server are also possible. These faults overload the computational resources to perform expensive executions (precisely, verification and authentication using expensive cryptographic techniques). These faults can bring a great overhead in terms of performance degradation.

It has been argued that it is impossible to achieve a reliable communication on top of unreliable channels, considering the fact that network links are prone to be broken even if two communicating nodes are correct. A reliable channel where message  $m$  send by process  $p$  (e.g., through the primitive  $send(m, p)$ ) will be eventually delivered to the destination process  $q$  (e.g., through the primitive  $receive(m, q)$ ). A communication protocol such as TCP which is built over a best-effort protocol (IP) gives no reliability guarantees.

However, it is possible to implement a reliable channel on top of fair channels (fair-links) through the use of retransmissions and acknowledgments. More precisely, the sender keeps re-transmitting message  $m$  periodically until it receives an acknowledgment (ACK) message from its destination server. Additionally, authentication and integrity can be guaranteed by using message authentication codes (MACs) on the exchanged messages and digital signatures if non-repudiation is also one of the concerns.

### 2.1.3 Verification & Authentication Mechanisms

A fundamental concern of a distributed system is authentication and data security [66, 109]. A distributed system is susceptible to a variety of security threats imposed by intruders as well as legitimate users of the system. Legitimate users are more powerful adversaries as they possess internal state information unknown to an intruder. An adversary can corrupt the system

state, modify or delete transmitted messages, replay old messages or perform any arbitrary attack affecting the system's reliability and availability. In distributed systems, secrecy and integrity are two main requirements for a secure communication achieved via authentication protocols that verifies the identity of senders and correctness of their messages. We now discuss the mechanisms used for verification and authentication purpose in most SMR (also distributed) protocols.

### 2.1.3.1 Cryptographic Hash Function

A cryptographic hash function is considered infeasible to invert, i.e., to recreate the input data from its hash value alone. This one-way hash function takes input data (message plaintext) and generates the hash value which is often called the message digest. For any message, it is easy to compute its hash value and infeasible to modify it without changing the hash. And it is practically difficult to find two different messages with the same hash value. These hash functions are inexpensive, fast to generate and commonly used in digital signatures for verifying the integrity of message files, password verification, generating Message Authentication Codes (MACs), indexing hash tables, to detect duplication of data, and as checksums to detect data corruption. The two most commonly used cryptographic hash functions are MD5 (128 bits) and SHA-2 (up to 512 bits). Hash digest is a cryptographic checksum that each server computes to verify a message. For example, both servers use the same algorithm and share a key to compute the digest of the message which is included in the packet. The receiving server must perform digest computation on the received message, and compare it to the original (included in the packet from the sender). If the message is modified in transit (the hash values will be different), the packet is rejected. Digest can only guarantee message integrity but not the authenticity of the message, i.e., a malicious server can alter the message and regenerate the digest (of the altered message) without being detected.

### 2.1.3.2 Message Authentication Code (MAC)

A message authentication code (MAC) is used to authenticate a message while providing integrity and authenticity assurances on the message. MAC is a keyed digest where digest is encrypted using a secret key shared between communicating servers. It is necessary for the involved servers to confidentially share the keys previously which is generally done using public-key cryptography. In a client-server architecture, when a client broadcasts a message, it includes a MAC authenticator consisting of a MAC for every server. Each

MAC is calculated over the digest of the message using a previously shared secret key between the client and the corresponding server. Each server, upon receiving the MAC authenticator can verify the content of the request from its MAC (dedicated for this server). Shared key cryptography ensures message authenticity while digest ensures its integrity. However, a faulty client or server can include some corrupt MACs in the authenticator to prevent a subset of servers from verifying the request. This is a well-known faulty behavior called *MAC attack*. It is also possible for an adversary to perform a *replay attack* where it can record a MAC authenticator and resend it back at a later time without getting detected.

### 2.1.3.3 Digital Signatures

Digital signatures [87] employ asymmetric cryptography (such as RSA [67]) ensuring authenticity, integrity, and non-repudiation of a message. Asymmetric or public key cryptography uses two keys: a public key and a private key where the public key is disclosed to everyone, but private key is known only to the entity (server). A valid digital signature validates the sender (authenticity) and this sender at a later time cannot deny having sent the message (non-repudiation). It also ensures that the message was not altered in transit (integrity). Digital signatures are calculated on the digest of a message. If the message changes, so will the digest (integrity). The digest is encrypted using the private key of the sender (authenticity and non-repudiation). This encrypted digest is the digital signature. The receiver will decrypt the message using the public of the sender and verifies the digest. Digital signatures are computationally expensive than MACs, but they can guarantee the property of non-repudiation. They are commonly used for software distributions, financial transactions, and also to detect forgery or tampering.

### 2.1.4 Fault Categorization

There exist different types of faults that can happen in a system (distributed, real-time, mission-critical, etc.) at any point of time. Failures can occur both in processes and communication channels due to any software and hardware faults. These faults can affect the system at different levels of degradation in terms of performance and dependability. Fault models are necessary in order to build robust fault-tolerant systems or algorithms with predictable behaviors in case of fault occurrences. Researchers have categorized these faults according to their nature and have proposed several protocols and mechanisms to handle them.

1. **Crash Fault.** It occurs when a server in a system operates correctly until some point of time but suddenly fails to produce any results. Other processes may not be able to detect this state. Crash faults occur at the process level and are further categorized as:

- (a) *Fail-stop.* Server fails to respond and stops forever until it is manually recovered. Such crash faults take longer to repair as they need human interventions. It is not a practical way to deal with a crash fault as it can be catastrophic if not repaired within a time limit.
- (b) *Fail-recover.* Server stops responding, but it is automatically recovered after a period of time. For handling such faults, there are solutions for instant recovery such as self-rebooting the system and self-restoration to the point of failure with checkpoint synchronization from other servers.

2. **Omission Fault.** It occurs when a server is up but fails to produce results. It happens mostly due to a faulty server or a faulty communication channel. Following are the possible omission faults at a server:

- (a) *Send Omission.* The server sends a message, but it never goes to the outgoing message buffer.
- (b) *Receive Omission.* The message is there at the server's incoming message buffer, but the server never receives it.

These faults are easy to detect and resolve in synchronous systems with the help of timeouts. If a message is sure to arrive, a timeout will indicate a fault at the sending server or in the communication link. However, this fails to distinguish from a timing fault.

3. **Timing Fault.** Timing faults can occur in synchronous distributed systems, where time limits are set for process execution, communications, and clock drifts. A timing fault occurs if any of these time limits is exceeded. For example, in mission critical systems like telecommunications, power distribution (to hospitals, data centers), aerospace communications, etc., it can be catastrophic if the timing constraints are not respected. For non-critical systems, performance is affected but timing faults can be ignored (if it is not so critical); for example, loading of a webpage, accessing emails, etc.
4. **Byzantine Fault.** It is any arbitrary fault that can occur at any time. It is the most general and worst possible fault semantics omitting intended execution paths. Byzantine faults envelop all of the above mentioned faults (crash, omission & timing). They include faults like sending

incorrect response to a request, out of order execution of requests (which may change the system state), flooding of correct servers with incorrect messages (having incorrect cryptographic operations) or intentional delay in sending messages to other nodes.

With the augmenting dependencies over service providers (over the internet), it is becoming critical for systems to deal with these arbitrary failures and preserve the Quality of Service (QoS) guarantees for their end-users. Several Byzantine Fault Tolerant (BFT) algorithms have been designed and proposed to help systems cope with such unpredictable faulty behaviors of the servers. We will discuss some of these algorithms later in this chapter (see Section 2.4).

Figure 2.1 represents the inclusion relation of all the above discussed faults.

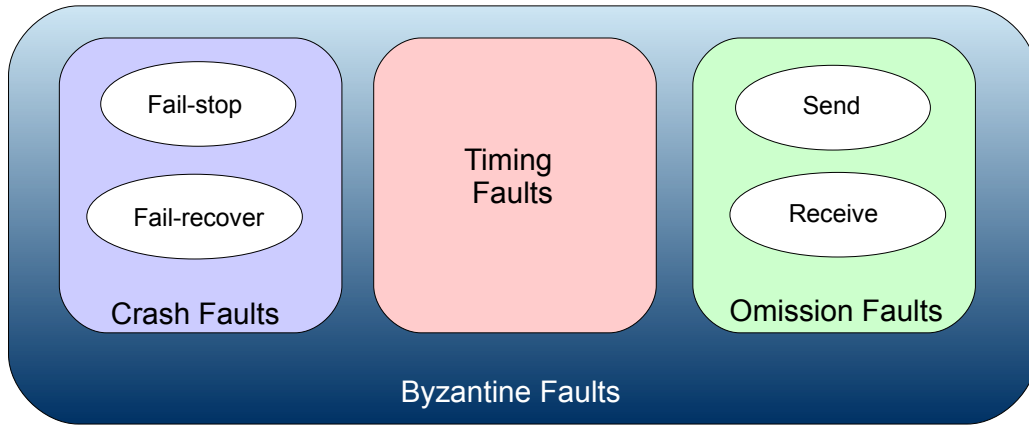


Figure 2.1: Inclusion relation of different fault types

## 2.2 State Machine Replication

State Machine Replication (SMR) is a technique to implement fault tolerant services by replicating these services on a set of servers [76, 94, 95]. In other words, copies of these services are maintained on different servers implementing state machines and clients communicate through SMR protocols. SMR technique is often implemented for guaranteeing system's performance, reliability and availability in the face of software and hardware failures.

According to Schneider [94], implementation of SMR must hold following three properties.

- *Initial state.* All the correct replicas must start with the same state. By same state we mean, having the same input and output conditions.

Arbitrary faults such as a crash, random machine reboot can bring some periods of inconsistencies. SMR employs techniques like checkpoint, local histories of all the servers, etc., to bring back the system to the same consistent state.

- *Determinism.* All the replicas should produce the same output for a given set of inputs. Determinism can be challenging to achieve due to complicated multi-threading processing or multiple cored physical machines. A deviation in output at replicas can determine a faulty state change in the replicas within a few finite number of steps. Indeed, determinism guarantees that status of multiple copies of a system will not diverge under the same inputs. In the context of replication, we consider implementing only deterministic state machines and not non-deterministic (where output depends on the inputs and internal states) due to the above stated obvious reasons.
- *Coordination.* All correct replicas process the client requests in the same order.

*Consensus* or *total order multicast* protocols [58] help the replicas to reach an agreement on the total ordering in the presence of partial synchrony.

The two main objectives of any SMR protocols are:

- *Robustness:* It is the ability to ensure availability and reliability of the services with consistent performance, despite contentions and failures in the system.
- *Consistency:* All the correct replicas produce consistent responses and their system states are always same.

These objectives are achieved through *Safety* and *Liveness* properties which we define next.

### 2.2.1 Definitions

There are 2 fundamental properties which every BFT-SMR protocol must satisfy during both fault-free and faulty scenarios. The primary aim of these protocols is to maintain the same internal state at each server where all the servers are consistent and execute the same request at any time. All the requests are executed automatically in a deterministic way, which means that they start with the same state, executes the requests in the same way and results in the same state at all the correct replicas. For example, in case of



a bank transaction, if the replicated servers do not possess the same state, it can be damaging and might result in a situation where rollback is unable to fix the issue. Considering the nature of communication channels, servers, and asynchronous networks, it becomes important for the service providers to guarantee the following properties.

### 2.2.1.1 Safety

*Safety* property states that all the correct servers execute the same request in the same order. According to the property, a system which starts in a correct state will always complete in a correct state after executing client's request. Strongly consistent services satisfying *Linearizability* [61] property can guarantee *Safety*.

### 2.2.1.2 Liveness

*Liveness* property states that all the requests coming from the correct clients will eventually be executed by all the correct replicas. SMR technique employs a replica as primary for giving sequence numbers to the incoming client requests. The correct replicas (with consensus) agree with the ordering by primary before executing the request. Total ordering ensures *Safety* property. Liveness is ensured by timeouts (bounded finite delay) at clients and replicas, i.e., under partial synchrony [51].

## 2.2.2 Message Primitives

State machine replication technique is implemented in a distributed system comprising a set of clients and servers where clients invoke an operation to a deterministic service implemented by the servers. The model is illustrated in Figure 2.2 where clients communicate with servers via request and response message primitives for an operation to be performed. In the rest of this manuscript, we will use the same model where an operation is either read-only or read-write. This separation is important because, in most BFT-SMR protocols, read-only operations can be processed without undergoing complex steps of consensus among replicas since the state of the service is not changed. This is an optimization employed by protocols to provide high performance during read-only operations.

Clients can request local logs at servers to discover inconsistencies, state changes while considering the computational overheads and performance degradation due to such operations. Whereas, servers are limited to provide responses and cannot initiate any communication with the clients. In the general

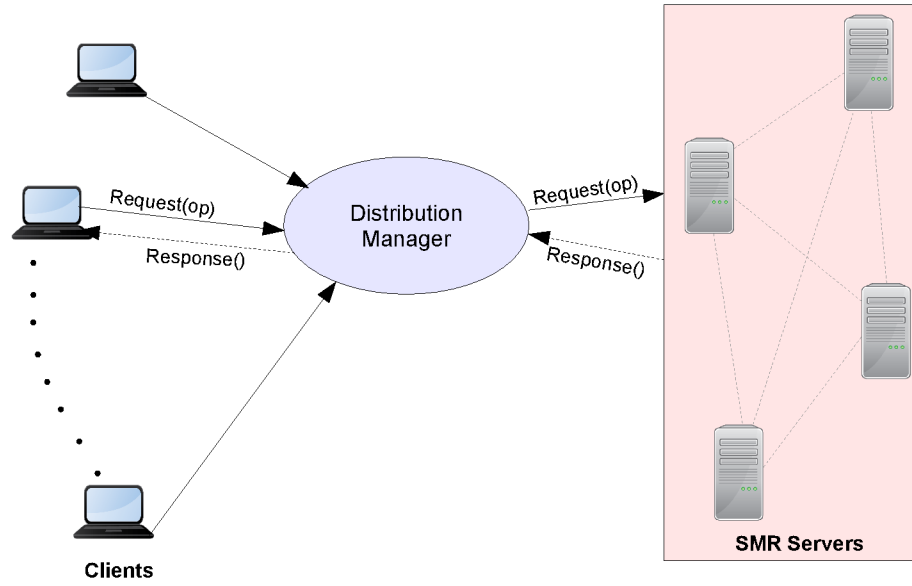


Figure 2.2: Request and response message primitives of a client-server model with state machine replication.

scenario of a distributed system, server upon receiving a request: (1) accepts a request, (2) executes a read-only or read-write operation, (3) takes a snapshot of the service state after executing the operation, (4) updates the local history with the new state, and (5) responds to the client.

### 2.2.3 Types of Replication

State Machine Replication technique is used to maintain a consistent state, guarantees on availability and reliability of an implemented service. There are two fundamental approaches to achieve the replication: (i) passive (primary-backup) replication [12] and (ii) active replication [75]. In passive replication, there is a dedicated primary replica to execute all client requests and periodically send state updates (changes) to the backup replicas. Furthermore, in case of a system failure, one of the backup replica becomes the new primary. However, this approach fails to tolerate other malicious behaviors of a primary such as, a primary can decide not to execute the client request or drop the status update at some replicas to bring inconsistencies in the future.

On the contrary, in active replication, client sends request to all the replicas, which execute the request cooperatively (in a coordinated way) and produce a reply. The client receives a response from all the replicas and can ensure their validity by comparing the responses. In this approach, a replica can be elected to define the ordering of the client requests (to ensure all repli-

cas execute the requests in the same order) and since the replicas are well synchronized before executing a request, no extra monitoring is required.

In this and forthcoming chapters, we will consider only active replication (as used by BFT protocols). Paxos [77] and PBFT [30] are the first protocols to tolerate crash faults and Byzantine faults, respectively, using active replication approach. We will discuss more protocols using the same approach for fault tolerance, later in this chapter.

### 2.2.4 Problem of Consensus

Coordination among replicas is a fundamental requirement for implementing a state machine replication where all correct replicas must agree on the same order for executing client requests. Conceptually this requirement can be satisfied by implementation of a total ordered multicast (atomic multicast) protocol [58]. Total ordering communication primitive and consensus problem, both subject to the same constraints and, thus, considered equivalent. The problem of consensus states that all the communicating processes propose different values for a request and they must agree on a single value with consensus. A consensus protocol is correct if it meets the following three conditions.

- *Integrity (Consistency)*. All agreed values must have been proposed by some process.
- *Agreement (Validity)*. All processes agree on the same 'correct' value and all decisions are final.
- *Termination*. All the correct processes eventually decide on a 'correct' value (within a finite number of steps).

Consistency property is violated if a process decides not to commit on the proposed value (this is possible due to slow processes or omission faults). Agreement is never reached if there are not enough processes validating the same proposal. Termination condition is violated if processes were never to agree.

Regardless of the faults, some processes cannot participate in the request execution leading to violations of above properties due to asynchronous distributed systems. This led to another well-known result in distributed computing, i.e., impossibility of deterministic consensus among processes in an asynchronous system [53], also called as *FLP impossibility*. This impossibility is a consequence of difficulty in identifying or detecting malicious processes in asynchronous systems. Considering a case, where a process  $p$  waits for a

message  $m$  from a process  $q$  before proceeding to the next step. In this scenario, process  $p$  cannot determine if the process  $q$  has crashed or is slow in responding. Process  $p$  might decide to wait longer or just proceed without the message from the process  $q$ . In either case, technical proofs of FLP impossibility demonstrate that (i) consensus protocol never terminates or (ii) different processes may decide different values. Thus making it impossible to have a consensus protocol satisfying the above defined properties, precisely, Termination and Agreement.

Since then, the consensus problem has been studied under different synchrony and failure assumptions. Initial research on database operating systems [55] and distributed database systems [101] presented solutions to the problem of consensus which were extensively studied by Byzantine community. Several BFT-SMR protocols [11, 30, 40, 56, 57, 104] were devised thereafter guaranteeing consensus properties in an asynchronous fault-prone distributed systems.

### 2.2.5 System Model

We assume a distributed system where servers (replicas) are connected in different network topologies, for example PBFT [30], RBFT [18], Aardvark [36] are fully connected whereas Chain [56], Ring [57] connect the replicas in a chain-like pattern (a replica followed by another) or completely disconnected like in OBFT [98].

**System Assumptions.** The system is defined as a set of clients and servers where a client sends a request in a closed loop, i.e., clients have to wait for the response of a request before sending a new request. We assume a finite client population where any number of them may be faulty and, a total of  $N$  replicas implement BFT-SMR deterministic protocol where up to ' $f$ ' number of replicas can behave maliciously. The total number of replicas,  $N$ , depend on the number of replicas a system can tolerate, precisely, ' $f$ '. The BFT-SMR protocols demonstrate that  $2f + 1$  replicas are enough to solve the problem of consensus but  $3f + 1$  replicas are required to tolerate and continue in the presence of  $f$  simultaneous Byzantine faults. We assume that there will be no more than  $f$  simultaneous faults at any moment.

Faulty servers behaving arbitrarily are subject to independent or adversary-coordinated failures. In case of identical replicas, a flaw on one will be replicated in the others as well, thus violating this assumption. To ensure the independence of failures, a technique such as N-Version programming is used to obtain different copies of the protocols or heterogeneous physical machines with different operating systems and hardware are used. We assume any number of clients may be faulty. Nevertheless, in some specific protocols such as

OBFT [98] where clients are assumed to be trusted and non-malicious. Clients are only prone to crashes.

**Network Assumptions.** The links between nodes are asynchronous and unreliable with synchronous intervals during which messages are delivered within a known bounded delay. We do, nevertheless, assume that if a node keeps re-transmitting a message, the message will eventually be received (partial synchrony [51], see Section 2.1.1.3). However, the *Liveness* property can only be ensured during periods of synchrony where a message reaches its destination within some fixed worst case delays [53]. Furthermore, the network itself may fail to deliver messages, delay them, duplicate them, delay them out of order, or even corrupt them.

**Cryptography Assumptions.** BFT protocols rely on cryptographic techniques such as collision-resistant hashing (digests), message authentication codes (MACs), public-key cryptography and digital signatures to ensure authenticity, integrity, and non-repudiation properties (see Section 2.1.3 for details).

**Adversary Assumptions.** We assume a Byzantine failure model, where node/(s) (replicas or clients) may behave Byzantine. We assume a strong adversary capable to manipulate and coordinate the malicious nodes to compromise the replicated service. However, we do assume that this adversary is computationally bounded and unable to break cryptographic techniques, i.e., it cannot produce a valid signature of a non-faulty (correct) node.

These assumptions are common to all the state of the art BFT protocols discussed in this manuscript (see Section 2.4).

## 2.3 Byzantine Fault Tolerance

Improving integrity, performance, availability and reliability of internet-based distributed computing has become challenging. The expanse of new computing technologies has potentially raised numerous threats to deployed services, compromising integrity due to failures such as software bugs, crash failures, omission faults and arbitrary malicious attacks and degrading other aspects of Quality of Services (QoS). Replication is an essential technique to survive these problems and maintain system reliability through replicating services at redundant servers (backup replicas) As discussed in the previous section, State Machine Replication (SMR) is used to make services fault-tolerant.

Traditional fault-tolerant systems such as Paxos [77], survives benign failures (i.e., crash or fail-stop faults) [78, 88]. However, the problem arises in the presence of arbitrary faults, called Byzantine [30, 80]. This problem was first

presented as Byzantine General's Problem by Lamport [80] where the generals of an army must decide if they want to attack the enemy or retreat in the presence of some traitors among them. These traitors can mischievously trick some generals, force them to change their decisions to be inconsistent with other generals or forge the messages to create distrust among loyal generals. Byzantine fault tolerance is possible if and only if all the loyal generals consistently commit to the same decision. And it was demonstrated that if there are  $N$  generals, out of  $f$  are traitors, then the problems created by traitors could only be handled with  $N > 3f$  generals<sup>1</sup>.

On the same lines of idea proposed by Lamport, BFT protocols were proposed to maintain resiliency against Byzantine (arbitrary) faults in distributed settings. From the perspective of distributed computing, Byzantine faults are those faults that force services to demonstrate an unpredictable behavior, different from the normal execution paths; such as sending inconsistent responses, producing wrong responses, corrupting messages and local states, intentionally delaying request processing, flooding the communication channels, etc. A BFT protocol requires at least  $3f + 1$  replicas to ensure *Safety* and *Liveness* among replicas, while tolerating up to  $f$  Byzantine replicas. Many BFT protocols have been designed so far, some ensuring consistency with more number of replicas, i.e.,  $5f + 1$  in Q/U [11] or trying to minimize the number of replicas using trusted components [37, 71] or enhancing the performance of the system in fault-free scenarios [56, 57, 74] or minimizing the performance degradation in the face of Byzantine faults [15, 18, 36, 105]. We will discuss some of the state-of-the-art BFT-SMR protocols in this section.

### 2.3.1 Understanding $3f+1$ Bound

Different system models require a different number of processes to implement consensus, client request ordering while tolerating up to  $f$  failures. Table 2.1 presents the minimum number of servers/replicas required by consensus protocols for tolerating crash [77, 78] and Byzantine faults, respectively [51] under synchronous and partially synchronous system models for ensuring their *Safety* and *Liveness*. The table also represents the minimum number of replicas that must execute the client requests to enable client always have the correct responses and for the system to be always consistent despite the presence of up to  $f$  faults. As can be seen in the table, under partial synchronous model, crash-prone system needs  $2f + 1$  replicas [77], while BFT system re-

<sup>1</sup> $N$  also represents the number of replicas required to tolerate  $f$  arbitrary (Byzantine) faults by a BFT protocol. Also, we will use  $f$  for faulty, arbitrary, Byzantine, malicious and traitors replicas, interchangeably

Table 2.1: Table presents some results from Dwork [51] showing the minimum number of replicas required for handling crash (or fail-stop) and Byzantine faults in synchronous and partially synchronous environments. It also presents the minimum number of replicas required to execute client request once the request is totally ordered.

Type of Fault	Synchronous	Partially Synchrony	Replicated execution
Crash or Fail-Stop	$f + 1$	$2f + 1$	$f + 1$
Byzantine	$3f + 1$	$3f + 1$	$2f + 1$

quires  $3f + 1$  replicas [80] for total ordering and tolerating  $f$  simultaneous faults. However, once the request is totally ordered (by at least  $2f + 1$  correct replicas),  $f$  fewer replicas are required to execute the request in both the fault models.

We now explain the bound of  $2f + 1$  replicas to ensure consistency for Byzantine faults with an example for read and write operations. We consider a replicated service with mutually exclusive read/write operations in an asynchronous environment. The service must be correct, consistent, ensuring *Safety* and *Liveness*, and always available and reliable even in the presence of faults. Assuming the total number of replicas is  $N$ , and up to  $f$  replicas can be Byzantine. In a BFT system, quorums consist of  $N - f$  replicas where  $f$  replicas might be the faulty replicas. Considering the first request to be a read operation which is carried out by a quorum (quorum A) of  $N - f$  replicas, out of which  $f$  replicas might be faulty. Similarly, during the second request, which is a write operation is carried out by another quorum (quorum B) of  $N - f$  replicas. Out of two quorums (which probably contain responses from different replicas) might contain responses from two sets of  $f$  replicas where these  $f$  replicas might be either Byzantine or just slow (e.g., due to network congestion).

So we can guarantee that the number of correct responses is at least  $N - 2f$  which does not contain responses from  $f$  Byzantine and  $f$  slow replicas. This would mean, that each quorum must have  $2f + 1$  replicas (Figure 2.3) where quorums of two consecutive operations will intersect in at least one correct replica, masking the responses from  $f$  malicious replicas, thus ensuring that no faulty replica violates *Safety* property. Finally, to ensure, there are no responses from the Byzantine or slow replicas, the responses from non-faulty replicas must be higher than  $f$  faulty responses, i.e.,  $N - 2f > f$  which is equivalent to  $N > 3f$ . Therefore, at least,  $3f + 1$  replicas are required to maintain *Safety* and *Liveness* where each step of the protocol must be

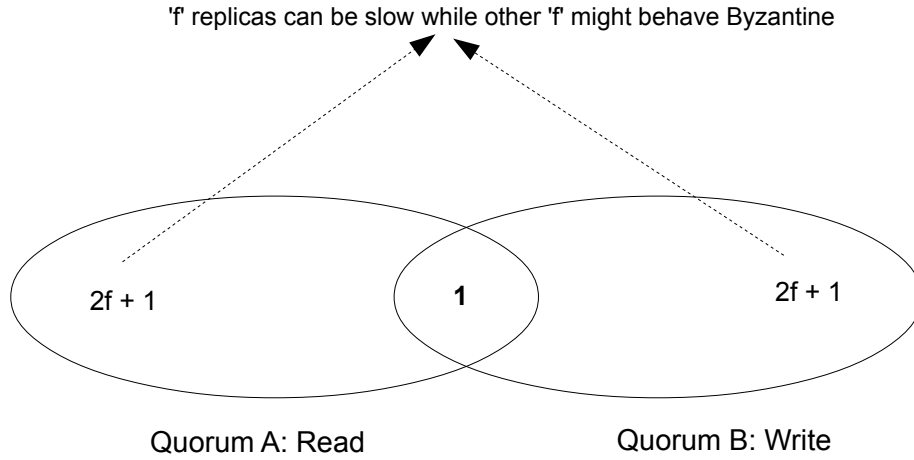


Figure 2.3: Read and write quorums need  $2f + 1$  replicas to intersect in at least one correct replica executing both the operations (read/write).

executed by at least  $2f + 1$  replicas. For any two quorums (A and B) of  $2f + 1$  responses for two consecutive operations, there is at least one correct replica which executed both the operations, reflecting the correctness of previous operations. The quorum of  $2f + 1$  replicas with another  $f$  which may not respond at all, define the threshold of  $3f + 1$ .

### 2.3.2 Types of Byzantine Behaviors

Any fault is a Byzantine fault that brings inconsistencies with the capabilities of degrading the performance of the system many folds. Out of many Byzantine faults, some faults are not observable and therefore, cannot be detected [59]. In this section, we discuss some of the detectable Byzantine faults which are intentionally induced by an adversary forcing an algorithm to deviate from its correct execution. We consider two categories of these Byzantine behaviors, they are as follows.

#### Context-Free Faults

Context-free faults are the faults triggered by a malicious node without any prior knowledge of the information at a node like messages exchanged, or content of the messages, or the state of the node, etc. These faults can occur independently without any understanding of underlying distributed algorithms or applications.

1. *CPU Load*: This fault is triggered by increasing load in terms of the number of concurrent clients sending requests.



2. *Replica Crash*: A Byzantine node crashes and terminates all future interactions with other replicas.
3. *Replica Hibernation*: A faulty replica delays all the communications (message exchange) with other replicas. It may also delay the processing of a request by a certain amount of time.
4. *Drop Packets*: A malicious replica accomplishes this attack by dropping packets for a particular replica(s), at a certain time of the day, for examples dropping  $x$  packet every  $n$  packets or every  $t$  seconds, or a randomly selected portion of the packets.

### Context-Dependent Faults

These types of faults need access to the information regarding the protocol specifications, the content of the messages including header and payload, the size of the messages, the signature of the node for forgery, etc.

1. *Message Delay*: Faulty replica intentionally delays sending of a specific message to increase the overall response time of the system.
2. *Corrupted Messages (header and payload)*: A faulty replica will corrupt the header to trigger the re-transmission of the messages with an intention of buffer underflow/overflow. Payload corruption is done to waste the CPU cycles in performing heavy cryptographic operations over verifying malicious information.
3. *Network Flooding*: A Byzantine replica randomly starts to send correct/incorrect messages to all the correct nodes to flood their incoming network.
4. *Message Authentication Code(MAC) Attack*: Clients send incorrect authenticators to make all the replicas accept the requests without verifying their MACs.
5. *Non-repudiation*: A Byzantine replica denies the authenticity of its signature on sending of a message. This attack is generally possible when using only collision-resistant hashing (digests) or MAC authenticators.
6. *Forge Signatures*: Forging some parts of a signature in an effort to convince correct replicas to commit to a wrong response.

We implemented some of the above mentioned faults in BFT-Bench (see Chapter 4) which were also considered in a robust state-of-the-art BFT protocols [15, 18, 36, 105] (see Section 2.4.3.1).

## 2.4 BFT Protocols at Present

BFT protocols have been the focus of research for well over a decade and have advanced many folds in terms of improved and robust performance. Research on these protocols has evolved from theoretical proposals to practical approaches. And in this section, we review the related work on some of the state-of-the-art BFT protocols under different categories. This section also demonstrates their comparative theoretical analysis.

### 2.4.1 BFT from Theory to Practice

The Byzantine General's Problem was first described theoretically by Lamport in [80]. It defines that a Byzantine agreement requires  $3f + 1$  replicas to tolerate up to  $f$  arbitrary faults under partial synchrony, i.e., a known fixed upper bound on the message delivery time [51, 53, 79] (see Section 2.1.1.3). Lamport presented the first theoretical model of Byzantine problem while PBFT introduced the first practical protocol which analyzes and performs evaluation in terms of performance metrics [80]. Since then PBFT has been considered the baseline for BFT protocols and its pattern is widely reused for its practical efficiency [15, 18, 24, 36, 105].

Numerous BFT protocols have been designed since PBFT, each one adding new features for improving the performance. All of these protocols can be broadly categorized under two groups: (1) protocols optimizing performance under fault-free settings, like Chain [104], Q/U [11], Zyzzyva [74], Ring [57], CheapBFT [71], HQ [40], Quorum [56], MinBFT [106] and OBFT [98] and, (2) protocols minimizing performance degradation under faulty scenarios including Spinning [105], RBFT [18], Aardvark [36] and Prime [15].

The engineering of BFT protocols followed several directions, varying from various communication patterns to adopting different cryptographic techniques to dependency over the third party trusted components for reducing the number of replicas to robust protocols for handling faulty scenarios [15, 18, 36, 37, 98, 106]. In the following, we present these two families of BFT protocols.

### 2.4.2 Group 1: Performance Enhancements in Fault-Free Conditions

According to the analysis of protocols in first group, one of the main concerns of researchers was to enhance the performance of BFT protocols in fault-free cases, while maintaining the *Liveness* and *Safety* properties in the presence of faults [11, 24, 37, 40, 56, 57, 71, 74, 86, 100, 104, 110]. For example,

Zyzyva [74], Zeno [100], ZZ [110], Q/U [11], HQ [40] and Quorum [56] allowed the replicas to be temporarily in inconsistent states and moved the task of detecting these inconsistencies to the client side of the protocol, improving both throughput and latency in fault-free scenarios. CheapBFT [71], BFT-TO [37], BFTMencius [86] and MinBFT [106] use trusted components for ordering the client requests, along with reducing the total number of replicas required to handle Byzantine faults to  $2f + 1$ . Chain [104], Ring [57] and Aliph [56] make improvements with regards to throughput. They also propose protocol switching at fault occurrence. During fault-free cases, they run a more simplified version of the protocol, and on detection of a fault, the protocol switches to a more robust protocol to handle faults. This is motivated by the fact that faults occur rarely in a system, and that it is more important to provide priority to fault-free cases. The following part of this section describes some of the state-of-the-art protocols in details under various categories.

#### 2.4.2.1 Agreement-Based Protocols

The primary replica is pivotal in an agreement-based protocol. It has the major responsibilities than any other replica in the system, and it often becomes the bottleneck and a deciding server for the performance of the system. Primary replica orders the client requests and forwards the same to other replicas. All the replicas with consensus agree to the ordering done by the primary and eventually executes the request in the same order, maintaining the *Safety* property. For most of the protocols, replicas send their responses directly to the client, thus preserving *liveness*, by executing the clients' request. Agreement-based protocols require a minimum of  $3f + 1$  replicas to tolerate up to  $f$  faults. Communication overhead is maximum for such protocols as they follow all-to-all communication among replicas for consensus. This at times, leads to poor performance and fault scalability issues.

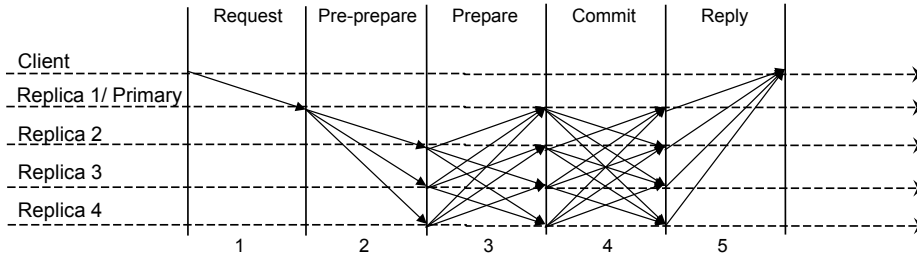


Figure 2.4: Communication pattern of PBFT

**PBFT.** PBFT [30], the first agreement based practical BFT protocol tolerates upto  $f$  simultaneous faults with  $3f + 1$  replicas. It is considered the most robust protocol; therefore, it has been used as the baseline for many protocols [15, 18, 24, 36, 74, 104, 105]. Figure 2.4 presents the widely used communication pattern of PBFT. PBFT has a dedicated replica called primary to order the requests of the clients. PBFT undergoes 3 rounds (phases) of message exchanges; PRE-PREPARE, PREPARE and COMMIT. Primary upon receiving a request from a client, assigns a sequence number and broadcasts a PRE-PREPARE message to all the replicas containing the ordered request. Once replicas receive PRE-PREPARE message, they verify the ordering and acknowledges the message by broadcasting a PREPARE message (containing the PRE-PREPARE message sent by primary) to all other replicas. Replicas upon receiving matching  $2f + 1$  PREPARE messages, commits the request in its local history and broadcasts COMMIT message to all replicas. On receiving COMMIT messages, replicas put them in their logs only if they are correctly signed, ordered and belongs to the same view. If a replica receives  $2f + 1$  COMMIT messages, it executes the request and responds to the client. The client commits the request only if it receives  $f + 1$  matching responses, otherwise it retransmits the request. Upon expiration of the timer at client or any other replica, it initiates a *view change* protocol to elect a new primary. All the replicas then send a VIEW-CHANGE message containing checkpoints and previously sent PRE-PREPARE messages as a proof of misbehavior. The new primary multicasts a NEW-VIEW message containing the new view number to all the replicas. Non-primary replicas verifies the correctness of the NEW-VIEW message; if correct, the protocol resumes to normal operation.

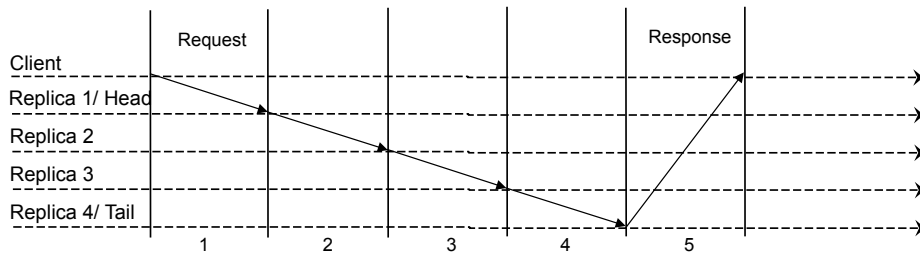


Figure 2.5: Communication pattern of Chain

**Chain.** Chain [56, 104] is an abortable speculative protocol with the communication pattern like a chain. Figure 2.5 presents the communication steps of the protocol where all the replicas are arranged in a chain fashion, starting with a head (primary) replica and ending with a tail replica. A client sends

a request to the head, which assigns a sequence number to the request. Each replica forwards the request to its successor except the tail, which replies to the client. A replica would only accept a message from its predecessor (except the head which accepts requests only from the clients). Chain ensures: (1) that content of a message is not modified by any malicious replica, (2) no replica is bypassed, and (3) reply sent by the tail (to a client) is correct. To provide these guarantees, Chain depends on lightweight MAC authenticators generated and verified by each replica in the Chain. Each replica (except Head) verifies  $f + 1$  MAC authenticators from its predecessor replicas and adds (except tail)  $f + 1$  MAC authenticators (corresponding to the next  $f + 1$  successor replicas) to the forwarding request. The last  $f + 1$  replicas include the digest of their histories (which has the forwarding request and its corresponding response) for the client. The client verifies these digests and if they match, client commits the request with the response obtained by the tail. This step ensures that reply sent by the tail is correct. In case of an incorrect response or no response, the client broadcasts a PANIC message to all the replicas. Similar to protocols, Quorum, Zyzzyva and HQ; replicas stop executing requests and send back a signed message containing their histories. The client waits for  $2f + 1$  matching histories and creates an abort history. Furthermore, upon fault detection, Chain switches to a backup protocol (eg., PBFT) to commit the requests. Chain's efficient implementation of pipeline topology brings two benefits: (1) reduced number of MAC operations at the bottleneck replica, and (2) better network usage (any replica communicates only with two other replicas). Nevertheless, these benefits makes sense only when the pipeline is completely fed. This can be done either by using a large number of messages or a large number of clients sending requests. It has been observed that Chain outperforms PBFT and Zyzzyva in terms of throughput when the number of clients is higher (at least more than 40). The pipeline pattern achieves consistently lower response-time than PBFT (due to the complex message exchange in PBFT). Chain demonstrates lower latency than Zyzzyva only when the pipeline is saturated.

**Ring.** Ring [57] is another speculative abortable BFT protocol like Chain that achieves high throughput (although less than Chain) during contention. Ring tries to overcome the drawbacks of other protocols such as the use of IP multicast, the presence of bottleneck replicas due to asymmetric replica processing, dependency over one primary replica for total ordering and unbalanced network bandwidth utilization. Ring uses ring topology where each replica has a successor and a predecessor (Figure 2.6). Like Chain, Ring protocol must ensure: (1) no replica in the ring is bypassed, (2) Byzantine

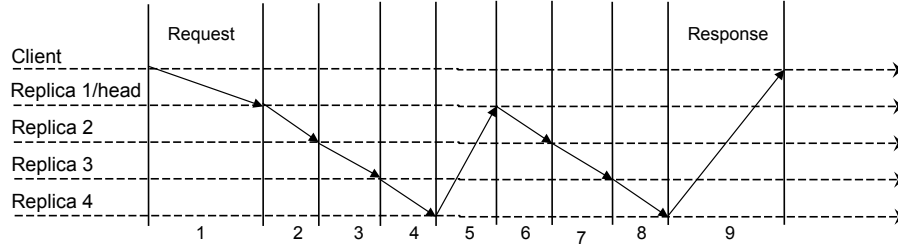


Figure 2.6: Communication pattern of Ring

clients cannot corrupt the total ordering of correct requests, and (3) the reply sent by the last replica is not forged. The client sends request to any of the replicas (in round robin fashion) called as *entry* replica and this submitted request is forwarded in the ring until it reaches the predecessor (called as *exit* replica) of the entry replica. A sequencer replica (can be any replica) assigns a sequence number to the request. The protocol undergoes another round of message passing in Ring which forwards the acknowledgment (ACK) message (for receiving the sequence number) and the *exit* replica replies to the client. Requests are executed in the second round only after they have received the ACK message. Ring uses the similar MAC authentication (Ring Authenticators) technique as of Chain but has additional ACK messages. In case of a fault, client cannot commit the request and sends a PANIC message to all the replicas and protocol switches to resilient mode. In the resilient mode, replicas and clients use signed messages which detect a faulty client, a faulty replica and also a faulty sequencer. In this mode, Ring switches to a new instance with a different configuration, and a new sequencer is elected. Due to the two rounds of Ring in fault free scenarios, Ring achieves the worst performance as compared to most of the state-of-the-art protocols in Group 1. However, under contention or large size messages (when the network or the primary replica becomes a bottleneck), Ring gradually outperforms some protocols in terms of performance. Ring demonstrates the best network and CPU utilization and it is the only protocol to consider the evaluation of these metrics.

**BFT-SMaRt.** BFT-SMaRt [24] is another BFT state machine replication protocol which is similar to protocols such as PBFT [30] and UpRight [34] but, improves reliability, modularity of components, multicore-awareness while providing reconfiguration support and flexible programming interface. BFT-SMaRt aims to be robust in terms of high performance in fault free executions and correctness during fault occurrences. Design principles of the protocol include: (i) *Tunable fault model* where it provides tolerance to Byzantine

faults using robust protocol and simplified version of SMR protocol (similar to Paxos [77]) to tolerate crash message corruptions, (ii) *Simplicity* to avoid all the performance optimizations used by other protocols such as pipelining [56, 104], resource efficiency with trusted components [37, 71, 106], speculation [56, 74, 100], and IP multicast [31, 74], (iii) *Modularity* is achieved by separating agreement protocol from client requests ordering and consensus, state transfer and reconfiguration, and (iv) *Multicore-awareness* to scale throughput by running separate hardware threads for signature verification and computational loads.

BFT-SMaRt protocol comprises of three core protocols considering above mentioned design principles.

*Total Order Multicast.* It is similar to the normal case execution of PBFT [30]. Protocol undergoes three communication steps (for consensus) with messages PROPOSE, WRITE, and ACCEPT. When a fault occurs, the protocol switches to synchronization phase where a new primary is elected using the state transfer protocol and reconfiguration techniques.

*State Transfer Protocol.* Protocol periodically creates logs after every certain number of request batches or takes snapshots at different points of executions. During fault occurrence, every replica sends these state information for creating a consistent state history. When all the correct replicas are consistent, they switch to a new view. *Reconfiguration.* During this, faulty replicas are replaced with new replicas from the same cluster, initiated with the same state as of correct replicas in the new view.

BFT-SMaRt displays higher throughput than PBFT, UpRight and JPaxos (Java implementation of Paxos) for a higher number of clients. Also, BFT-SMaRt undergoes reconfiguration during a crash.

#### 2.4.2.2 Quorum-Based Protocols

Another category of protocols is quorum-based. Replicas do not communicate with each other to reach an agreement, rather they execute the request as send by the client directly. After execution, they send their responses to the client without any consensus. For such protocols, clients send their requests to a set of replicas forming a quorum, which execute the requests independently and assign their own ordering. We have explained this briefly under  $3f + 1$  bound in the previous section. Client commits the request only if it receives the required number of matching responses (depends on the protocol specifications) and the ordering is consistent for each replica. The performance of quorum based protocols is higher than agreement based (when there is no contention), but in case of a fault or an inconsistency, they are a lot more expensive. Quorum-based protocols cannot handle contention due to lack of



request ordering mechanism.

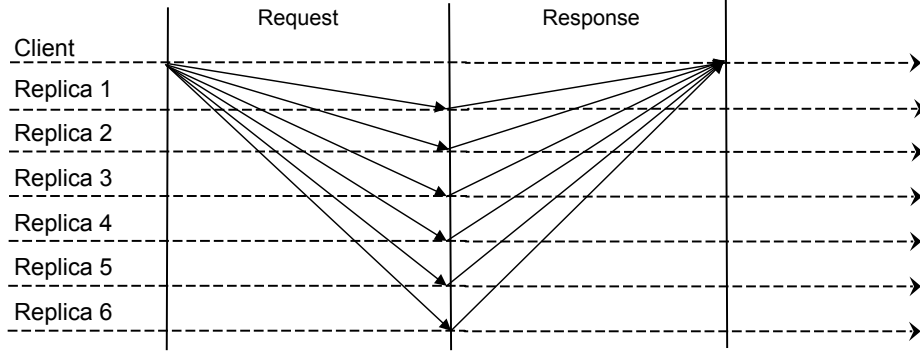


Figure 2.7: Communication pattern of Q/U

**Query/ Update (Q/U).** Query/Update (Q/U) [11] protocol is a quorum based fault-scalable Byzantine fault protocol ensuring better performance than PBFT under no contention. It needs  $5f + 1$  replicas unlike commonly required  $3f + 1$  to tolerate  $f$  Byzantine faults. Figure 2.7 presents the communication pattern of the protocol. It requires only one round-trip of message exchange between a client and replicas to commit a request. Namely, a client sends a request to a preferred quorum of replicas and these replicas speculatively executes it and respond directly to the client. Q/U employs no primary for ordering the incoming client requests. Consistency is maintained through periodic exchange of up-to-date history at each replica with other replicas. Clients ensure that they always obtain the final and correct version of the history. For performance efficiency, clients can contact a quorum of  $4f + 1$  replicas, but it results in out of order execution of requests and outdated histories at the remaining  $f$  replicas. These  $f$  replicas induce the cost of synchronization phase where they demand up-to-date histories from at least  $f + 1$  replicas. This phase ensures that history is not manipulated by more  $f$  faulty replicas. Q/U cannot handle contention as it can cause concurrent updates leading to different versions on different replicas. This is usually detected by the client that initiates an expensive recovery phase where all the replicas are brought to the same version. The performance of Q/U gradually decreases due to a high number of concurrent updates; therefore, Q/U is also not scalable where there are a large number of clients. But under no contention, Q/U is scalable in terms of a number of Byzantine faults tolerated by the system. Performance of Q/U decreases by only 36% when  $f = 5$  (in comparison to  $f = 1$ ) whereas the performance of other BFT protocols decreases by 83%.



**HQ.** Hybrid Quorum [40] replication protocol is another quorum-based protocol which improves over the shortcomings of PBFT [30] (i.e., the quadratic cost of inter-replica communication is unnecessary when there is no contention and poor fault scalability) and Q/U [11] (i.e., Q/U requires a large number of replicas which not only augment the hardware cost but also increase the number of possible points of failure and it suffers concurrency issues under contention). HQ requires  $3f + 1$  replicas and uses two approaches; (i) in the absence of contention, it uses a lightweight quorum protocol which requires one round-trip (between client and replicas) for read operations and two round-trips for write operations, and (ii) when contention occurs, it uses agreement-based techniques (protocol similar to PBFT) to agree upon the ordering of contending requests. Therefore, when a client receives conflicting responses (or dissimilar history digest), client requests conflict resolution by sending RESOLVE messages with *conflict certificate* (formed from responses and history digests). To resolve the conflicts, HQ use BFT-SMR protocol (PBFT), to reach agreement on a deterministic ordering of the conflicting messages. Results of HQ exhibit that it performs better than Q/U and PBFT in contention-free scenarios and scales acceptably for  $f$  up to 5. Under contention, HQ achieves better performance than Q/U and similar to PBFT. Nevertheless, with a rise in contention, contention resolution leads to gradual degradation in performance.

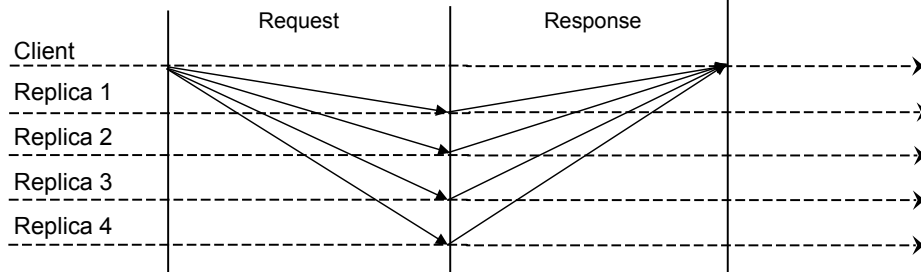


Figure 2.8: Communication pattern of Quorum

**Quorum.** Quorum [56] is designed to be abortable with the same communication pattern as Q/U in fault-free cases, however, it requires  $3f + 1$  replicas instead of  $5f + 1$  replicas (unlike Q/U). Quorum is targeted for system conditions that do not involve asynchrony, failures or contention. Figure 2.8 presents the communication pattern where a client sends the request to all replicas. These replicas independently execute the request (again without any message exchanges among them), update their local history and reply to the client (with the digest of up-to-date history). If the client receives matching

responses from all the  $3f + 1$  replicas and all the histories match, client commits the request; otherwise, it invokes a panicking mechanism. Client sends a PANIC message to all the replicas. Upon reception of PANIC message, replicas stop executing the requests in their queues and send a signed message containing their history to the client. Client upon receiving  $2f + 1$  signed messages, generates an abort history and aborts Quorum; and, switches to a backup protocol which uses the history for maintaining consistency across replicas (in this case, it is PBFT) [56]. Similar to Q/U, Quorum does not tolerate contention. Quorum achieves the best performance (high throughput and minimum latency) in comparison to other BFT protocols, but only with a fewer number of clients. Nevertheless, with  $f > 1$ , the Quorum outperforms Q/U as Q/U requires additional  $2f$  replicas forcing the client to perform additional MAC computations. Furthermore, Quorum also reduces the overhead of hardware cost of extra  $2f$  replicas.

There are other protocols like Scrooge [96] which further aim to reduce the number of replicas required for a quorum while tolerating  $f$  Byzantine faults.

### 2.4.2.3 Speculation-Based Protocols

In these protocols, replicas respond to the client's request without going through the expensive 3 phase commit protocol as in agreement based. Replicas optimistically agree with the ordering proposed by the primary replica and responds to the request. Replicas do not care about the inconsistencies, it matters only to the clients. If there is a problem, like primary sending different operations or assigning different sequence numbers to requests, the correct client will eventually detect the fault and informs the replicas to rollback to a consistent state. The client or replicas can initiate the view change upon any malicious behavior. Speculation-based protocols improve upon performance in best case scenarios, i.e., partial synchrony and fault free.

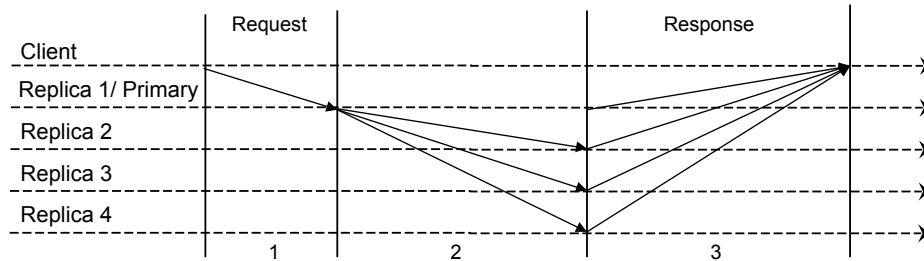


Figure 2.9: Communication pattern of Zyzzyva

**Zyzyva.** Zyzyva [74] is a speculative, high throughput BFT protocol that improves the performance of PBFT. Similar to many state-of-the-art protocols, Zyzyva also requires  $3f + 1$  replicas to ensure *Safety* and *Liveness* in presence of up to  $f$  Byzantine faults. Figure 2.9 illustrates the message exchange pattern of Zyzyva. During a fault-free scenario, the client sends the request to the primary which is in charge of assigning a sequence number to all the incoming client requests. Primary then forwards the ordered request to other replicas. All the replicas speculatively execute the request and send the response to the client along with the digest of their local histories. Speculative approach bypasses the expensive agreement phase (like in PBFT) for achieving total ordering of the messages before executing (committing) the request. If the client receives  $3f + 1$  mutually-consistent matching responses with complete history, client commits. If the client receives between  $3f$  and  $2f + 1$  mutually-consistent matching responses, it creates a COMMIT certificate which contains the SPEC-RESPONSE messages from responding replicas and broadcasts it to all the replicas. Replica upon receiving a COMMIT message, makes sure that the history in the message is consistent with their local history. If consistent, replica responds with a LOCAL-COMMIT message. If there exists two different ordering for the same request, replica initiates a *view change* protocol with a proof of misbehavior (POM) message. In case, a client receives less than  $2f + 1$  matching responses, it resends the request to all the replicas. Upon receiving the request directly from the client, all the non-primary replicas send the CONFIRM-REQ message containing the request to the primary for ordering. If primary confirms with CONFIRM-REQ (assigning the sequencing number as well), replicas execute the request and respond to the client. If the replicas fail to receive the confirmation, they create the POM against the faulty primary and initiate the *view change* protocol to elect a new primary. Under the fault-free scenario, Zyzyva exhibits better performance than PBFT and Q/U but it drops sharply upon fault occurrence due to expensive *view change* protocol.

Atul et al. proposed Zeno [100], a protocol based on Zyzyva aiming to replace strong consistency (linearizability) with a weaker guarantee (eventual consistency), i.e., clients can temporarily miss each other's updates, but when the network becomes unstable, states from all replicas are merged for consistency (having the replicas to agree on a total order for all requests). There is another protocol, called ZZ [110], proposed by Timothy et al., that reduces the replication cost from  $2f + 1$  to practically  $f + 1$  during the normal case and activate additional  $f$  replicas only upon failures.

## 2.4.2.4 Client-Based Protocols

Client-based protocols aim to prevent faulty replicas to attack, mislead or delay correct replicas, by preventing inter-replica communication. For these protocols, replicas rely upon or communicate only with the client and their identity remains secret to other replicas in the system. Protocols completely depend on the correctness of their clients. They assume their clients to be non-faulty, honest, but crash-prone. Such protocols show comparable performance to agreement and quorum based protocols.

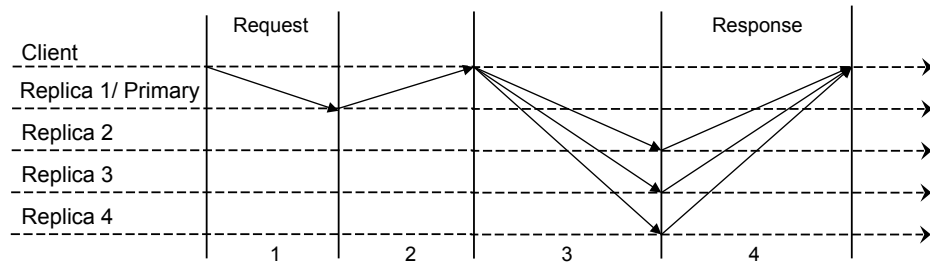


Figure 2.10: Communication pattern of OBFT

**OBFT.** Obfuscated BFT (OBFT) [98] is a client based abortable BFT protocol ensuring complete independence of failure among replicas unlike other BFT protocols which include inter-replica communications. Replicas are absolutely unaware of each other's existence and communicate only with the clients. This prevents a strong adversary from colluding with other replicas and impacting the system's performance. Therefore, the role of clients become very crucial in OBFT. OBFT considers a strong assumption that clients cannot be malicious (but are prone to crashes); since they can violate consistency. A malicious client can send two different requests to two distinct subsets of the replicas and behaves against each subset as if there was a single request. This assumption is very strong and thus limits the usability of the protocol for certain applications where clients are trusted members of the organization. OBFT requires  $3f + 1$  replicas to maintain *Safety* and *Liveness*. However, protocol uses only  $2f + 1$  replicas at a time constituting to an Active replicas set and the extra  $f$  Passive replicas are used upon fault occurrence. Figure 2.10 presents the communication pattern of OBFT. OBFT launches the speculative phase on  $2f + 1$  Active replicas where a client sends a request to the primary replica. Primary replica assigns a sequence number to the request and executes it. Primary then sends the request with a sequence number and its corresponding response to the client. The client then forwards this ordered request to the remaining Active replicas which execute the request and send

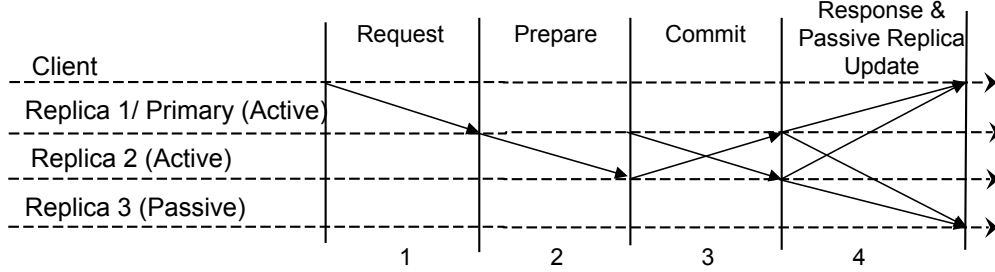
their responses to the client. If the client receives matching  $2f + 1$  messages, it commits the request otherwise launches the recovery phase. During recovery, a client sends a PANIC message to all the Active replicas. Upon receiving PANIC message, replicas stop executing new requests and send a signed *Abort* history to the client. The client waits for matching  $f + 1$  abort histories and send an INIT message to all the  $3f + 1$  replicas. The replicas then respond with ACK messages and the first  $2f + 1$  replicas (that respond the first) form the new Active set. This phase replaces the suspicious (either faulty or slow) replicas with correct replicas from the Passive set (forming a new Active set) and resumes to speculative phase in the new view.

#### 2.4.2.5 Trusted Component-Based Protocols

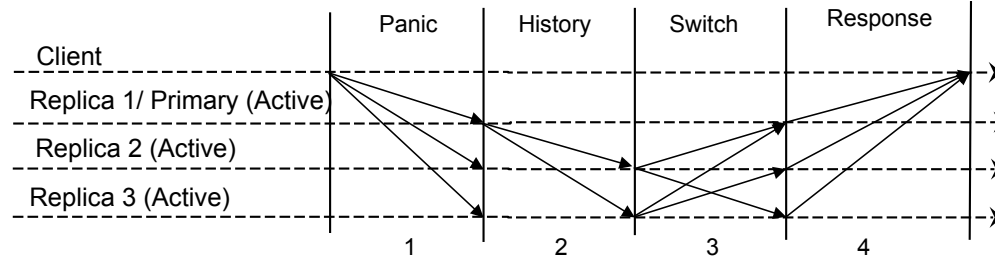
Considering the fact of impossibility to achieve a consensus in an asynchronous system, trusted component-based protocols depend on external entities for partial synchrony unlike other protocols which use timeouts and delays. These protocols exploit components like Abortable Timely Announced Broadcast (ATAB) [86] to enable correct servers compute operations during synchronous period and blacklist the faulty servers responding out of bound, Trusted Timely Computing Base (TTCB) [106] to provide total ordering services or Trusted Ordering (TO) Wormholes [37] which are part of every server and have their dedicated communication channels. All these external entities are considered to be fault-free, but can crash and have periods of unavailability. Such protocols display comparable performance with protocols in Group 1, for fault free configurations. And they also minimize the number of replicas required from  $3f + 1$  to  $2f + 1$  for tolerating the same number of  $f$  faults.

**CheapBFT.** CheapBFT [71], is a resource-efficient BFT system relying on a FPGA-based trusted subsystem called Counter Assignment Service in Hardware (CASH) which tunes a protocol to the minimal resource usage during normal cases (i.e., fault free scenarios). CASH is used for message authentication and verification, and to prevent **equivocation**; that is, the ability of a server to generate conflicting messages for other servers in a system. The trusted CASH subsystem may fail only by crashing and its key remains secret even at Byzantine replicas. This implies that an attacker cannot gain physical access to a replica. The protocol runs a composite agreement-based protocol and saves resources by exploiting passive replication with lesser number of physical machines. To achieve this, each replica is composed of a trusted CASH subsystem initialized with a secret key (this key is shared among all the subsystems of replicas) and is uniquely identified by an id (corresponding to the replica that hosts the subsystem).

The protocol consists of three sub-protocols: (1) normal case protocol, CheapTiny, (2) transition protocol, CheapSwitch, and (3) fall-back protocol, MinBFT [106].



(a) Communication pattern of CheapTiny



(b) Communication pattern of CheapSwitch

Figure 2.11: Communication pattern of two sub-protocols of CheapBFT

Figure 2.11(a) and 2.11(b) present the communication patterns of CheapTiny and CheapSwitch, respectively.

*CheapTiny.* During normal case (fault free cases) executions, CheapTiny requires only  $f + 1$  replicas to agree on client requests and execute them. However, CheapTiny is incapable of tolerating any type of fault.

*CheapSwitch.* CheapTiny switches to CheapSwitch upon suspecting or detecting any faulty behavior of replica(s). It activates the  $f$  additional (passive) replicas to participate in consensus and execution. This protocol brings all correct replicas into a consistent state.

*MinBFT.* Upon achieving consistency, CheapBFT temporarily executes MinBFT protocol where all  $2f + 1$  are active. This protocol creates a new view with  $f$  non-faulty replicas making a new active set before eventually switching back to CheapTiny. MinBFT ensures progress, consistency and tolerates up to  $f$  faults.

The results of CheapBFT demonstrate that it outperforms BFTSMaRt [24] and MinBFT [106]. This is due to the fact of using minimal number of resources with minimal number of message exchanges in comparison to MinBFT

and BFTSMaRt which depends on PBFT communication pattern and undergoes 3 rounds of message exchange.

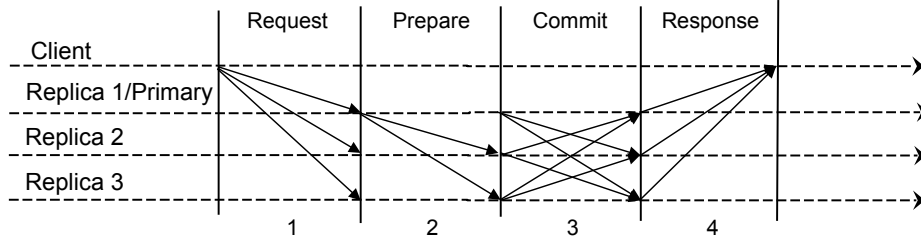


Figure 2.12: Communication pattern of MinBFT during normal case executions

**MinBFT and MinZyzyva.** Giuliana et al. proposed MinBFT and MinZyzyva [106] which require only  $2f + 1$  replicas, instead of usual  $3f + 1$ . This reduction is achieved via a tamper proof distributed component called Trusted Timely Computing Base (TTCB) which provides ordering services and verification simplicity. Another trusted service USIG is a local service that exists in every replica which is used to prevent equivocation (explained previously in CheapBFT). USIG assigns a unique identifier along with a certificate to every request, implying that the ordering was verified by the trusted service. During the normal case executions, both the MinBFT (non-speculative) and MinZyzyva (speculative) run in the minimum number of communication steps, 4 and 3 steps, respectively.

*MinBFT.* Figure 2.12 demonstrates normal case operation. A client sends a request to all servers, but only primary creates a PREPARE message containing a sequence number and a unique identifier from USIG. Primary then forwards PREPARE message to all the replicas. Upon receipt of PREPARE message, replicas verify the message (using USIG) and multicast a COMMIT message to other replicas. Upon receiving  $2f + 1$  matching COMMIT messages, servers accept the request, execute it and return a response to the client. Client upon receiving matching  $f + 1$  responses, commits the requests. In case, a client does not have  $f + 1$  matching responses or  $f + 1$  backup replicas suspect primary to be faulty, a *view change* protocol is executed which elects a new primary and the protocol continues normally. *View change* protocol works in the same way as of PBFT [30] but with only  $2f + 1$  replicas and help of trusted components.

*MinZyzyva.* During gracious execution, MinZyzyva works exactly like Zyzyva but with  $2f + 1$  replicas. Moreover, the ordering of requests by primary and verification of different messages at replicas is done using USIG. Figure 2.13



illustrates the non-gracious executions of MinZyzyva which happens when the network is too slow or one of more servers are faulty, therefore, the client will never receive  $2f + 1$  matching responses. The client during this execution collects  $f + 1$  responses as a proof of misbehavior, to initiate a *view change* protocol to elect a new primary.

MinBFT and MinZyzyva benefit in terms of cost with fewer replicas to tolerate  $f$  faults, their resilience and management complexities. Even with the overhead induced by trusted components, these protocols perform better than PBFT and Zyzyva.

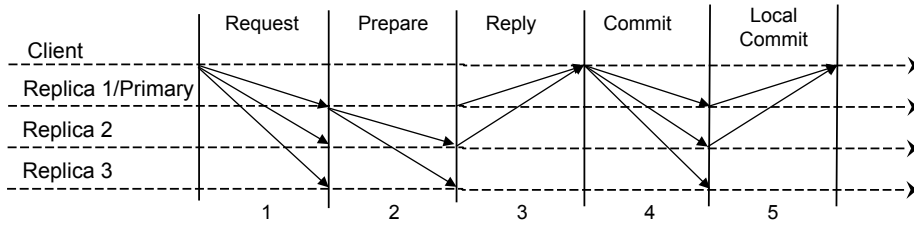


Figure 2.13: Communication pattern of MinZyzyva during non-gracious execution

Correia et al. proposed BFT-TO [37], an algorithm for asynchronous BFT-SMR with only  $2f + 1$  replicas by leveraging a trusted component called *Trusted Ordering* (TO) wormhole. Only the algorithm was proposed and was never implemented. There is another protocol BFT-Mencius [86] based on a new communication primitive, Abortable Timely Announced Broadcast (ATAB), and does not use signatures. The protocol was implemented using Scala and its results are comparable to PBFT. The implementation in Scala adds an overhead of implementing other protocol in Scala too for performing comparative analysis.

Several papers considered theoretical evaluation as an alternative, where throughput is related to the number of cryptographic operations on the bottleneck replica, and latency is related to the number of communication steps in the critical path of the protocol [56, 74]. Table 2.2 provides the theoretical analysis of the above discussed protocols in terms of throughput, latency and minimum number of replicas required to tolerate up to  $f$  simultaneous Byzantine faults. We observe that the Group 1 protocols over the years have enhanced in reducing the cost of replication, optimized in terms of number of cryptographic operations performed (most protocols use computation of MACs over public-key encryption) and ideally provide comparable performance to the performance of a non-replicated system, but all in non-faulty settings. Nevertheless, performance of these protocols degrades in the pres-



Table 2.2: Theoretical Analysis of BFT protocols aiming to enhance the performance in fault free scenarios

Type of Protocol	BFT Protocol	Throughput	Latency	Replication Cost	Cost of Bad Runs
Agreement-Based	PBFT [30]	$8f + 3$	5	$3f + 1$	1 view change
	BFT-SMaRt [24]	$\approx$ PBFT	5	$3f + 1$	1 view change
	Chain [56]	$2f + 2$	$3f + 2$	$3f + 1$	Switch to PBFT + PBFT execution
	Ring [57]	$4f + 4$	$7f + 2$	$3f + 1$	Switch to resilient mode of Ring
Quorum-Based	Q/U [11]	$4f + 2$	2	$5f + 1$	1 view change
	HQ [40]	$4f + 2$	4	$3f + 1$	Switch to PBFT + PBFT execution
	Quorum [56]	2	2	$3f + 1$	Switch to PBFT + PBFT execution
Speculation-Based	Zyzyva [74]	$3f + 2$	3	$3f + 1$	Checkpointing + 1view change
	Zeno [100]	$\approx$ Zyzyva	3	$3f + 1$	1view change
Client-Based	OBFT [98]	2	4	$3f + 1$	1 view change
Trusted Component	CheapBFT [71]	$4f + 2$	4	$2f + 1$	Switch to MinBFT
	MinBFT [106]	$f + 5$	4	$2f + 1$	1 view change
	MinZyzyva [106]	2 + signatures	3	$2f + 1$	Switch to Zyzyva
	BFT-TO [37]	$8f + 3 +$ wormhole ops	5	$2f + 1$	—
	BFTMencius [86]	$\approx$ PBFT	5	$2f + 1$	—

ence of arbitrary failures. This is mainly due to the inherent design defects or complex implementations of the BFT algorithm.

### 2.4.3 Group 2: Minimizing Performance Degradation in Faulty Conditions

One could argue that BFT protocols are meant to efficiently tolerate faulty behaviors no matter how rarely they occur. The protocols of Group 1 which were once considered to be Byzantine fault tolerant, crashed during faulty scenarios. This is due to the fact that their prototypes were never evaluated under faulty conditions. From the second group, Aardvark introduced the notion of robust BFT protocols, i.e., maintaining a constant performance in the presence of few Byzantine faults [36]. Robust BFT protocols such as Spinning [105], Prime [15], Aardvark [36], and RBFT [18] are meant to efficiently handle some worst-case malicious Byzantine behaviors.

Table 2.3 gives the list of attacks performed by some state-of-the-art protocols to evaluate performance in faulty conditions. Each protocol considers

some Byzantine fault scenarios when evaluating their prototypes and measures the performance in terms of latency and throughput for each case.

Table 2.3: Performance evaluation of robust state-of-the-art protocols under different types of attacks

BFT Protocol	Byzantine Fault Scenarios	Throughput Evaluation	Latency Evaluation
Aardvark [36]	Client floods the replicas	Yes	No
	Replica floods the correct replicas	Yes	No
	Client sending inconsistent MAC authenticator	Yes	No
	Intentional delay of responses on a specific client	Yes	No
	Intentional pre-prepare delay by primary replica	Yes	No
Prime [15]	Intentional pre-prepare delay by primary replica	Yes	Yes
Spinning [105]	Intentional pre-prepare delay by primary replica	Yes	Yes
RBFT [18]	Client floods the correct replicas with invalid requests or inconsistent MAC authenticators	Yes	No
	Intentional delay by primary on specific client	No	Yes
	Replica floods the correct replicas	Yes	No

Aardvark [36], Spinning [105], Prime [15] and RBFT [18], all employ different approaches for reducing the harm caused by a malicious primary. Either they allow replicas to expect a minimal acceptable throughput from the primary [36], or they change the primary with every batch of requests [105]. Furthermore, these protocols implement various fault adaptive mechanisms to handle some fault behaviors. Aardvark allows the replicas to expect a minimal acceptable throughput  $T$  from the primary (level of  $T$  is raised periodically) which enable them to monitor the performance of primary. The inability of the primary replica to maintain the  $T$  leads to frequent view changes. With this mechanism system throughput remains high in the presence of faulty primary (mainly slow primary which is unable to maintain  $T$  at least at  $f + 1$

replicas). Spinning [105] proposed another approach for reducing the damages introduced by a malicious primary by changing the primary every batch of requests. But there is no mechanism to detect a faulty primary which delays a request ordering just by a little less than the timeout at the client. Like Aardvark, Prime [15] also moved the task of monitoring the performance to replicas. In Prime [15], the network performance is monitored by the replicas, while the primary periodically sends messages (with or without request ordering) at a constant rate. This mechanism allows the replicas to expect a constant frequency of messages and enable them to identify a slower primary. But Prime fails to improve the performance over Aardvark and Spinning when a malicious primary colludes with a faulty client. Lastly RBFT [18], introduced the concept of maintaining a constant performance during a fault occurrence. Authors of RBFT demonstrated that the performance of Aardvark, Spinning, and Prime is reduced by at least 78% when there is a fault while RBFT shows a degradation of only 3%. This is due to the fact that RBFT runs  $f + 1$  multiple instances of its protocol, where only one of the instances executes the requests. The other  $f$  instances are meant to monitor the difference in throughput at different instances. RBFT also implements few fault adaptive mechanisms like Aardvark, to handle some fault scenarios.

On the other hand, the evaluation of BFT protocols has also raised several questions. In [36], authors present results showing that the throughput of PBFT [30], Zyzzyva [74], HQ [40] and Q/U [11] falls to 0 when encountering a malicious client while these results are theoretically impossible. Thus, BFT prototypes often perform poorly and in an unexpected way in the presence of faults, violating *Liveness* property.

#### 2.4.3.1 Robust Protocols

Recently, a lot of attention has been given to make BFT protocols robust, i.e., protocols minimizing performance degradation during fault occurrences (attacks from clients and Byzantine servers). The fact that Group 1 protocols are fragile and offer less performance guarantees in case of attacks, motivated the researchers to design robust protocols [15, 18, 36, 105]. According to robust protocols, the main reason for performance degradation is due to the bottleneck replica, mostly primary (also known as a principal server). Robust protocols design mitigate the responsibilities of the primary. This is achieved by frequently changing the primary [36, 105] or by having multiple instances with different primary for each instance [18]. Robust protocols show promising results with minimum performance degradation [18] but with a lower performance during fault free scenarios in comparison to the Group 1 protocols.

**Aardvark.** Clement et al. present Aardvark [36], a protocol similar to PBFT which is effective and robust, unlike protocols Q/U, Chain, Ring, Zyzzyva, PBFT and HQ. Aardvark follows the same communication as of PBFT (see Figure 2.4); it requires  $3f + 1$  replicas where one of the nodes is primary, responsible for ordering client requests. Aardvark employs various fault adaptive mechanisms to provide strong *Safety* and *Liveness* guarantees to ensure system's availability not only during *gracious* intervals (synchronous network, timely, fault free replicas and correct clients) but also during *uncivil* execution intervals (network links and correct servers are timely but with up to  $f$  Byzantine servers and any number of faulty clients). Aardvark also bypasses *fragile optimizations* which only improve best-case performance and introduce expensive alternative protocol paths, making the system vulnerable to faulty nodes.

Mechanisms implemented by Aardvark are as follows.

1. *Malicious Clients.* Aardvark does not rely on clients for anything except sending their requests. It implements a hybrid signature/MAC authentication construct to safeguard against manipulative faulty clients. This is due to the fact that MAC-based authentication makes the system vulnerable to unfaithful (inauthentic) messages; worse, it fails to correctly identify the malicious client/server. Aardvark diligently exercises signatures by using it only for authenticating client requests. Rest of the communication continues to use MACs. It uses Rabin-Williams signature scheme [84, 108] to disproportionately place the computational load of signature generator at the clients rather than on the servers. A client submitting a request, signs the request and then authenticates the signed request with a MAC. Servers upon receipt of a request, authenticates the MAC. If it is not valid, the request is discarded otherwise server verifies the signature. If the signature is invalid, all the future requests from this client are discarded to prevent the system from the spurious load. This mechanism amortizes the cost of verifying signatures and brings down the overhead in line with Group 1 protocols [30, 57, 74, 104]. Furthermore, to mitigate the additional cost of using signatures, Aardvark exploits multi-core commodity machines by processing client requests in one core and messages received by servers in other core. Isolating the computations performed for a request at replicas from messages send/received by the replicas, facilitates the protocol to advance.
2. *Vulnerable Primary.* Unlike past BFT protocols which conservatively trigger a *view change* to replace a malicious primary, Aardvark is a lot aggressive on view changes. It relies on an adaptive throughput mechanism to prevent a primary from achieving a tenure and encourages it to

work hard to maintain its position. That means, a primary providing adequate throughput remains a primary for at least 5 seconds (grace period); also, a primary has only 5 seconds to make inappropriate progress. Once a *view change* is triggered, minimum accepted throughput is defined for the primary. At every checkpoint, performance of the protocol is measured. If the observed throughput is less than the required throughput, primary is considered faulty and a *view change* is triggered. The grace period is long enough to allow a primary to provide required throughput. The acceptable throughput value is set to 90% of the maximum throughput observed in previous  $n$  views. Periodically, this value is increased by a factor of 0.01, until a primary fails to provide the expected throughput. These throughput adaptive frequent view changes do induce a performance loss of at most 10%. Furthermore, Aardvark uses PBFT as its baseline, but its performance is lower than PBFT due to these fault adaptive mechanisms.

3. *Byzantine Servers*. Aardvark exploits features of multiple network interfaces (Network Interface Controller (NIC)) to separate network traffic of clients from servers. This limits the load imposed by generic network flooding by relying on distinct network devices to communicate with each replica. Each replica has  $3f + 1$  network interfaces; one for replica-to-client communication and others for inter-replica communications. This prevents network traffic from clients to slow down the replica traffic and vice-versa. Aardvark employs flood adaptive mechanism by allowing a replica to detect excessive flow of messages from a replica than necessary. This will ensure, that a spurious replica is stressing the network with messages. Upon detection of network flooding from a server (imposing excessive load), dedicated physical links are deactivated for a while to prevent system's performance degradation.

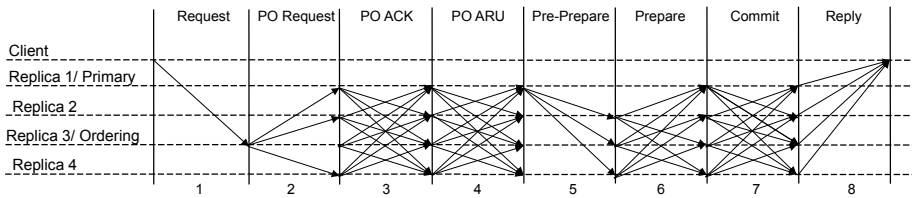


Figure 2.14: Communication pattern of Prime

**Prime.** Amir et al. observed the imperfection of existing Byzantine-resilient replication protocols in the presence of Byzantine faults and present Prime [15]

overcoming these drawbacks. Prime requires  $3f + 1$  replicas where clients send their requests to any of the replicas in the system and that replica then assigns an order to the request. Figure 2.14 illustrates the communication steps of Prime, which are similar to PBFT. Prime was motivated from the possible attack on PBFT where an attacker can delay sending of pre-prepare messages (by the duration just less than the timer at other replicas) without letting the protocol undergo multiple view changes. The attacker benefits on the delay introduced by it without ever getting detected as malicious and successfully brings down the performance to almost zero. In the protocol, once a replica receives a client request, it assigns attributes such as a query,  $r$  and request ordering  $s$  and sends a PO-REQUEST message (containing  $r$  and  $s$ ) to other replicas. Upon receiving PO-REQUEST, replicas send PO-ACK message to all containing  $s$  and  $n$  ( $n$  is a list of replicas sending PO-REQUEST). A replica that receives  $2f$  PO-ACK sends PO-ARU message contains the proof with the replicas which have agreed on the pre-scheduling of the request with same  $r$ ,  $s$  and  $n$  attributes. The PO-ARU message is a proof of accepted sequence numbers by  $n$  replicas and it keeps replicas aware of the requests being executed by other replicas. When the primary receives  $2f$  matching PO-ARU messages, it knows that requests have been pre-ordered and can now be ordered. It then sends pre-prepare messages to all the replicas and the protocol continues as PBFT. Prime allows replicas to monitor network performance by analyzing the maximum time taken by the primary replica to order requests. Thus, it becomes essential of the primary to periodically send scheduling messages (even empty) to allow replicas identify a malicious replica. Even the replicas expect to receive these messages at a certain frequency which is calculated using three parameters: (1) latency between replicas, (2) frequency of primary replica for sending ordered requests, and (3) constant considering variable latency of the network. Through this mechanism, it is easy to detect a faulty primary if it fails to respond within the required frequency and trigger a *view change*.

Prime benefits over performance maintenance in the presence of faults, but the performance reduces many folds during fault free scenarios. This is because, Prime does not use MACs, but signatures. The addition of a new ordering phase and use of signatures for ensuring integrity, authenticity, and non-repudiation, helps in fault detection and performance monitoring. However, induces non-negligible cost and leads to lower performance than Group 1 protocols.

**Spinning.** Spinning [105], another BFT protocol based on PBFT but with an effort to mitigate performance attacks by changing the primary after every

batch of pending client requests. This means, that instead of changing the primary upon detecting it to be faulty, Spinning changes the primary very single batch of requests without any inter-replica exchange of messages. During normal operation, it follows the communication steps of PBFT (see Figure 2.4). It has no view change operation, but rather *merge* operation which collects the information from different servers to decide if the requests in the previous views should be executed in the new view or not. In Spinning, client requests are sent to all replicas. Upon receipt of a request from a non-primary replica, it starts a timer to receive the request ordering message (PRE-PREPARE message) from the current primary (of the view). If the timer expires, after the maximum time duration  $T_{acc}$ <sup>2</sup> to accept the request, the current primary is considered faulty and blacklisted. The protocol changes its state from normal to merge. *Merge* operation ensures *Safety* and reliability of the system upon expiration of the timer at non-primary replicas. Another replica becomes a primary and the value of  $T_{acc}$  is doubled. If there are already  $f$  replicas in the blacklist, the oldest in the list is removed to ensure *Liveness* property. This operation is not a change the view, but to agree to the requests from the previous views which were accepted and have to be executed by all the correct servers (this excludes all the blacklisted replicas). *Merge* operation collects information from all the servers to agree to the requests to be executed considering loss of messages, not sent messages, acceptance of requests by few correct servers and not all of them.

Unlike Aardvark, Spinning does not require clients to sign their requests. Spinning uses only MAC authentication, thus improving the performance over Aardvark. However, this makes the protocol vulnerable to MAC attacks. If a malicious client sends valid messages only to a subset of correct replicas, then primary would be forced to trigger a merge operation. This attack will possibly make the protocol spend the majority of its time executing the merge operation. This would reduce the performance to almost zero, but signing of a request can always be employed to handle such faults. Also, as previously said,  $T_{acc}$  is a static parameter, faulty replicas can intentionally trigger the timeout. A faulty primary can delay the request ordering by a little less than this parameter, without ever being detected as malicious. This behavior would greatly reduce the throughput during the view when the faulty replica is the primary. Several experiments were performed to analyze this intentional delay by primary and it was observed that throughput drops by 99% [18].

---

<sup>2</sup> $T_{acc}$  is a static system defined parameter and its initial value is defined by the system user



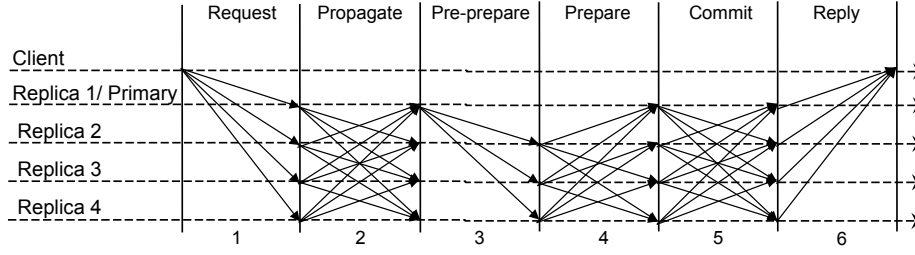


Figure 2.15: Communication pattern of RBFT

**RBFT.** Redundant-BFT (RBFT) [18] like Prime observed the malicious behavior of the primary replica in primary-dependent existing BFT protocols targeting high throughput, where primary can smartly degrade the performance of the system without being detected by correct replicas. RBFT follows the pattern of PBFT except the communication step to exchange the propagate message among replicas before primary sends pre-prepare message with request ordering to all replicas. RBFT executes multiple instances of the same protocol, each with a different primary replica executing on a different physical machine. All the protocol instances order the requests, but only the requests ordered by the *master* instance are effectively executed. Other instances are called *backup* instances, they order requests only for monitoring performance, in order to check if the master instance is providing the required performance. If the master instance is slower, then backup instances can trigger a *view change* and a new primary is elected at each protocol instance. RBFT is intended for open loop systems and it is presumed that backup instances would never be faster than the master instance. It exploits the multicore architecture of physical machines and uses multiple network interface controllers (NICs) for ensuring robustness.

RBFT runs  $f + 1$  protocol instances which are necessary and sufficient to detect a faulty primary and ensure robustness of the protocol. Backup instances run monitoring module to compute the throughput and if  $2f + 1$  nodes observe the ratio of performance of master instance to that of the back instance is lower than a given threshold, the primary of the master is considered Byzantine. It is also important that  $f + 1$  instances receive the same client requests. The client sends the requests (REQUEST message) to all the replicas and once the request is verified by the replica (only after it receives  $f + 1$  copies of the same request), it sends a PROPAGATE message to all other nodes. This step ensures that every correct node will eventually receive the request as long as the request has been sent to at least one correct node. RBFT follows the steps of PBFT from this point, with prep-prepare,



prepare and commit messages. Furthermore, RBFT implements a fairness mechanism between clients by monitoring the latency at each, which ensures that client requests are processed fairly.

## 2.5 Benchmarking Tools

With the growing demand for cloud services, it has become challenging and essential to guarantee performance and dependability aspects in the presence of real world fault scenarios. There exist many benchmarks which we discuss next.

### 2.5.1 Performance Benchmarks

Until recently, much emphasis has been given to performance benchmarks and many tools have been proposed that include Open System Testing Architecture (OpenSTA) [6], a distributed software testing architecture that evaluates performance and resource utilization in the presence of realistic heavy loads simulating activities of virtual users. It considers web servers, application servers, database servers and operating systems under test. Pylot [6], a free open source for testing performance and scalability of web services. SIPp [6] is a performance testing tool for the SIP protocol. SLAMD Distributed Load Generation Engine [6] is a Java-based application designed for stress testing and performance analysis of network-based applications. DBMonster [6] is a framework that test performance of SQL database driven applications under heavy loads. There are many more such performance benchmarking tools for various computer systems. This explains that users are most inclined towards analyzing performance, resource consumption (CPU utilization, network behavior and memory usage) and their associated costs.

Research has been made in the field of Byzantine fault tolerant protocols to guarantee *Liveness* and *Safety* properties in the presence of faults. Many protocols have been proposed which result in better performance during fault free scenarios but less effort has been made to evaluate these protocols in the face of faulty behaviors. There are some protocols, that considered evaluating the protocols under different types of faults, but compromises on the performance during fault free conditions. It is a tradeoff between performance and incorporating fault tolerant mechanisms which makes it difficult to analyze and choose the best protocol among all. Byzantine fault tolerance benchmark is a solution which can help the BFT system users to evaluate the prototypes in different scenarios for performance and dependability evaluations, with an ease of injecting different faultloads and workloads. However, even after a

decade of research on BFT protocols and benchmarks, considerably less effort has been done for benchmarking BFT systems. Such systems are defined next.

#### 2.5.1.1 a/b Microbenchmarks

Castro et al. [31] devised a micro-benchmark for stress testing the BFT protocols using different sizes of request and response messages. It is denoted as a/b microbenchmark where 'a' and 'b' are sizes of request and response messages in KBs, respectively. Microbenchmark uses a simple service emulating a real service, but has no state. The request operations receive arguments from the clients and return zero-filled results without performing any actual computations. Experiments can be performed for both read-only and read-write operations considering different arguments and request/response message sizes. For example, in a 0/0 benchmark, a client sends a null request and receives a null reply. In the 4/0 benchmark, a client sends a 4KB request and receives a null reply. In the 0/4 benchmark, a client sends a null request and receives a 4KB reply. It is worth noting that, computations and I/O operations (over the request data and operation) at both client and server side would introduce varying delays depending on the size of the request/response messages.

#### 2.5.1.2 Hermes Framework

Hermes [29], developed under TRONE project, is a diagnostic tool that can be used to inject faults in BFT-SMaRt [24]. Hermes allows system developers to evaluate the performance of BFT-SMaRt implementation by allowing them to inject faults and observe the corresponding behaviors. Hermes architecture allows injection of different faults such as crash faults, network faults, corrupted headers and forged signatures. It contains a fault injection orchestrator, responsible for injecting multiple faults across different nodes. But the actual fault injection is performed by the Hermes runtime which is integrated in every node. It considers injection of four types of attacks:

- *Attack 1*: Crash of malicious nodes
- *Attack 2*: Malicious nodes forge their payload size with MAX\_INT.
- *Attack 3*: Malicious nodes delay prepare messages to 90% of the timeout set at all the nodes for receiving the prepare messages.
- *Attack 4*: Malicious nodes delay prepare messages to 5 times of the timeout value.

The evaluation was performed in a simulated distributed environment with 11 virtual machines, where 10 machines ran BFT implementations and 1 was reserved for clients. Hermes measures the throughput and latency during fault free and faulty scenarios where the number of clients sending the requests vary from 1 to 11. Even though Hermes can inject various faults, it is limited to performance benchmarking of only BFT-SMaRt. Hermes is built in AspectJ which could easily be integrated with the code base of BFT-SMaRt (developed in Java). Most of the other protocols are implemented in C and their integration with Hermes C version is ambiguous. This creates an overhead of implementing all the protocols in Java despite the fact how complex BFT algorithms and their implementations are. However, Hermes was able to detect some implementation issues in BFT-SMaRt while evaluating the prototype under faults.

### 2.5.2 Dependability Benchmarks

The primary objective of benchmarking the dependability is to provide generic and reproducible ways of characterizing the system behavior in the presence of faults and also its impact on the performance. On the industry side, work at Sun Microsystems [65] defines a high level framework specifically for availability benchmarking during failure of hardware components, installation of software patches and system recovery. Another work by IBM, the Autonomic Computing initiative [5], develops benchmarks for analyzing system's self-management, self-configuration, self-healing and self-protection capabilities. In academia, Berkeley University proposed a dependability benchmark to assess human assisted recovery process [27]. There are many other dependability benchmarks available in the domains of hardware, cluster computing, database systems, web services, operating systems, online transaction processing (OLTP) systems, web-servers, etc. [8, 22, 50, 70, 92, 97, 107].

Considerable work has been done in designing fault and load injection systems to quantitatively assess the consequences of faulty behaviors in a system both at software and hardware levels [8, 13, 32, 60, 102]. Hsueh et al. [62] provide a good survey of fault injection techniques and tools for testing software dependability. Fault injector like Loki [32] injects various faults based on the partial view of global state change in a distributed system. Orchestra [43, 44] uses interception approach to inject communication faults in the different layers of a protocol by manipulating, dropping, injecting messages. Doctor [60] is an integrated software fault injection environment for injecting CPU specific faults, network and memory loads and network communication faults. Ferrari [69] (Fault and Error Automatic Real-Time Injection) also injects CPU,

memory, and bus faults but uses software traps. FTaPE [103] (Fault Tolerance and Performance Evaluator) allows injection of faults using mode registers in CPUs, memory locations, and disk subsystems. Xception [38] injects more realistic faults depending on underlying system applications and processors, advantaging from advanced debugging and performance monitoring features. Ballista [45] is a black box software testing tool that uses combinational tests of correct and incorrect values for parameters to subroutine calls, methods and functions. Finally, CLIF [47] is a load injection framework for distributed platforms where it deploys, controls and monitors load injectors and captures performance and resource consumption upon varying loads. Bobelin et al. [26], also considered proposing security-aware architecture with an easy expression of security requirements and an actual enforcement of those requirements.

Apart from the research on performance benchmarks for BFT systems, advancements have been made in developing BFTSim [99], a simulation environment and Achilles [21], a tool to detect trojan messages in distributed systems.

**BFTSim** [99] intends to implement BFT protocols from pseudocode in the high-level declarative language and compare these protocols under identical crypto-primitives, workload and network conditions. It implements three BFT protocols, precisely, PBFT, Q/U and Zyzzyva and validates the performance against the published results. Simulator observes that a protocol's performance characteristics are primarily inherent in its high-level design, not the particulars of its implementation. However, other state-of-the-art protocols, such as Chain, Ring, Aliph, OBFT, Aardvark, Prime, Spinning, etc., did not consider using the simulator due to the underlying complex high-level declarative language. Also, some protocols assume different network conditions, such as, exploitation of multiple core architecture [24], and multiple network interfaces [18, 36]. Furthermore, as a known fact, BFT protocols are complex in nature, therefore, their re-implementation is an overhead for BFT developers. Radu et al. proposed a tool, **Achilles** [21], that searches trojan messages in a distributed system. Trojan messages are messages that seem correct to the receiver, but cannot be generated by any correct sender [21]. Such messages reside in untested, uncommon code paths and are difficult to track using regular testing mechanisms. Such messages can have a major impact on the performance of a distributed system. For example, Amazon S3 storage system went down for several hours due to the propagation of single corrupted bit to the whole system corrupting the entire stored information [1]. Achilles evaluated PBFT for detection of trojan messages and rediscovered the previously known MAC attack vulnerability. This attack is possible due to the simplicity of verification by servers on the client's requests. Servers would accept client

(malicious) requests without checking the MACs. Achilles can be a useful tool for determining coding flaws in the distributed protocols.

From our comprehensive analysis, we can conclude that it is necessary to primarily focus on the practical evaluation of BFT implementations under different workload and faultload settings from the viewpoint of adapting them to real world systems. For realizing the same, it is important to design a generalized, high-level, easy to use, performance and dependability benchmark with faultload and workload injectors.

## 2.6 Discussion

This chapter embarks on the study of distributed system characterizations which are extensively used in the representation of many replication techniques such as agreement and consensus, state machine replication (SMR) and quorum. We studied SMR replication thoroughly, that forms a baseline for all the BFT protocols proposed so far. This chapter provides a comprehensive research on Byzantine (arbitrary) behaviors, analysis of the upper bound on the number of replicas required simultaneously to tolerate up to  $f$  Byzantine faults and many existing BFT protocols. These protocols were classified into two groups, precisely, (i) performance enhancements in fault-free scenarios, and (ii) minimizing performance degradation in faulty conditions. Group 1 primarily consists of proprietary protocols aiming to reduce the cost (in terms of the number of replicas) while increasing the performance of the BFT system in the absence of faults. However, Group 2, so-called robust protocols, aims to provide performance guarantees (i.e., minimum degradation in performance in faulty settings in comparison to fault-free scenarios) in the presence of Byzantine attacks. Additionally, this chapter provides a theoretical analysis of all the BFT protocols and lists the Byzantine fault scenarios handled by robust protocols. This chapter also provided a complete taxonomy on the known and published state-of-the-art BFT protocols in an effort of helping the future researchers working on BFT techniques.

Furthermore, we studied the existing work on performance and dependability benchmarks not only for BFT systems, but also in other domains such as web servers, database systems, application servers, grid and cluster computing. Our investigation of existing benchmark tools expresses absence of a standard benchmarking approach for empirically evaluating competing BFT systems in terms of Quality of Service (QoS) metrics under various workloads and faultloads settings.

We were further motivated by the work of Kanoun et al. [8]. Their techniques guided us with the basic building blocks required in designing a per-

formance and dependability benchmark using complex workload, faultload for different distributed systems.

Analysis of the state-of-the-art BFT protocols and benchmarking frameworks motivated us to design a comprehensive performance and dependability benchmark tool for evaluating BFT systems with the following objectives.

- **QoS metrics scalability.** By this we mean, that multiple metrics must be analyzed in parallel without limiting the analysis to performance and dependability aspects in terms of latency, throughput, availability and reliability. A benchmark must also consider analyzing other metrics such as cost, security, testability, interoperability, scalability, etc and must be scalable to incorporate them. It should also provide some high level statistics such as the number of unsuccessful and successful operations, and number of incorrect responses. It must be structured to produce low-level statistics such as CPU and network utilization, a number of operations performed by each server in the system and size of data read and written during each operation.
- **Heterogeneity.** It is well-known that distributed systems are complex and consider various workloads, faultloads described via multiple characteristics, features and dimensions. In real-world scenarios, we observe complex workload where some applications are computationally heavy while others are data access intensive. However, few have both bulky computational and data access characteristics. Similarly, we have recognized various faulty behaviors commonly occurring in distributed systems, such as operator mistakes, network failures, hardware and software faults. Considering our analysis, a benchmark must provide a simple way to define different workloads and faultloads attributes, and inject them.
- **Adoptability.** Benchmarking framework must be easy to understand and use by the researchers and as well as amateur end-users for analyzing underlying system's quality measurements.

Thereupon, our exhaustive survey affirms a need of comprehensive benchmark to evaluate performance, dependability and other QoS metrics of distributed systems under various workloads and faultloads.



# A General Architecture for Performance and Dependability Benchmarking of BFT Protocols

---

## Contents

---

<b>3.1</b>	<b>Overview and Objectives . . . . .</b>	<b>66</b>
<b>3.2</b>	<b>Dependability and Performance Benchmarking Specifications and Validations . . . . .</b>	<b>68</b>
3.2.1	Categorization . . . . .	69
3.2.2	Measures . . . . .	70
3.2.3	Experimental Dimensions . . . . .	70
3.2.3.1	Faultload . . . . .	71
3.2.3.2	Workload . . . . .	72
3.2.3.3	Measurement . . . . .	72
<b>3.3</b>	<b>General Benchmarking Architecture and Framework for Distributed Protocols . . . . .</b>	<b>73</b>
3.3.1	Benchmarking Steps . . . . .	73
3.3.2	High-level class Diagram of Performance and Dependability Benchmark Architecture . . . . .	75
3.3.3	Overview of Communication Primitives Operation by Orchestrator . . . . .	78
<b>3.4</b>	<b>BFT-Bench: Case Study of Benchmarking BFT Protocols . . . . .</b>	<b>79</b>
3.4.1	Faultload Dimensions . . . . .	80
3.4.2	Workload Dimensions . . . . .	81
3.4.3	Measurement Analysis . . . . .	82
3.4.4	Potential Benchmark Users . . . . .	82



<b>3.5</b>	<b>Benefits of General Architecture . . . . .</b>	<b>83</b>
3.5.1	Reduction in Software Development Cost . . . . .	83
3.5.2	Extensibility . . . . .	83
3.5.3	Reusability . . . . .	84
3.5.4	Testability . . . . .	84
<b>3.6</b>	<b>Summary . . . . .</b>	<b>84</b>

---

Until recently, for distributed systems, a benchmark implicitly refers and limits to performance benchmarking. But there are many aspects which converge on the importance of dependability measurement of today's distributed systems. Industries define easy to use standard benchmarks for measuring the performance of a system in a deterministic and reproducible manner, but lack in characterizing standards for dependability analysis. However, with novel explorations, increasing demands and advantages of distributed systems, particularly in the aspects of cloud computing has compelled the researchers and developers to fabricate solutions to benchmark the dependability and performance of these services and distributed protocols.

A dependability benchmark is intended to characterize the system behavior in the presence of faults which could potentially include component failures, hardware or software design defects, arbitrary faults and disruptions due to dynamic environments. Available dependability and performance state-of-the-art benchmark solutions do not acknowledge many specific challenges faced by a distributed system when intending to use distributed protocols. We believe that there is a strong need for designing a performance and dependability benchmark which evaluates the behavior of distributed protocols under faulty and highly demanding real-world settings. Therefore, in this thesis, we intend to devise a benchmark for analyzing dependability and performance of distributed protocols, namely Byzantine Fault Tolerant protocols, which increase the reliability and availability metrics of a system in the presence of arbitrary failures [80].

In this chapter, we present a generic architecture of a benchmark for evaluating performance and dependability aspects of distributed protocols. We describe each component of the benchmark, responsible for the generation of various faultloads and workloads, their injections, and measurement and analysis of performance and dependability attributes. We have used the presented generic architecture to design a software prototype, BFT-Bench (explained in details in the next Chapter 4), which evaluates different implementations of BFT protocols. We also demonstrate the benefits of building a generic architecture and its easy adoption to enable researchers and developers in industry and academia to extend it for analyzing various distributed protocols for reliable file systems, verification, communications, different replication techniques, etc., in the presence of different faulty behaviors.

## 3.1 Overview and Objectives

With the increasing dependency on cloud services and its pay-as-you-go models, users have become more interested in evaluating the reliability and availability metrics of such services along with other high level statistics like performance, scalability and cost. Furthermore, with growing on-demand nature of the cloud services, it has become challenging and essential to guarantee Quality of Service (QoS) attributes of the systems in the presence or absence of different faulty behaviors. Until recently, substantial emphasis has been given to evaluate performance aspects of various systems while lesser progress has been made to benchmark distributed protocols. The primary objective of any benchmark is to provide generic and reproducible ways of characterizing system behavior in the presence of faults and also analyze their impact on the performance and dependability metrics.

Clearly, a lot of work is still required concerning the empirical assessment of dependability and performance aspects. However, Madeira et al. [83] point out some of the recent works addressing coupling of such issues, such as bringing together performance benchmarking and dependability assessments [7], field measurement and fault injection [64], field measurement and modeling [68], fault injection and modeling [17], and use of standard performance benchmarks as a workload for dependability evaluation [39].

From the literature, we have seen that there is no scientific approach or framework developed so far that can help the researchers and developers (from industry or academia) to evaluate the dependability quality aspects of an underlying distributed protocols in cloud systems. And in this path we explore and design a comprehensive benchmark framework for measuring QoS metrics of distributed protocols and their comparison with competing solutions.

From a practical point of view, dependability is a promising approach to assess QoS metrics related to the behavior of a distributed system and its protocols in the face of various faults. Thus, any dependability benchmark should be clear enough to allow implementation of the specifications in a system to benchmark the dependability and complete understanding of benchmark results.

DBench report [8] provides a basic foundation of dependability benchmarking and defines 3 major guidelines that should be followed while designing a benchmark. These include (1) implementing the specifications, (2) performing the experiments with uniform conditions and (3) analysis of results. Any benchmarking framework should be independent of the underlying infrastructure and must consider a wide range of workload characteristics, including a number of concurrent users, and data vs compute-intensive jobs.

Designing a benchmark framework for distributed protocols is complex and time-consuming as it needs a thorough understanding of the algorithm to determine the corner faulty cases, ways to inject these faults in a system and definitions for evaluating different QoS metrics. Distributed protocols are the building blocks of a cloud system and thus, it becomes necessary to analyze their performance and dependability before they are integrated in the real world settings. It is equally substantial to clearly define the objectives while designing a benchmarking framework. Therefore, we recognize:

- Considering a system model which defines the protocols under study with all the considered assumptions (probably common to all to have a unified testing environment).
- Defining workloads and faultloads.
- Defining components and modules and relationships among them such as fault injection module's relationship with faultload and the system under test for injecting a fault.
- Building components for the framework with minimum coupling.

This chapter makes the following contributions:

- We present the main dependability benchmark dimensions as per DBench [8] report considering categorization which defines the application area and benchmarking purpose, measures to define nature, type and assessment methods, and experimentation to define the system under test, faultloads, workloads and measurements of considered QoS metrics. While defining these dimensions we also consider benchmarking validation key features that includes representativeness, repeatability, reproducibility, portability, non-intrusiveness, scalability, benchmarking time and cost and simplicity.
- We illustrate the architecture of high-level and low-level classes which can be adapted easily for integrating implementations of comparable distributed protocols.
- We demonstrate the adaptability and usability of general architecture proposed in this chapter by instantiating the integration of Byzantine Fault Tolerant protocols. We propose a performance and dependability benchmark framework BFT-Bench build using the components and modules of general architecture such as generation of faultloads and workloads, their injection, and analysis and measurements of performance and dependability metrics using output logs and monitoring reports.

The proposed general architecture can be adopted for developing a benchmark for any type of distributed protocol for empirical and effective measurement of performance, cost, availability, security, etc. BFT-Bench framework (detailed in the next chapter) is designed specifically to analyze and compare implementations of BFT protocols under different fault settings and various workloads. The design specifications of the framework demonstrate easy integration of various BFT protocols with minimum code changes in the software prototypes to inject some context-based faults. We graphically illustrate the evaluation of performance and dependability aspects for these BFT prototypes using BFT-Bench framework.

## 3.2 Dependability and Performance Benchmarking Specifications and Validations

As said in the report of DBench project [8], a meaningful framework for dependability and performance measurements is the one which clearly states and understands all the impacting dimensions. This gives an essential insight to the problem space, ways to catalog and determine the measurable aspects of performance and dependability benchmarks. Along with dimensions, it is equally important to consider a set of properties that should be satisfied and verified while conducting experiments for analyzing the performance and dependability metrics. For our approach, while designing the generic architecture, we considered the same classifications: (1) Categorization, (2) Measure, and (3) Experimentation, and the same set of validation properties like representativeness, portability, reproducibility, scalability, benchmarking time and cost, etc., which are mostly verified during measure and experimental phase of the benchmark specifications.

*Representativeness* reflects how closely workload of the benchmark corresponds to the actual workload of a real system and how well injected faults resemble the real faults. *Repeatability* is a property which guarantees statistically equivalent results when a benchmark is run more than once in the same environment, whereas *reproducibility* is the property which guarantees that another benchmark user obtains statistically equivalent results when the benchmark is implemented from the same specifications for evaluating the same system. *Portability* refers to the usability of the benchmark for various systems within the same application domain. This feature enables the benchmark users to compare different systems, services, protocols and modules. A benchmark is said to be *non-intrusive* if it requires minimum changes in the system under test. *Scalability* allows benchmark users to test systems of differ-

### 3.2. Dependability and Performance Benchmarking Specifications and Validations 69

ent sizes. For example, in case of BFT protocols, the size of a system changes depending on the number of faults it can tolerate. A number of nodes required are 4, 7, 10 for  $f = 1, 2$  and 3, respectively. Benchmark *time and cost*, is the time required to receive the results from a benchmark which includes system setup and preparation time, running experiments, and data monitoring and analysis.

Categorization	
Considered System	Byzantine Fault Tolerance Protocols
Application Area	SLA oriented cloud systems
Benchmark Users	Researchers, Developers
Measures	
Quantitative	Throughput, Response Time, Availability, Reliability, Resource Utilization
Qualitative	Representativeness, Portability, Scalability, Reproducibility, Non-intrusiveness, Benchmarking time and cost
Experimental Dimensions	
Workload	#concurrent clients, Size of the request message
Faultload	Fault type, Fault trigger time, Fault location
Measurements	Performance, Dependability, Cost, Resource Utilization

Figure 3.1: Specifications and Validations of a Performance and Dependability Benchmark

Figure 3.1 demonstrates the performance and dependability benchmarking dimensions for any distributed protocol. In our case, we consider BFT protocols.

#### 3.2.1 Categorization

Categorization allows us to define the system in consideration (or system under test) and benchmarking context. It also characterizes benchmarking context which includes defining benchmark users or performers, and the purpose of building a benchmark tool. Figure 3.1 illustrates these parameters from the perspective of analyzing and comparing different implementations of BFT protocols so that the best among all (depending on the performance and dependability measurements and system requirements) can be integrated into cloud systems to guarantee QoS metrics.

Describing the considered system allows us to define the nature of the system, application area and operating environment. This allows understanding of the system at abstraction and functional layer levels. It enables the users to perform evaluations of specific measures on key components, functionalities and features. We consider contrasting various BFT protocols and how their implementations handle faults and maintain their performance and dependability aspects. Application area and operating environment impact the selection of measures during experimentation that include types of faults to be injected. We consider PaaS cloud models which intend to run a BFT protocol for tolerating up to  $f$  Byzantine faults. Thus, we consider the impact of cloud users running/developing an application and using underlining features of a model. Performance and dependability benchmarking are useful for PaaS cloud providers willing to integrate BFT protocols and aspire in performing the comparative analysis of available BFT implementations in the face of physical, design and arbitrary faults. Cloud providers become the prospective benchmark users performing evaluations to assess the capabilities and weaker areas of a BFT protocol.

### 3.2.2 Measures

For any dependability and performance benchmark, it is essential to construe the measures to be assessed under various conditions. Measures allow the characterization of the system in a qualitative and quantitative manner. Qualitative measures are defined in terms of features related to capabilities and properties like representativeness, portability, scalability, reproducibility, non-intrusiveness, benchmarking time, and cost. Whereas, qualitative measures include system availability, response time, throughput of the system, reliability, in the presence of faults. The occurrence of faults in a system leads to performance degradation, but it is not always true for dependability. The system may remain available and reliable. Thus, we consider defining performance and dependability measures both under fault-free and faulty conditions. We also define comprehensive measures that include a number of faults a system is intended to handle, different faulty models, etc.

### 3.2.3 Experimental Dimensions

Figure 3.2 illustrates the various modules of performance and dependability architecture with experimental aspects. It also defines system under benchmarking or system under test which is a targeted system to be evaluated under various faultloads, workloads for measuring QoS metrics. All these modules

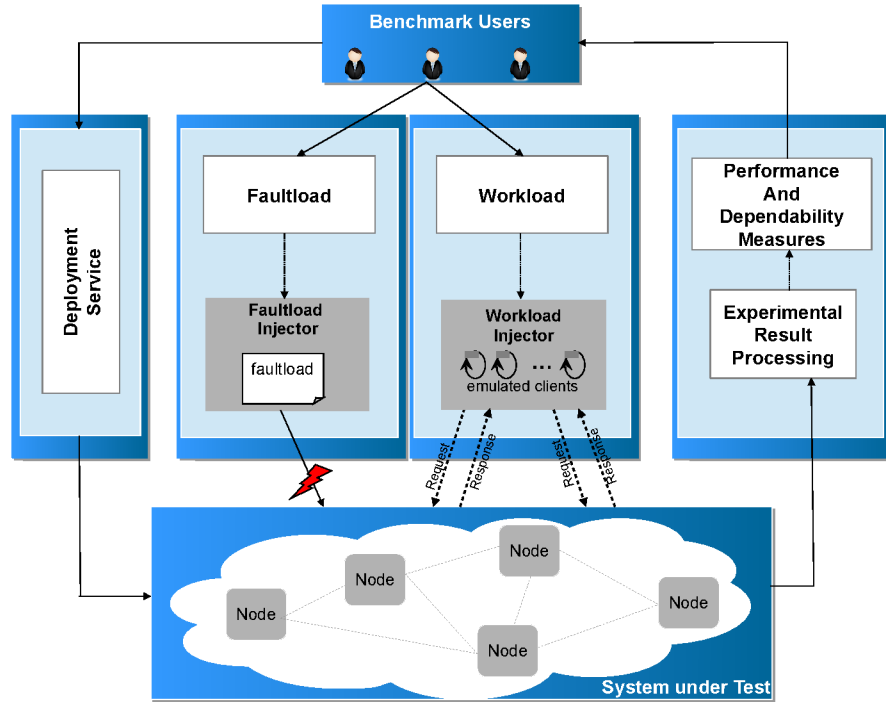


Figure 3.2: Various modules of a Generic Performance and Dependability Benchmark

are part of orchestrator which is responsible for inter-module communication, data exchanges, and metrics analysis.

### 3.2.3.1 Faultload

The faultload describes a set of faults that are intended to emulate the real world threats or faults a system can encounter at any point of time. We categorize these faults as (1) Internal faults (hardware or software faults) which mainly consider a node crash, deviation from the correct algorithm path, run-time issues, and the arbitrary behavior of a node, for example, delaying of messages, sending corrupted messages to other nodes, etc., and (2) External faults which depend entirely on the cloud system. More often it has been noticed that a significant high workload leads to random faults and disruption in the system (mainly due to node failures). Therefore, we also consider such types of workloads as faultloads (under Section 4.4.3, we define a fault type called system overloading, where the workload is considered as a faultload).

For our simplicity, we focus on injection of only internal faults primarily node crash, intentional message delay, network flooding, etc., while making



sure that key features of benchmark validation are satisfied (representativeness, portability, easy injection of faults, etc.) For the convenience of benchmark users, we generate the faultload file where users just need to input *what* fault is to be injected, *when* (at what time) it is injected and *where* (location in the system, for example, node address) it is injected.

### **3.2.3.2 Workload**

The workload is an abstraction of the actual work that an instance or a set of instances is going to perform. From the practical point of view, the primary aim of any benchmark is to emulate the real world workloads to examine the system's actual behavior in handling it. It represents actual profile of a considered system. We consider fabricated workload for evaluating the metrics where we define the number of concurrent clients sending requests in the system and size of each request/response message. Workload plays a vital role and makes an impact on performance (with low load, the system maintains its performance but during high load, performance degrades) and dependability (with high load, it is possible that system breaks down and is unable to serve clients' requests).

### **3.2.3.3 Measurement**

Measurements performed on the target system allow the observation of the behavior of a system upon injection of workloads and faultloads. The measures of interest are then obtained from processing these measurements. Basic measurements include the identification of outcomes of system under benchmarking/test upon injection of workloads and faultloads. Another important aspect related to the assessment of the behavior of the system is in the time domain. For all performance benchmarking, timing related measurements have formed the basis for evaluating any system. We extend the same for benchmarking dependability metrics too. For any system under test, we make the fault-free evaluations as the baseline for comparison with the analysis of performance and dependability metrics in the presence of faults. Primarily, we consider the following time-dependent metrics which are measured from the perspective of a client:

#### **Performance Analysis**

- *Latency* is calculated as the time required by a system to respond to a job. In other words, it the total time elapsed from the moment a user submits a request until it receives the corresponding response.

- *Throughput* is calculated in terms of a number of client requests (transactions or jobs) executed by the system per unit of time.

#### Dependability Analysis

- *Availability* is a measure of time a system is responding to requests of clients.
- *Reliability* is the ability of a system to always respond with correct responses to the client's requests, during a period of time.

## 3.3 General Benchmarking Architecture and Framework for Distributed Protocols

In this part of the chapter, we discuss the various steps required for conducting the dependability and performance benchmark of a distributed system or protocol or component and the high-level class architecture of the general benchmark.

### 3.3.1 Benchmarking Steps

Benchmarking is achieved in several steps forming a *benchmarking scenario*. It consists of three major steps, *analysis* step, *experimental* step and *assessment* step. The experimental step is further divided into 2 phases; *load injection* and *statistics monitoring*.

Figure 3.3 gives the three steps for performance and dependability benchmarking and their interrelations.

**The analysis step** consists of defining the categorization and measure dimensions for the system under test. This step can also be expressed as a load generation step as benchmark users generate workloads and faultloads. Before the target system is initialized, benchmark users can define system configuration parameters which require basic information such as a number of clients sending the request (in the beginning) and fault type (a system is initiated under fault-free conditions). They also characterize the faultload parameters where they define *what* fault is injected, *where* it is injected and *when* it is injected into the target distributed system. Once the loads are defined, it enters into the second step of benchmarking.

**The experimentation step** pertains to carry out the experimental part of the benchmark, i.e., injecting the workload and faultload, and collecting

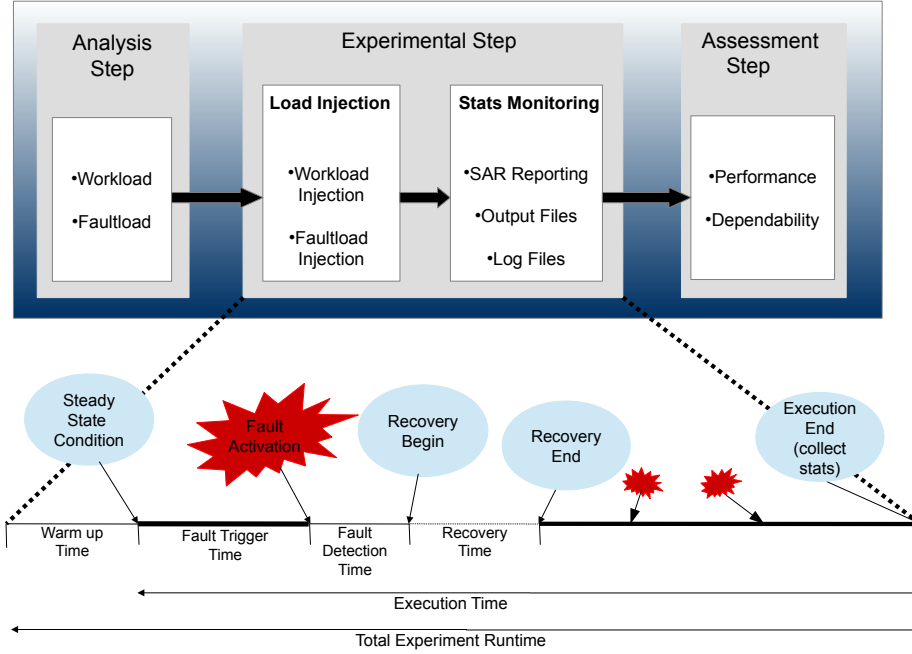


Figure 3.3: Benchmarking steps for performance and dependability benchmarking

statistical monitoring reports via output files, log files and SAR generated files as the results of system execution.

1. *Load Injection Phase*: Experimentation is tightly related to the target system, defined workload and faultload during the analysis step. Workload injector instantiates a concurrent number of clients as defined in the workload to stress test the system under benchmarking whereas faultload injector triggers the type of fault at the specified time at a given location (node).
2. Similarly, *Statistics Monitoring Phase*: measurements and monitoring is tightly related to the target system accessibility and important modules. This phase specifies the kind of outputs to be produced during the experimentation to assess the performance and dependability metrics.

Every experiment runs for a *warm up* period and then an *execution period*. Warm up period gives enough time to the system to stabilize before the statistics monitor starts collecting the data. During the *warm up* phase, system runs on the basis of the configuration parameters and no workloads and faultloads are injected. Once the experiment reaches the *execution* period, workload and

faultload are initiated and triggered at the time mentioned in the parameters which are in accordance with the beginning of the execution period. Actual statistics are collected during *execution phase* for future analysis. Benchmark user can define the time until which the experiments run. Once the execution ends, the system collects the SAR reports, output and log files to further analysis to determine the performance and dependability benchmarking.

The last step, i.e. **assessment step** analyzes various reports from the previous step and computes throughput, latency, availability and reliability of the system under test. During this phase, we also measure resource utilization in terms of CPU usage, network bandwidth consumption and some low-level statistics like number of rejected requests, the number of times a request was re-transmitted, etc. These results can be easily presented in the form of graphs, pie charts, line diagrams for better comparison and analysis with other competing target systems.

#### 3.3.2 High-level class Diagram of Performance and Dependability Benchmark Architecture

Figure 3.4 illustrates the classes, methods and some of the member variables of the general architecture presented in Figure 3.2. Most of the modules of high level class diagram are generic and do not need any additional implementations (features wise) before integrating a new target system. However, implementations of some classes like SystemUnderBenchmarking, faultload, workload, and request are system dependent and require additional implementation. Considering faultload, it makes sense to inject a type of fault on one system but may not hold the same significance when injected in another target system. Classes such as FaultInjector, workloadInjector and Orchestrator are independent of any system and require no modifications and can be used as plug and play modules.

We now describe each of these classes and some of their associated methods at implementation level.

1. **Faultload.** This class considers fault parameters from class *fault* and generates a faultload to be injected into the system under test. Fault describes the specifications related to the fault to be injected. It defines the *type* of fault, *time* at which it is triggered in the system and *where* in the target system (location of the fault). All the faults are user-defined.
2. **FaultloadInjector.** This class includes a method that injects a fault in the target system. It also consists of onEvent methods which trigger when it is the time for injecting a fault depending on the elapsed *warm*

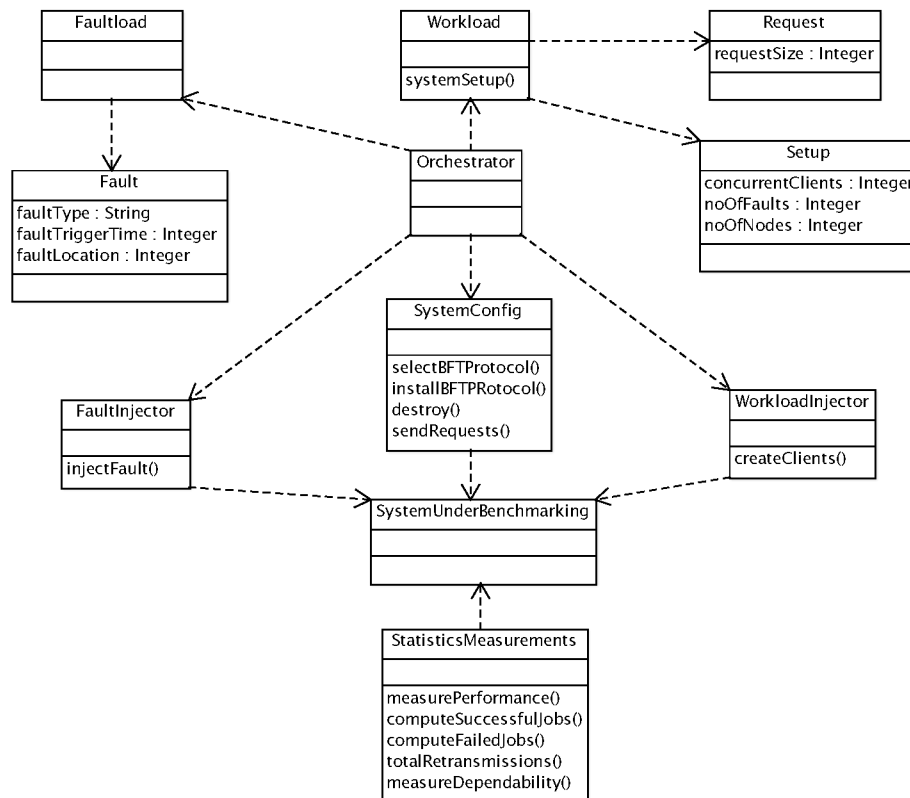


Figure 3.4: High-level Class Diagram of Generic Benchmark Architecture

*up* and *execution time*. This class, thus, monitors experimental specified times for injecting a fault accordingly.

3. **Workload.** This class consists of the methods to setup the initial target system for benchmarking. For measuring the dependability of any system (BFT protocols in our case), its needs to be instantiated. Methods of workload class consider input data (request from the users), number of nodes in the system that runs a distributed protocol, etc. Request class considers the request message and its size, whereas setup class accounts for total number of clients sending requests (initial), number of faults target system must handle and the number of nodes running the protocol.
4. **WorkloadInjector.** This class creates the number of concurrent clients (emulation of real world system users) waiting for their requests to be served. But the clients are allowed to send requests only in FIFO order, where a client sends a request and waits for a response before sending another request. Each client is given a fair chance to send their requests, sending their requests in a round-robin fashion.
5. **SystemConfiguration.** This class includes the methods that are responsible for preparing the target system under test. It consists of methods to select one out all the competing distributed protocols (solving the same problem), install them on the distributed cloud network (it can be public, private, hybrid or any local setting as described in Section 1.1), destroy the setup once the experimentation is over and finally, responsible to distribute requests (coming from the clients) in the FIFO order.
6. **SystemUnderBenchmarking.** This class is actually responsible for injecting faultloads and workloads depending on the parameters provided by the faultload and workload injector classes, respectively. This class contains methods which identifies the complete system, considering the locations of the nodes, topological arrangements of these nodes in the system, their properties, like whether it is a primary node or a backup node. It is important and necessary as benchmark users might want to inject a fault in a particular node. For example, failing of a primary node (master node) in a protocol impacts the performance and dependability of a system in a much critical way than the failing of a backup node (slave node). Also, this class is responsible for starting and stopping the nodes once an experiment starts or ends. It also includes methods that provide statistics such as SAR (System Activity Reporting) reports generated at each node in the system.

7. **Measurements.** This class has methods to compute statistics depending on the information gathered from `SystemUnderBenchmarking` class. It includes methods that compute latency (depending on the send and receipt of the request and its corresponding response), throughput (depending on the total number of successful requests served by the system), availability (depending on the time the system is responding which includes correct or incorrect responses) and reliability (depends on the number of correct responses by the system before it breaks down due to any arbitrary fault). It also measures the resource utilization depending on the usage of CPU and network bandwidth. This class computes some other low-level statistics related to the total number of re-transmissions of a request, etc.
8. **Orchestrator.** This class is the backbone of all the classes communicating with others and is responsible for providing data between the classes. This class basically orchestrates the whole system from starting until the end of the evaluation considering the parameters, configurations, and experimental results from/to the benchmark users.

### 3.3.3 Overview of Communication Primitives Operation by Orchestrator

As seen in the above sections, *orchestrator* is the main component of the benchmark and its primary goal is to provide the orchestration among all the modules, for communication, data exchange and result analysis. It communicates directly with the benchmark users, enabling them to prepare the target system for evaluations and comparisons. The interaction between orchestrator and a target system is built on top of two communication primitives, namely *injectFaultAction* and *StatNotification*.

Figure 3.5 demonstrates the communication using these primitives.

#### **InjectFaultAction.**

The main responsibility of *InjectFaultAction* is to inject faults according to the specifications provided by the orchestrator, determined from the parameters defined by the benchmark users. *InjectFaultAction* uses the necessary network infrastructure information to inject a fault in the specific node.

#### **StatNotificationAction.**

Orchestrator uses send and receive procedures for sending a request and receiving a response from the system under test and performs measurements as per the collected statistics. *StatNotificationAction* communication primitive is mainly responsible for sending the status report of all the nodes in the

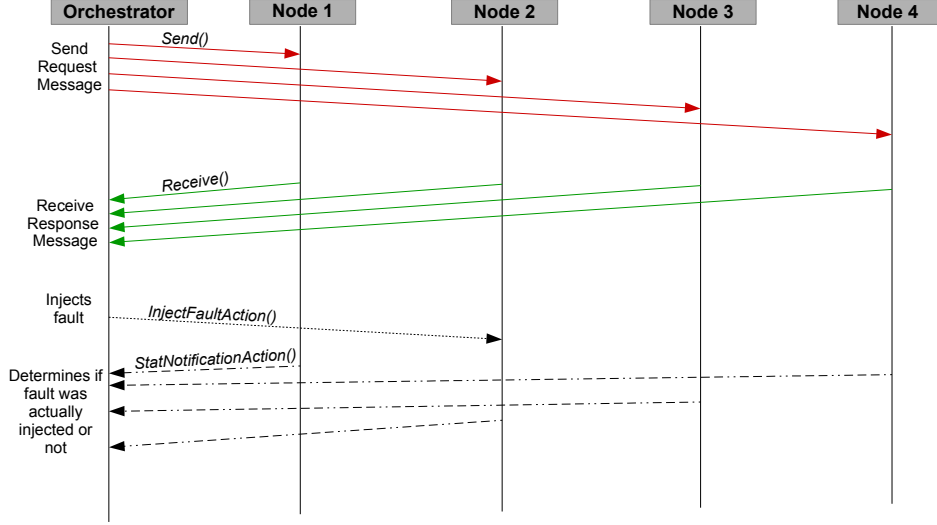


Figure 3.5: Communication Primitives Overview at Orchestrator

system. This enables the orchestrator to determine if the fault was correctly injected at the right location or not, and is the fault performing the right actions it is supposed to execute. This also allows the benchmark users to test the system with corner cases and various malicious faults.

Orchestrator send and receives messages in a synchronous manner from the manager in the target system (mechanism explained in details in the next section) and performs analysis of performance and dependability metrics. Orchestrator monitors the *warm up* and *execution time* and once its the time for triggering the fault, orchestrator calls `InjectFaultAction`. Manager of target system sends `StatNotificationAction` to the orchestrator in an synchronous manner notifying about the injection of the fault and its impact through statistics reporting.

### 3.4 BFT-Bench: Case Study of Benchmarking BFT Protocols

In this section, we illustrate a case study of implementing performance and dependability benchmark for BFT protocols called BFT-Bench using the generic benchmark architecture proposed in the previous section. This section not only presents the guidelines to integrate multiple implementations of BFT protocol but also demonstrates how easy it gets to design a benchmark framework using a general architecture. We have considered three BFT protocols for our case study, namely, PBFT [30], Chain [56] and RBFT [18]. We have



briefly explained them in Chapter 2 demonstrating the communication pattern. They are presented again in Chapter 4 from the perspective of fault injection and fault handling mechanisms. As discussed previously, performance and dependability benchmark must achieve some specification and validation key features:

1. Providing a clear definition of System Under Benchmark (SUB) or System Under Test (SUT).
2. It should define the qualitative and quantitative measures.
3. Considering the experimental dimensions, it should characterize different faultload and workload possible for the testing the SUBs and means to inject them.
4. Benchmark must describe measurements to be performed such as throughput, latency, availability, reliability, cost, etc.

BFT-Bench framework measures performance and dependability of the various BFT protocols in presence of arbitrary faults, primarily considering replica crash (a node completely halts and makes no progress), intentional message delay (node starts to delay sending of messages to intentionally increase the response time), network flooding (node sends malicious/corrupt messages to flood the network) and system overloading (demonstrates the maximum workload a system can handle and its impact on performance and dependability metrics, before breaking down).

### 3.4.1 Faultload Dimensions

A faultload file in BFT-Bench framework describes the type of fault to be injected, time at which it is to be injected, and the location in the system where it is injected. All the BFT protocols are meant to handle any arbitrary fault. But considerably less effort has been made to analyze the performance and dependability of the implementations of these BFT protocols in the face of various real world fault scenarios. Due to this, BFT protocols still encounter reluctance in adoption by practitioners. Therefore, BFT-Bench provides a mean to inject arbitrary faults in the BFT system and analyze the system behavior in terms of performance and dependability attributes. We consider injection of three types of Byzantine faults:

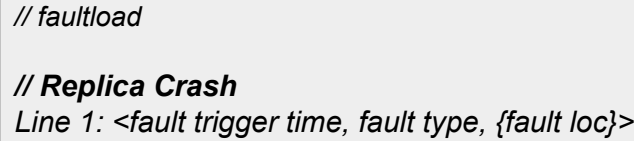
*Replica Crash.* In this fault, a node crashes and completely stops making a progress. When a primary node crashes, backup nodes stop receiving messages from primary and undergoes a *view change* protocol to elect a new

primary node. Whereas, when a backup replica crash, the system still continues and does not need a *view change* protocol. All the nodes can still reach a consensus and commit to the client's request. In BFT-Bench benchmark, we consider failing of primary as it has a higher impact on the performance and dependability than the crash of backup nodes.

*Message Delay.* In this fault, any node can start to delay the messages it is suppose to send to other nodes. But we select primary to delay a certain message (due to its high impact on performance) by a period of time, which is sufficient enough to hurt the performance of the system, but inadequate to detect the faulty primary and trigger a *view change* protocol to replace it.

*Network Flooding.* In this fault, node starts to send corrupt malicious messages to flood the network of correct nodes, with an intention of wasting their computational resources in validation corrupt messages. Some BFT protocols apply flood adaptive mechanisms to deal with such a behavior while some do not.

Figure 3.6 illustrates an example of a faultload for a replica crash.



```
// faultload

// Replica Crash
Line 1: <fault trigger time, fault type, {fault loc}>
```

Figure 3.6: An example of faultload descriptor

### 3.4.2 Workload Dimensions

A workload file describes the number of clients sending the requests concurrently and the request size. We have considered request size as one of the key parameters (higher the size of the request, higher is the time taken by the nodes to perform cryptographic verification and validations). The BFT-Bench framework considers fault type *system overloading* which is purely workload dependent.

*System Overloading.* In this fault, we do not inject any Byzantine fault, rather we inject a heavy workload (in terms of the number of concurrent clients sending requests), which some BFT protocol implementations are able to sustain while others succumb to the breakdown of the system.

### 3.4.3 Measurement Analysis

BFT-Bench is used to measure the performance and/or dependability of BFT protocols using various faultloads and workloads defined above. Framework produces runtime statistics for performance, namely, throughput and latency, and dependability, namely, availability and reliability. It also produces statistics related to resource utilization in terms of CPU usage and network bandwidth consumption. Some low-level statistics such as total number of request re-transmissions, and number of failed vs successful jobs (although all the BFT protocols should result in successful responses to maintain *Safety* and *Liveness* properties (discussed in Chapter 2 under Section 2.2.1)).

### 3.4.4 Potential Benchmark Users

With the growing dependency on cloud systems for different types of service, namely, infrastructure, platform, application, it becomes equally important for these services to be reliable, efficient, available, performing and safe. As seen until now, these performance and dependability metrics are questionable when the cloud systems encounter arbitrary faults. QoS parameters cannot be guaranteed, leading to huge loss and trust of cloud users. Considering the situation, BFT protocols have a bright future as they intend to handle arbitrary faults. But, it is fairly critical to test the BFT protocols to build confidence that they are capable of handling various real world faulty scenarios. The BFT-Bench framework is proposed to solve the problem of benchmarking the BFT protocols in the presence of different fault models and analyze their performance and dependability metrics. The users of BFT-Bench can be:

1. End users (distributed system, cloud providers) intending to use BFT protocols for their systems.
2. End users willing to compare different available BFT prototypes in the presence of arbitrary faults.
3. Researchers and developers design robust BFT protocols for constant performance during the presence of faults, as it was when there were no faults.
4. Researchers contemplating different implementations to modify the software design of the algorithm to integrate robust mechanisms.

## 3.5 Benefits of General Architecture

In this section, we point out some of the key features and benefits of the proposed generic benchmark architecture in the previous section and how it reduces the cost and effort of the researchers and users, benchmarking a distributed protocol. We consider the parameters that define the characteristics of a software depending on operational, transitional and maintenance grounds that include software development cost, extensibility, reusability, and testability.

### 3.5.1 Reduction in Software Development Cost

Cost is measured in terms of time required to develop a dependability and performance benchmark which is primarily determined by the coding effort. Considering the generic architecture proposed in the previous section (see Section 3.3), we consider the cost of development and reuse of the components. There are some parts of the benchmark which are system dependent like faultload and workload. For any new fault type, it requires changes in the faultload, fault injector and corresponding change in the system under test. Similarly, for workload. Cost of development, thus, considers the effort required to introduce and test different types of faults. In case of BFT-Bench, the only effort required (in terms of implementation) was implementing additional methods/messages to recognize injection of faults at replica and client side.

### 3.5.2 Extensibility

This characteristic measures, how easy it is to add a new functionality; a good software design is likely to provide high extensibility. Higher the extensibility, lesser the time it requires to add a new functionality as changes in the current code are minimum. This means that system is adaptable and modular enough to incorporate new changes without leading to regressions. In the generic architecture for a benchmark, it is not difficult to integrate new fault models as it will only require some modifications in the code of the target system to allow fault injections. Faultload and fault injector are flexible and adaptable and require minimal code changes, if required. It also depends highly on modularity of the design.

### 3.5.3 Reusability

This characteristic measures how different units of logic of a benchmark are separated so that when a change is necessary, it is only performed in one place in the current code. Higher the modularity, easier is to integrate more features. Benchmark architecture design is quite modular where the functionality of each component does not coincides with each other and they demonstrate low coupling. Components are reusable with minimal code change effort.

### 3.5.4 Testability

This characteristic measures how much automation and coverage, the code tests have. This not only covers unit testing, but integration testing, and injection of different faults and workloads in the system. It gives more liberty to the benchmark users to perform analysis and measurements under different scenarios and extend the evaluation to analysis of other QoS metrics, such as cost, security, and scalability.

## 3.6 Summary

In this chapter, we introduced the generic architecture for building performance and dependability benchmarks for distributed systems. We presented the essential building blocks (components or modules) for developing a new benchmark. They consist of (i) module for defining and injecting various workloads and fault models, (ii) component to automatically deploy system under test and launch experiments, and (iii) module to measure and analyze performance and dependability attributes (adapted to incorporate other QoS metrics). We believe, for any benchmark (generic or specific), it is necessary for it to follow design guidelines and provide the following. First, it must provide end-users an easy way to define and inject various workloads and faultloads, encompassing different fault models and load conditions. This would also users to effectively analyze system's behavior under diverse settings. Second, the benchmark must provide mechanisms to empirically evaluate various QoS parameters with easy adaptability (to add more), for example, performance, dependability, security and cost. Finally, it must be easy to use, portable on a wide range of platforms, systems and cloud infrastructures.

This chapter also presents some details to use the generic architecture for building BFT-Bench (discussed at length in the subsequent chapter) to benchmark BFT protocols under various Byzantine fault models.

# BFT-Bench: Performance and Dependability Benchmarking Framework for BFT Protocols

---

## Contents

---

<b>4.1</b>	<b>Background . . . . .</b>	<b>88</b>
<b>4.2</b>	<b>Objectives of BFT-Bench . . . . .</b>	<b>89</b>
<b>4.3</b>	<b>Design Principles of BFT-Bench Framework . . . . .</b>	<b>90</b>
4.3.1	BFT Protocols in Consideration . . . . .	90
4.3.1.1	PBFT: A Practical BFT Protocol . . . . .	90
4.3.1.2	Chain: Performance Enhancement in Fault Free Conditions . . . . .	91
4.3.1.3	RBFT: Minimizes Performance Degradation in Presence of Faults . . . . .	92
4.3.2	Fault Types in Consideration . . . . .	93
4.3.2.1	Replica Crash . . . . .	93
4.3.2.2	Message Delay . . . . .	94
4.3.2.3	Network Flooding . . . . .	94
4.3.2.4	System Overloading . . . . .	94
<b>4.4</b>	<b>Overview of BFT-Bench . . . . .</b>	<b>94</b>
4.4.1	Cluster Setup . . . . .	95
4.4.2	BFT Protocol Selection . . . . .	96
4.4.3	Faultload . . . . .	96
4.4.3.1	Fault Trigger Time . . . . .	96
4.4.3.2	Fault Type . . . . .	96
4.4.3.3	Fault Parameters . . . . .	96
4.4.4	Workload . . . . .	98
4.4.4.1	Concurrent Clients . . . . .	98

4.4.4.2	Message Size . . . . .	98
4.4.5	Fault Injection . . . . .	98
4.4.5.1	Injection of Replica Crash . . . . .	100
4.4.5.2	Injection of Message Delay . . . . .	100
4.4.5.3	Injection of Network Flooding . . . . .	101
4.4.5.4	Injection of System Overloading . . . . .	101
4.4.6	Performance and Dependability Analysis in BFT-Bench	102
4.4.6.1	Performance . . . . .	103
4.4.6.2	Dependability . . . . .	103
4.4.6.3	Cost . . . . .	103
4.4.6.4	Network Level Statistics . . . . .	104
<b>4.5</b>	<b>Automatic Deployment of Experiments . . . . .</b>	<b>104</b>
<b>4.6</b>	<b>Using BFT-Bench . . . . .</b>	<b>104</b>
<b>4.7</b>	<b>Portability of BFT-Bench . . . . .</b>	<b>105</b>
4.7.1	Portability of Workload Injection . . . . .	105
4.7.2	Portability of Fault Injection . . . . .	106
4.7.3	Portability of Performance and Dependability Analysis	106
4.7.4	Portability of Automatic Experiment Deployer . . . . .	106
<b>4.8</b>	<b>Summary . . . . .</b>	<b>106</b>

---

Recently BFT protocols have gained popularity due to frequent occurrences of Byzantine faults causing high performance degradation and unavailability of cloud systems. Constant enhancements have been proposed to state-of-the-art BFT protocols with focuses on improving the performance in fault-free scenarios (see Section 2.4.2) while some minimizing performance degradation in faulty cases (see Section 2.4.3).

Considerably less progress has been made to analyze the robustness and effectiveness of BFT protocols under real world conditions where nodes can demonstrate arbitrary/malicious behaviors. Our comprehensive study in previous chapters motivated us to design BFT-Bench framework, the first performance and dependability benchmark tool for evaluating BFT protocols under identical settings.

The contributions of this chapter are as follows:

- We present BFT-Bench, a framework to evaluate the performance and dependability metrics of BFT protocols in the face to real world faulty and non-faulty scenarios to enable the researchers and developers analyze their effectiveness and robustness in practice.
- We describe the design principles of BFT-Bench framework which includes automatic deployment of experiments on cloud clusters (public, private or a local configuration), selection of BFT protocol under assessment, definition and injection of faultloads and workloads, and monitoring of QoS metrics using generated output and log files.
- Lastly, we discuss the portability and usability of BFT-Bench to incorporate new fault models and other BFT protocols. Although the current BFT-Bench prototype includes integration of three BFT protocols, but it can be easily extended to incorporate other BFT implementations. Section 4.7 describes the portability aspects of BFT-Bench framework.

This chapter is organized as follows. Section 4.1 provides the brief outline of the BFT systems. Section 4.2 discusses objectives of BFT-Bench. Section 4.3 provides the design principles describing BFT protocols and fault types in consideration for performing comparative analysis using BFT-Bench framework. Section 4.4 presents the overview of BFT-Bench. Sections 4.5 and 4.6 respectively describe the automatic deployment of experiments and how end-users can use BFT-Bench framework for evaluating their BFT protocols and conduct measurement analysis. Section 4.7 demonstrates the portability of BFT-Bench which makes it easier to integrate other BFT protocols. Finally, Section 4.8 presents the summary of the contributions of this chapter.



## 4.1 Background

Distributed systems must handle arbitrary malfunctioning, deceptive computer components misleading other parts of the system with incorrect information, hardware failures, and operator mistakes. These malicious behaviors are termed as Byzantine (arbitrary) faults where a server can produce an incorrect response to a client's request, sends corrupt messages, disobeys the ordering of the requests, etc. BFT protocols are used to maintain a resiliency against such Byzantine behaviors. A BFT protocol manages the communication among the nodes and clients in the presence of partial synchrony [51] to identify a malicious behavior in the system. However, the BFT protocols often suffers from multiple orders of magnitude reductions in performance and systems become unavailable for a long duration in the presence of Byzantine behaviors, violating the SLA contracts. This makes it difficult for the system programmers to trust and adopt BFT protocols for building a fault tolerant cloud system without worrying about the complex nature of BFT protocols.

System developers intend to know the answers to many questions before trusting and incorporating a BFT protocol in their underlying systems. They demand responses on effectiveness, robustness, performance and dependability aspects of the protocols in presence of Byzantine behaviors. They need to know what kind of faults a system can tolerate? Can it really handle malicious faults, including common faults like crash and complex faults like intentional message delay by a node? What is the maximum number of faults a system can handle before breaking down? What is the maximum workload a system can endure? And is it possible to test different BFT implementations under same settings and perform a comparative analysis to select the most efficient and most reliable for the required cloud system. Therefore, it is not only important but necessary to develop a generalized, high-level, easy to use, performance and dependability benchmarking framework for analyzing BFT implementations with mechanisms to define and inject various workloads and faultloads.

All the BFT protocols consist of a primary/head node and backup replicas. Users of cloud clusters (using BFT protocols) submit their jobs to the primary node which is responsible for ordering/scheduling the client's requests. The other replicas undergo agreement (by message exchanges among themselves) and commit phases before executing the ordered requests. By default, all the requests are scheduled in FIFO order. Under fault free conditions, all the replicas execute the incoming requests and share their state with other replicas to maintain consistency. Once a request is executed, replicas send back the responses to the client. Client upon receiving consistent matching

responses from all the replicas, client commits the request. In case of faulty scenarios, replicas go through a *view change* to maintain a correct primary for correct ordering, consistency and *Liveness* of the system. Replicas and clients maintain history logs and share them periodically for failure recoveries.

## 4.2 Objectives of BFT-Bench

In this section we define the objectives of BFT-Bench framework.

- **QoS Metrics Analysis:** BFT-Bench assesses and evaluates performance and dependability aspects of the implementations of BFT protocols. We consider latency (or response time) and throughput for measuring the performance, reliability and availability for evaluating dependability. We determine the cost of using the BFT-SMR techniques by cloud services in terms of the number of replicas used by a BFT protocol for handling Byzantine faults. We also consider low-level statistics like network bandwidth and CPU utilization, the number of re-transmissions of a request, failed vs successful jobs, etc.
- **Heterogeneous Parameters for Evaluation:** BFT-Bench characterizes various types of loads such as faultload and workload for evaluating different BFT implementations. Precisely, faultload is defined by *type* of fault to be injected, *time* at which the fault will be triggered and *location* (replica in the cluster) where the fault will be induced. For each type of fault, there are few related parameters which complete a faultload. Similarly, workload is characterized by the number of concurrent clients (i.e., the users of the system) and size of the request message send by each client.
- **Usability:** BFT-Bench is an easy to use framework where configuration, deployment of experiments and analysis of results are automated. It is independent of any cloud infrastructure; thus, can be deployed on public, private, hybrid or any local cloud configurations. Statistics monitoring of experimental evaluation can easily be comprehended with the smooth generation of graphs and charts.

The intention of BFT-Bench framework is to enable researchers and developers working on cloud systems to understand and evaluate BFT protocols by injecting various faulty scenarios commonly occurring (affecting) in a real world cloud environment under different and varying workloads.

## 4.3 Design Principles of BFT-Bench Framework

BFT-Bench is intended to be an open framework, that includes BFT protocol prototypes, and that may include new BFT protocols. This chapter defines the system model of BFT-Bench framework where we consider different types of BFT protocols for evaluation and different types of faults injected in the system to emulate the real world settings. We describe the assumptions made at the system level for nodes and the network, and Byzantine failure model considered for a Byzantine environment. This section provides a brief overview of the protocols considered for performance evaluation and finally we introduce different types of faulty behaviors used in BFT-Bench.

### 4.3.1 BFT Protocols in Consideration

In the rest of this manuscript, we consider the following state-of-the-art BFT protocols: (i) PBFT for being the first practical BFT protocol [30], (ii) Chain for its performance efficiency in fault-free conditions [56], and (iii) RBFT as an instance of robust protocols that minimizes performance degradation in presence of failures [18]. For an effective comparative analysis of these BFT protocols, we consider the same system assumptions for all. We assume a distributed system where  $N$  servers/replicas<sup>1</sup> are connected in a specific network topology, for example, PBFT [30] and RBFT [18] are fully connected whereas Chain [104] connects the replicas in a chain-like pattern (a replica followed by another). The system is defined as a set of clients and servers where a client sends a request in a *closed loop*, i.e., a client has to wait for the response of a request before sending a new request. We assume a finite client population where any number of them may be faulty, and at most  $\frac{N-1}{3}$  replicas can behave maliciously. The links between nodes are asynchronous and unreliable with synchronous intervals during which messages are delivered within a known bounded delay. We do, nevertheless, assume that if a node keeps re-transmitting a message, the message will eventually be received (partial synchrony [51]). However, the *Liveness* property can only be ensured during periods of synchrony [53].

#### 4.3.1.1 PBFT: A Practical BFT Protocol

PBFT [30] is considered the baseline of BFT protocols and its communication pattern is used by many protocols such as Aardvark [36], RBFT [18], Spin-

<sup>1</sup>We will use the terms server, node or replica alternatively throughout this paper.

ning [105], Prime [15] and BFTSMaRt [24]. The communication pattern of PBFT is presented in Figure 2.4. The protocol ensures *Liveness* and *Safety* properties as long as there are no more than  $f$  faulty nodes. In order to maintain *Safety*, the primary must first assign a sequence number to each incoming request during the pre-prepare step. The two following steps, prepare and commit, are dedicated to the exchange and validation of the sequence numbers proposed by the primary. *Liveness* is ensured by detecting whether the primary performs the correct ordering or not, within a dedicated time. If *Liveness* becomes challenged, a *view change* is triggered accordingly, implying that a new replica will replace the current primary. This *view change* mechanism is essential for all BFT protocols relying on a primary for the ordering, but could imply some conceptual variations depending on the underlying protocols [74].

Additional detail 1: In PBFT, the communication pattern is not affected by the actual presence or absence of faulty nodes.

Additional detail 2: Even if PBFT is strong enough to ensure *Liveness* and *Safety* in the presence of attacks, some Byzantine behaviors can substantially decrease its performance. For instance the pre-prepare delay attack, which has been the focus of several protocol proposals [18, 36, 105].

#### 4.3.1.2 Chain: Performance Enhancement in Fault Free Conditions

Chain [56] is designed to handle a high load of requests. It is dedicated towards improving throughput in fault-free settings, while maintaining the ability to detect inconsistencies. As the name suggests, it has a chain-like communication pattern that greatly benefits from the batch optimization (multiple messages in one batch to avoid expensive authentication computations for every single message) because unlike PBFT, the head of the chain does not send the ordered requests to all the replicas but only to its successor, and does not need to be authenticated by everyone. This greatly reduces the number of authentications at the bottleneck replica. Chain is most efficient when it is completely fed, i.e., when the network link between any 2 servers is fully loaded [56]. This can be done by employing a large set of clients sending a large number of requests. Thus, Chain gracefully handles a high load of requests. Chain improves the throughput in fault-free settings among other protocols from Group 1 (see Section 2.4.2), while maintaining the ability to detect inconsistencies. But Chain is unable to ensure Byzantine fault tolerance by itself, and must rely on a protocol switching mechanism when subject to failures. The switching mechanism is activated when Chain cannot continue to make a progress upon detecting Byzantine failures. Although the switching

mechanism has been proposed theoretically, its prototype lacks switching to PBFT in the presence of faults.

#### 4.3.1.3 RBFT: Minimizes Performance Degradation in Presence of Faults

Even though PBFT is theoretically strong enough to ensure *Liveness* and *Safety* in the presence of attacks, some Byzantine behaviors substantially decrease its performance when evaluated practically [18]. For instance, delay attack where a primary intentionally delays the sending of pre-prepare message to other replicas to increase the latency of the system, has been the focus of several protocol proposals including RBFT [18, 36, 105]. Redundant-BFT (RBFT) [18] strengthens the architecture of PBFT and also incorporates fault adaptive mechanisms to deal with certain faulty behaviors. Figure 2.15 illustrates the communication pattern of the protocol. Like other protocols, RBFT requires  $3f + 1$  replicas and relies on a primary replica for ordering the client's requests. RBFT runs  $f + 1$  multiple instances of the same protocol in parallel, but the requests are executed only by one of the instances called master instance while other  $f$  instances are called backup instances. Each backup instance has its own primary which orders the incoming requests in order to monitor the difference of throughput between the master instance and itself. If the performance at master and backup instances differ by a definite threshold<sup>2</sup>  $T$  at less than  $2f + 1$  replicas, the primary replica at master instance is considered faulty and a *view change* is triggered, where a new primary is elected at every instance. RBFT further implements few fault adaptive defensive pathways for handling faulty behaviors of both clients and replicas. RBFT exploits the multicore architectures of today's world machines to run multiple instances in parallel for efficiency and robustness.

We selected RBFT as it shows improvements over Aardvark, Prime and Spinning. Although Aardvark was the first to test BFT implementations under faulty scenarios, but it considered fewer cases than RBFT. Spinning and Prime were also evaluated under some fault types (see Table 2.3). RBFT the most robust protocol, not only evaluated the prototype under maximum fault scenarios, but also demonstrated that the performance degrades only up to 3% in the presence of faults.

---

<sup>2</sup>The value of threshold depends on the ratio of observed throughput between fault-free conditions to the throughput observed under attack

### 4.3.2 Fault Types in Consideration

We assume a Byzantine failure model, in which any node (replica) or client can behave arbitrarily. The network itself may fail to deliver messages, delay them, duplicate them, delay them out of order, or even corrupt them. A malicious node can perform these behaviors intentionally. We assume a strong adversary that can manipulate and coordinate these malicious nodes to compromise the replicated service. However, we do assume that this adversary is computationally bounded and unable to break cryptographic techniques like collision-resistant hashing, digests, Message Authentication Codes (MACs), encryptions and digital signatures. Apart from the above assumptions for Byzantine model, we define different types of faults injected for evaluating the BFT prototypes using the BFT-Bench framework.

The performance of the prototypes for these state-of-the-art protocols (see Section 4.3.1) is greatly affected by the faulty behaviors [18, 30, 36]. Therefore, it becomes extremely critical to analyze the performance and resiliency of BFT implementations in real world faulty settings. We characterize four types of faults from the perspective of software and hardware components, out of many feasible faults (few were listed in Section 2.3.2). Some intentional malicious behaviors like message delay, network flooding make it challenging to identify a faulty network from a misbehaving server. And this often leads to an incorrect diagnosis of the fault and becomes critical to preserve system's performance, integrity, and availability in times of such Byzantine faults. To imitate these Byzantine behaviors, replicas are made to drift from the correct protocol expectations. Hardware faults include unpredictable events like power outage which eventually results in a crash. Software faults, for example, can be related to the addition of a delay before forwarding messages, or flooding a system with corrupted information.

#### 4.3.2.1 Replica Crash

Crash of a server is a common performance failure that can happen in a system. Upon a crash, the server stops completely and do not participates in any further communication with the clients or the servers. Most of the industries like Salesforce, Amazon, Oracle, etc., rely on Paxos [77, 78] for handling crash. But during the occurrence of Byzantine faults, they face challenges of disrupted availability. BFT protocols consider crash as yet another Byzantine fault.

#### 4.3.2.2 Message Delay

Delaying the sending of messages benefits from the difficulty to distinguish a faulty replica from a slow network. When a replica starts to delay messages, it slows down all future operations depending on these messages. As described in Section 2.4, most of the BFT protocols ensure *Safety* property by reaching an agreement on the total order of execution of the requests. If the messages containing these information are delayed, then the whole protocol is delayed, leading to performance degradation. This Byzantine behavior is especially critical when it occurs at the primary replica.

#### 4.3.2.3 Network Flooding

Network flooding is meant to overload both the network and the computational resources with malicious messages which cannot be said invalid until verified. This message verification consumes a lot of computational cycles and prevents the resources from focusing on the correct messages.

#### 4.3.2.4 System Overloading

Overloading the system with a large number of requests sent by many concurrent clients can prove to be catastrophic and can degrade the performance to a large extent. Although none of the servers behave maliciously in this attack, but continuous increase in the number of clients can eventually deteriorate the performance or lead to system failure.

## 4.4 Overview of BFT-Bench

We present BFT-Bench, a novel framework that allows empirical evaluation and comparison of state-of-the-art and new BFT systems. Figure 4.1 described the major components of BFT-Bench, such as: (i) *Cluster setup* component allows users to prepare the cluster for launching BFT protocols and perform experiments. (ii) *BFT protocol selector* launches the BFT implementation on the required number of nodes. (iii) *Fault Injection* that triggers the fault scenarios in the underlying BFT system; (v) *Load Injection* that injects the number of concurrent clients accessing the underlying system; and (v) *Statistics Monitoring* that collects monitoring information, and reports performance and dependability statistics of the system. BFT-Bench is a testbed to evaluate performance and dependability of BFT systems. In this section we define the building blocks of BFT-Bench framework in details.

Conceptually, cluster runs cloud user's services where each node in the cluster is running a BFT protocol for maintaining *Liveness* and *Safety* of the system. Users of these cloud services issue a large number of requests to be executed in order. BFT-Bench emulates multiple clients concurrently accessing the cluster. BFT-Bench can also imitate commonly occurring real world faults which have been catastrophic in some situations, for example [49].

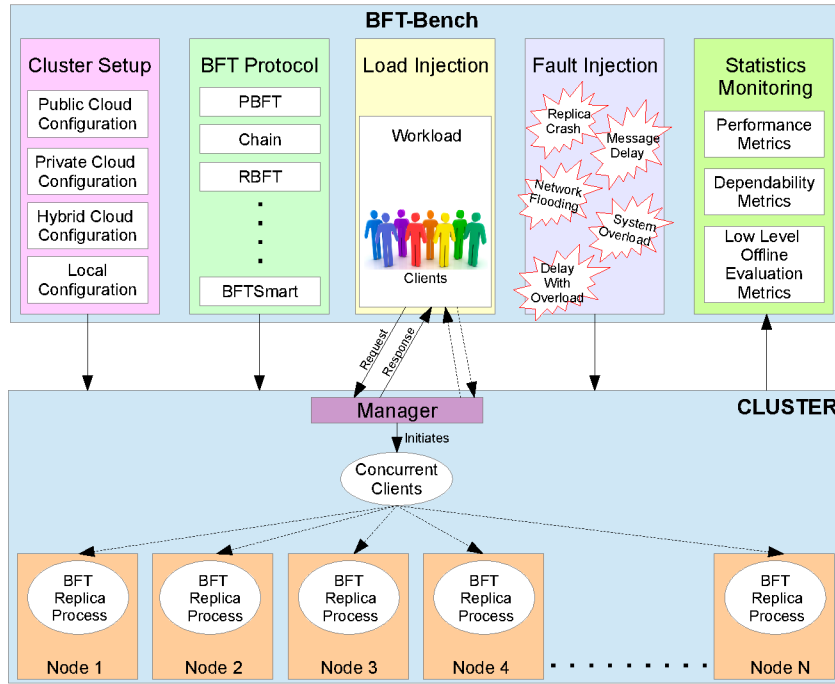


Figure 4.1: Overview of BFT-Bench Framework

#### 4.4.1 Cluster Setup

BFT-Bench framework can be used in any existing cloud configurations (see Section 1.1) varying from public clouds to local distributed systems. BFT-Bench allows testers to deploy BFT protocols and run experiments irrespective of the underlying cloud settings. Installation of BFT systems and experimental analysis remain the same and does not change with different cloud configurations. Cluster setup component allows the users to prepare cluster for launching BFT protocols and perform experiments. It reads the configuration file which mentions the number of required nodes ( $N$ ) and a total number of handled faults ( $f$ ).



## 4.4.2 BFT Protocol Selection

BFT protocol selector launches the prototype of the BFT protocol considered for evaluation on the required number of nodes (nodes designated in the previous step). For evaluating the working of BFT-Bench, we consider three BFT protocols, PBFT, Chain and RBFT (defined previously in this chapter) but can be easily extended and adopted for other BFT prototypes. In case of node failures or crash faults, it relaunches the prototypes once a new experiment commences. Protocol selector takes care of the protocol switching and its installation. BFT-Bench framework launches only one protocol at a time.

## 4.4.3 Faultload

Faults can occur accidentally or can be induced intentionally. To test the practical implementations of BFT protocols in faulty environments, we deliberately trigger our faults at a fixed time. We thus do not consider accidental failures. Users of BFT-Bench framework can generate synthetic faultloads involving different faulty behaviors. Figure 4.2 presents the structures of faultloads for different fault types considered individually or in combinations (see Section 4.3.2) where each line (line numbers) corresponds to one type of fault. Each faultload contains various information which we describe below.

### 4.4.3.1 Fault Trigger Time

The *fault trigger time* contains the time at which the fault must be triggered. It is the time to be elapsed from the time the experiment is launched.

### 4.4.3.2 Fault Type

Byzantine faults encompass numerous faulty behaviors. Nevertheless, we keep our focus on the four faults (mentioned in Section 4.3.2), where each one is designated with a specific keywords *replica crash*, *message delay*, *network flooding* and, *system overloading*, respectively. We also consider the combination of these faults, such as message delay with system overloading. This would mean injection of message delay fault when the number of concurrent clients in the system is increasing.

### 4.4.3.3 Fault Parameters

Different faults may require additional fault parameters at the time of fault injection. According to the type of fault to be injected, fault parameters

```

// faultload

// Replica Crash
Line 1: <fault trigger time, fault type, {fault loc}>

//Message Delay
Line 2: <fault trigger time, fault type, {fault loc, delay time, message type}>

//Network Flooding
Line 3: <fault trigger time, fault type, {fault loc, message size}>

//System Overloading
Line 4: <fault trigger time1, fault type, {#clients1}>
      <fault trigger time2, fault type, {#clients2}>
      .
      .
      <fault trigger timeN, fault type, {#clientsN}>

//Combination of 2 types of faults
Line 5: <fault trigger time1, fault type, {#clients1}>
      <fault trigger time2, fault type, {fault loc, delay time, message type}>
      <fault trigger time3, fault type, {#clients3}>
      .
      .
      <fault trigger timeN, fault type, {#clientsN}>

```

Figure 4.2: Faultloads for different types of faults considered in Section 4.3.2

might vary. For *replica crash*, *message delay* & *network flooding*, the location of the fault must be specified, whereas in *system overloading*, the location is irrelevant since no replica acts faulty. For *network flooding*, the size of the corrupted messages is an important factor, as larger the size of the messages, larger will be the time consumed during cryptographic operations. For *message delay*, the value of the delay introduced before sending a message by the faulty replica, must be specified.

These fault parameters have a huge impact on the performance and dependability of the whole system. We now briefly define the fault parameters we considered for injecting the four different types of faults.

1. **fault loc**: It defines the replica at which the fault will be triggered. This parameter is required for *replica crash* to know which replica will crash, for *message delay* to know which replica will start delaying the messages and for *network flooding* to know which replica will send corrupt messages in an effort to degrade the performance by increasing network and CPU utilization at correct replicas.
2. **delay time**: It defines the time by which a replica delays the sending of messages to the other replicas. This parameter is required by *message delay*.

3. **message type**: It defines the type of the message that will be delayed by *delay time* before being sent to the other replicas. This parameter is used by *message delay*.
4. **message size**: It is the size of the invalid/corrupt messages send by a faulty replica to correct/non-faulty replicas when causing network flooding. This parameter is used by *network flooding* fault type. Larger the size of the message, higher is the impact on performance.
5. **#clients**: It defines the number of concurrent clients sending their requests to the servers. We define this parameter only in case of *system overloading*.

#### 4.4.4 Workload

BFT-Bench allows users to determine the impact of varying workload on performance and dependability aspects in the presence or absence of a Byzantine fault. The workload is characterized by the number concurrent clients issuing requests to the cloud service and also the size of each request message. Following are the parameters of a workload.

##### 4.4.4.1 Concurrent Clients

It is the number of clients sending requests to the BFT system. Client requests are executed in FIFO order in a closed loop, where a client submits a request, waits for the request to get processed and receives a response, before sending another request.

##### 4.4.4.2 Message Size

It is the size of the client request/response messages exchanged with the BFT system. It is an important parameter as large size messages affect BFT system performance, due to time-consuming cryptographic operations executed by BFT protocols. BFT-Bench includes a client emulator implementing multi-client behavior, where each client process sends requests to the underlying BFT system, and receives corresponding responses.

#### 4.4.5 Fault Injection

The overall architecture of fault injection in BFT-Bench is described in Figure 4.3. The cluster runs BFT protocols on  $N + 1$  nodes, where  $N = 3f + 1$  nodes

are dedicated to replicas, and one node hosts concurrent clients. We run the fault injector in another node in the same cluster. Faultload injector uses faultload (see Section 4.3.2) to determine *which* type of fault is to be triggered from *fault type*, at *what* time this fault will be injected from *fault trigger time*, and other required *fault parameters* (Section 4.4.3.3). The fault injector runs a daemon that communicates directly with the replicas to trigger faults. For instance, in case of *replica crash*, the daemon waits until the *fault trigger time* is reached, then calls the fault injector of *fault type* which interacts with the replica defined in *fault location* to trigger the fault. Practically, once a fault is injected, it persists until the end of the experiment. In the absence of a faultload, the fault injector lets the system run in fault-free settings. Some modifications are enforced in the BFT implementations to enable them to interact with the fault injector daemon.

In the following, we describe for each fault type introduced in Section 4.3.2, how BFT-Bench implements its injection in a BFT system.

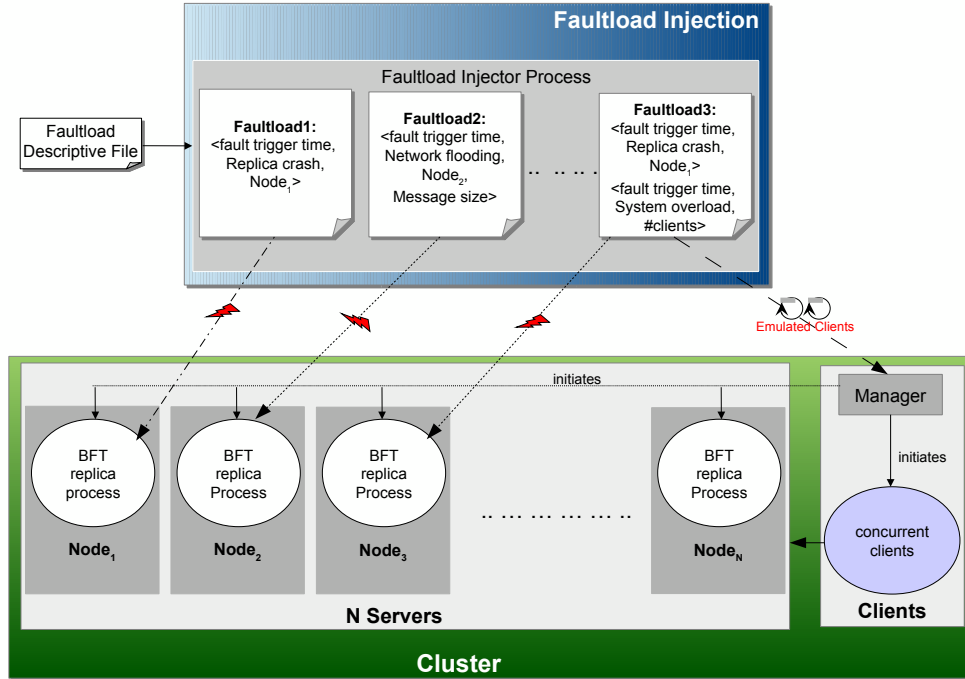


Figure 4.3: Architecture of Faultload Injection

#### 4.4.5.1 Injection of Replica Crash

The simplest way to implement a replica crash is to completely shut down the whole machine where it is hosted. However, since each replica is isolated on a dedicated node and to avoid repetitive reboots, we simply shutdown the replica process on the targeted node. Practically, when *fault trigger time* is reached, the fault injector daemon remotely connects to the targeted node and kills the replica process. After being killed, the replica process is not restarted until the end of the experiment and no new replica is added to the system. This fault is implementation independent, thus requires no changes in the prototypes.

We consider the impact of crashing the primary versus crashing of a non-primary replica for RBFT and PBFT. Since the primary replica is responsible for ordering of incoming requests, crash of it leads to expensive *view change* protocol, degrading the overall performance. Whereas, in case of a non-primary crash, protocol continues as replicas need only  $2f + 1$  matching responses. In case of Chain, location of the fault does not matter as the crash of any replica will break the pipeline and expensive switching protocol will be enabled.

#### 4.4.5.2 Injection of Message Delay

It is an implementation dependent fault and needs changes in the BFT implementations. All the considered BFT protocols rely on a common code base (RBFT and Chain use PBFT as an underlining protocol), and the functions for sending and receiving messages follow the same pattern in all prototypes. Therefore, we introduced a few messages to be recognized in addition to the ones currently used, to allow fault injector daemon inject faults. Practically, a delay message is now recognized by all the prototypes, and when a replica receives it from a fault injector daemon, it triggers the following Byzantine behavior: instead of sending messages according to the protocol specifications, the replica process sleeps during the given value of *delay time* provided in the fault parameters before resuming to send any messages to other replicas. Our implementations can delay a subset of messages, to trigger this fault. Moreover, once a replica becomes faulty, it keeps delaying messages until the end of the experiment.

From the communication pattern of the considered protocols, we observe that the impact of delaying the pre-prepare message is maximum in comparison to delay of any other message type. But this holds true only for RBFT and PBFT. All the replicas upon receiving prepare messages, wait for the corresponding pre-prepare message and start a timer. Upon failing to receive

the message before the expiration of a timer, replicas start to exchange messages to identify the faulty primary and trigger a *view change*. View change and election of new primary is time-consuming and degrades the performance, whereas delaying of other messages does not trigger view change in RBFT and PBFT. For Chain, there is only a single message type and all the nodes play the same role (except the head, which orders the requests as well). Also, there is no timer mechanism at replicas in Chain as replicas are not anticipating for messages from their predecessor.

#### 4.4.5.3 Injection of Network Flooding

Network flooding is a common denial-of-service attack that could be performed both by clients and servers. We consider flooding by a replica since in practice the replicas are often co-located on the same cluster, making it easier to monopolize network links. Just like message delay, this fault required to introduce additional messages. Practically, a flooding message is now recognized by the prototypes, and when received from the fault injector daemon, the replica triggers the following Byzantine behavior: the faulty replica enters an infinite loop, where it continuously transmits corrupted messages of a chosen size (*message size*) to other replicas until the end of the experiment. We keep authenticating the messages with the faulty replica's public key in order to exhaust not only the network but the computational resources of the other replicas (the messages cannot be declared invalid until verified).

All the replicas with network flooding fault (primary or non-primary) will impact the performance in the same way. In case of primary, it will send the correct messages (prepare message with ordering, pre-prepare message, etc.) along with corrupted messages. Similarly other non-primary replicas will send malicious messages with prepare and commit messages.

#### 4.4.5.4 Injection of System Overloading

Clients are not instantiated the same way in all the prototypes. In Chain and RBFT, a manager is responsible for initiating and managing the clients, while in PBFT, the clients are independent and can be deployed on multiple nodes. As mentioned earlier, the workload injector can vary *#clients* after the experiment has commenced. To do so, the fault injector daemon remotely connects to the node in charge of hosting concurrent clients and starts additional client processes.

#### 4.4.6 Performance and Dependability Analysis in BFT-Bench

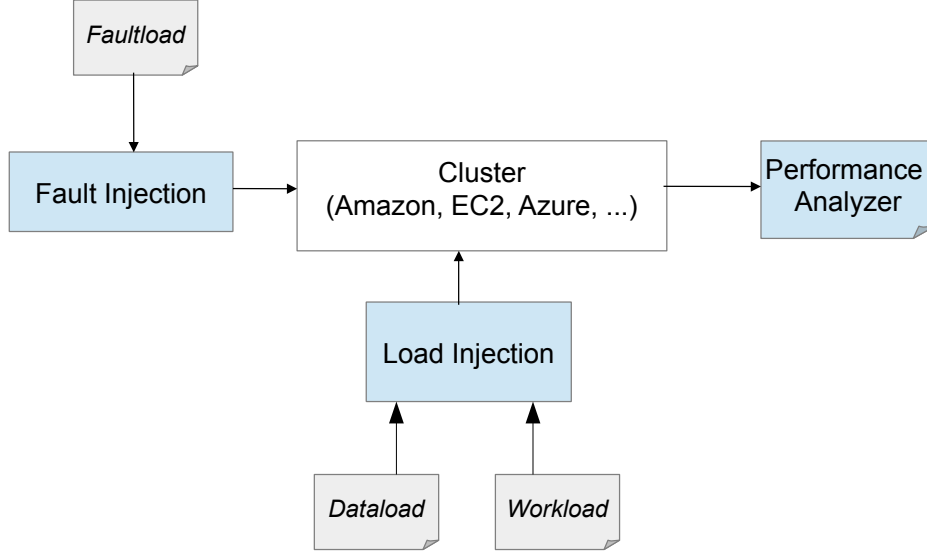


Figure 4.4: Performance and Dependability Benchmarking Architecture of BFT-Bench

BFT-Bench framework is used to evaluate and compare dependability and performance metrics of BFT protocols. Figure 4.4 presents the high level architecture of the benchmark. Following steps are performed to analyze the performance and dependability for various BFT protocols: (1) build a faultload with specifications for injecting a fault (i.e., a specific Byzantine fault scenario), (2) build a workload for setting up the test environment, (3) conduct experiments based on faultload and workload, and (4) collect statistics for measuring performance and dependability of the BFT protocol under test. Upon experimental execution, BFT-Bench produces statistics for performance metrics, namely *throughput* and *latency*. They have been the two main performance parameters considered when evaluating BFT protocols, both experimentally and theoretically. We focus on the experimental evaluation of the prototypes, thus, we do not take into account any theoretical analysis. BFT-Bench also considers the measurement of dependability metrics namely *availability* and *reliability*. Along with measurements of performance and dependability aspects, BFT-Bench also computes network level and low-level statistics, considering network bandwidth usage, CPU utilization, the number of successful vs failed requests, the number of re-transmissions of each request, etc.

#### 4.4.6.1 Performance

1. **Throughput.** It is the number of client requests handled by the system per unit of time. It can be evaluated both from the client or the replicas side of the protocol. Nevertheless, in order to avoid unnecessary operations on the replicas side, we choose to perform the evaluation of throughput from the client side of the protocol. Just like latency, the evaluation of throughput can be performed using timestamps, thus, we do not need to collect any additional information. We evaluate throughput in the performance analyzer by summing the number of response timestamps collected on the client processes per unit of time.
2. **Latency or Response Time.** It is the time elapsed from the moment a client submits a request until the complete response is received by this client. To evaluate latency, we rely on the timestamps of these two events. We measure latency from the client side of the protocol by collecting and bringing these timestamp values to the performance analyzer. Then, performance analyzer evaluates the latency by computing the difference between the response timestamps and the request timestamps, for all the requests send by all the clients per unit of time.

#### 4.4.6.2 Dependability

1. **Availability.** It is measured in terms of time when the service is available, i.e., the service is responding. It is the ratio of the time the service was returning responses (correct or incorrect) to the total time the service was meant to run. It is usually measured over a period of time, in terms of days, months or years.
2. **Reliability** is measured as the ratio of successful client requests over the total number of requests send by the clients over a period of time.

Theoretically, all BFT protocols should be 100% reliable and available. The experimental evaluation (see Chapter 5) describes how well they perform in practice.

#### 4.4.6.3 Cost

1. **Replica Cost** is measured in terms of the total number of replicas required to handle the  $f$  simultaneous Byzantine faults. As we defined before, that each BFT protocol requires different number of servers, it becomes an important parameter when it comes to the cost of using BFT systems in real world applications.



#### 4.4.6.4 Network Level Statistics

1. **Network Bandwidth Usage** is defined in terms of the size of data send and received by any server per second. Network bandwidth is used as a synonym for data transfer rate, i.e., the amount of data that can be carried from one point to another in a given time period (usually a second). Network bandwidth is usually expressed in bits per second.
2. **CPU Utilization** is the amount of work that a computer system performs in processing the computations on available resources. It is the amount of work handled by a CPU. Actual CPU utilization varies depending on the amount and type of computing tasks. Certain tasks require heavy CPU time, while others require less because of non-CPU resource requirements. BFT-Bench calculates CPU usage over a small interval of time, considering the time spent by the CPU over real time.

BFT-Bench measures some offline metrics along with above mentioned online metrics. BFT-Bench provides statistics related to cost of computations performed by the cluster, status of each request send by the clients (success or failure), the total number of re-transmissions of each request, etc.

## 4.5 Automatic Deployment of Experiments

BFT-Bench framework allows testers to automatically deploy the prototypes of BFT protocols, run extensive experiments under various Byzantine behaviors on a cluster in any cloud infrastructure. The cloud infrastructure and the size of cluster (number of nodes required by a BFT protocol to run the experiments) are two main configuration parameters of BFT-Bench. The framework acquires on-demand resources maintained by the public or private cloud providers like Amazon EC2, Azure, etc, where one node is dedicated for running load and fault injectors, and the rest of the nodes are used for deploying test environment (installation of BFT prototypes). Once the cluster is set up, BFT-Bench automatically runs the experiment scripts for injecting different faults at various times while monitoring and reporting the execution traces of each experiment for performance and dependability benchmarking. Resources (nodes in the cluster) are automatically released once BFT-Bench terminates.

## 4.6 Using BFT-Bench

BFT-Bench user defines faultload and workload; and then BFT-Bench framework automatically deploys the experiments on the reserved resources of the

cloud infrastructure and injects the user defined workloads and faultloads thereafter. BFT-Bench creates and initializes the concurrent clients as per the workload. Clients send requests in FIFO order until the experiment terminates. If user defines a faultload, then according to the parameters, fault injector daemon triggers the specified fault during the experiment run. Once the experiment ends, the running processes are terminated on all servers. For each experiment, processes are restarted. Every experiment is independent and runs with different settings, producing distinct measurement reports for future evaluations. Every experiment may run a number of times to produce an average statistics and variance reports.

BFT-Bench is an easy to use framework with very few parameters used for its configuration. BFT-Bench configuration file mentions *warm up time*, *execution time*, number of nodes, etc. Once the experiment starts the *execution period*, workload and faultload can be injected at the defined times. Since we are evaluating different prototypes, we do not provide any default settings for BFT-Bench to avoid any complications.

## 4.7 Portability of BFT-Bench

The prototype of BFT-Bench measures performance and dependability aspects of few BFT protocols, namely PBFT, Chain and RBFT. However, it can be easily adopted for other BFT implementations like Ring, Aardvark, Prime, Spinning, MinBFT, OBFT, etc. Most of the prototype of BFT-Bench testbed is general and can be easily applied, except some parts like fault injection integration where it requires some changes to source code to incorporate triggering of a Byzantine behavior. In the following, we describe all the portable parts of the framework and how to extend them for other BFT protocols.

### 4.7.1 Portability of Workload Injection

Workload injection depends on the testers intention on how rigorously he/she wants to test the BFT system and what are his/her requirements or expectations. Workload injection is used to send the requests (transactions jobs) of varying sizes (it depends on the applications being used) of the imitating clients to the cloud cluster running the BFT protocols. Portability of workload injection is pretty simple and straightforward and does not require any change or new modifications unless there are some application specific loads.

### 4.7.2 Portability of Fault Injection

Fault injection depends on the type of fault to be triggered in the system. For some fault types, it is not necessary to modify the source code, but for some, it is required to add additional tasks to enable the node to trigger a fault upon receipt of a fault injection message. For instance, triggering replica crash is simple where we just use a system to kill a process running at a node at definite time (*fault trigger time*). This does not require any implementation level changes for triggering the crash of a node. On the other hand, faults like message delay where we delay sending of pre-prepare messages to other replicas. It requires integration of new code which informs the node to start delaying a particular message type. Thus, the injection of faults of these types needs to be adapted to any new BFT protocol being considered for evaluation. But for all faults, the framework should adapt to the actual names of the underlying processes.

### 4.7.3 Portability of Performance and Dependability Analysis

BFT-Bench measures performance and dependability aspects, but can be extended to incorporate other Quality of Service (QoS) metrics. BFT-Bench also measures network and CPU utilization at each server by using the files produced by *Server Activity Reporting* (sar commands for unix) for monitoring individual CPU stats, memory usage, network stats, I/O activities, etc. Low-level statistics such as the number of successful and failed requests, the number of times a request is re-transmitted, are extracted from the log files generated at all servers and clients. Pattern matching is used to extract the necessary information from the generated reports. Thus, BFT-Bench must adapt to these pattern recognition for every new QoS parameter.

### 4.7.4 Portability of Automatic Experiment Deployer

The BFT-Bench allows automatic deployment of experiments which can be effortlessly adapted to launch other BFT protocols, create workloads and faultloads, start and stop experiments and generate required QoS metrics.

## 4.8 Summary

In this chapter, we presented BFT-Bench, a comprehensive benchmark tool for evaluating the dependability and performance of BFT protocols under various

real-world faulty and non-faulty settings. BFT-Bench framework allows end-users to define and inject various workloads and faultloads, automatic deployment of experiments, and produce extensive dependability and performance statistics. It also generates few low-level statistics such as CPU and network utilization, the number of successful vs unsuccessful operations, etc. This chapter also discussed fault and load injectors, automatic deployment of experiments, and portability of BFT-Bench to incorporate other BFT protocols such as Aardvark, Spinning, Prime, Zyzzyva, OBFT, etc. [15, 36, 74, 98, 105]. We demonstrate the adaptability and ease of use of BFT-Bench for injecting Byzantine faults and also mechanisms to define new arbitrary behaviors. BFT-Bench successfully achieves the main objectives of any benchmark, i.e., is to provide practical ways for:

1. *Characterizing* the dependability of modules or a system.
2. *Easy detection* of errors or faults in design specifications and its practical implementations.
3. *Identification* of weaker segments and violations of protocol blueprints.
4. *Obtain* insights into the design decisions of the protocol developers.
5. Compare the dependability and performance of competitive solutions/protocols based on different measurable aspects under unified environmental settings.

This enables the benchmark users to give more attention and provide improvements of the implementations by fixing the identified bugs or issues to enhance the dependability levels (either by using software wrappers or adding fault tolerance mechanisms).

Furthermore, this work unfolds interesting perspectives in terms of investigating other faulty behaviors, heterogeneous workloads (computational or data-access intensive or both) and evaluation of other QoS metrics such as cost, security, scalability, etc. We believe, BFT-Bench aids researchers and practitioners to better analyze and evaluate various aspects of BFT protocols in a systematic manner.



# Experimental Evaluation

---

## Contents

---

<b>5.1</b>	<b>Experimental Setup . . . . .</b>	<b>110</b>
5.1.1	Hardware Settings . . . . .	110
5.1.2	Software Settings . . . . .	111
<b>5.2</b>	<b>Comparative Evaluation under Faulty Scenarios . . .</b>	<b>112</b>
5.2.1	Presence of Replica Crash . . . . .	114
5.2.1.1	Performance Analysis . . . . .	115
5.2.1.2	Dependability Analysis . . . . .	116
5.2.1.3	System and Network Level Statistics . . . . .	116
5.2.2	Presence of Message Delay . . . . .	118
5.2.2.1	Performance Analysis . . . . .	119
5.2.2.2	Dependability Analysis . . . . .	121
5.2.2.3	System and Network Level Statistics . . . . .	121
5.2.3	Presence of Network Flooding . . . . .	122
5.2.3.1	Performance Analysis . . . . .	123
5.2.3.2	Dependability Analysis . . . . .	124
5.2.3.3	System and Network Level Statistics . . . . .	124
5.2.4	Presence of System Overloading . . . . .	126
5.2.4.1	Performance Analysis . . . . .	127
5.2.4.2	Dependability Analysis . . . . .	128
5.2.4.3	System and Network Level Statistics . . . . .	129
5.2.5	Combination of Different Types of Faults . . . . .	130
5.2.5.1	Performance Analysis . . . . .	131
5.2.5.2	Dependability Analysis . . . . .	132
<b>5.3</b>	<b>Summary . . . . .</b>	<b>132</b>

---

In this chapter, we present a comparative analysis of the three BFT protocols considered for evaluation under fault free scenarios and when facing different fault behaviors, presented in Section 4.3.2. We specify our experimental settings used for performing the experiments followed by comparative graphical representations of performance and dependability metrics evaluations of each protocol under different faultloads and workloads. We also monitor some offline statistics like CPU utilization, network bandwidth usage, etc., for all the scenarios under scrutiny.

## 5.1 Experimental Setup

This section provides hardware and software settings used for performing experiments. Hardware setup comprises of cluster configurations and network utilities while the software setting is regarding the installation of BFT protocols, assigning values to parameters for initiating and launching the experiments.

### 5.1.1 Hardware Settings

All our experiments were conducted on a cluster running in Grid’5000 composed of 34 nodes [28]. Each node hosts two 4-core Intel Xeon E5420 QC processors at 2.50GHz frequency with 8GB of RAM and 160GB SATA of storage space. Table 5.1 presents the hardware configuration of the cluster used for conducting experiments.

All the machines in the cluster are interconnected through 1 or 2 Giga-bit Ethernet and have only a single network interface. We created multiple virtual network interfaces (also known as aliasing) on a single physical Network Interface Controller (NIC) to exploit the robustness of RBFT [18, 36]. In the experiments we consider a system capable of handling only up to one Byzantine fault, i.e.,  $f = 1$ . Therefore, the cluster needs 4 nodes ( $3f + 1$ ) for running BFT protocol instances. We reserve 2 extra nodes, one for concurrent clients and one for hosting BFT-Bench framework. The reason to limit the number of Byzantine faults to 1 is due to the difficult NIC configuration for RBFT with more than one fault in Grid’5000 settings. Authors of RBFT conducted some of their experiments on a cluster composed of ten built-in network interfaces which made it easier to configure with  $f = 2$ . In practice, this limitation does not hold for PBFT and Chain as they do not require multiple virtual network interfaces. We performed some extra experiments to determine the sustainable cluster size by each protocol based on the number

Table 5.1: Hardware Configuration of the Cluster in Grid'5000

Cluster	CPU	Memory	Storage	Network
G5K I	4-core 2-CPU 2.5 GHz Intel Xeon E5420 QC	8 GB	160 GB SATA	1 Gbit Ethernet

of faults they can handle. PBFT is able to tolerate up to 2 simultaneous faults (7 replicas) before terminating and Chain remains available with a maximum of 3 faults (10 replicas). This observation is made under fault free scenario and no Byzantine faults were triggered. For effective practical comparison of BFT prototypes we used the same environment, i.e., same configuration, same cluster, same faultloads and workloads.

### 5.1.2 Software Settings

We have used original versions of the code bases for the three protocols in consideration<sup>1</sup>. However, we modified and added some code in their implementations for triggering the faults which we explained in Section 4.4.5. All the authors usually performed the experiments using echo service for evaluating their protocols, but unlike them, we introduce a delay of 30 ( $\pm 10\%$ ) milliseconds before sending any response to the client. This delay is meant to emulate the computations that a real service would perform during the commit phase of all protocols. We believe this is more realistic than just sending empty (echo) messages. We keep our focus on intentionally triggering the faults at replicas and not the accidental failures which can occur at network or hardware level. We model the faults by forcefully deviating execution of instructions of the BFT algorithms.

We use BFT-Bench for evaluating throughput and latency for each faulty behavior (similar to *a/b* microbenchmark by Castro and Liskov [30]). BFT-Bench also produces runtime statistics for dependability metrics. Low-level metrics such as CPU utilization and network usage are assessed offline. Finally, all the experiments were performed three times to calculate the average and standard deviations.

We consider *total experiment runtime* as the sum of *warm up time*<sup>2</sup> and *execution time*<sup>3</sup> where *warm up time* is set to 180s, and *execution time* to

<sup>1</sup>Code base of PBFT was downloaded from <http://www.pmg.csail.mit.edu/bft/#sw> whereas RBFT and Chain implementations were obtained directly from authors [18, 56].

<sup>2</sup>*warm up time* allows the system to stabilize before the statistics monitor starts to collect the execution traces.

<sup>3</sup>*execution time* is the actual monitoring phase where system collects all the statistics



either 600s or 800s. We exclude the results collected during *warm up period* from all our results as no actual workload or faultload is injected during this time. The *fault trigger time* is  $\frac{1}{2}$  of (*execution time*), i.e., 300s for fault types - *replica crash*, *message delay* and *network flooding* (timer of the *fault trigger time* starts with *execution time*). Once a server becomes malicious upon expiration of *fault trigger time*, it remains faulty until the end of the experiment. Independent of the protocols, during fault-free case (also corresponding in our experiments to the time elapsed before a fault gets triggered) we mostly achieve a peak throughput of around 32/33 requests per second. This is the consequence of introducing the 30 ( $\pm 10\%$ ) milliseconds of simulated computation before committing the incoming requests.

## 5.2 Comparative Evaluation under Faulty Scenarios

In this section, we present a comparative analysis of the three BFT protocols, PBFT, Chain and RBFT when facing different types of faults presented in Section 4.3.2 using BFT-Bench. It also demonstrates the loopholes of these BFT implementations. Table 5.2 provides the list of different faults considered for evaluation of prototypes for BFT protocols.

We inject one Byzantine fault, which means only one replica can be faulty in a system at any point of time. All the faults are performed independent of each other except fault type, *delay with overloading*, where we inject *message delay* fault in the primary while increasing the *#clients* at different intervals. This enables us to evaluate the prototypes with the varying (increasing) workload in presence of a Byzantine fault, *message delay*. Faultload for each fault is defined at the beginning of the experiment, but workload can also be defined at runtime. Faultload cannot be modified once the experiment starts, whereas workload is variable and depends on the injected fault type.

Table 5.3 provides a comprehensive analysis of PBFT, Chain and RBFT when faults are triggered in the system. In all the graphs representing evaluations of performance and dependability aspects, the first half corresponds to fault free scenarios, i.e., there is no fault injection. To perform each experiment, we define few initial values, such as (i) experiment start time, (ii) request/response message size and (iii) number of concurrent clients.

Defined as:

$\langle \text{start time, size, \#clients} \rangle$ .

---

for future analysis.

Table 5.2: List of different faultloads considered for evaluation

<b>fault_type</b>	<b>fault_loc</b>	<b>Description of the fault</b>
<i>Replica Crash</i>	primary	server stops and terminates all further communications
<i>Message Delay</i>	primary	primary delays sending pre-prepare messages to other replicas
<i>Network Flooding</i>	any non primary replica	faulty replica floods correct replicas with malicious messages to overload the network and computational resources
<i>System Overloading</i>	-NA-	#clients sending requests increases with time
<i>Delay with Overloading</i>	primary	it is a combination of two faults: message delay and system overloading

Latency and throughput of each experiment depend on the initial values. We observe throughput is mostly limited to 32/33 requests per second (with any  $\#clients$ ) due to induced computational delay at each replica while latency increases with increase in  $\#clients$ . During a fault free scenario, system is always available and maintains reliability. All the replicas adhere to the instructions of BFT algorithms and do not deviate from the correct paths.

As studied, each protocol has a primary/head ( $Replica_1$ ), responsible for handling the incoming client requests. The primary replica has the highest computational load as it manages all the client requests, their ordering, client connections and extra cryptographic operations (for verifying authenticity of the client and its request). Interestingly,  $Replica_4$  in PBFT is used slightly more compared to other non-primary replicas as we assigned it the responsibility for sending the encrypted response to the client instead of primary. Other times, all the replicas just send MAC of the responses to the clients. This is an optimization performed in PBFT by its authors. Similarly for Chain, the last replica performs more than the middle replicas as it is sending the responses to clients like PBFT. Due to the high CPU usage at primary, primary becomes a bottleneck when there are a high number of concurrent clients and also, in case of the Byzantine attacks at primary.

In all protocols, there are two network interfaces: replica-to-replica communication and replica-to-client communication. We consider measuring the total amount of data send (txkB/s) and received (rxkB/s) per second by each replica to/from clients and other replicas. All the graphs illustrating network utilization present normalized total data (total of incoming and outgoing data at each replica). Generally, we observe that primary in PBFT and RBFT uses the network more intensively than other replicas. This is primarily due to the

Table 5.3: Comprehensive analysis of fault handling by each BFT protocol

	<b>PBFT</b>	<b>Chain</b>	<b>RBFT</b>
<i>Replica Crash</i>	✓	×	×
<i>Message Delay</i>	✓	✓	✓
<i>Network Flooding</i>	✓	×	✓
<i>System Overloading</i>	✓ fails at clients > 80	✓ fails at clients > 120	✓ fails at clients > 35
<i>Delay with System Overloading</i>	✓ Max. #clients not evaluated	✓ Max. #clients not evaluated	✓ fails at clients > 10

incoming channel being used more often for receiving messages from replicas and clients. For other replicas, it is only replica-to-replica communication except for the *Replica<sub>4</sub>* which uses slightly more bandwidth due to optimization for sending encrypted response (rather than MAC) to clients. In case of Chain, head and tail are under utilized while replicas in the middle are used extensively. It can be explained by the fact that other replicas receive and send twice the number of MAC authenticators as of head and tail. Head replica receives MAC authenticators from a client for itself and next  $f + 1$  replicas. It then generates  $f + 1$  MAC authenticators for its successors. So the replica in total will have  $3f + 2$  MAC authenticators on the incoming ( $f + 1$ ) and outgoing ( $2f + 1$ ) channel. For the tail, it creates MAC authenticator just for the client leading to under utilization of outgoing channel. Consequently, uneven network bandwidth usage becomes an important, impacting factor that results in lower throughput of a protocol.

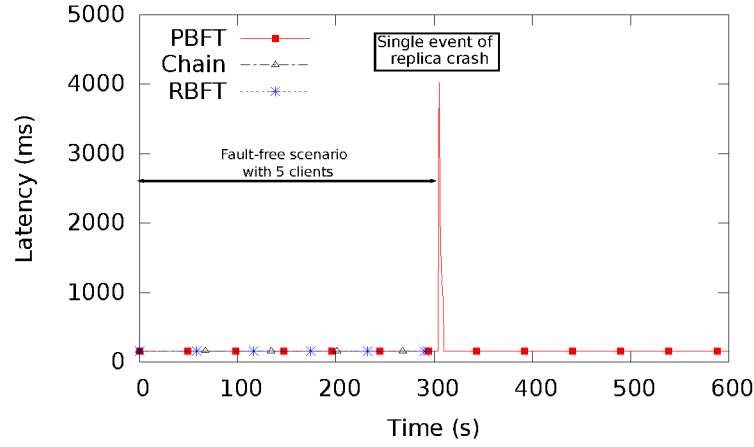
We now consider evaluation of different fault types and their impact on performance and dependability aspects using BFT-Bench.

### 5.2.1 Presence of Replica Crash

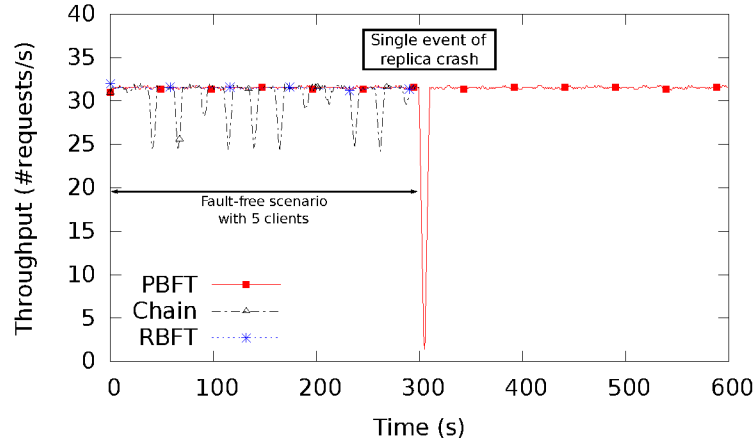
Here, we consider the Byzantine behavior described by the following faultload (see Figure 4.2):

$\langle 300s, replica\ crash, \{primary\} \rangle$ ,

defining *fault trigger time*, *fault type* and *fault location*, respectively.



(a) Latency



(b) Throughput

Figure 5.1: Performance evaluation of PBFT, Chain, RBFT when primary crashes

### 5.2.1.1 Performance Analysis

We implement this fault by killing the server process running at the primary, i.e., primary is automatically shutdown in the middle of the execution run at *fault trigger time*. For our evaluation, we consider crashing of primary over crashing of a non-primary replica due to its higher impact on performance. Since the primary replica is responsible for ordering the incoming requests, its crash leads to expensive *view change* protocol initiated by other replicas, thus degrading the overall performance. In our experiments, all the backup replicas wait for 5 seconds before considering primary to be unresponsive. In case of non-primary crash, protocol continues as replicas need only  $2f + 1$

matching responses. Figure 5.1 presents the performance of the prototypes when primary crashes. In the results for PBFT, we observe a sudden increase in latency (Figure 5.1(a)), and throughput (Figure 5.1(b)) drops sharply upon crash of the primary. This is due to the *view change* protocol, which replaces the faulty primary.

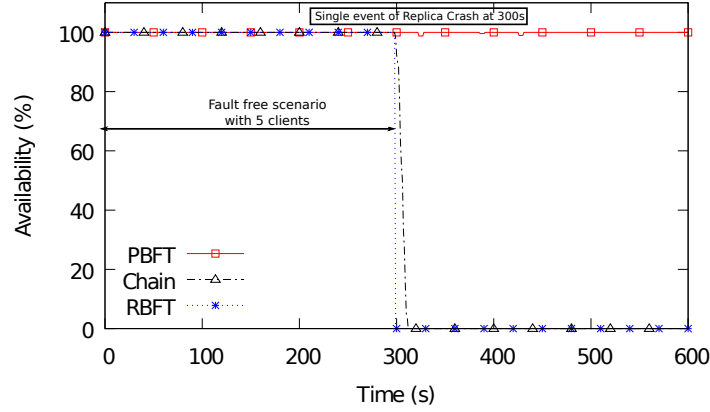
Prototypes for Chain and RBFT fail to respond once the primary crashes. Upon a crash, Chain cannot maintain its pipeline structure as the successor of the crashed server never receives any messages. Chain must switch to PBFT upon crash, but, unfortunately, this mechanism is not present in the original prototype. We would have observed the same performance as PBFT if switching was possible [20]. In RBFT, clients broadcast requests to all replicas. During crash fault, client enters an infinite request re-transmission loop while attempting to send a request to the crashed replica. This is due to the absence of a crash handling mechanism at the client side.

### 5.2.1.2 Dependability Analysis

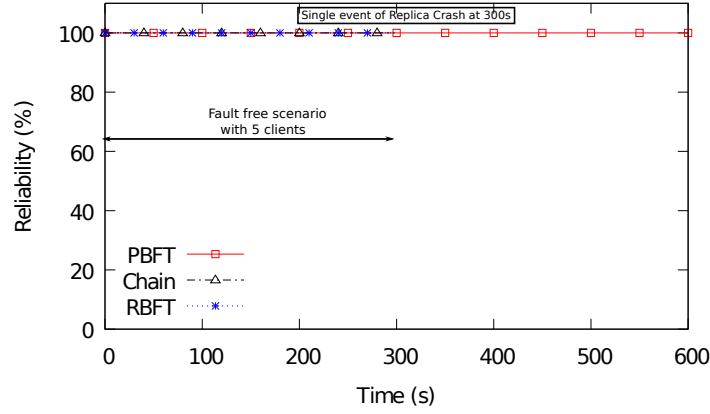
Figure 5.2(a) demonstrates the impact on the availability of the system upon the crash of the primary. This result can be easily predicted on the basis of performance evaluation of BFT prototypes. As soon as the primary crashes, availability of RBFT suddenly drops to zero while Chain drops slowly. It is due to the fact that other replicas in the chain (after the head replica) are processing the requests they have (exploiting the pipeline pattern of Chain). The tail will eventually send the responses coming from the pipeline to the respective clients. But the head will not take any further requests and availability will eventually hit zero. PBFT handles the primary crash by going through a *view change* and thus continues to make a progress. Figure 5.2(b) illustrates that reliability for PBFT is 100% even after the fault occurrence but Chain and RBFT are no more available leading to undetermined reliability. The requests of the clients will not complete and undergo infinite re-transmissions before terminating. If the system was available after primary crashes, the system would have continued to be reliable. We conclude this according to the definition of reliability that is a measurement of the total number of successful correct responses to the total number of requests send by the clients.

### 5.2.1.3 System and Network Level Statistics

Figure 5.3 represents the system and network level statistics of PBFT before the crash, upon a crash and after the crash. From Figures 5.3(a) and 5.3(b), we observe that PBFT is able to maintain almost the same CPU and network bandwidth utilization before and after the crash with a new elected primary



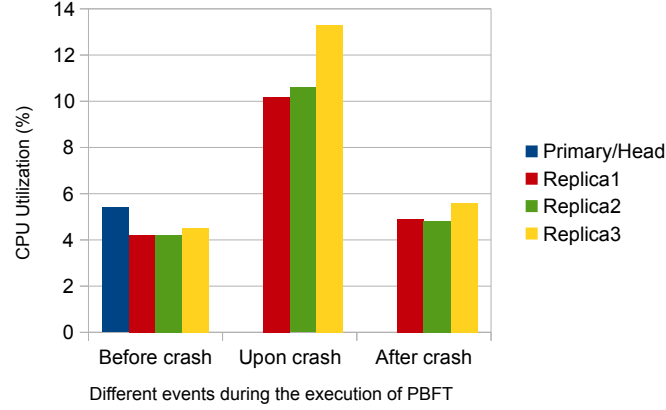
(a) Availability



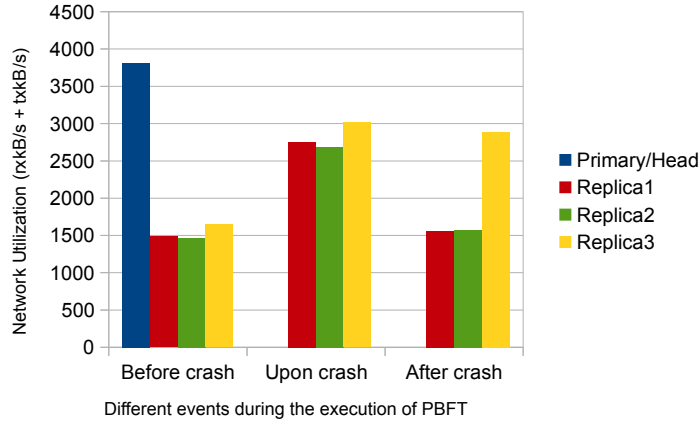
(b) Reliability

Figure 5.2: Dependability analysis of PBFT, Chain, RBFT when primary replica crashes

(*Replica<sub>4</sub>* in our case). Upon a crash, CPU utilization at primary (*Replica<sub>1</sub>*) becomes zero and other replicas start to use the CPU rapidly (slightly more than double of before crash). Upon crash, we observe more data exchange. This increment in network bandwidth and CPU utilization is due to the numerous message exchanges of *view change* protocol to elect a new primary and a number of MAC authentications performed at each replica, respectively. After the crash, newly elected primary (*Replica<sub>4</sub>*) uses more CPU as it performs client-to-replica and replica-to-replica communications and send-/receives more messages due to client interactions for requests and responses than other replicas in the system.



(a) CPU Utilization



(b) Network Utilization

Figure 5.3: CPU and Network utilization of PBFT in presence of primary replica crash

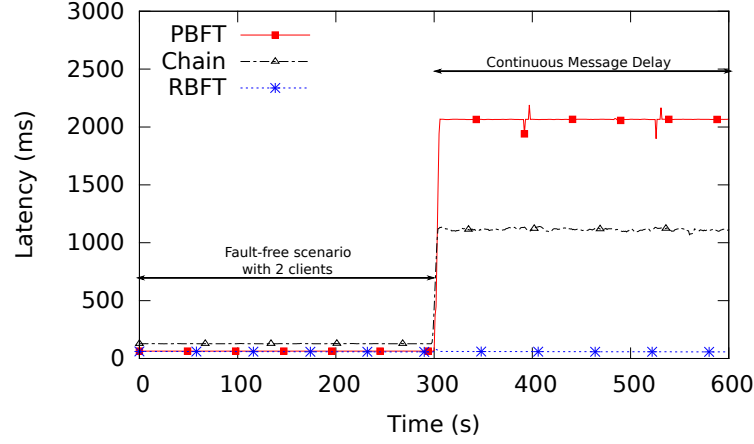
### 5.2.2 Presence of Message Delay

This behavior has faultload:

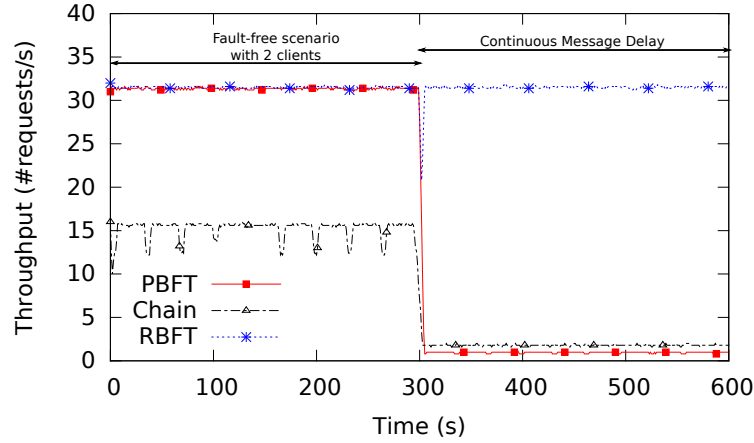
$\langle 300s, \text{message delay}, \{\text{primary}, 500, \text{pre-prepare}\} \rangle$

where values correspond to *fault trigger time*, *fault type*, *fault location*, *delay time* and *message type* (in reference to line 2 of Figure 4.2). We implement this fault by forcing the primary to delay sending of pre-prepare messages by 500ms in case of PBFT and RBFT, and the only message send by the head to its successor in case of Chain. We perform this fault with  $\#clients$  sending the requests equals 2.

## 5.2.2.1 Performance Analysis



(a) Latency



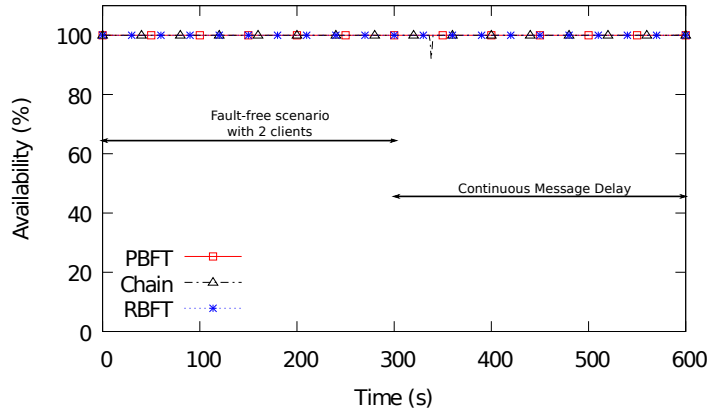
(b) Throughput

Figure 5.4: Performance evaluation of PBFT, Chain and RBFT in presence of message delay fault at primary

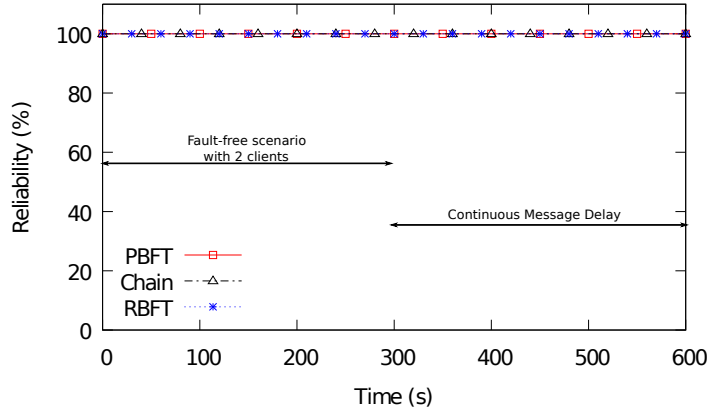
We observe that impact of delaying the pre-prepare message (it contains the request ordering number) is higher in comparison to the delay of any other message. But this holds true only for RBFT and PBFT. All the replicas upon receiving a request, wait for its corresponding pre-prepare message. Upon failing to receive the message before the timer expires, replicas go through an agreement to trigger the *view change* protocol to replace the faulty primary. View change and new primary election protocols are time-consuming and degrade the performance. View change is not triggered when other message types are delayed. Notice that in Chain, the same behavior could be observed



if the delay fault occurs in any replica, due to its pipeline structure. In Figure 5.4(b), we can observe that peak throughput of Chain is 15 while for RBFT and PBFT it is 32 (before the fault is triggered). This is due to the fact, that Chain is not completely fed to exploit its pipeline pattern. Also, every request in Chain undergoes the commit phase, sequentially introducing a delay of at least 120ms (simulated computation delay). Since requests are executed in a closed-loop, fewer requests are sent to the Chain protocol, consequently less requests are executed, thus, the throughput is lower than the throughput of PBFT and RBFT. In presence of fault, the peak throughput falls from 32 to 2 requests per second for PBFT and Chain, and latency increases accordingly in Figure 5.4(a). This throughput is relevant since no more than 2 requests



(a) Availability



(b) Reliability

Figure 5.5: Dependability analysis of PBFT, Chain, RBFT in presence of message delay fault at primary

can be executed per second because of the 500ms delay. We do not observe

the same behavior for RBFT as it implements a robust mechanism to handle this fault<sup>4</sup>. RBFT uses a delay adaptive mechanism where it inspects the total number of pre-prepare messages received by a replica within a bounded interval. If the number is less than the one expected, a *view change* is triggered. This is practically analyzed from the difference observed between the throughput at primary and backup instances of RBFT [18]. Therefore, we do not see a degradation in performance after fault injection except at the moment when the fault was triggered. Throughput drops by 40% approximately during the time taken by *view change* protocol to replace the faulty primary. This test could be performed for a maximum of 5 clients as the prototype of RBFT fails to make a progress upon fault injection. In PBFT and Chain, *view change* protocol is not triggered due to the absence of fault handling mechanisms.

### 5.2.2.2 Dependability Analysis

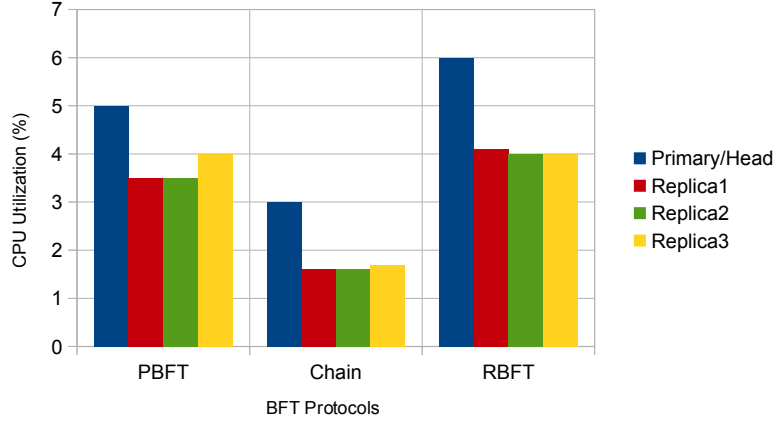
Figure 5.5(a) displays that all the three prototypes continue even after primary starts to delay the pre-prepare messages. Availability has no impact as the system is always available, but responds slower which we clearly see in the performance (latency) analysis. Figure 5.5(b) illustrates that system is always reliable for all the considered BFT protocols. It is due to the fact that BFT systems continue to work normally by responding to the client's requests but with a delay injected by the primary. This delay triggers the timeout at clients and requests are re-transmitted. Primary do not add any delay to re-transmitted requests as they are already ordered and pre-prepare messages of these requests have already been sent.

Therefore, during the message delay, overall system remains available and reliable except that the responses are delayed to the clients, triggering re-transmissions of delayed requests. Hence, we see the impact only on the performance and not on the dependability metrics of this fault.

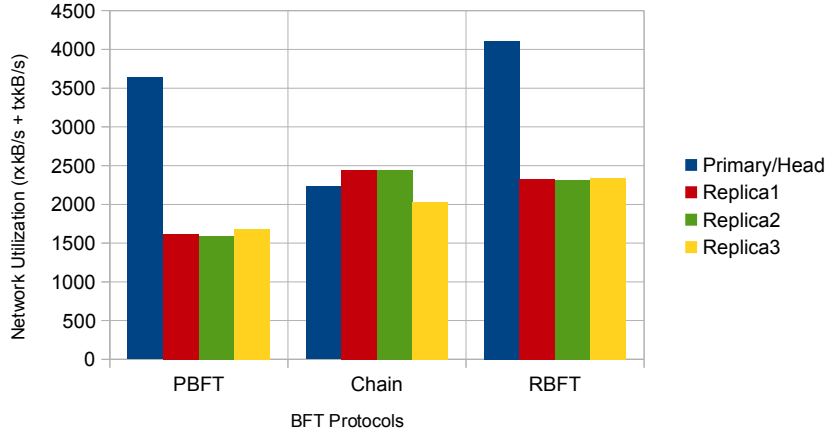
### 5.2.2.3 System and Network Level Statistics

Figures 5.6(a) and 5.6(b) illustrate CPU and network utilization, respectively. We observe a difference in CPU utilization of primary for all the protocols as it delays pre-prepare messages and processes trigger time for sending these messages. Delay by primary triggers the timeout (for receipt of pre-prepare messages), which enables replicas to begin exchange messages with each other and primary. This increases the utilization of network bandwidth at each replica. Also, when timeout at a client expires for a response to a request,

<sup>4</sup>RBFT follows the design and specifications of Aardvark [36] for handling this behavior.



(a) CPU Utilization



(b) Network Utilization

Figure 5.6: CPU and Network usage in presence of message delay fault at primary with  $\#clients = 2$

it retransmits the request. Thus, further augmenting the network bandwidth for replica-to-client communications.

### 5.2.3 Presence of Network Flooding

Figure 5.7 presents the performance of PBFT & RBFT when a non-primary replica starts to flood (sends as many malicious/corrupt messages as possible) other replicas.

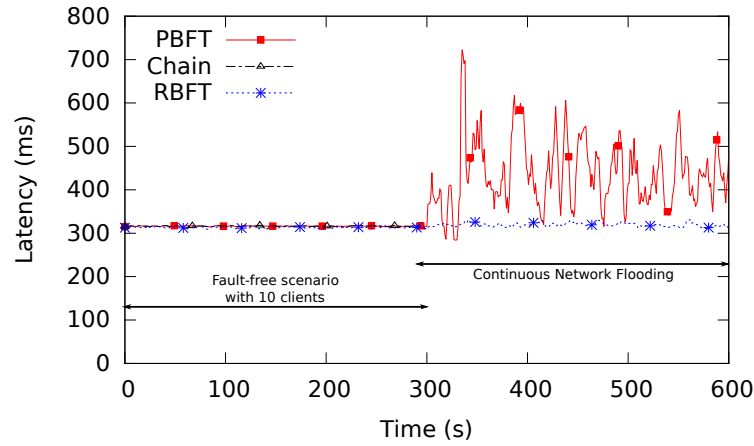
Faultload used is:

$\langle 300s, network\ flooding, \{Replica_2, 4KB\} \rangle$ ,

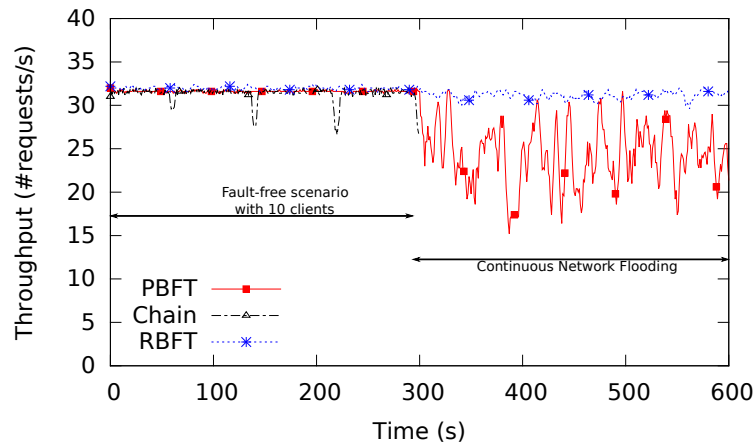
where  $Replica_2$  will start to flood other servers with corrupt messages of size

4KB at 300s (as defined in line 3 in Figure 4.2). To perform this experiment, we consider  $\#clients$  to be 10.

### 5.2.3.1 Performance Analysis



(a) Latency



(b) Throughput

Figure 5.7: Performance evaluation of PBFT, Chain and RBFT in presence of network flooding by a non primary replica

We implement this behavior by forcing a replica to enter an infinite loop of continuous transmission of malicious messages to other servers until the end of the experiment (corrupt messages have the same size as of request messages). We observe that any replica, either primary or non-primary would impact the performance in the same way. The results illustrate that Chain makes no

progress upon fault injection while the performance of PBFT becomes sporadic. This is due to the expensive, time consuming cryptographic operations performed over corrupt messages by all the replicas in PBFT and successor replica in Chain. Inability to handle corrupt messages introduces a gap in the communication pattern and lack of protocol switching mechanism holds the Chain from continuing.

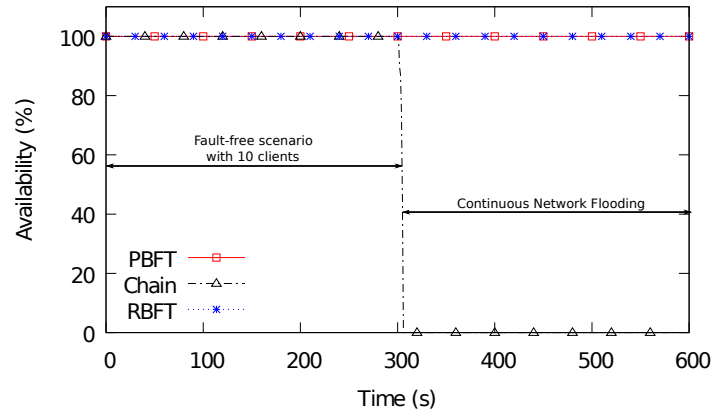
RBFT uses multiple NICs to avoid malicious clients and replicas from flooding client-to-replica & replica-to-replica communications. RBFT also employs flood adaptive mechanism where non-faulty replicas can detect a flooding replica and blacklists it [18]. Flood protection enables a non-faulty replica to monitor the number of messages (including correct & malicious messages) received. If a non-faulty replica receives more than a specific number of messages from a particular replica in a period of time, then it can label this replica as faulty and initiates a blacklisting protocol. When this happens, RBFT closes the NIC of the misbehaving replica for some time but after a given period it rejoins the system again. Due to this, we observe slight variations in performance with up to 5% of degradation.

### 5.2.3.2 Dependability Analysis

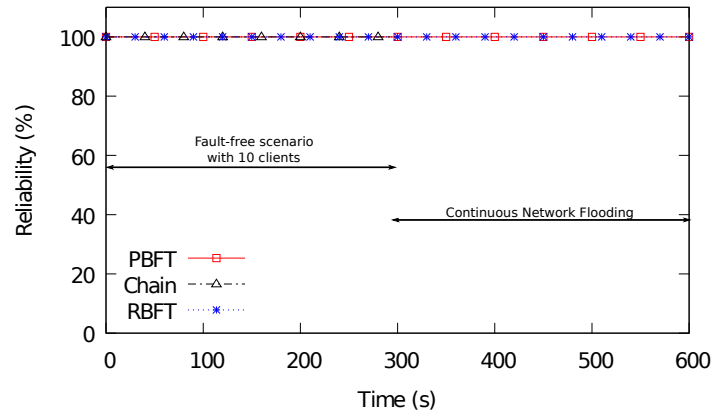
In Figure 5.8(a) we demonstrate the availability of RBFT, PBFT and Chain when network is flooded by one of the non-primary replicas (*Replica<sub>2</sub>*). We observe that PBFT and RBFT are always available while Chain succumbs to unavailability after the fault is triggered. RBFT handles network flooding with flood adaptive mechanisms and PBFT prototype is robust enough to manage to continue. Whereas for chain, this scenario is similar to crash as the replica succeeding the faulty replica cannot handle the flood and terminates. According to Figure 5.8(b), we can again conclude that BFT systems are 100% reliable but not 100% available. In case of Chain, the availability reduces to 0, but the reliability of the system cannot be questioned as the system terminates to progress without producing any incorrect responses. There are no responses at all. Having no response, cannot be equated to an incorrect response. Therefore, reliability remains undetermined (it cannot be concluded to non-reliability of the system).

### 5.2.3.3 System and Network Level Statistics

For evaluation of CPU utilization and network bandwidth usage, we do not consider Chain as it terminates when a replica starts to flood other replicas. We consider only RBFT and PBFT as they continue during this fault. Figure 5.9(a) illustrates the CPU utilization of RBFT and PBFT with network



(a) Availability

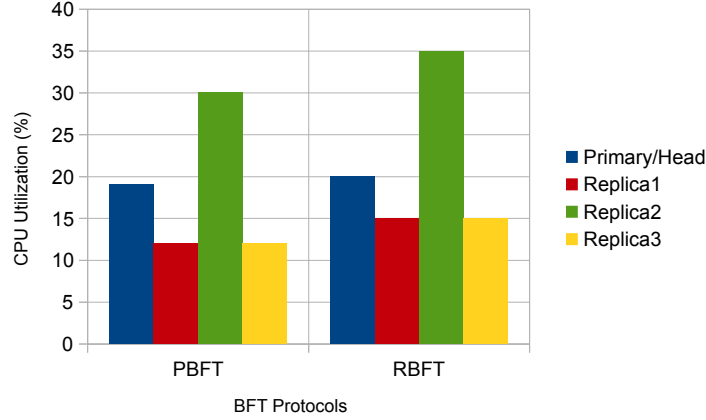


(b) Reliability

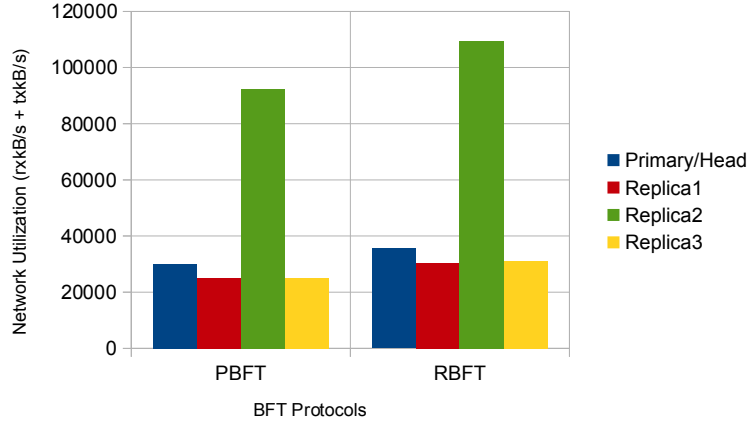
Figure 5.8: Dependability analysis of PBFT, Chain, RBFT in presence of network flooding by a non primary replica

flooding in the presence of 10 concurrent clients. Computation at all the nodes increases as they also perform authentication and validation of all corrupted messages (corrupted messages are discarded once authentication fails). CPU usage is maximum at *Replica<sub>2</sub>* as it is generating those malicious messages for all replicas. RBFT and PBFT behave similarly in this aspect.

Figure 5.9(b) demonstrates usage of network bandwidth at each replica for RBFT and PBFT. Fault replica, *Replica<sub>2</sub>*, sends a lot of corrupt messages on its outgoing channel and these messages are evenly distributed to all other replicas to their incoming channels, increasing the network utilization at all the replicas. But *Replica<sub>2</sub>* uses the maximum network bandwidth by sending a large number of malicious messages of equal size to correct messages.



(a) CPU Utilization



(b) Network Utilization

Figure 5.9: CPU and Network usage in presence of network flooding by a non primary replica with  $\#clients = 10$

#### 5.2.4 Presence of System Overloading

Figure 5.10 presents the performance of protocols under contention but in fault free conditions. This fault type measures the maximum workload *i.e.*  $\#clients$  a protocol can handle before discontinuation. We do not inject any Byzantine fault ( $f = 0$ ). We implement this fault by simply increasing  $\#clients$  during the experiment run.

Faultload (see line 4 in Figure 4.2) for this experiment is:

$\langle 200s, \text{system overloading}, \{20\} \rangle,$

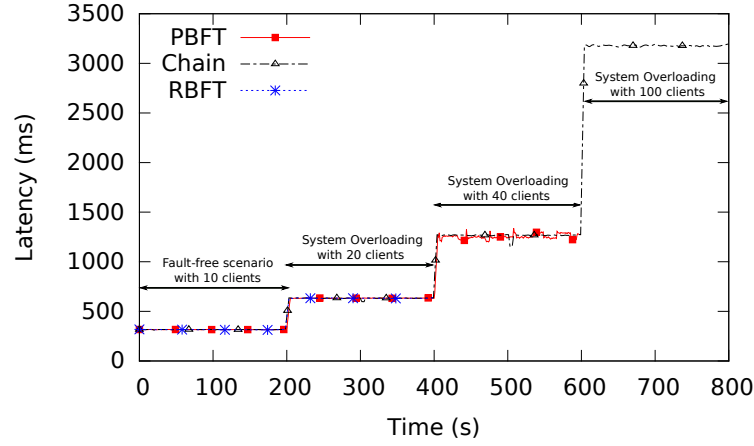
$\langle 400s, \text{system overloading}, \{40\} \rangle,$

$\langle 600s, \text{system overloading}, \{100\} \rangle,$

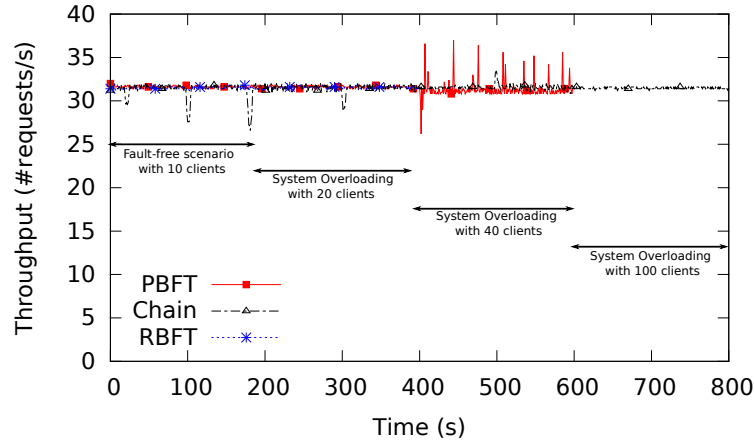
where  $\#clients$  increase from 10 (at time  $(t) = 0$ ) to 20 at  $t = 200s$ , then 40

at  $t = 400s$ , and finally 100 at  $t = 600s$ .

#### 5.2.4.1 Performance Analysis



(a) Latency



(b) Throughput

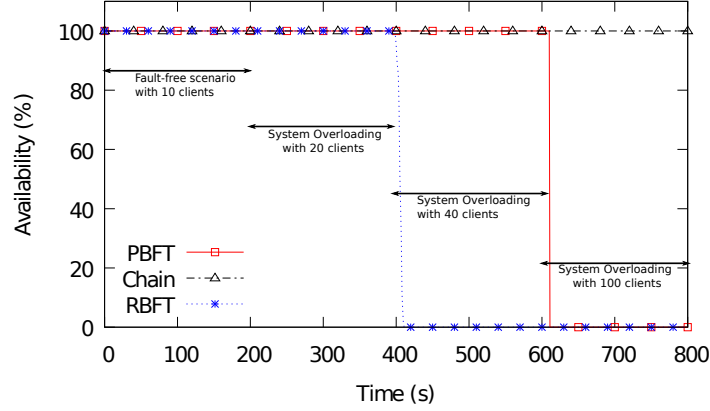
Figure 5.10: Performance evaluation of PBFT, Chain and RBFT when system is overloaded with increasing number of clients at every 200s

From the results, we observe that all the protocols achieve the peak throughput with 10 clients (Figure 5.10(b)), and the latency increases with the number of clients in the system (Figure 5.10(a)). At  $t = 400s$ , the RBFT prototype cannot handle more than 20 clients, while PBFT and Chain continue to progress. At  $t = 600s$ , the PBFT prototype does not appear as reliable as before. It is unable to handle the load of 100 clients and thus terminates.

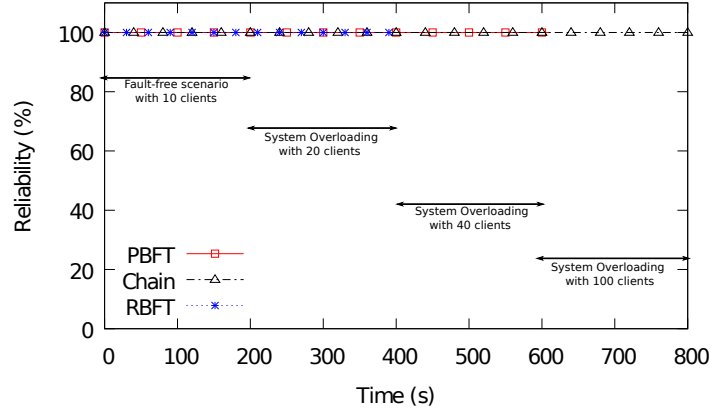


However, Chain survives 100 clients with a constant throughput but latency continues to augment (with increasing  $\#clients$ ).

#### 5.2.4.2 Dependability Analysis



(a) Availability



(b) Reliability

Figure 5.11: Dependability analysis of PBFT, Chain, RBFT when system is overloaded with increasing number of clients at every 200s

Figure 5.11(a) presents that RBFT and PBFT stop once they cannot handle a certain number of clients. In this fault, no Byzantine fault is injected and only the workload is increased with time. These results prove that the prototypes of these BFT systems cannot handle heavy, realistic workload. To be acknowledged by the real world practitioners, it is important for these protocols to demonstrate the ability to handle the workloads and faultloads of real world applications. From the analysis, we conclude that Chain is able

to handle more  $\#clients$ . We didn't test Chain for its maximum load in the presence of message delay fault.

According to Figure 5.11(b) we observe that with time when PBFT and RBFT gets unavailable, reliability becomes undetermined. For Chain, it remains reliable and available all the time (until it can serve all the concurrent clients).

#### 5.2.4.3 System and Network Level Statistics

Figure 5.12 illustrates the CPU utilization at each replica for all the considered BFT protocols with increasing  $\#clients$  at different intervals of time. For all the protocols, we observe that replica receiving the client requests, i.e., the primary has the maximum CPU load. It is due to the replica-to-client communication. For Chain, head performs more work in comparison to other protocols as it computes and verifies more MAC authenticators for every incoming request (Chain profits from batching). CPU utilization for PBFT and RBFT is comparable as they both follow the same communication pattern (of PBFT). RBFT halts when the number of clients increases to 40 and PBFT stops at 100 clients while Chain continues until 100 clients (that's why there are no bars at these times for PBFT and RBFT).

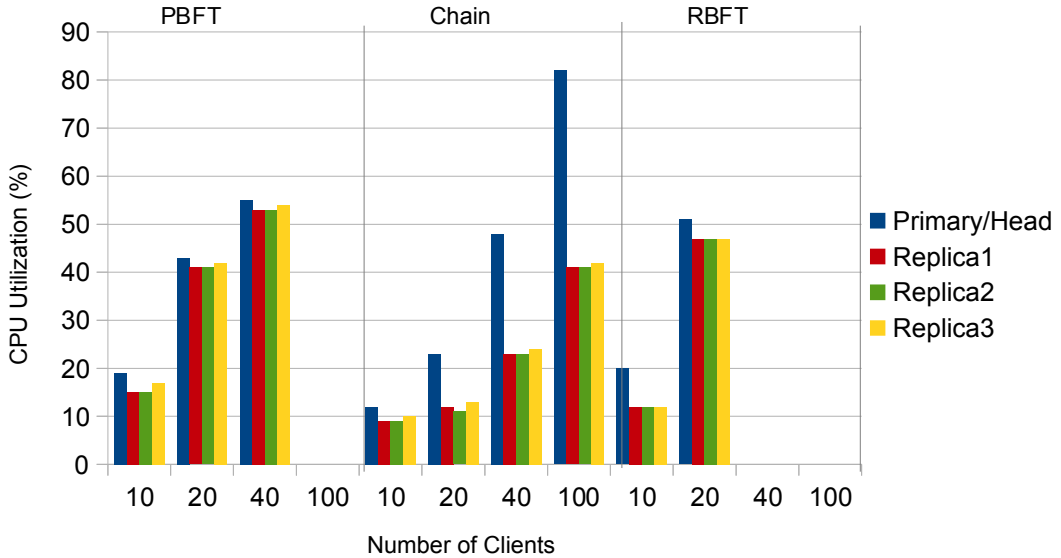


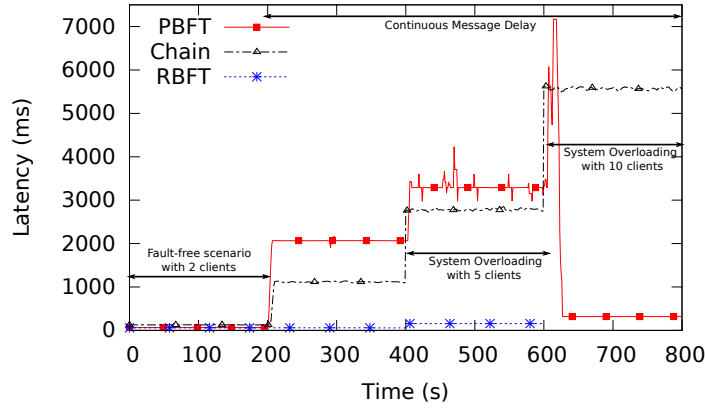
Figure 5.12: CPU Utilization of PBFT, Chain and RBFT when system is overloaded with increasing number of clients at every 200s

### 5.2.5 Combination of Different Types of Faults

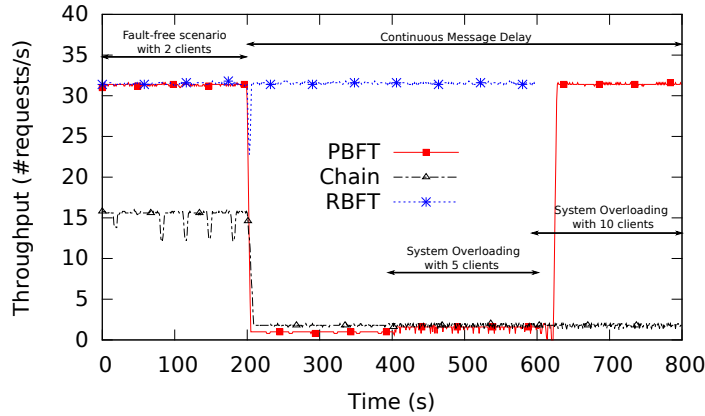
In this experiment, we inject a combination of faults, i.e., *message delay* in presence of *system overloading*. This enables us to determine the maximum number of clients a BFT system can survive when there exists a Byzantine fault in the system. Figure 5.13(a) and 5.13(b) present latency and throughput, respectively, upon injection of delay fault in presence of increasing workload ( $\#clients$ ).

The faultload (line 5 in Figure 4.2) is:

$\langle 200s, message\ delay, \{primary, 500, pre-prepare\} \rangle,$   
 $\langle 400s, system\ overloading, \{5\} \rangle,$   
 $\langle 600s, system\ overloading, \{10\} \rangle.$



(a) Latency

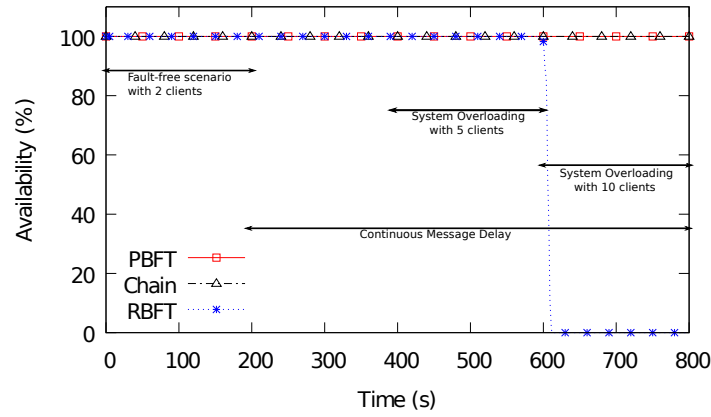


(b) Throughput

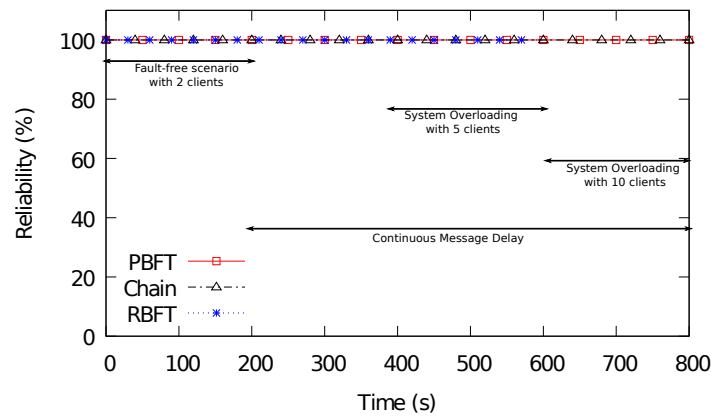
Figure 5.13: Performance evaluation of PBFT, Chain and RBFT in presence of system overloading with message delay fault. At every 200s, number of clients increases in the system

## 5.2.5.1 Performance Analysis

The experiment launches in a fault free environment with  $\#clients = 2$ . At  $t = 200s$ , delay fault is injected at the primary while keeping constant  $\#clients$ . Until 400s, the result is same as message delay (Section 5.2.2). At  $t = 400s$ , we increases  $\#clients$  to 5. Now we observe that RBFT continues to handle the delay fault (exploiting delay adaptive fault mechanism) without a significant effect on the performance while Chain and PBFT demonstrate an increase in latency and their throughput continues to be 2 requests per second. Finally at  $t = 600s$ ,  $\#clients$  grows to 10. RBFT cannot handle contention (more than



(a) Availability



(b) Reliability

Figure 5.14: Dependability analysis of PBFT, Chain and RBFT in presence of system overloading with message delay fault. At every 200s, number of clients increases in the system

10 clients) and terminates, however, Chain and PBFT continue to advance.

PBFT undergoes a *view change* where the faulty primary is replaced with a new primary. This phenomenon occurs due to the expiration of the timers at backup replicas waiting to get pre-prepare messages for a request received earlier. With the new primary, we observe the same performance of fault free scenario. Chain continues to worsen in terms of latency, but maintains a constant throughput of 2. It is interesting to observe that in the presence of message delay fault, workload handled by all the protocols reduces by many folds. This clearly demonstrates ineffectiveness of the protocols when a fault occurs under high workload conditions.

### 5.2.5.2 Dependability Analysis

Figure 5.14(a) is a combination of Figures 5.5(a) and 5.11(a). As demonstrated previously, RBFT can handle a maximum of 20 clients in fault free scenario and with message delay fault, maximum *#clients* can only be 10. We see the same behavior in the figure 5.14(a). At 600s, RBFT fails to continue and system becomes unavailable, whereas PBFT and Chain advances. We are sure that PBFT will fail once *#clients* reaches more than 40 (we did not perform this analysis). According to Figures 5.14(a) and 5.14(b), we observe that PBFT and Chain continue to be available and reliable while RBFT becomes unavailable. Therefore, reliability cannot be determined until the time system is available once again.

## 5.3 Summary

The evaluation affirms our motivation that prototypes for most of the considered state-of-the art protocols (for Chain and PBFT) have been assessed only in fault-free scenarios which make them challenging to be used in real time systems where faults are prone to happen. Since RBFT uses fault-adaptive mechanisms for certain types of faults, it is able to maintain a constant performance with a small percentage of degradation during these malicious behaviors. However, it still fails to sustain all types of injected faults, as illustrated it terminates the progress in the most prominently occurring crash fault. Our experimental results confirm our above statements.

We also observe that availability is 100% when the system is up and running, but upon fault injection, for some protocols, availability drops to 0% upon fault injection. According to the conducted experiments, all BFT protocols are 100% reliable (i.e. maintains consistency across replicas) until the system is available. Once the protocols fail upon fault injection, reliability cannot be measured. Theoretically, all BFT protocols must be 100% reliable

---

and available in the presence or absence of various faults, which is not the case in practice. Performance and dependability evaluations by BFT-Bench affirms that prototypes have not been tested rigorously under various fault models. It also demonstrates many loopholes in the BFT implementations which cause the termination of the prototypes when a fault is triggered.



# Conclusions and Perspectives

---

## Contents

---

6.1	Conclusions . . . . .	136
6.2	Perspectives . . . . .	137
6.3	Publications . . . . .	138
6.4	Acknowledgments . . . . .	138

---



## 6.1 Conclusions

The cloud computing represents a significant shift our society has gone through. It radically changes the way enterprises manage IT - like servers, data centers, OS, middleware and clustering. With the increasing on-demand computing, performance and dependability have become the important requirements of today's critical cloud services and data centers. The growth of cloud technologies, their applications, service oriented models, have raised many types of failures and attacks paradigms. Such faults have been termed as Byzantine faults.

Byzantine Fault Tolerance (BFT) is a general approach to make distributed systems, theoretically, tolerate arbitrary faults. BFT has been investigated extensively in previous years, with two main families of BFT protocols, (i) protocols that enhance performance in fault-free cases, and (ii) protocols that minimize performance degradation in the presence of some types of faults. Although considerable effort has been made to study Byzantine faults and achieve BFT protocols, but yet they fail to convince the practitioners for adopting them in real world settings. Serious concerns have been raised in the last decade for theoretically dealing with such behaviors, but not many attempts have been proposed to bring Byzantine fault tolerance to the sight of potential users. This is mainly due to the significant lack of practical analysis of QoS metrics, on common grounds. Evaluations of BFT protocols have been conducted in simplified settings which fail to challenge the prototypes in worst case scenarios, i.e., presence of arbitrary faults, high contention, malicious clients, etc. Also, no constant improvements are made to any of these prototypes after their initial release. To the best of our knowledge, there is no practical solution to identify and inject various Byzantine behaviors for evaluating performance and dependability levels of BFT protocols.

In this thesis, we underwent extensive research on technical and scientific know-how in designing a dependability and performance benchmark for distributed systems. We proposed a generic software architecture in an effort to help designers and researchers developing benchmark solutions for distributed protocols for analyzing various QoS aspects. We further used the generic architecture as a building block for BFT-Bench.

This thesis presented BFT-Bench, the first framework for empirically evaluating BFT implementations to quantify their dependability and performance levels under different faulty behaviors and workloads. BFT-Bench framework tests three state-of-the-art BFT protocols, automatically deploys them, al-

lows to generate different types of faults, injects them at different locations and different rates, and computes performance and dependability measures. We also presented the experiments conducted with BFT-Bench. The evaluation results show that BFT-Bench is able to successfully compare various BFT protocols, in various faulty behaviors.

We wish to make BFT benchmarking easy to adopt by developers and end-users of BFT protocols. BFT-Bench framework aims to help researchers and practitioners to better analyze and evaluate the effectiveness and robustness of BFT systems.

## 6.2 Perspectives

This work unlocks the way of evaluating BFT protocols under real world settings, bridging the gap between the research and practical usability of BFT. While this work concentrates on presenting the current version of BFT-Bench with some BFT protocols and their related fault types, we believe that the proposed approach can be easily extended to other BFT protocols listed under Section 2.4, and many other faulty behaviors. Addition of different fault models will open up the interesting prospects for analyzing other QoS metrics such as security, scalability, cost, etc. In this thesis, we have interested in evaluating performance and dependability aspects, considering throughput, latency, availability and reliability, in addition to some low-level network statistics and cost of using BFT services in terms of the number of physical resources used.

BFT-Bench can also interest developers and end-users to model it as a selection tool for determining the suitable BFT prototype abiding to required real world settings and QoS parameters. Such an example is of a non-critical system which would care more for performance rather than dependability guarantees under high contention.

The proposed benchmark can be used to aid researchers from Byzantine community to trace implementation issues and tune their source codes and default parameters (such as various timeouts at client and server side) accordingly. These modifications and corrections will not only improve the original implementations many folds, but will also boost the robustness and effectiveness of BFT prototypes. An integration of a debugging tool can ease the process of determining bugs.

Another technical perspective of this work could be considering high-level application domains such as database systems, web services, and web servers, that runs on BFT protocols under comparison. For example, PBFT imple-

mented Byzantine fault-tolerant NFS service using their algorithm while authors of RBFT proposed using their protocols for applications like Zookeeper [63] or Boxwood asynchronous API [82].

Furthermore, cloud providers can cater BFT-as-a-Service to their customers, specifically to ones deploying mission critical applications. This would imply Byzantine fault tolerance could be proposed as a cloud service, on-demand, adapted to application needs.

## 6.3 Publications

Some of our work have been published in international conferences, and few are under submission.

1. Divya Gupta, Lucas Perronne, Sara Bouchenak. BFT-Bench: Framework to Evaluate Robustness and Effectiveness of BFT Protocols in Practice, 7th ACM/SPEC International Conference on Performance Engineering, Delft, The Netherlands, March 12-18, 2016.
2. Divya Gupta, Lucas Perronne, Sara Bouchenak. BFT-Bench: Framework to Evaluate Robustness and Effectiveness of BFT Protocols in Practice, 6th ACM Symposium on Cloud Computing, Hawai'i, USA, August 27-29, 2015. Poster.
3. Divya Gupta, Lucas Perronne, Sara Bouchenak. BFT-Bench: Towards a Practical Evaluation of Robustness and Effectiveness of BFT Protocols, 16th IFIP International Conference on Distributed Applications and Interoperable Systems, Heraklion, Crete, June 6-9, 2016 (To appear).

## 6.4 Acknowledgments

This work was partly supported by AMADEOS (Architecture for Multi-criticality Agile Dependable Evolutionary Open System-of-Systems) which is a collaborative project funded under the European Commission's FP7 (FP7-ICT-2013-610535) with University of Grenoble as one of the partners.

All our experiments were conducted on the Grid'5000 experimental testbed, developed under the INRIA ALADDIN development action with support from CNRS, RENATER and several Universities, as well as other funding bodies.

# Bibliography

- [1] Amazon s3 availability event: July 20, 2008. retrieved on 2013-07-20. <http://status.aws.amazon.com/s3-20080720.html>.
- [2] Cloud Computing. [https://en.wikipedia.org/wiki/Cloud\\_computing](https://en.wikipedia.org/wiki/Cloud_computing).
- [3] Cloud outage collection. <http://ventures.tpedersen.net/errata/cloudstatus/cloud-outage-collection>.
- [4] e-fiscal project state of the art repository.
- [5] Ibm: The autonomic computing initiative. <http://www.ibm.com/autonomic>.
- [6] Performance Test Tools. <http://www.opensourcetesting.org/performance.php>.
- [7] A recommendation for high-availability options in tpc benchmarks. <http://www.tpc.org/information/other/articles/ha.asp>.
- [8] Dependability Benchmarking Project. <http://webhost.laas.fr/TSF/DBench/>, 2004.
- [9] Distributed Application Architecture , 2009.
- [10] It's probable that you've misunderstood 'Cloud Computing' until now, 2010.
- [11] M. Abd-El-Malek, G. R. Ganger, G. R. Goodson, M. K. Reiter, and J. J. Wylie. Fault-scalable byzantine fault-tolerant services. In *SOSP*, pages 59–74, 2005.
- [12] P. A. Alsberg and J. D. Day. A principle for resilient sharing of distributed resources. In *Proceedings of the 2Nd International Conference on Software Engineering, ICSE '76*, pages 562–570. IEEE Computer Society Press, 1976.
- [13] G. Alvarez and F. Cristian. Centralized failure injection for distributed, fault-tolerant protocol testing. In *Distributed Computing Systems, 1997., Proceedings of the 17th International Conference on*, pages 78–85, May 1997.

- [14] P. S. N. F. N. P. V. Alysson Bessani, Miguel Correia. Intrusion tolerance: The "killer app" for bft protocols (?). In *BFTW3: Why? When? Where? Workshop on the theory and practice of Byzantine fault tolerance*, 2009.
- [15] Y. Amir, B. A. Coan, J. Kirsch, and J. Lane. Byzantine replication under attack. In *DSN*, pages 197–206, 2008.
- [16] L. Arantes, R. Friedman, O. Marin, and P. Sens. Probabilistic byzantine tolerance for cloud computing. In *34th International Symposium on Reliable Distributed Systems (SRDS'15)*, Sept. 2015.
- [17] J. Arlat, A. Costes, Y. Crouzet, J.-C. Laprie, and D. Powell. Fault injection and dependability evaluation of fault-tolerant systems. *Computers, IEEE Transactions on*, 42(8):913–923, Aug 1993.
- [18] P.-L. Aublin, S. B. Mokhtar, and V. Quéma. Rbft: Redundant byzantine fault tolerance. In *ICDCS*, pages 297–306, 2013.
- [19] A. Avizienis, J.-C. Laprie, B. Randell, and C. E. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. Dependable Sec. Comput.*, 1(1):11–33, 2004.
- [20] J.-P. Bahsoun, R. Guerraoui, and A. Shoker. Making bft protocols adaptive.
- [21] R. Banabic, G. Candea, and R. Guerraoui. Finding trojan message vulnerabilities in distributed systems. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '14*, pages 113–126, New York, NY, USA, 2014. ACM.
- [22] R. Barbosa, J. Karlsson, Q. Yu, and X. Mao. Toward Dependability Benchmarking of Partitioning Operating Systems. In *IEEE/IFIP 41st International Conference on Dependable Systems Networks (DSN), 2011*, pages 422 –429, june 2011.
- [23] D. Bernstein, E. Ludvigson, K. Sankar, S. Diamond, and M. Morrow. Blueprint for the intercloud - protocols and formats for cloud computing interoperability. In *Internet and Web Applications and Services, 2009. ICIW '09. Fourth International Conference on*, pages 328–336, May 2009.
- [24] A. Bessani, J. Sousa, and E. E. Alchieri. State machine replication for the masses with bft-smart. In *Dependable Systems and Networks (DSN)*,

- 2014 44th Annual IEEE/IFIP International Conference on, pages 355–362. IEEE, 2014.
- [25] K. Birman, G. Chockler, and R. van Renesse. Toward a cloud computing research agenda. *SIGACT News*, 40(2):68–80, June 2009.
- [26] L. Bobelin, A. Bousquet, J. Briffaut, J.-F. Couturier, C. Toinard, E. Caron, A. Lefray, and J. Rouzaud-Cornabas. An advanced security-aware cloud architecture. In *High Performance Computing Simulation (HPCS), 2014 International Conference on*, pages 572–579, July 2014.
- [27] A. Brown and D. A. Patterson. Towards availability benchmarks: A case study of software raid systems. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference, ATEC '00*, pages 22–22, Berkeley, CA, USA, 2000. USENIX Association.
- [28] F. Cappello, E. Caron, M. Dayde, F. Desprez, Y. Jégou, P. Primet, E. Jeannot, S. Lanteri, J. Leduc, N. Melab, et al. Grid’5000: A large scale and highly reconfigurable grid experimental testbed. In *Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing*, pages 99–106. IEEE Computer Society, 2005.
- [29] A. Casimiro, P. Verissimo, D. Kreutz, F. Araujo, R. Barbosa, S. Neves, B. Sousa, M. Curado, C. Silva, R. Gandhi, and P. Narasimhan. Trone: Trustworthy and resilient operations in a network environment. In *Dependable Systems and Networks Workshops (DSN-W), 2012 IEEE/IFIP 42nd International Conference on*, pages 1–6, June 2012.
- [30] M. Castro and B. Liskov. Practical byzantine fault tolerance. In *OSDI*, pages 173–186, 1999.
- [31] M. Castro and B. Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst.*, 20(4):398–461, 2002.
- [32] R. Chandra, R. Lefever, M. Cukier, and W. Sanders. Loki: a state-driven fault injector for distributed systems. In *Dependable Systems and Networks, 2000. DSN 2000. Proceedings International Conference on*, pages 237–242, 2000.
- [33] Q. Z. H. C. Chunye Gong, Jie Liu and Z. Gong. The characteristics of cloud computing. In *Parallel Processing Workshops (ICPPW), 2010 39th International Conference on*, pages 275–279, Sept 2010.
- [34] A. Clement, M. Kapritsos, S. Lee, Y. Wang, L. Alvisi, M. Dahlin, and T. Riche. Upright cluster services. In *SOSP*, pages 277–290, 2009.

- [35] A. Clement, M. Marchetti, E. Wong, L. Alvisi, and M. Dahlin. Bft: The time is now. In *Proceedings of the 2Nd Workshop on Large-Scale Distributed Systems and Middleware*, LADIS '08, pages 13:1–13:4, New York, NY, USA, 2008. ACM.
- [36] A. Clement, E. L. Wong, L. Alvisi, M. Dahlin, and M. Marchetti. Making byzantine fault tolerant systems tolerate byzantine faults. In *NSDI*, pages 153–168, 2009.
- [37] M. Correia, N. F. Neves, and P. Veríssimo. Bft-to: Intrusion tolerance with less replicas. *Comput. J.*, 56(6):693–715, 2013.
- [38] D. Costa, H. Madeira, J. Carreira, and J. Silva. Xception™: A software implemented fault injection tool. In A. Benso and P. Prinetto, editors, *Fault Injection Techniques and Tools for Embedded Systems Reliability Evaluation*, volume 23 of *Frontiers in Electronic Testing*, pages 125–139. Springer US, 2003.
- [39] D. Costa, T. Rilho, and H. Madeira. Joint evaluation of performance and robustness of a cots dbms through fault-injection. In *Dependable Systems and Networks, 2000. DSN 2000. Proceedings International Conference on*, pages 251–260, 2000.
- [40] J. A. Cowling, D. S. Myers, B. Liskov, R. Rodrigues, and L. Shrira. Hq replication: A hybrid quorum protocol for byzantine fault tolerance. In *OSDI*, pages 177–190, 2006.
- [41] V. Cunsolo, S. Distefano, A. Puliafito, and M. Scarpa. Volunteer computing and desktop cloud: The cloud@home paradigm. In *Network Computing and Applications, 2009. NCA 2009. Eighth IEEE International Symposium on*, pages 134–139, July 2009.
- [42] K. Danielson. Distinguishing cloud computing from utility computing, 2008.
- [43] S. Dawson, F. Jahanian, and T. Mitton. Orchestra: A fault injection environment for distributed systems. Technical report, In 26th International Symposium on Fault-Tolerant Computing (FTCS), 1996.
- [44] S. Dawson, F. Jahanian, and T. Mitton. Orchestra: a probing and fault injection environment for testing protocol implementations. In *Computer Performance and Dependability Symposium, 1996., Proceedings of IEEE International*, pages 56–, Sep 1996.

- [45] J. DeVale, P. Koopman, and D. Guttendorf. The ballista software robustness testing service. In *Testing Computer Software, 1999., In Proceedings of*, 1999.
- [46] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Commun. ACM*, 8(9):569–, Sept. 1965.
- [47] B. Dillenseger and E. Cecchet. Clif is a load injection framework. In *In Workshop on Middleware Benchmarking: Approaches, Results, Experiences, OOPSLA 2003*, 2003.
- [48] S. Dolev. *Self-stabilization*. MIT Press, Cambridge, MA, USA, 2000.
- [49] K. Driscoll. Real system failures. In *DASHlink*. NASA, 2012.
- [50] J. Durães, M. Vieira, and H. Madeira. Dependability Benchmarking of Web-Servers. In *Proceedings of 23rd International Conference on Computer Safety, Reliability and Security (Safecom’2004)*, pages 297–310, 2004.
- [51] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)*, 35(2):288–323, 1988.
- [52] F. Fich and E. Ruppert. Hundreds of impossibility results for distributed computing. *Distrib. Comput.*, 16(2-3):121–163, Sept. 2003.
- [53] M. J. Fischer, N. A. Lynch, and M. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985.
- [54] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles, SOSP ’03*, pages 29–43, New York, NY, USA, 2003. ACM.
- [55] J. Gray. Notes on data base operating systems. In *Operating Systems, An Advanced Course*, pages 393–481, London, UK, UK, 1978. Springer-Verlag.
- [56] R. Guerraoui, N. Knezevic, V. Quéma, and M. Vukolic. The next 700 bft protocols. In *EuroSys*, pages 363–376, 2010.
- [57] R. Guerraoui, N. Knezevic, V. Quema, and M. Vukolic. Stretching bft. Technical report, Technical Report EPFL-REPORT-149105, EPFL, 2011.
- [58] V. Hadzilacos and S. Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical report, 1994.



- [59] A. Haeberlen, P. Kouznetsov, and P. Druschel. The case for byzantine fault detection. In *Proceedings of the 2Nd Conference on Hot Topics in System Dependability - Volume 2*, HOTDEP'06, pages 5–5, Berkeley, CA, USA, 2006. USENIX Association.
- [60] S. Han, K. Shin, and H. Rosenberg. Doctor: an integrated software fault injection environment for distributed real-time systems. In *Computer Performance and Dependability Symposium, 1995. Proceedings., International*, pages 204–213, Apr 1995.
- [61] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990.
- [62] M.-C. Hsueh, T. Tsai, and R. Iyer. Fault injection techniques and tools. *Computer*, 30(4):75–82, Apr 1997.
- [63] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, pages 11–11, Berkeley, CA, USA, 2010. USENIX Association.
- [64] R. K. Iyer and D. Tang. Fault-tolerant computer system design. chapter Experimental Analysis of Computer System Dependability, pages 282–392. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1996.
- [65] J. M. J. Zhu and I. Pramanick. R3 - a framwork for availability benchmarking. In *Proc. Int. Conf. on Dependable Systems and Networks (DSN 2003)*, pages 86–87, San Francisco, CA, USA, 2003. DSN.
- [66] A. Jiwa, T. Hardjono, and J. Seberry. Beacons for authentication in distributed systems. *J. Comput. Secur.*, 4(1):81–96, Jan. 1996.
- [67] J. Jonsson and B. Kaliski. Public-key cryptography standards (pkcs) #1: Rsa cryptography specifications version 2.1. 2003.
- [68] M. Kalyanakrishnam, Z. Kalbarczyk, and R. Iyer. Failure data analysis of a lan of windows nt based computers. In *Proceedings of the 18th IEEE Symposium on Reliable Distributed Systems*, SRDS '99, Washington, DC, USA, 1999. IEEE Computer Society.
- [69] G. Kanawati, N. Kanawati, and J. Abraham. Ferrari: a flexible software-based fault and error injection system. *Computers, IEEE Transactions on*, 44(2):248–260, Feb 1995.

- [70] K. Kanoun and L. Spainhower. *Dependability Benchmarking for Computer Systems*. Wiley-IEEE Computer Society Pr, 2008.
- [71] R. Kapitza, J. Behl, C. Cachin, T. Distler, S. Kuhnle, S. V. Mohammadi, W. Schröder-Preikschat, and K. Stengel. Cheapbft: Resource-efficient byzantine fault tolerance. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 295–308. ACM, 2012.
- [72] R. King. Pivotal's head of products: We're moving to a multi-cloud world, 2014.
- [73] J. Kohlas, B. Meyer, and A. Schiper, editors. *Dependable Systems: Software, Computing, Networks, Research Results of the DICS Program*, volume 4028 of *Lecture Notes in Computer Science*. Springer, 2006.
- [74] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. L. Wong. Zyzzyva: Speculative byzantine fault tolerance. *ACM Trans. Comput. Syst.*, 27(4), 2009.
- [75] L. Lamport. The implementation of reliable distributed multiprocess systems. *Computer Networks*, 2:95–114, 1978.
- [76] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.
- [77] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.
- [78] L. Lamport. Paxos made simple. *SIGACT News*, 32(4):51–58, 2001.
- [79] L. Lamport. Lower bounds for asynchronous consensus. In *Future Directions in Distributed Computing*, pages 22–23, 2003.
- [80] L. Lamport, R. E. Shostak, and M. C. Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982.
- [81] B. W. Lampson. Atomic transactions. In *Distributed Systems - Architecture and Implementation, An Advanced Course*, pages 246–265, London, UK, UK, 1981. Springer-Verlag.
- [82] J. MacCormick, N. Murphy, M. Najork, C. A. Thekkath, and L. Zhou. Boxwood: Abstractions as the foundation for storage infrastructure. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6*, OSDI'04, pages 8–8, Berkeley, CA, USA, 2004. USENIX Association.

- [83] H. Madeira, K. Kanoun, J. Arlat, D. Costa, Y. Crouzet, M. D. Cin, P. Gil, N. Suri, and H. Madeira. Towards a framework for dependability benchmarking.
- [84] D. Mazières, M. Kaminsky, M. F. Kaashoek, and E. Witchel. Separating key management from file system security. In *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles, SOSP '99*, pages 124–139, 1999.
- [85] P. M. Mell and T. Grance. Sp 800-145. the nist definition of cloud computing. Technical report, Gaithersburg, MD, United States, 2011.
- [86] Z. Milosevic, M. Biely, and A. Schiper. Bounded delay in byzantine-tolerant state machine replication. In *SRDS*, pages 61–70, 2013.
- [87] C. NIST. The digital signature standard. *Commun. ACM*, 35(7):36–40, July 1992.
- [88] B. M. Oki and B. H. Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing, PODC '88*, pages 8–17, New York, NY, USA, 1988. ACM.
- [89] M. C. Pease, R. E. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *J. ACM*, 27(2):228–234, 1980.
- [90] M. K. Reiter. The rampart toolkit for building high-integrity services. In *Theory and Practice in Distributed Systems*, pages 99–110. Springer, 1995.
- [91] M. Rouse. What is a multi-cloud strategy, 2014.
- [92] A. Sangroya, D. Serrano, and S. Bouchenak. Benchmarking dependability of mapreduce systems. In *Reliable Distributed Systems (SRDS), 2012 IEEE 31st Symposium on*, pages 21–30, Oct 2012.
- [93] R. D. Schlichting and F. B. Schneider. Fail-stop processors: An approach to designing fault-tolerant computing systems. *ACM Trans. Comput. Syst.*, 1(3):222–238, Aug. 1983.
- [94] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4):299–319, Dec. 1990.

- [95] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)*, 22(4):299–319, 1990.
- [96] M. Serafini, P. Bokor, D. Dobre, M. Majuntke, and N. Suri. Scrooge: Reducing the costs of fast byzantine replication in presence of unresponsive replicas. In *DSN*, pages 353–362, 2010.
- [97] D. Serrano, S. Bouchenak, Y. Kouki, T. Ledoux, J. Lejeune, J. Sopena, L. Arantes, and P. Sens. Towards qos-oriented sla guarantees for online cloud services. In *Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on*, pages 50–57, May 2013.
- [98] A. Shoker, J.-P. Bahsoun, and M. Yabandeh. Improving independence of failures in bft. In *Network Computing and Applications (NCA), 2013 12th IEEE International Symposium on*, pages 227–234, Aug 2013.
- [99] A. Singh, T. Das, P. Maniatis, P. Druschel, and T. Roscoe. Bft protocols under fire. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI’08, pages 189–204, Berkeley, CA, USA, 2008.
- [100] A. Singh, P. Fonseca, P. Kuznetsov, R. Rodrigues, and P. Maniatis. Zeno: Eventually consistent byzantine-fault tolerance. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, NSDI’09, pages 169–184, Berkeley, CA, USA, 2009. USENIX Association.
- [101] D. Skeen and M. Stonebraker. A formal model of crash recovery in a distributed system. *Software Engineering, IEEE Transactions on*, SE-9(3):219–228, May 1983.
- [102] A. Tchana, N. De Palma, B. Dillenseger, and X. Etchevers. A self-scalable load injection service. *Software: Practice and Experience*, 45(5):613–632, 2015.
- [103] T. K. Tsai and R. K. Iyer. Measuring fault tolerance with the ftape fault injection tool. In *Proceedings of the 8th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation: Quantitative Evaluation of Computing and Communication Systems*, MMB ’95, pages 26–40, London, UK, UK, 1995.
- [104] R. van Renesse and F. B. Schneider. Chain replication for supporting high throughput and availability. In *OSDI*, pages 91–104, 2004.

- 
- [105] G. S. Veronese, M. Correia, A. N. Bessani, and L. C. Lung. Spin one's wheels? byzantine fault tolerance with a spinning primary. In *SRDS*, pages 135–144, 2009.
  - [106] G. S. Veronese, M. Correia, A. N. Bessani, L. C. Lung, and P. Veríssimo. Efficient byzantine fault-tolerance. *IEEE Trans. Computers*, 62(1):16–30, 2013.
  - [107] M. Vieira, N. Laranjeiro, and H. Madeira. Benchmarking the Robustness of Web Services. In *Proceedings of the 13th Pacific Rim International Symposium on Dependable Computing*, PRDC '07, pages 322–329, 2007.
  - [108] H. Williams. A modification of the rsa public-key encryption procedure (corresp.). *Information Theory, IEEE Transactions on*, 26(6):726–729, Nov 1980.
  - [109] T. Y. C. Woo and S. S. Lam. Authentication for distributed systems. *Computer*, 25(1):39–52, Jan. 1992.
  - [110] T. Wood, R. Singh, A. Venkataramani, P. Shenoy, and E. Cecchet. Zz and the art of practical bft execution. In *Proceedings of the sixth conference on Computer systems*, pages 123–138. ACM, 2011.
  - [111] F. J. Y. J. SONG and B. REED. Bft for the skeptics. In *BFTW3: Why? When? Where? Workshop on the theory and practice of Byzantine fault tolerance*, 2009.