Scalable Byzantine State Machine Replication: Designs, Techniques, and Implementations

Balaji Arun

Dissertation submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Computer Engineering

Binoy Ravindran, Chair
Dongyoon Lee
Cameron D. Patterson
Freek Verbeek
Haibo Zeng

May 10, 2021
Blacksburg, Virginia

Keywords: Byzantine Fault Tolerance, State Machine Replication
Copyright 2021, Balaji Arun

Scalable Byzantine State Machine Replication: Designs, Techniques, and Implementations

Balaji Arun

(ABSTRACT)

State machine replication (SMR) is one of the most widely studied and used methodology for building highly available distributed applications and services. SMR replicates a service across a set of computing hosts, and executes client operations on the replicas in an agreed-upon total order, ensuring linearizability of the replicated shared state. The problem of determining a total order reduces to one of computing *consensus*.

State-of-the-art consensus protocols are inadequate for newer classes of applications such as Blockchains and for geographically distributed infrastructures. The widely used Crash Fault Tolerance (CFT) fault model of consensus protocols is prone to malicious and adversarial behaviors as well as non-crash faults such as software bugs. The Byzantine fault-tolerance (BFT) model and its trust-based variant, the hybrid model, permit stronger failure adversaries. However, state-of-the-art Byzantine and hybrid consensus protocols have performance limitations in geographically distributed environments: they designate a primary replica for proposing total-orders, which becomes a bottleneck and yields sub-optimal latencies for far-away clients. Additionally, they do not scale to hundreds of replicas and provide consistent performance as the system size grows.

To overcome these limitations and develop highly scalable SMR solutions, this dissertation presents two *leaderless* consensus protocols, namely ezBFT and Dester, for the Byzantine and hybrid models, respectively. These protocols enable every replica to receive and order client commands. Additionally, they exchange command dependencies to collectively order commands without relying on a primary. Our experimental evaluations in a 7-node geographically distributed setup reveals that ezBFT improves client-side latency by as much as 40% over state-of-the-art BFT protocols including PBFT, FaB, and Zyzzyva. Dester, for the hybrid model, reduces latency by as much as 30% over ezBFT.

Next, the dissertation presents a new paradigm called DQBFT for designing consensus protocols that can scale to hundreds of nodes in geographically distributed environments. Since leaderless protocols exchange command dependencies, they do not scale to hundreds of nodes. DQBFT overcomes this scalability limitation by decentralizing only the heavy task of replicating commands and centralizing the process of ordering the commands. While DQBFT can be used to enhance existing primary-based protocols, Destiny is a hybrid instantiation of the DQBFT paradigm using linear communication for better scalability than naive instantiations. Experimental evaluations in a 193-node geographically distributed setup reveal that Destiny achieves $\approx 3\times$ better throughput and $\approx$50% better latency than state-of-the-art BFT protocols including Hotstuff, SBFT, and Hybster.

Lastly, the dissertation presents two techniques for designing and implementing BFT protocols with reduced development costs. The dissertation presents BUMBLEBEE, a methodology for manually transforming CFT protocols to tolerate Byzantine faults using trusted execution environments that are increasingly available in commodity hardware. BUMBLEBEE is based on the observation that CFT protocols are incapable of tolerating non-malicous non-crash faults, but they are nevertheless deployed in many production systems. BUMBLEBEE provides a Generic Algorithm that can represent protocols in both CFT and hybrid fault models, thus allowing easy construction of hybrid protocols using CFT protocols as baselines. The dissertation constructs hybrid instantiations of CFT protocols including Paxos, Raft, and M^2Paxos. Experimental evaluations of the hybrid variants reveal that they perform at par with native hybrid protocols, but incur a 30% overhead over their CFT counterparts.

Hybrid protocols rely on the integrity of trusted execution environments, which are increasingly subject to security exploits. To withstand exploits, the dissertation presents DuoBFT, a protocol that exposes both the BFT and hybrid fault models within a single consensus protocol. This enables consensus under both fault models within the same protocol and without additional redundancy, allowing DuoBFT to achieve the performance of hybrid protocols and the security of BFT protocols. Experimental evaluations reveal that DuoBFT achieves the best of both hybrid and BFT fault models with less than 10% overhead.

Scalable Byzantine State Machine Replication: Designs, Techniques, and Implementations

Balaji Arun

(GENERAL AUDIENCE ABSTRACT)

Computers are ubiquitous; they perform some of the most complex and safety-critical tasks such as controlling aircraft, managing the financial markets, and maintaining sensitive medical records. The undeniable fact is that computers are faulty. They are prone to crash and can behave arbitrarily. Even the most robust computers such as those that are sent to the outer space eventually fail. External phenomenon such as power outages and network disruptions affect their operation.

To make computing systems reliable, researchers and practitioners have long focussed on interconnecting many individual computers and programming them to effectively be duplicates of one another. This way when one computer fails in a system, the rest of the computers still ensure that the system as a whole is operational. Duplication requires that multiple computers effectively perform the same task. In order for multiple computers to perform the same task together, they should first agree on the task. More generally, since computing systems perform multiple tasks, they should agree on the sequence of tasks that they will individually perform and follow the agreement. This is what is known as the State Machine Replication technique.

State Machine Replication (SMR) is a powerful technique that is applicable to numerous computing applications. Blockchain systems, the technology behind the cryptocurrencies such as Bitcoin and Ethereum, uses the SMR technique. In the context of Blockchain, the added challenge in that some of the computers involved in SMR can be programmed by adversarial parties and could act in a way to jeopardize the integrity of the whole system. For Bitcoin and Ethereum, this could mean embezzlement of hundreds or even millions of dollars worth of cryptocurrencies. Certain SMR systems are capable of tolerating such intrusions and ensure system integrity. Such systems are deemed to be Byzantine tolerant.

This dissertation presents designs, techniques, and implementations of Byzantine State Machine Replication systems. The problems addressed in this dissertation are those that plague existing Byzantine SMR systems making them suboptimal for newer applications such as Blockchains. First, when computers that participate in SMR are spread around the world, their performance is dependent on the communication latencies between any two pair of computers. Second, the number of computers required is proportional the number of adversarial computers that need to be tolerated. Consequently, certain SMR systems for Blockchains require hundreds of computers to tolerate heavy adversarial behavior. Many existing SMR technique perform poorly under these scenarios. The techniques presented in this dissertation address various permutations of these challenges.

Dedication

To my family.

Acknowledgments

I would like to express my foremost gratitude to my advisor Professor Binoy Ravindran for his guidance and support throughout my PhD. He always encouraged me to solve hard research problems and supported me relentlessly throughout the challenges that arose. I am glad that he introduced me to the exciting world of distributed systems research during my Undergraduate days, which was the stepping stone for this exciting PhD journey.

I thank Professor Roberto Palmieri for co-advisement during my pre-PhD days. The reading groups he hosted and the long discussions with him contributed to my fundamental understanding of distributed systems and consensus protocols. Those conversations certainly inspired me to venture deeper into the subject.

Next, I thank Dr. Sebastiano Peluso for all the collaborations and advice during the initial part of my Graduate School. It was a pleasure working with him. Solving some of the most complex problems were much easier with him around.

I thank the committee members Professor Cameron Patterson, Professor Dongyoon Lee, Professor Freek Verbeek, and Professor Haibo Zeng for providing their valuable insights and feedback on my work. I thank Freek Verbeek for providing the insights and ideas pertaining to formal verification of my work.

Throughout my period at the System Software Research Group, I have had the pleasure of meeting some of the most talented and hardworking people who inspired me in many ways. I appreciate their company. Thank you for all the fun-filled Walleyball games, which were a big stress reliever. Thanks to Ho-Ren Chuang aka. Jack for his company and friendship.

I dedicate this dissertation to my parents and grandparents. This dissertation would not have been possible without their support, love, and dedication. My dad inspired me to pursue Graduate School and facilitated the journey towards this PhD. My mom ensured my wellbeing with weekly prepared meals and regular weekend feasts. In short, they assisted me with life in many ways, so I could focus more on research. I am extremely grateful for their support.

I cannot thank Lekha enough for her support throughout this endeavor. I am grateful for her company during the critical periods and the final and the most stressful days of my PhD journey. Our relationship certainly grew stronger during this time.

This work is supported in part by US National Science Foundation under grant CNS 1523558, Air Force Office of Scientific Research under grants FA9550-15-1-0098 and FA9550-16-1-0371, and Office of Naval Research under grant N00014-17-1-2297.

Contents

List of Figures **xiv**

List of Tables **xvi**

1 Introduction **1**

1.1 Motivation . . . 1

1.1.1 Replication . . . 2

1.1.2 Fault Models . . . 2

1.1.3 Challenges . . . 4

1.2 Summary of Research Contributions . . . 5

1.2.1 ezBFT . . . 7

1.2.2 Dester . . . 8

1.2.3 Bumblebee . . . 8

1.2.4 DQBFT . . . 8

1.2.5 Destiny . . . 9

1.2.6 Flexible Hybrid Protocols . . . 9

1.2.7 DuoBFT . . . 9

1.2.8 Folly . . . 10

1.3 Dissertation Organization . . . 10

2 Background **11**

2.1 State Machine Replication . . . 11

2.2 The Consensus Problem . . . 11

2.3 Timing Models . . . 12

2.4 Fault Models . . . 12

2.4.1 The Crash Fault Model . . . 13

2.4.2 Byzantine Fault Model . . . 13
2.4.3 Hybrid Faults . . . 14
2.5 Byzantine Consensus . . . 14
2.5.1 Consensus with Trusted Subsystems . . . 15
2.6 Example: Comparison of Fault Models . . . 16
2.7 The BFT Quorum System . . . 20

3 Related Work **22**
3.1 The Rise of Fault-Tolerant Distributed Computing . . . 22
3.2 Byzantine Fault Tolerance . . . 23
3.3 Crash Fault Tolerance . . . 27
3.3.1 Paxos-like Protocols . . . 27
3.3.2 Leaderless Protocols . . . 27
3.3.3 Hybrid Fault-Tolerant Protocols . . . 28
3.4 Protocol Transformations . . . 29
3.5 Summary of Past and Related Protocols . . . 29

4 ezBFT **31**
4.1 Introduction . . . 31
4.2 Protocol Overview . . . 33
4.3 Protocol Design . . . 35
4.3.1 System Model . . . 35
4.3.2 The Fast Path Protocol . . . 35
4.3.3 Execution Protocol . . . 38
4.3.4 The Slow Path Protocol . . . 38
4.3.5 Triggering Owner Changes . . . 42
4.3.6 The Owner Change Protocol . . . 43
4.3.7 Correctness . . . 44
4.4 Evaluation . . . 45

4.4.1 Client-side Latency . . . 45
4.4.2 Client Scalability . . . 48
4.4.3 Server-side Throughput . . . 48
4.5 Conclusions . . . 49

5 Dester **50**
5.1 Overview . . . 51
5.1.1 Background . . . 51
5.1.2 Intuition and Contributions . . . 52
5.2 Preliminaries . . . 53
5.2.1 System Model . . . 53
5.2.2 Fast Decisions and Quorums . . . 53
5.3 Protocol Design . . . 55
5.3.1 The TruDep Subsystem . . . 55
5.3.2 USIG Subsystem . . . 56
5.3.3 Overview . . . 57
5.3.4 Protocol Details . . . 57
5.3.5 Correctness . . . 64
5.4 Evaluation . . . 66
5.4.1 Client-side Latency . . . 67
5.4.2 Server-side Throughput . . . 69
5.4.3 Scalability . . . 70
5.4.4 Fault Tolerance . . . 71
5.4.5 Summary of Results . . . 71
5.5 Conclusions . . . 72

6 Bumblebee **73**
6.1 Preliminaries . . . 74
6.1.1 Model . . . 74

6.1.2 Communication Predicates . . . 75
6.1.3 Consensus Problem . . . 76
6.2 Transformation Methodology . . . 76
6.2.1 Lifting Abstractions: Generic Algorithm . . . 77
6.2.2 Correctness of Generic Algorithm . . . 79
6.2.3 Instantiations of Parameters . . . 81
6.2.4 $\mathcal{P}_{good}$ and $\mathcal{P}_{cons}$ Implementations . . . 82
6.2.5 Transformation Steps . . . 84
6.3 Examples . . . 85
6.3.1 Paxos Transformation . . . 85
6.3.2 Raft Transformation . . . 86
6.3.3 M^2Paxos Transformation . . . 86
6.4 Evaluation . . . 87
6.4.1 USIG . . . 87
6.4.2 Performance of Hybrids . . . 88
6.4.3 Scalability of Hybrids . . . 89
6.4.4 Fault Tolerance of Hybrids . . . 89
6.4.5 etcd . . . 90
6.5 Conclusions and Limitations . . . 91

7 DQBFT **92**
7.1 Introduction . . . 92
7.1.1 Towards Partial Decentralization . . . 92
7.2 Protocol Design and Description . . . 93
7.2.1 Design . . . 93
7.2.2 Protocol Description . . . 95
7.2.3 Correctness . . . 99
7.3 Conclusions . . . 101

8 Destiny **103**

8.1 Protocol Design . . . 104

8.1.1 System Model and Cryptography . . . 105

8.1.2 The ThreshSign Subsystem . . . 105

8.1.3 Linear Hybster . . . 106

8.1.4 Protocol . . . 109

8.2 Evaluation . . . 110

8.2.1 Performance . . . 113

8.2.2 Scalability . . . 113

8.2.3 Batching . . . 114

8.2.4 Availability under Failures . . . 114

8.2.5 Summary of Results . . . 115

8.3 Conclusions . . . 115

9 Flexible MinBFT **116**

9.1 Revisiting MinBFT . . . 116

9.1.1 USIG Trusted Service . . . 116

9.1.2 The MinBFT Protocol . . . 117

9.2 MinBFT with Flexible Quorums . . . 118

9.3 Correctness of Flexible MinBFT . . . 120

10 DuoBFT **126**

10.1 Introduction . . . 126

10.1.1 Towards providing a dual fault model . . . 126

10.2 Preliminaries . . . 128

10.3 Protocol Design . . . 130

10.3.1 Preliminaries . . . 130

10.3.2 Protocol . . . 131

10.3.3 Learners . . . 134

10.3.4 Proof . . . 135
10.3.5 Optimizations . . . 138
10.4 Discussion . . . 138
10.5 Evaluation . . . 139
10.6 Conclusions . . . 141

11 Folly **142**
11.1 Preliminaries . . . 143
11.1.1 Ordering Guarantees in BFT protocols . . . 143
11.1.2 Byzantine Ordered Consensus . . . 143
11.1.3 Pompe . . . 144
11.2 Overview . . . 145
11.2.1 System Model . . . 145
11.2.2 Key Techniques . . . 146
11.3 Protocol . . . 147
11.3.1 Pre-Consensus Ordering . . . 147
11.3.2 Consensus . . . 148
11.3.3 Post-Consensus Procedure and Execution . . . 149
11.3.4 Correctness . . . 149
11.4 Evaluation . . . 151
11.5 Conclusions . . . 152

12 Conclusions and Future Work **153**
12.1 Future Work . . . 155
12.1.1 Additional Fault Models within DuoBFT . . . 155
12.1.2 Byzantine Ordered Consensus with DQBFT paradigm . . . 156
12.1.3 Generic Algorithm for Leaderless Protocol Transformation . . . 156
12.1.4 Automatic Bumblebee Transformations . . . 156
12.1.5 Flexible Leaderless Protocols . . . 157

Bibliography **158**

Appendices **172**

Appendix A TLA+ Specification of ezBFT **173**

List of Figures

2.1 Failure hierarchy. 14

2.2 An example of normal execution in three different failure models. 17

4.1 An example of a fast path execution. 37

4.2 ezBFT: An example of a slow path execution. 40

4.3 An example of a slow path with a faulty replica. 41

4.4 Median latencies for Experiment 1. All primaries are in US-East-1 region. The latency is shown per region as recorded by the clients in that region. Error bars indicate 99%ile. 46

4.5 Best case for Zyzzyva and the effect of moving primary to different regions. Experiments reveal ezBFT's effectiveness. Legend entries show primary's location in parenthesis. 47

4.6 Latency per location while varying the number of connected clients (1 – 100) per region. 48

4.7 Throughput of ezBFT and competitor BFT protocols. 49

5.1 Classic Majority Quorum vs. Dester Fast Quorum. Despite loss of f (shaded) nodes, the remaining nodes constitutes a majority in a fast quorum but not so in a classic quorum. 54

5.2 Dester: An example execution. 63

5.3 Client-side latency at each of the seven regions. 68

5.4 Throughput of Dester and competitor BFT protocols. 70

5.5 Scalability of Protocols. 71

5.6 Dester Fault Tolerance. 72

6.1 Throughput of SGX certifier vs Go-based certifier. 88

6.2 Throughput vs latency of Hybrid protocols. 89

6.3 Throughput vs latency of Hybrid Raft vs Original Raft as number of nodes is varied. In the legend, the number inside the parenthesis indicate the number of nodes in that experiment. . . . 90

6.4 Original etcd vs Hybrid Raft etcd. . . . 91

7.1 DQBFT's separation of replication and ordering. . . . 95

7.2 DQBFT Protocol Execution using the PBFT protocol. . . . 96

8.1 Normal communication steps in Linear Hybster and Destiny. . . . 104

8.2 Destiny Protocol Execution. . . . 107

8.3 Evaluation of DQBFT and other consensus protocols. . . . 112

8.4 Throughput under leader failure. . . . 113

9.1 MinBFT Normal Execution. . . . 117

10.1 DuoBFT Normal Protocol. . . . 131

10.2 DuoBFT View Change Protocol. . . . 132

10.3 DuoBFT Commit Rules. . . . 134

10.4 Latency vs. Throughput with Batch Size of 1. . . . 140

10.5 Latency vs. Throughput with Batch Size of 64. . . . 141

11.1 Folly example execution. Command c is proposed by r_1 to assign a timestamp. Despite r_3 returning a timestamp that is too large, picking the median value ensures that the timestamp returned by the correct replicas is assigned for c. Note that the ordering phase takes only four communication steps. . . . 147

11.2 Latency versus Throughput. . . . 151

List of Tables

2.1 Summary of labels used by different protocols and their meaning. 19

3.1 Table 3.1: Summary of Past and Related Protocols. The dissertation's protocols are also included in the table (boldfaced) for perspective. 30

4.1 Zyzzyva's [83] latencies (in *ms*) in a geo-distributed deployment with primary at different locations. . Columns indicate the primary's location. Rows indicate average client-side latency for commands issued from that region. For example, the entry at the 4th row and the 3rd column shows the client-side latency for commands issued from India to the primary in Japan. Lowest latency per primary location is highlighted. 32

8.1 Maximum tolerated failures for different N.. 111

Chapter 1

Introduction

1.1 Motivation

Computers are fault-prone: they often crash, slow down, and become corrupt. Yet, they play a crucial role in many mission-critical systems in numerous application domains (e.g., cloud systems [5, 29, 47, 79], cyber-physical systems [27, 137], cryptocurrency [118, 139]). Building reliable systems using fault-prone computers has a very long history [38, 104]. The rise of networked computing systems as exemplified by the Internet presents new sets of challenges as they introduce another dimension of unreliability: it is difficult to ensure timely and reliable communication between computers over a network. With the explosion and commodification of the Internet, the risk of malicious actors compromising enterprise infrastructure systems via cyberattacks is becoming more pronounced [20]. The challenge in building reliable and safe networked systems that can withstand such complex and uncertain behaviors has accelerated research in secure, fault-tolerant distributed computing [38].

The last decade has witnessed an ever-increasing proliferation of online services enabled by ubiquitous cloud platforms. Cloud providers charge for per unit of computational resource utilized, thus minimizing the total cost of ownership of cloud resources. This has enabled businesses ranging from startups to large-scale enterprises to take advantage of cloud's flexibility to develop highly available (i.e., round-the-clock) applications and services. Moreover, the ubiquity of globally distributed datacenters allows computational resources to be instantiated at different datacenters around the world, enabling new classes of geographically distributed systems [2, 4]. Leveraging these newer capabilities to build high-performance distributed systems is a challenge.

Over the past decade, a new paradigm of distributed systems called Blockchain [118] that involves multiple untrusted parties communicating over a network to provide a completely decentralized service to users has gained significant traction. The idea, which was initially popularized by Bitcoin [118], a cryptocurrency, has been used to solve numerous problems that involve multiple untrusted parties, particularly, in financial application domains [139]. This new paradigm has prompted researchers to revisit and reexamine the literature on fault-tolerant distributed systems from a decentralization perspective.

1.1.1 Replication

Replication has been the primary solution for solving the reliability problem in distributed systems. In this solution approach, copies of a software are run on multiple host computers, but they provide the illusion of a single computer to an end user. Failure of a few hosts does not cause total loss in availability of the service as other hosts can still serve the user. Replication has also shown to be effective in improving the scalability of software, as multiple hosts can process larger computational workloads than a single host. Replication has been used to build highly reliable systems in a number of application domains including safety-critical systems such as power grids [27], cyber-physical systems such as avionics [78], and business- or mission-critical systems such as databases [3, 47], key-value stores [5], and transactional systems [29].

Replication is particularly challenging when sharing of the software's state (e.g., database tables, indices, data structures) is involved. Replicas, i.e., computing hosts that maintain a copy of the shared state, must communicate and coordinate to ensure consistency (i.e., linearizability [73])[1] of the (replicated) shared state. Hosts and networks that are faulty, unreliable, and even malicious complicate the coordination that is necessary for ensuring state consistency. This complexity only exacerbates when hosts are spread across different geographic locations around the world and communicate and coordinate via high-latency network links. Yet, maintaining consistency (i.e., linearizability) is necessary to ensure that different replicas do not have different versions of the shared state.

The state machine replication (SMR) approach has been widely used to build fault-tolerant replicated services [88, 125]. In this approach, every replica in the system starts from the same initial state, execute client-issued operations (e.g., look-up, update operations in a key-value store) in the same order – i.e., a *total order* – across all replicas, and thereby reach the same final state, ensuring linearizability. Consensus protocols can be used to build replicated state machines [125]. Simply put, consensus is the problem of selecting a value among a set of proposed values, which allows a total order to be determined. Paxos [95] is a well-known implementation of consensus that enables SMR. Implementations of consensus exist in many practical systems [27, 79]

1.1.2 Fault Models

While distributed systems can be designed to cope with different kinds of faults, categorizing the faults allows for consensus protocol designs that efficiently tolerate a given class of faults,

[1] The literature on distributed systems defines a number of different forms of consistency, e.g., linearizability [73], sequential consistency [25], eventual consistency [36]. This dissertation focuses on linearizability as it enables high programmability: in this form of consistency, the behavior of a concurrent software object (that encapsulates a state) is always indistinguishable from that of its sequential version, which enables no difference in how interfacing applications are programmed and debugged. We refer the reader to [11] for an excellent exposition of different consistency models.

called a fault model. Historically, two fault models have extensively been studied in the literature: the crash fault model (CFT) and the Byzantine fault model (BFT). The CFT model includes faults such as machine crashes and network outages, and requires a minimum of $N = 2f+1$ replicas to tolerate a maximum of f failures. Example CFT consensus protocols include Paxos [95], EPaxos [117], Caesar [21], and M^2Paxos [122]. Many SMR systems that are in wide-spread production use by enterprises today are based on the CFT model (e.g., Google's Spanner [2, 47], Amazon's Dynamo [54], Microsoft's Azure [4], CockroachDB [3], ZooKeeper [79]).

In contrast, the BFT model permits a stronger failure adversary. In addition to crash faults and network outages, this model covers malicious and adversarial behaviors, as well as non-crash faults such as hardware errors, corrupted data, software bugs, data losses, and intrusions. BFT protocols require a minimum of $N = 3f + 1$ replicas to tolerate a maximum of f Byzantine failures. Example BFT consensus protocols include PBFT [41, 42], FaB [108, 109], Zyzzyva [83], and Aliph [26].

Owing to their tolerance for malicious behaviors, BFT protocols have been used to solve consensus in Blockchain settings [31]. A Blockchain consists of a *ledger*, a data structure that is replicated across a set of nodes, on which transactions (i.e., a sequence of operations) issued by clients, are executed in an agreed-upon total order, which ensures linearizability of the replicated (data structure) state. The problem of determining a total order reduces to one of consensus. For the class of *permissioned* ledgers, nodes are untrusted, so the problem further reduces to BFT consensus.[2] As a matter of fact, state machine replication is the most common way to implement permissioned ledgers [31].

The BFT model, in which the adversary is allowed to take control of faulty machines as well as the network in a coordinated way, is superfluous when such a strong adversary is unlikely. Precisely, for a class of distributed systems such as storage and database systems that are maintained by a single entity (e.g., a single enterprise organization), non-crash faults typically include errors in the hardware, stale or corrupted data from storage systems, memory errors caused by physical effects, bugs in software, hardware faults due to ever smaller circuits, and human mistakes that cause state corruptions and data losses. Therefore, for such systems, the trust-based Byzantine fault model [49] can yield greater resource savings while still tolerating Byzantine faults. This can particularly improve the reliability of storage and database systems, without the need for a special consistency verification process [111] or fault-recovery mechanisms [14]. Trust-based BFT protocols, called *hybrid* protocols, employ a small trusted subsystem that can only fail by crashing, reduce the number of replicas required for replication from $3f + 1$ to $2f + 1$, matching the CFT requirement. Though some early hybrid protocols used ASICs and FPGAs, which resulted in high development complexity and very low performance [82, 98], the advent of trusted execution environments (TEEs) within commodity hardware such as Intel SGX [52] and ARM Trustzone [100] has

[2] In contrast, in permissionless ledgers, which seeks to avoid a central authority, any node can participate [31]. In permissioned ledgers, nodes are untrusted, but openness and anonymity are not goals [31].

significantly reduced the performance overheads for hybrid protocols [32]. A software-based subsystem can be safe-guarded within such trusted environments providing the required level of trust.

1.1.3 Challenges

State-of-the-art BFT protocols have weaknesses that inhibit their performance and scalability in geographically distributed (*geo-distributed* hereafter) systems. On the one hand, for applications such as Blockchain systems that operate at the scale of hundreds of nodes, many existing BFT protocols do not scale: SBFT [64], PBFT [41], and Hybster [32] reveal rapidly declining performance as system size grows. On the other hand, scalable BFT protocols such as RCC [70] and MirBFT [129] do not consider node slowdowns that are typical in datacenter deployments. Hence, the maximum performance of these protocols are limited by the performance of the slowest node.

Geo-distributed deployments of BFT and hybrid protocols pose an important challenge: taming the high communication latency. Since replicas need to communicate with each other and with clients to reach consensus, the number of communication steps incurred directly impacts latency, as each step involves sending messages to potentially distant nodes. Moreover, state-of-the-art BFT and hybrid protocols are primary-based: a replica in the system is bestowed the *primary* status and is responsible for assigning the total-order for executing the requests issued by clients. This can result in sub-optimal performance: clients may originate from different regions around the world, but their requests must be sent to the primary even when other replica nodes are present much closer to the client than the primary. Thus, such clients will perceive sub-optimal latencies for processing their requests. Second, the primary must handle all client requests, irrespective of where they originate, thereby performing more work than other replicas. This causes load imbalance among the replicas, causing the primary to become a resource bottleneck, hampering scalability.

A *leaderless* protocol can solve the aforementioned problems. A client can send its requests to the geographically-closest replica and can continue to do so as long as that replica is *correct* (i.e., non-Byzantine). The replica can undertake the task of finding an order among concurrent *conflicting* requests in the system, execute the request on the shared state, and return the result. (A pair of requests are conflicting if they operate on the same part of the shared state.) Since individual replicas share the workload as equal contributors in the system, performance scales. Leaderless consensus protocols suited for geo-distributed deployments with bounded communication steps [21, 117] have previously been studied for the CFT model. These protocols track dependencies between client commands to construct a total order among conflicting commands. However, protocols that focus on the aforementioned characteristics, namely, leaderless operation with bounded communication and command commutativity, do not exist for the BFT model.

A plethora of distributed systems in production use today such as Google Spanner [47],

etcd [5], CockroachDB [3], and Azure CosmosDB [4] adopt CFT consensus protocols. However, these systems can tolerate many of the fatal non-crash faults [48] by adopting a hybrid protocol. Today, this requires swapping out the existing CFT protocol implementation for a hybrid fault-tolerant one. This causes unnecessary friction in software development as the new protocol would require rigorous testing and validation. If, instead, additional mechanisms are added to CFT protocols to tolerate Byzantine failures, then development would become more incremental than a "ground-up" construction and thus would ease and facilitate adoption in practice. In doing so, the crux of CFT protocols should be maintained: requiring only $2f + 1$ replicas to tolerate a maximum of f failures and reduced complexity compared to other fault models. In addition, the trusted subsystem must be minimal enough to easily ensure that their implementations are bug-free.

While the dependency-based ordering mechanism for leaderless protocols can improve performance in geo-distributed deployments, it can prevent scalability to hundreds of replicas. The performance of leaderless protocols is inversely proportional to the rate of conflicting requests, since highly conflicting commands produce large dependency graphs that must be consistently shared by replicas and processed. Hence, mechanisms that do not introduce additional complexities to the consensus protocols such as dependency tracking is key for scalability.

The hybrid model depends on the integrity of TEEs to ensure correctness of consensus, in particular, that the TEE can only fail by crashing, to ensure linearizability. However, recent security exploits [131] of TEEs such as Intel SGX question this presumption. Emerging mitigations [131] of such exploits show promise in this direction. While there are convincing reasons to build protocols using such trust mechanisms, care should be taken to ensure that correctness of consensus is preserved at all times.

For certain class of Blockchain applications [101, 139], it is necessary to reach consensus on commands as soon as they are received to ensure that the system is fair to all the clients. However, primary-based BFT protocols can unfairly discriminate certain clients by choosing to delay their requests. Byzantine Ordered Consensus protocols [141] prevent such behavior and allow replicas to define an order that the primary must follow. Existing approaches [141] in this space are slow because they involve the sequential execution of two long protocol stages, which can increase client-side latencies.

This dissertation addresses these challenges.

1.2 Summary of Research Contributions

This dissertation underscores that state-of-the-art SMR solutions have performance and scalability limitations in geo-distributed deployments, among others, and argues for BFT protocol designs that are not primary-based to overcome such limitations.

The dissertation presents BFT protocols for geo-distributed deployments in three categories:

- **Leaderless protocols**. The dissertation presents two sets of techniques for building high performance, leaderless consensus protocols for the BFT and hybrid models. The protocols allow any replica to order client commands by leveraging the dependency-based ordering technique. Replicas track the dependencies between client commands based on command *conflicts*, and use that information to decide the order in which to execute commands. The dissertation presents two BFT protocols, ezBFT (Chapter 4) and DESTER (Chapter 5) that use this technique.

- **Partially decentralized protocols.** The dissertation presents a new paradigm for building high performance, highly scalable consensus protocols for the BFT and hybrid models. The key technique introduced by this paradigm is the separation of the replication and ordering steps that occur in tandem in primary-based consensus protocols and executing those two steps concurrently. Unlike leaderless protocols, these partially decentralized protocols do not track dependencies and thus, are not affected by conflicts. This intrinsic simplicity in design allows protocols under this paradigm to scale to hundreds of replicas. The dissertation presents two protocols, DQPBFT (Chapter 7) and Destiny (Chapter 8) that use this technique.

- **Byzantine Ordered Consensus protocol.** The dissertation proposes a low-latency consensus protocol for the Byzantine Ordered Consensus model [141]. While ensuring fairness in ordering is vital, this property comes at the cost of higher latencies in existing protocols [141]. To overcome this, the dissertation proposes to minimize the number of communication steps in the protocol's critical path, even though the overall number of communication steps remain the same. Removing communication steps from the critical path improves client-perceived latency. The dissertation presents a protocol, Folly (Chapter 11) that uses this technique.

The dissertation also proposes a complementary technique to aid in the design and development of hybrid protocols. While there exists a plethora of distributed systems in production use today that adopt CFT consensus protocols (e.g., Google Spanner [47], etcd [5], CockroachDB [3]), they cannot tolerate non-crash non-Byzantine faults that are becoming increasingly common. For such systems, the dissertation presents an approach called BUMBLEBEE (Chapter 6), to transform CFT protocols to tolerate Byzantine faults with the help of trusted hardware.

The BFT and hybrid models have a crucial performance versus trusted subsystem trade-off. BFT protocols are slower but do not require any trusted subsystem, while hybrid protocols are faster but requires TEEs. In order to provide the best of both these fault models, the dissertation presents a single protocol called DuoBFT (Chapter 10) that simultaneously enables consensus under the two fault models.

In summary, the dissertation makes the following research contributions:

- ezBFT [23]: a leaderless protocol for solving BFT consensus quickly, i.e., in three communication steps, under low contention. This is done by exploiting commutativity of client requests and by exchanging request dependencies (Chapter 4).
- Dester: a leaderless protocol for solving hybrid consensus quickly, i.e., in three communication steps, under low contention. This is done by leveraging the integrity of TEEs, exploiting commutativity of client requests, and by exchanging request dependencies (Chapter 5).
- Bumblebee: a three-step methodology for transforming existing CFT protocols to tolerate Byzantine faults using trusted subsystems present in commodity hardware. This enables leveraging the investment in CFT protocols for developing BFT protocols (Chapter 6).
- DQBFT: a new paradigm for designing BFT protocols that divides consensus into two parts: decentralizing the heavy task of replicating and partially ordering commands, and centralizing the process of totally ordering the commands (Chapter 7).
- Destiny: a protocol instantiation of the DQBFT paradigm that leverages a trusted subsystem to reduce consensus costs (Chapter 8).
- Hybrid Flexible Quorums [24]: a flexible quorum technique for the hybrid fault model that separates the relationship between the system size and the number of tolerated faults and reduces the size of normal quorums while increasing the size of view-change quorums (Chapter 9).
- DuoBFT [24]: a BFT protocol with two features: low-cost resilience and dual fault assumptions. It exposes both the BFT and hybrid fault models to the clients and allows the clients to select the model that suits the application (Chapter 10).
- Folly: a Byzantine Ordered Consensus protocol that decreases the number of communication steps required to assign ordering indicators to commands before the consensus phase, improving performance (Chapter 11).

1.2.1 ezBFT

First, the dissertation presents ezBFT, a leaderless BFT protocol that enables every replica in the system to process the requests received from the clients. Doing so (i) significantly reduces the client-side latency, (ii) distributes the load across replicas, and (iii) tolerates faults more effectively. Importantly, ezBFT delivers requests in *three* communication steps in normal operating conditions. To enable leaderless operation, ezBFT exploits a particular characteristic of client commands: *interference*. In the absence of concurrent interfering commands, ezBFT's clients receive a reply in an optimal three communication steps. To understand how ezBFT fares against state-of-the-art BFT protocols, we implemented ezBFT

and conducted an experimental evaluation using Amazon's AWS EC2 infrastructure [15], deploying the implementations in different sets of geographical regions. Our evaluation reveals that ezBFT improves client-side latency by as much as 40% over prior BFT protocols including PBFT, FaB, and Zyzzyva.

1.2.2 Dester

Second, the dissertation presents DESTER, a leaderless, hybrid protocol that is built from the *ground-up* for achieving high performance in geo-distributed environments. DESTER incorporates a novel trusted subsystem, called TRUDEP, for achieving low latency, and allows any replica to propose and commit client commands in two communication steps in most practical situations, while clients minimize latency by sending commands to the geographically-nearest replica. We analyze DESTER, and establish its correctness. By performing an experimental evaluation using the AWS EC2 infrastructure, we show that DESTER provides up to 60% reduction in latency for most clients and better availability guarantees than state-of-the-art BFT protocols. In addition, DESTER provides $1.25\times - 2.4\times$ better throughput than existing throughput-oriented BFT protocols at system sizes up to 19 replicas.

1.2.3 Bumblebee

Third, the dissertation presents BUMBLEBEE, a three-step manual methodology for transforming CFT protocols to tolerate Byzantine faults by incorporating a trusted subsystem realized using TEEs. The methodology introduces a parameterized Generic Algorithm that can be used to instantiate both CFT and hybrid protocols by substituting different parameters. By manually representing a CFT protocol using the Generic Algorithm, we show that it is possible to transform the CFT protocol into a hybrid protocol with very little effort. We demonstrate the feasibility of the BUMBLEBEE approach by transforming common CFT protocols including Paxos, Raft and M^2Paxos. into their hybrid counterparts. Our experimental evaluations using the AWS EC2 infrastructure show that the transformed protocols exhibit performances similar to native hybrid protocols. By incorporating the BUMBLEBEE approach in the consensus protocol of a popular production use distributed key-value store, etcd [5], we show that overheads due to hybridization are less than 30% in terms of throughput.

1.2.4 DQBFT

DQBFT is a new paradigm for designing BFT protocols that addresses three major performance and scalability challenges that plague past protocols: (i) high communication costs to reach geo-distributed agreement, (ii) uneven resource utilization hampering performance,

and (iii) performance degradation under varying node and network conditions and high-contention workloads. Specifically, DQBFT divides consensus into two parts: 1) durable command replication without a global order, and 2) consistent global ordering of commands across all replicas. DQBFT achieves this by decentralizing the heavy task of replicating commands and centralizing the process of ordering the commands. We show that the technique is general and applicable to primary-based BFT and hybrid protocols.

1.2.5 Destiny

We instanstiate a BFT protocol using the DQBFT paradigm: Destiny. Destiny combines three techniques to achieve high performance and scalability: i) using a trusted subsystem to decrease consensus's quorum size, ii) using collectors and threshold signatures to attain linear communication costs, and iii) reducing client communication. Our experimental evaluations using a 193 node intercontinental setting reveal that Destiny achieves significant performance gains over prior art including Hybster [32], Hotstuff [140], and SBFT [64]: ≈3x better throughput and ≈50% better latency. Furthermore, the evaluations also show that Destiny is scalable to nearly 300 replicas.

1.2.6 Flexible Hybrid Protocols

We enable a notion called Flexible Quorums in the hybrid fault model by revisiting the quorum intersection requirement in hybrid protocols. We show that quorums with a view need not intersect while quorums across views must intersect to ensure safety. With this intuition, we decrease the size of normal quorums such that they do not intersect, while increasing the size of view-change quorums to ensure that they intersect with all other quorums. Consequently, the dependency between the system size and the number of tolerated faults is relaxed, allowing a variable number of faults to be tolerated for a given system size. We apply the Flexible Quorums technique to MinBFT [135], a state-of-the-art hybrid protocol, to produce Flexible MinBFT. Our experimental evaluations show that Flexible MinBFT improves throughput by up to 30% over MinBFT.

1.2.7 DuoBFT

Next, the dissertation presents DuoBFT, a BFT protocol that provides two features: low-cost resilience and dual fault assumption. First, by enhancing a fraction of replicas in the system with trusted components, DuoBFT enables commit decisions in the hybrid fault model with quorums that are about half the size of regular Byzantine quorums. Second, DuoBFT exposes both the hybrid and BFT fault models to the clients and allows them to make commit decisions under either of those models. By performing an experimental

evaluation study, we show that DuoBFT provides the best of both hybrid and BFT fault models with minimal overhead of <10% in terms of throughput.

1.2.8 Folly

Finally, the dissertation presents a new Byzantine Ordered Consensus protocol, Folly, to address the latency problem that uses only four communication steps to assign ordering indicators to commands before the consensus phase. The original Byzantine Ordered Consensus protocol, Pompe [141], uses six communication steps and fine-tuned delays to order commands. By reducing six communication steps to four steps, Folly yields significant latency savings of up to 35% over Pompe, while providing similar throughput as Pompe. Thus, Folly better competes with traditional BFT protocols than Pompe.

1.3 Dissertation Organization

The rest of the dissertation is organized as follows. Chapter 2 discusses the necessary background to better understand the dissertation's contributions including the consensus problem definition, the CFT and BFT fault models, and the BFT quorum system and consensus. Chapter 3 discusses past and related works, provides a taxonomy of those protocols that is relevant from the dissertation's perspective, and also catalogues the dissertation's protocols in the taxonomy.

Chapter 4 describes, analyzes, and evaluates ezBFT. Chapter 5 describes, analyzes, and evaluates DESTER. Chapter 6 describes, analyzes, and evaluates BUMBLEBEE.

Chapter 7 describes the DQBFT paradigm. Chapter 8 describes Destiny, analyzes it, and evaluates it along with other DQBFT protocols.

Chapter 9 presents Flexible Hybrid Quorums and the Flexible MinBFT protocol. Chapter 10 describes, analyzes, and evaluates DuoBFT along with the Flexible MinBFT protocol.

Chapter 11 describes the Byzantine Ordered Consensus problem, and presents, analyzes, and evaluates Folly.

Finally, Chapter 12 concludes the dissertation and outlines direction for future research.

Chapter 2

Background

This chapter provides the adequate background information for understanding the rest of this dissertation. Section 2.1 describes the problem of replication in distributed systems, while Section 2.2 explains the problem of consensus. Section 2.4 describes some prolific and relevant fault models employed in distributed systems. Finally, Section 2.6 compares and contrasts, with examples, three fault models, namely CFT, BFT, and Hybrids, that are important for understanding the contributions of this dissertation.

2.1 State Machine Replication

State Machine Replication is a general paradigm for implementing distributed systems that replicate shared state. Under this paradigm, a set of nodes collectively participate to implement a single state machine. Each of the nodes have the same initial state. The state machine moves from one state to another by executing client commands, such as database transactions and key-value store operations. The commands are replicated and executed atomically in the same order across all the nodes. Atomic execution is vital to ensure that either a state transition is successful or unsuccessful, without leaving any intermediate state. The state machine should function despite some faulty nodes to provide a useful service. Consensus is a primitive that is most commonly used to implement State Machine Replication.

2.2 The Consensus Problem

Consensus is the problem of agreeing on a value among a set of values proposed by different nodes. A consensus protocol provides the following properties [95]:

- *Nontriviality*: A request can only be agreed upon if it was proposed by a client.
- *Stability*: Once a request has been agreed for execution, no replica can revert its decision.
- *Consistency*: Two different replicas cannot agree on different request for execution.

2.3 Timing Models

Since nodes in a distributed system communicate by message-passing to ensure that the replicated state is consistent, the assumptions on network delays and local clocks decide the extent to which the state can be kept consistent in the face of failures. The literature defines three timing models [38]: synchrony, asynchrony, and eventual synchrony.

Under synchrony, there is a bound on the maximum network delays and clock of different nodes are tightly synchronized. This allows nodes to identify faulty nodes by depending on the network delays. If a node p does not receive a message in time from another node q, then q is faulty. The synchrony assumption allows for consensus protocols that agree on a value with messages from a majority of the processes. Thus, protocols under this model can tolerate up to minority arbitrary failures.

The asynchrony timing model assumes no bound in maximum network delays or synchronized clocks between nodes. The famous FLP result [60] shows that it is impossible to solve consensus deterministically under this assumption that tolerates even one fault. There are two ways to overcome the FLP result. The first is to assume that only faulty processes ever fail and correct processes remain correct. Under this assumption, the faulty process can be identified using timeouts. This assumption does not allow malicious processes because it is impossible to distinguish between a correct process whose messages are delayed by a malicious process and a faulty process. The second method to overcome the FLP result is to introduce randomized in protocols [38].

The eventual synchrony timing model is more practical than the previous models, since it assumes that the network oscillates between synchonous and asynchrous periods. This allows protocols under the model to make progress during synchronous periods, while still ensuring safety during asynchrous periods. This model is practical because generally the networks are synchronous mostly and messages are delivered within time, and after a period of asynchrony, get become synchrous again.

2.4 Fault Models

A distributed system is subject to a variety of failures: nodes can fail, network can be faulty, and either nodes or the network can exhibit arbitrary or even malicious behavior. We assume that when a node fails by crashing, all the components fail at the same time. However, when Byzantine nodes fail, it is impossible to predict if they have really crashed or not. Thus, categorizing the different faults into fault models, lead to algorithm designs that use resources efficiently to solve consensus for specific fault models. In the rest of this section, we discuss the three fault models that is dissertation depends on.

2.4.1 The Crash Fault Model

The Crash Fault Model is the simplest of the fault models, in which a process stop executing protocols steps. Under this model, a correct process is expected to execute the algorithm correctly and exchange messages with other processes in a timely manner for some time t. Beyond, the process may stop sending messages and executing the protocol to other processes. A process is said to be *faulty*, if it crashes some time during its execution; otherwise, it is said to be *correct*. Such protocols are said to be adopt the *crash-stop* model.

A consensus protocol under this fault model guarantees the safety properties by enforcing a limit on the number of processes that can be faulty at any given time. N denotes the total number of processes while f denotes the maximum number of tolerated faults. To guarantee safety, crash fault-tolerant protocols typically require $N = 2f + 1$ processes to tolerate f failures. Furthermore, to guarantee progress even under failures, such protocols depend on a quorum, instead of all, of the processes.

In practice, it is unlikely that no more than f processes during the entire execution of the protocol. Either crashed processes must be recovered or new processes must be be added to the protocol to ensure progress. Therefore, in practice, *crash-stop* protocols are enhanced with recovery mechanisms that help a crashed process rejoin the protocol after a restart. They are, hence, said to adopt the *crash-recovery* model. Typically, protocols in this fault model persist some state information to local stable storage to aid during the recovery procedure.

Some of the notable and prominent examples of protocols that tolerate crash faults are: Paxos [95], Viewstamped Replication [119], and Raft [120].

2.4.2 Byzantine Fault Model

The Byzantine Fault Model is perhaps the most general of the fault models. Under this model, a process can behave *arbitrarily*; that is, it may deviate from the protocol specification. Protocols adopting this fault model are the most complex due to the many possible failure executions they must cope with.

Figure 2.1a illustrates the different kinds of failures in the Byzantine failure model. Broadly, Byzantine failures can be classified into *Omission* faults and *Commission* faults [45]. Under omission faults, a process may fail to send one or more messages specified by the protocol, but sends no incorrect messages. Crash faults are a subset of omission faults. Commission faults include all other non-omission faults in which a process may send a message not specified by the protocol. This includes *equivocation*, i.e., the ability of a process to make conflicting statements without being detected as having an intent to compromise consistency.

Because processes can equivocate, Byzantine fault-tolerant protocols require more than $N = 2f + 1$ processes to tolerate f failures. Specifically, N should be equal to $3f + 1$. The

additional f replicas are required to overshadow the malicious acts (i.e., equivocation) of the Byzantine replicas. Unlike CFT, protocols in the BFT model do not guarantee safety beyond f failures, thus it is important to proactively recover processes to avert any calamities [42].

(a) Byzantine Fault Model. (b) Hybrid Fault Model.

Figure 2.1: Failure hierarchy.

2.4.3 Hybrid Faults

The hybrid fault model is very similar to the Byzantine fault model, except that the processes depend on a trusted subsystem to tolerate arbitrary faults. In this model (see Figure 2.1b), the ability of the replicas to equivocate is stripped away using a trusted subsystem [44, 98]. A trusted subsystem certifies the message sent by the replicas to ensure that a malicious replica cannot cause different correct replicas to execute different operations as their *i-th* operation. For example, in *Hybster* [32], replicas attest messages with *monotonically increasing counter certificates* using the `TrInX` trusted subsystem. Therefore, replicas cannot sign different messages using the same counter value without being detected.

Thus, additional resources that were required to detect equivocation in the BFT model are unnecessary in the hybrid model. One fundamental change is that the number of replicas required to tolerate f failures reduces from $3f + 1$ to $2f + 1$. In the hybrid model, at most f replicas may behave arbitrarily with the exception of the trusted subsystem, which can only fail by crashing.

2.5 Byzantine Consensus

A Byzantine Fault-Tolerant (BFT) consensus protocol consists of a collection of nodes that agree on the order of client-issued commands and execute them in the agreed order. The protocol functions in a series of views, where in each view, a primary replica proposes and sequences commands, which are executed by all non-faulty replicas in the prescribed order. Before executing the commands, correct replicas must ensure that (i) the commands are replicated at enough correct replicas and (ii) enough correct replicas observe the same sequence of commands from the primary. This function is carried out by the *Agreement*

algorithm, by exchanging command and state between replicas. Some BFT agreement algorithms (e.g. PBFT [41]) commit in three phases and require consent from a supermajority (67%) of replicas, while some others (e.g. SBFT [64]) require consent from all replicas and commit in two phases during good periods.

When a primary replica ceases to make timely progress or misbehaves by sending different sequence of commands to different replicas, the *View Change* algorithm is invoked by non-faulty replicas to replace the faulty primary. The primary of the new view, determined by the view number, collects the replica-local states of enough replicas, computes the initial state of the new view, and proceeds with the agreement algorithm in new view. If a view change does not complete in time, another view change is triggered for a new primary.

Furthermore, replicas use the *Checkpoint algorithm* to limit their memory requirements by garbage collecting the states for those commands that have been executed by enough correct replicas. Replicas exchange information to produce the checkpoint state. When some replicas fall behind the rest of the system, the checkpoint state is used to bring them up to date via the *state transfer* algorithm.

2.5.1 Consensus with Trusted Subsystems

Broadly described, Byzantine failures can be classified into *Omission* faults and *Commission* faults [45]. Omission faults include a replica failing to send one or more messages specified by the protocol, but sends no incorrect messages. Crash faults are a subset of omission faults. Commission faults include all other non-omission faults in which a replica may send a message not specified by the protocol. This includes *equivocation*, i.e., a replica's ability to make conflicting statements to compromise consistency without being detected.

In the hybrid fault model, a trusted subsystem employed to prevent replicas from equivocating [44, 98]. A trusted subsystem is a local service that exists at every replica, and certifies the messages sent by the replicas to ensure that malicious replicas cannot cause different correct replicas to execute different sequences of operations. The trusted subsystem, typically, consists of a monotonically increasing counter paired with a attestation mechanism (signatures/message authentication codes). The trusted subsystems assigns a unique counter to a message and generates a cryptographic attestation over the pair, thus each outbound message is bound to a unique counter value. When correct replicas receive the message pairs, they process them in increasing counter value order. Thus, when a faulty replica sends two different messages to two different correct replicas, only one will process the message, while the other will wait for the message with the missing counter value, eventually detecting equivocation.

Since equivocation is prevented using the trusted subsystem, f additional correct replicas that were required for traditional BFT protocols to balance the impact of f malicious replicas are no longer required in the hybrid fault model. The result is smaller *quorums* – the subset

of replicas that is used to make decisions at different phases of consensus. The quorum size of most traditional BFT protocols is $3f + 1$; hybrid protocols improve this to $2f + 1$.

2.6 Example: Comparison of Fault Models

In this section, we highlight the fundamental design differences between the BFT, CFT, and the hybrid classes of protocols. We begin by comparing three example protocols, Paxos [95], PBFT [41], and MinBFT [135], to illustrate their design differences and then proceed with a more general description.

Paxos vs PBFT vs MinBFT

We provide a walk-through of identical normal-case executions for each of the three protocols and highlight where each protocol performs uniquely. Note that we describe the three protocols hand-in-hand and only highlight the differences explicitly. Paxos, PBFT, and MinBFT are leader-based protocols, i.e., they designate a replica to propose the order for client commands. A normal-case execution involves a set of replicas agreeing on the order of the commands submitted by the clients. For brevity, we exclude the view-change/leader-election protocol. Figure 2.2 provides the visual illustration. Paxos, PBFT, and MinBFT are leader-based protocols, i.e., they designate a replica to propose the order for client commands.

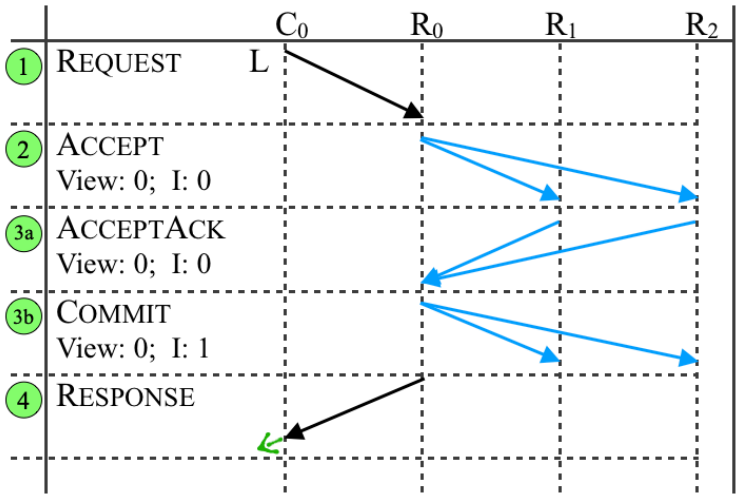
(1) For each protocol, the execution starts when the client sends a request to the leader replica containing a command to be executed on the replicated state. For PBFT and MinBFT, the client signs its requests to ensure that a potential Byzantine leader does not tamper with the requests without detection.

(2) As soon as the leader receives the request, it assigns the command with a sequence number (*instance number* in Paxos and *order number* in MinBFT), which defines the execution order. The leader `proposes` the command with the sequence number to all the replicas. PBFT certifies the message with a *message authentication code* (MAC), while MinBFT certifies with a trusted subsystem-produced MAC. Also, note that in our example, Paxos and MinBFT require three nodes each, while PBFT requires four.

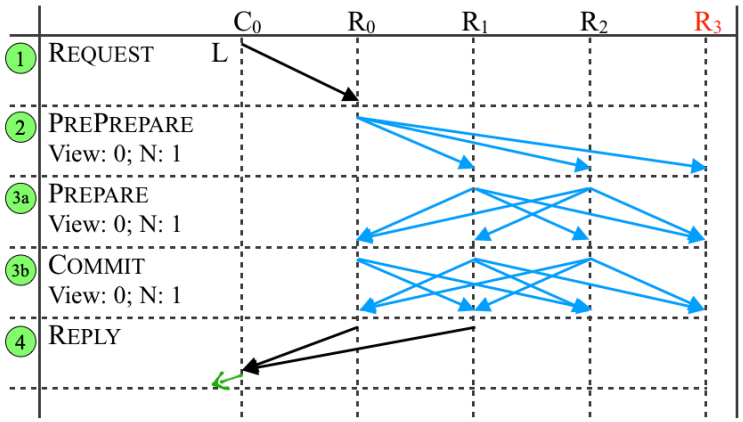
The replicas receive the proposal from the leader.

(3) **MinBFT** replicas acknowledge the proposal to each other in addition to the leader. Replicas wait for a majority of responses to commit and execute the command.

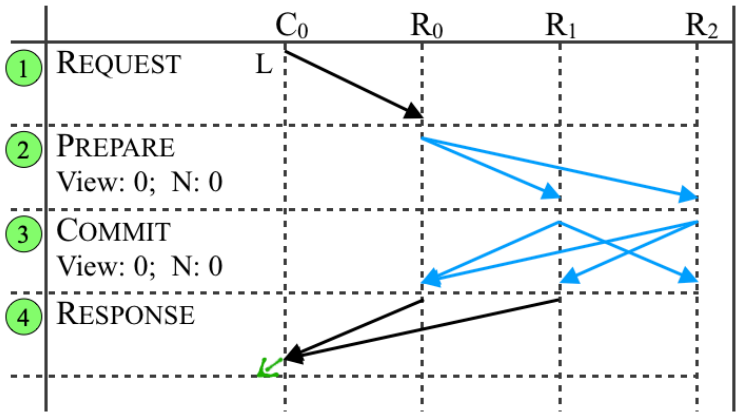
(3a) **Paxos** replicas reply back to the leader acknowledging the proposal. **PBFT** replicas exchange the proposal with each other to ensure that they received the same



(a) Paxos.



(b) PBFT.



(c) MinBFT.

Figure 2.2: An example of normal execution in three different failure models.

proposal from the leader (i.e., to ensure no equivocation). The proposal is validated if a majority of nodes respond with the same proposal from the leader.

3b **Paxos** leader, upon receipt of a majority of acknowledgments, commits the request, and sends a `commit` message to the replicas to do the same. **PBFT** replicas exchange commit messages. They execute the command upon collecting a majority quorum of these messages and reply to the client.

4 **Paxos** client waits until it receives a reply from the leader. **PBFT** and **MinBFT** clients wait until they receive identical replies from at least $f + 1$ replicas. This is because, waiting for only one potentially Byzantine replica may provide an incorrect response.

The rest of this section generalizes the differences more broadly and presents some key insights.

Replicas

Consensus algorithms consist of two main components: agreement and leader election. We analyze the classes of protocols with respect to these two components. We note that consensus protocols designed for different fault models use different terminologies to define equivalent constructs; Table 2.1 summarizes the labels used in rest of this paper.

Leader Election or View Changes

One of the fundamental design differences between CFT and BFT protocols lie in how they elect a designated replica to propose commands, called *leader election* and *view change*, in the CFT and BFT models, respectively. Here, we only refer to single leader (or primary)-based protocols. During leader election in the CFT model, multiple replicas compete to gain leadership for a particular (logical) period. If an elected leader fails to make progress, another election takes place.

If such a scheme was applied in the BFT model, a malicious replica may repeatedly win an election but will fail to make progress (i.e., propose commands to the state machine). Thus, protocols in the BFT model use a different approach, also known as *view change*. Each logical period, during which a stable leader is active, is identified using a monotonically increasing number, called *view*. For each view number, the replica that will be responsible for proposing commands in that view is predefined. This ensures that a correct leader is established after at most f view changes, where f is the maximum number of Byzantine replicas. In practice, the leader replica is obtained from the view number v using the formula $v \bmod N$, where N is the number of replicas.

Table 2.1: Summary of labels used by different protocols and their meaning.

Label	Meaning
View, Term, Ballot	Logical period in which the protocol makes progress.
Instance, Log Index, Order, Sequence	Positions in the replicated log to which client commands are assigned.
Leader, Primary, Owner	Distinguished node responsible for proposing order to client commands. An owner is the leader for a subset of client commands indicated by objects they access.
View Change, Leader Election, Term Change	Process of replacing a leader/primary with another one.

Agreement

Once a stable leader is established, the leader starts proposing client commands to be applied on the replicated state. In a CFT protocol, this agreement process usually consists of two phases: (i) the leader proposes a command to be replicated to other replicas, and replicas acknowledge the receipt of the command; and (ii) the leader finalizes the command to all replicas after acknowledgment. The last phase is unnecessary if the replicas simply acknowledge each other in addition to the leader. This is an optimization that many CFT protocols implement to minimize communication steps (and reduce client-perceived latency) [95, 117, 122]. For example, in Figure 2.2a, the `Commit` step can be removed; instead, replicas broadcast their `AcceptAck` messages to each other and a quorum of those messages constitute a commit indication.

However, for BFT protocols, replicas acknowledging to each other is not an optimization but a requirement. Since the replicas including the leader can be Byzantine, messaging with each other is the only way for correct replicas to detect equivocation. Thus, generally BFT protocols perform two such acknowledgement: one to ensure that the replicas have received the same proposal from the leader, and a second to acknowledge that a majority of the replicas agree with that proposal.

Hybrid protocols also need to perform all-to-all message exchanges, but only once. Due to the equivocation prevention mechanism, replicas can be certain that the leader cannot equivocate without detection. Thus, the most a leader can equivocate is to propose the same command twice with two different attestations, but a correct replica can detect this using the metadata of the last signed client request.

Client

In the CFT model, the client sends a request to one of the replicas and waits until it receives a reply with the result of the operation. In contrast, in the BFT model, the client sends a

request to one of the replicas, but expects a reply from at least $f + 1$ of the replicas. This is because, the client cannot trust a single reply since it may be from a Byzantine replica. Therefore, a client must wait until it receives $f + 1$ matching replies to validate the result. Due to the assumption of Byzantine replicas in the hybrid fault model, the clients in this model use the same reply validation mechanism.

A property that is unique to the BFT and hybrid models is that the client messages are signed in the BFT model using the client's private key in order to prevent Byzantine replicas from tampering with commands that should be executed by the state machine.

2.7 The BFT Quorum System

In this section, we overview the quorum system used in PBFT and revisit the composition of different quorums that lead to setting the number of tolerated faults in a Byzantine setting to one-third of the total replicas.

A value proposed by the primary undergoes two rounds of voting before being committed. The first round of votes ensures that replicas receive the same value and that they agree to vote on that value in the next round. The second round of votes ensures that the value is agreed by enough correct replicas to tolerate failures.

In Flexible BFT [106], the authors dissect the different quorums in partially synchronous Byzantine protocols, and intuitively illustrate how these systems work. We summarize below.

The leader proposes a value to all the replicas in the system. A replica accepts the value if it is a valid message i.e. the signature is valid and that it has not voted for any other value in the same view. It votes on the value and sends it to all other replicas. A set of q_{lck} votes form a lock certificate, and the value is said to be *locked*. For each of exposition, we use a common variable to represent both the quorum fraction and the quorum name.

When a replica collects a lock certificate, it votes on it and broadcasts its vote to other replicas. A set of q_{unq} votes for the lock certificate form a commit certificate, and the proposed value is said to be committed at the acceptors. A replica will only vote on the same locked value in all subsequent views. It will only unlock a value if at least q_{ulck} other replicas are locked on a different value. A learner will learn the commit decision when it receives q_{unq} votes for the value and q_{cmt} votes for the lock certificate.

Thus, there are four quorums: q_{lck} is used to lock on the value. q_{ulck} unlocks on the value. q_{unq} votes on the value and q_{cmt} votes on the certificate for clients to commit. These quorums must interest at least one correct replica to guarantee protocol's safety. However, not all quorums need to intersect with each other.

Quorum intersection within a view. To guarantee that a locked value is the only value that is voted on, the q_{lck} quorum and the q_{unq} quorum must intersect at at least one honest

replica. That is, $q_{lck} + q_{unq} - 1 > f$, where f is the fraction of total faults.

Quorum intersection across views. To guarantee that a value is not committed, if it has been unlocked, we require that $q_{ulck} + q_{cmt} - 1 > f$. A value is voted for commitment in a given view, only if has been locked in that view. Thus, by ensuring that a value is not unlocked before committing, honest replicas are certain that the value remains locked when voting to commit the value.

Liveness. To ensure progress within a view, there cannot be more than $1-max(q_{lck}, q_{ulck}, q_{unq}, q_{cmt})$ Byzantine failures, as otherwise it cannot be guaranteed that an honest replica is present in midst of these quorum intersections to drive the consensus forward.

To form the Byzantine quorums used in PBFT, the two acceptor and two client quorums can be consolidated by setting $q_{lck} = q_{ulck}$ and $q_{unq} = q_{cmt}$. Thus, the number of tolerated faults is $q_c + q_r - 1$ and the number of Byzantine faults tolerated is $1 - q_c$. Thus, setting $q_r = q_c = 0.67$, the total fraction of Byzantine and total faults tolerated is 0.33 (one-third faults). Flexible BFT allows setting different values for q_c and q_r to achieve different total faults and Byzantine fault thresholds.

Chapter 3

Related Work

The research on fault-tolerant distributed computing systems spans over five decades. In this chapter, we only scratch the surface and present the most relevant results that directly impact the results presented in this dissertation.

3.1 The Rise of Fault-Tolerant Distributed Computing

During the mid-seventies, the drive to use computers to fly commercial aircraft spurred the research to build reliable, fault-tolerant computing systems [94]. One notable project, called the SIFT project [137], introduced some of the early concepts that would later be coined as the Byzantine Generals problem [96], by Leslie Lamport.

Around the same time, the intent to replicate databases over an ARPA-like network led to ideas for totally ordering events in a distributed system. Leslie Lamport specified that a distributed system can be realized using a sequential state machine over a network of processes [88]. In his seminal work "Time, Clocks and the Ordering of Events in a Distributed System" [88], Lamport describes how to implement an arbitrary distributed state machine over a network of processes by totally ordering the input events using timestamps. Lamport showed that it is possible to partially order events based on the causality exhibited during communication between processes. By assigning timestamps from logical clocks to events and using the observation that messages must be sent in order to be received, a happens-before partial ordering on the events at concurrent, communicating processes can be established. A total ordering of events can be obtained by consistently ordering the partial ordering of events at all processes.

In [88], Lamport assumed that the processes never fail and that all the messages are delivered. Later, in [87], Lamport presented the first fault-tolerant algorithm that discusses arbitrary (Byzantine) failures. The algorithm expects that processes have tightly synchronized clocks, and assumes a synchronous timing model, that is, there exists an upper bound on the message delays between non-faulty processes.

The FLP Result. It quickly became clear that the network and process synchrony assumptions play a role in the design of consensus algorithms, in addition to the kinds of faults exhibited by the system. The impossibility result of Fischer, Lynch, and Patterson [60]

provided the important foundational knowledge on the limitations to which fault-tolerant algorithms can be designed. Particularly, they showed that it was impossible to devise a deterministic consensus algorithm in the asynchronous model with only a single failure. Since their result applied only to deterministic and asynchronous protocols, various designs that circumvented the impossibility followed. Ben-Or [33] proposed a non-deterministic algorithm in which processes vote for a value of a single bit. When processes don't agree on the value, they will randomly change their value to vote for in the next round. While the algorithm took an exponential number of rounds, it circumvented the FLP impossibility result and tolerated faults up to half the number of processes under the asynchronous timing model. Similarly, other algorithms [43] that considered the eventual synchrony timing model were developed.

3.2 Byzantine Fault Tolerance

Since Lamport et al. formulated the Byzantine Generals problem [97], numerous solutions have been proposed to solve the agreement problem in the face of Byzantine failures. These solutions have varied widely in terms of their fault assumptions and the timing models. A detailed review of these solutions can be found in the books on distributed computing [38, 104].

After numerous theoretical results, Castro and Liskov presented the first practical design and implementation of a Byzantine State Machine Replication protocol. The protocol, PBFT [41, 42] solved consensus in three communication steps (excluding two steps for client communication) using $3f + 1$ nodes and using quorums of $2f + 1$ nodes. f is the maximum number of Byzantine faulty nodes that the system can tolerate and make progress. PBFT assumes a weak synchrony model making it suitable for asynchronous networks, unlike many earlier protocols that required either synchrony or took many rounds to reach agreement. Consequently, a plethora of designs and implementations of BFT protocols that improved over the PBFT algorithm followed. Below, we group these protocols by their novel, characterizing features, and discuss their merits.

Speculation. Some partially-synchronous BFT protocols [23, 64, 83, 108] use speculation to make commit decisions using fewer communication steps. These protocol adopt fast BFT quorums that are at least 50% larger than normal BFT quorums, and reduce communication delays under favorable conditions. At times, a fast quorum may not be enough or unattainable, in which case, the replicas fall back to a slow protocol with normal quorums and additional communication steps. Thus, such protocol collect larger quorum than normal protocols in the best case, and spend more communication steps than normal protocols in the worst case.

Similarly, Thunderella [121] and Sync Hotstuff [10] commit optimistically under the partial synchrony model using quorums of size $\geq 3/4$, and fallback to a synchronous slow commit

rule. DUOBFT's guarantees are completely based in the partial synchrony model.

SACZyzzyva [69] is a recent protocol that uses a trusted counter to increase the resilience of Zyzzyva to tolerate f slow nodes. The protocols requires only $f + 1$ replicas to host the trusted counters in a system with $3f + 1$ replicas. Similarly, DUOBFT also reduces the number of replicas that host the trusted counter in addition to support two fault models. Moreover, with Flexible MinBFT, we also provide the flexibility to choose f independently of N, the number of replicas.

Tentative Execution and Optimistic Agreement. Some protocols like PBFT and MinBFT use tentative execution [40] to execute the proposed operations before the final commit step. This can improve the overall performance under favorable conditions. Furthermore, the optimistic agreement [55] technique uses only a subset of replicas to run agreement, while the remaining replicas update their state passively. Such techniques are orthogonal and applicable to DUOBFT as well.

Request Classification. The Q/U [8] protocol was the first to achieve consensus in two communications steps when there are no faults and update requests do not concurrently access the same object. Q/U defines a simplified version of conflicts. Requests are classified as either reads or writes. Reads do not conflict with reads, while write conflicts with reads and writes. HQ [53] is similar to PBFT with a special optimization to execute read-only requests in two communication steps and update requests in four communication steps under no conflicts. HQ's definition of conflict is the same as Q/U's.

Hierarchical Protocols. Steward [18, 19] and GeoBFT [71] follow the hierarchical fault model by using a combination of crash and Byzantine fault-tolerant mechanisms. The replicas are divided into groups. Replicas with a group run a BFT protocol while inter-group agreement is achieved using a crash-fault tolerant protocol. However, the protocol exposes a single combined fault model to the learners: the protocols can tolerate fz failures in z groups of which at most f can happen in a single group. Such techniques are aimed towards WAN deployments.

Flexible Quorums. In Flexible Paxos [76], Howard et al. introduced the notion of flexible quorums in the crash fault model. Malkhi et al. then developed the flexible quorums approach in the Byzantine fault model. [106]. The Flexbile Hybrid Quorums presented in this paper can be seen as the hybrid variant of the flexible quorums technique.

Composition in Consensus. Aliph [68] builds a BFT protocol by composing three different sub-protocols, each handling a specific system factor such as contention, slow links, and Byzantine faults. Under zero contention, the sub-protocol *Quorum* can deliver agreement in two steps with $3f + 1$ nodes by allowing clients to send the requests directly to the nodes. However, as contention or link latency increases, or as faults occur, Aliph switches to the sub-protocol *Chain* whose additional steps is equal to the number of nodes in the system, or to the sub-protocol *Backup* which takes at least three steps. Although the idea of composing simpler protocols is appealing in terms of reduced design and implementation complexities,

the performance penalty is simply too high, especially in geo-distributed settings.

Rotating Primary Protocols. By rotating the primary role after each commit decision, static-ordering protocols [133, 134, 140] control the slowdown caused by malicious replicas. However, this technique does not effectively use replicas' resources and is prone to under-utilization as primary-based protocols. However, a Byzantine replica can delay its commands without detection reducing the overall server-side throughput.

Multi-Primary Protocols. To overcome the overhead of primary-backup protocols, various multi-primary proposals have been proposed. These solutions largely allow multiple replicas to be a primary and order commands received from the clients. These solutions differ in how they prevent primary replicas from exhibiting Byzantine behavior. The RCC [70] technique runs agreement in round, where in each round multiple active primary replicas reach consensus on a command. At the end of each round, the agreed commands are executed in a determistic order by correct replicas. When primary replicas cease to make progress, a separate consensus protocol stops the replica from proposing the round. MirBFT [129] uses primary and leader model, where the primary decides replicas that can propose (called leaders). The algorithm run in epochs, where each epoch mostly lasts as long as the leaders make progress. The sequence numbers are divided among the leaders. MirBFT is pessimistic permits only good leaders to participate in epochs and penalizes bad leaders by disallowing participation for at least one epoch.

Leaderless Protocols. Dispel [136] uses multiple instance of deterministic Binary consensus protocol to reach agreement in parallel at different replicas. The Binary consensus protocol itself relies on a request dissemination mechanism such as Reliable Broadcast [81]. Unlike other leaderless protocols [116], Dispel is not affected by contention. However, similar to other multi-primary protocols such as RCC, it requires that at each round, a value is decided for the consensus instance handled by each replica in the system.

Other leaderless protocols such as Aliph [26] rely on interference relationships to order commands in a way such that interfering commands are always executed in the same order across all replicas. This introduces additional complexities into the protocols such as additional communication steps and larger quorums leading to poor performance at scale.

Leaderless protocols such as HoneybadgerBFT [113] and BEAT [56] have been proposed in the asynchronous fault model where any replica can propose commands. However, these solutions use randomized agreement protocols and do not offer any ordering properties. Lamport [91] proposed a leaderless version of Byzantine Paxos that relies on synchrony and a virtual leader. However, the protocol itself takes many rounds to elect a leader.

Request Dissemination. Request dissemination [17, 45, 50, 72] has been proposed previously as a means to relieve the workload on the primary. These solutions use reliable broadcast that increases the overall latency. Dissemination is a common technique used to alleviate the primary's workload [17, 72]. The replicas receive client requests and disseminate them on to the other replicas. The primary orders the requests by proposing the request hashes

instead of the entire payloads. A key requirement is that the requests must be disseminated to enough correct replicas before the primary can propose an order. A common protocol used for this purpose, the Byzantine Reliable Broadcast [81] takes three communication steps and utilizes quadratic number of messages to disseminate each request. Such additional steps and messages lead to poor performance in wide-area deployments.

Cross Fault Tolerance (or XFT). The XFT model [103] assumes a weaker adversary that fully controls either the malicious replicas or the network, but not both. Protocols under this model, Elpis [62] and XPaxos [103] tolerate t Byzantine failures in a $2t + 1$ system as long as a majority of replicas are correct and communicate synchronously. XFT is suitable for geographically distributed systems, where an adversary cannot compromise t replicas and cause network partitions among correct ones at the same time. as long as the trusted component is not compromised.

Workload Analysis and Weighted Voting. AWARE [34] is a mechanism for geographically distributed BFT protocols that optimizes, at runtime, client latency in different geographical locations. AWARE builds on top of WHEAT [128] which assigns weighted voting power to nodes such that nodes closer to clients can contribute more to the quorum than far away nodes. This is orthogonal to our contribution.

Thin Clients. Troxy [99] provides an alternative mechanism for using threshold signatures to reduce client communication complexity to linear. A proxy that uses a trusted execution environment mediates the communication between the replicas and clients, enabling clients to verify the replicas with only a single message from the trusted proxy. Troxy is aimed at making Byzantine tolerance transparent to legacy clients and is orthogonal to our contribution.

Adversaries. Different kinds of adversaries have been explored in prior works. Both the BAR and a-b-c fault models [13, 106] consider an adversary that does not collude. With the hybrid fault model, we consider an adversary that does not break the protections around the trusted component, but they can otherwise collude with other Byzantine replicas.

Diverse learners. Bitcoin [118] uses a probabilistic commit rule that depends on the depth of the confirmation. Typically a block depth of six implies a commit with a very high probability, although a block depth of one is enough to commit for some learners. The Cross fault-tolerant (XFT) [103] model offers two kinds of learners: learners that follow the crash fault model under the asynchronous timing model, or learners that from the Byzantine fault model under the synchronous timing model.

3.3 Crash Fault Tolerance

3.3.1 Paxos-like Protocols

Lamport introduced Paxos [95] to solve consensus in the presence of crash failures in the asynchronous network model. It required $2f + 1$ replicas to tolerate f non-malicious failures. The protocol became immensely popular due to its applicability in datacenter-centric systems, where non-malicious faults are nearly non-existent. Furthermore, interest spurred in Paxos-like protocols both in academia and the industry. Numerous storage systems [29, 47] that use Paxos were developed. At the same time, researchers focused on improving the original Paxos protocol to produce an ever-growing number of Paxos variants. Some notable results include the Fast Paxos [90], Generalized Paxos [89], and Flexible Paxos [75] protocols.

The Fast Paxos protocol presented a single communication round consensus algorithm without involving the leader, by exploiting situations in which concurrent proposals by multiple replicas are rare. Fast Paxos was one of the first protocols to introduce the notion of fast consensus. The leader would step in to only resolve conflicts, leading to slow path consensus. The Generalized Paxos protocol proposed the notion of commutativity to improve the chances of guaranteeing fast consensus without the leader even when concurrent commands conflict with each other. This is achieved by exploiting the commutativity property of commands. When commands commute, executing them in any order at different replicas on the same initial state will produce the same resulting state. Thus, only non-commuting commands need to be totally ordered, while it is enough to reliably commit non-commuting commands with a partial order.

Even though Paxos is widely popular, it is not very understandable since it only discusses the core algorithm and defers the implementation specifics such as storage that are required to build a complete, working system. Raft [120] was proposed as an understandable consensus algorithm that addresses all the components required to build a working implementation. Raft has grown immensely popular and is used in various open-source as well as enterprise systems [3, 5].

3.3.2 Leaderless Protocols

Leaderless and multi-leader protocols [21, 107, 117, 122] have been proposed for the CFT model.

EPaxos [117] is a multi-leader consensus protocol that, unlike Mencius, exhibits high availability despite replica crashes as long as a majority of the nodes are up and running. The protocol works in two phases: a fast phase that is reached under no conflicts and an additional slow phase that is required under conflicts. A conflict is induced when two commands access the same object in the underlying state and one of them is a write. In EPaxos, every

replica collect command dependencies and form a graph, such that conflicting commands are part of the same graph. The graph is deterministically executed by the replicas.

Caesar [21] improves EPaxos by providing a protocol that commits commands in the fast path with very high probability under heavy conflicts. In M^2Paxos [122], a replica can order a request if it *owns* the object that the request accesses. Otherwise, it forwards the request to the right owner or acquires ownership. Acquiring ownership means becoming the primary for some subset of objects, and in CFT-based protocols, any replica can propose to be an owner of any subset of objects at any point in time.

SDPaxos [142] proposes a crash fault-tolerant protocol that separates the replication and ordering concerns, and uses a consensus protocol for both the tasks. The advantage of such an approach is that it is agnostic to workload contention. However, the use of a single primary for the ordering protocol introduces availability challenges that are largely avoided by other leaderless protocols.

3.3.3 Hybrid Fault-Tolerant Protocols

The literature is rich in protocols that adopt the hybrid fault model [102]. While early protocols depended on an attested append-only log abstraction provided by the trusted component [44], the counter-based abstraction [98] became widely adopted due to its simplicity, and have been adopted by numerous protocols [32, 82, 102, 135]. We used MinBFT to perform our analysis and construction, because of its presentation as the hybrid counterpart to PBFT, and its use of a simple counter-based trusted attestation mechanism.

A multitude of works in the past have proposed the use of trusted hardware to provide increased fault tolerance at lower cost in distributed systems. A2M [44] is one of the first efforts to show that a small trusted component can significantly improve security in distributed systems. A2M uses an append-only log backed by a trusted module and demonstrated its use with A2M-PBFT, a PBFT variant with $2f + 1$ node requirement. Due to the high overheads of maintaining the append-only log that can grow indefinitely, TrInc [98] was proposed as a smaller and simpler alternative to A2M. TrInc proposes a monotonically increasing trusted counter backed by a key. BUMBLEBEE uses a variant of the TrInc subsystem.

Efforts such as [82, 135] reduce the complexity of existing BFT protocols using trusted hardware. MinBFT and MinZyzzyva [135] are protocols obtained by incorporating a trusted counter implementation, called *USIG*, into PBFT and Zyzzyva [83], respectively. CheapBFT [82] uses a *CASH* subsystem that is implemented using FPGAs.

Hybster-COP [32] is a variant of Hybster that boosts throughput by executing multiple instances of the consensus protocol, albeit with a single primary. OMADA [58] tackles variability in node hardware and network bandwidth by using multiple agreement groups, and varies the number of groups per replica and the number of requests per group to balance the workload among replicas.

EBAWA [134] incorporates trusted components into the Spinning primary approach [133] to minimize the client-side latency in geo-distributed deployments.

FastBFT [102] uses SGX and threshold secret shares to improve the performance of BFT protocols. However, the use of secret sharing as the cryptographic technique requires creating a set of secret key shares for each command, since the shares need to be exposed for committing each command. This requires expensive computational and additional network resources that could be potentially used for replication.

3.4 Protocol Transformations

Techniques for transforming CFT protocols into BFT protocols have been investigated before. Nysiad [74] translates Byzantine faults into crash faults by adding additional nodes, called *guards*, which detect and prevent Byzantine behavior. Each node requires at least $3f + 1$ guards to tolerate f Byzantine failures. Furthermore, [74] uses a complex attestation mechanism to verify the integrity of sent messages.

Correia et al. [51] present a method for transforming asynchronous crash tolerant protocols to Byzantine tolerant protocols. The mechanism involves using trusted components and applying a set of transformation steps to an CFT algorithm and proving the correctness of the resulting algorithm under Byzantine failures. Furthermore, the transformation methodology requires certain assumptions such as a mute-ness failure detector in addition to the trusted component. In contrast, we provide a parameterized generic algorithm that has been validated for correctness. For transformation, we require only that the parameters that satify their respective properties are chosen. Furthermore, we do not make any more assumptions than requiring a trusted component.

In [92], Lamport presented a transformation of the Paxos protocol to tolerate Byzantine faults by adding f additional replicas and modifying both phases of the protocol. The resulting protocol was similar to PBFT [41, 42], in terms of complexity than CFT protocols. In contrast, our intent is to achieve Byzantine fault tolerance in existing CFT protocols, without the need for additional nodes and communication steps, thereby reducing overheads and also achieving high performance.

3.5 Summary of Past and Related Protocols

We provide a consice summary of related protocols in Table 3.1. To provide a big picture perspective of the dissertation's contributions, all algorithms of the dissertation are also placed in this table (boldfaced), including **Destiny** (Chapter 8), **DQ-PBFT** (Chapter 7), **Linear Hybster** (Chapter 8), **ezBFT** (Chapter 4), **Dester** (Chapter 5), **Flexible MinBFT** (Chapter 9) and **DuoBFT** (Chapter 10).

	BFT	**Hybrid**	**CFT**
Lower-bound	PBFT [41]	MinBFT [135]	Paxos [95]
Speculative	Zyzzyva [83]	MinZyzzyva [135]	Fast Paxos [90]
Speculative + Leaderless	**ezBFT (Ch. 4)**	**Dester (Ch. 5)**	EPaxos [116] Caesar [21] Atlas [59]
Linear Communication	SBFT [64]	**Linear Hybster (Ch. 5)**	N/A
Multi-primary	RCC [70], MirBFT [129], **DQ-PBFT (Ch. 7)**	**Destiny (Ch. 8)**	M^2Paxos [122] WPaxos [12]
Rotating-primary	Hotstuff [140] Spinning [133]	EBAWA [134]	Mencius [107]
Flexible Quorums	Flexible BFT [106]	**Flexible MinBFT (Ch. 9)**	Flexible Paxos [75]
Hierarchical	Steward [18] GeoBFT [71]	-	N/A
Byzantine Ordered Consensus	Pompe [141] **Folly (Ch. 11)**	-	N/A

Table 3.1: Table 3.1: Summary of Past and Related Protocols. The dissertation's protocols are also included in the table (boldfaced) for perspective.

Chapter 4

Leaderless BFT with ezBFT

4.1 Introduction

This chapter addresses the challenges that arise from deploying a Byzantine fault-tolerant protocol on a geographically distributed set of nodes. Particularly, achieving low client-side latencies and high server-side throughput under the high communication latencies of a WAN is challenging, since replicas need to communicate with each other and the clients to reach consensus. The number of communication steps incurred directly impacts the latency, as each step involves sending messages to potentially distant nodes. Thus, protocols such as Zyzzyva [83], Q/U [8], and HQ [53] use various techniques to reduce communication steps. These techniques, however, do not reduce client-side latencies in geographical deployments, where the latency per communication step is as important as the number of communication steps. In other words, a protocol can achieve significant cost savings if the latency incurred during a communication step can be reduced.

The downside of such lack of locality is most manifest for primary-based BFT protocols such as PBFT and Zyzzyva: a replica is bestowed the *primary* status and is responsible for proposing the total-order for all client requests. While the clients that are geographically nearest to the primary may observe optimal client-side latency, the same is not true for distant clients. A distant client will incur higher latency for the first communication step (of sending the request to the primary). Additionally, the primary-based design limits throughput as the primary carries significantly higher load. To validate these hypotheses, we deployed Zyzzyva [83] in a 4-replica geographical deployment with nodes located in the US (Virginia), Japan, India, and Australia (Sydney), using Amazon's EC2 infrastructure. We deployed clients alongside each replica to inject requests, and measured the client-side latencies by changing the primary's location. Table 4.1 shows the results. We observe that the lowest latencies are when the primary is co-located within the same region.

Past works suggest that it is important to rotate the primary-role in order to reduce the negative impacts of Byzantine replicas [133]. For geo-distributed applications, this technique will produce suboptimal latency SLAs, because the appropriate latency SLA, in this case, is the maximum of the latencies at every primary site. A leaderless protocol can solve the aforementioned problems. A client can send its requests to the nearest replica and can continue to do so as long as the replica is *correct*. The replica can undertake the task of finding an order among all the concurrent requests in the system, executing the request on

Table 4.1: Zyzzyva's [83] latencies (in *ms*) in a geo-distributed deployment with primary at different locations. . Columns indicate the primary's location. Rows indicate average client-side latency for commands issued from that region. For example, the entry at the 4th row and the 3rd column shows the client-side latency for commands issued from India to the primary in Japan. Lowest latency per primary location is highlighted.

	Virginia (US)	Japan	India	Australia
Virginia (US)	**198**	238	306	303
Japan	236	**167**	239	246
India	304	242	**229**	305
Australia	303	232	304	**229**

the shared state, and return the result.

While leaderless protocols [21, 117, 122] have been previously proposed for the CFT model, the transition from CFT to BFT is not straightforward and will yield a sub-optimal solution. In [92], additional communication steps as well as more number of messages within each step are shown to be fundamental for such transformations. This will result in sub-optimal server-side throughput since more messages should be certified by replicas, (exponentially) increasing the degree of computation. Moreover, additional communication steps will increase client-side latency.

Motivated by these concerns, this chapter presents EZBFT, a leaderless BFT protocol. EZBFT enables every replica in the system to process the requests received from the clients. Doing so (i) significantly reduces the client-side latency, (ii) distributes the load across replicas, and (iii) tolerates faults more effectively. Importantly, EZBFT delivers requests in *three* communication steps in normal operating conditions. To enable leaderless operation, EZBFT exploits a particular characteristic of client commands: *interference*. In the absence of concurrent interfering commands, EZBFT's clients receive a reply in an optimal three communication steps. When commands interfere, both clients and replicas coherently communicate to establish a consistent total-order, consuming an additional zero or two communication steps. EZBFT employs additional techniques such as client-side validation of replica messages and speculative execution to reduce communication steps in the common case, unlike CFT solutions [21, 117]. The experimental evaluations on the AWS EC2 infrastructure consisting of geographically distributed nodes reveals that EZBFT improves client-side latency by as much as 40% over PBFT, FaB, and Zyzzyva. By minimizing the latency at each communication step, EZBFT provides a highly effective BFT solution for geo-distributed deployments.

The rest of the chapter is organized as follows. Section 4.2 overviews EZBFT, and Section 4.3 presents a complete algorithmic design and correctness arguments. An experimental evaluation of EZBFT is presented in Section 4.4.

4.2 Protocol Overview

This section overviews ezBFT and highlights the novelties that enable it to tolerate Byzantine failures and provide optimal wide-area latency.

ezBFT can deliver decisions in three communication steps from the client's point-of-view, if there is no contention, no Byzantine failures, and synchronous communication between replicas. The three communication steps include: (i) a client sending a request to any one of the replicas (closest preferably); (ii) a replica forwarding the request to other replicas with a proposed order; and (iii) other replicas (speculatively) executing the request as per the proposed order and replying to the client. These three steps constitute ezBFT's core novelty. To realize these steps, ezBFT incorporates a set of techniques that we summarize below and explain in detail in the next section.

First, the ezBFT replicas perform speculative execution of the commands after receiving the proposal messages from their respective command-leaders (see below). With only one replica-to-replica communication, there is no way to guarantee the final commit order for client commands. Thus, the replica receiving the proposal assumes that other replicas received the same proposal (i.e., the command-leader is not Byzantine) and that they have agreed to the proposal. With this assumption, replicas speculatively execute the commands on their local state and return a reply back to the client.

Second, in ezBFT, the client is actively involved in the consensus process. It is responsible for collecting messages from the replicas and ensuring that they have committed to a single order before delivering the reply. The client also enforces a final order, if the replicas are found to deviate.

Third, and most importantly, there are *no* designated primary replicas. Every replica can receive client requests and propose an order for them. To clearly distinguish the replica proposing an order for a command from other replicas, we use the term *command-leader*. A *command-leader* is a replica that proposes an order for the commands received from its clients. For clarity, all replicas can be command-leaders. To ensure that client commands are consistently executed across all correct replicas, ezBFT exploits the following concepts:

- **Command Interference.** ezBFT uses the concept of *command interference* to empower replicas to make independent commit decisions. If replicas concurrently propose commands that do not interfere, they can be committed and executed independently, in parallel, in any order, and without the knowledge of other non-interfering commands. However, when concurrent commands do interfere, replicas must settle on a common sequential execution. A command encapsulates an operation that must be executed on the shared state. We say that two commands L_0 and L_1 are interfering if the execution of these commands in different orders on a given state will result in two final states. That is, if there exists a sequence of commands Σ such that the serial execution of Σ, L_0, L_1 is not equivalent to Σ, L_1, L_0, then L_0 and L_1 are interfering.

- **Instance Space.** An instance space can be visualized as a sequence of numbered slots to which client-commands can be associated with. The sequence defines the execution order of requests, and the role of a consensus protocol is to reach agreement among a set of replicas on a common order. However, to accommodate our requirement that every replica can be a command-leader for their received requests, each replica has its own instance space. Thus, EZBFT's role is not only to reach consensus on the mapping of client-commands to the slots within an instance space, but also among the slots in different instance spaces.

- **Instance Number.** An instance number, denoted I, is a tuple of the instance space (or replica) identifier and a slot identifier.

- **Instance Owners.** An owner number, O, is a monotonically increasing number that is used to identify the owner of an instance space. Thus, there are as many owner numbers as there are instance spaces (or replicas). This number becomes useful when a replica is faulty, and its commands must be recovered by other replicas. When replicas fail, another correct replica steps up to take ownership of the faulty replica's instance space. The owner of a replica R_0's instance space can be identified from its owner number using the formula $O_{R0} \bmod N$, where N is the number of replicas. Initially, the owner number of each instance space is set to the owner replica's identifier (e.g., $O_{R0} = 0$, $O_{R1} = 1$, and so on).

- **Dependencies.** Due to the use of per-replica instance spaces, the protocol must agree on the relationship between the slots of different instances spaces. EZBFT does this via dependency collection, which uses the command interference relation. The dependency set $\mathcal{D}$ for command L is every other command L' that interferes with L.

- **Sequence Number ($\mathcal{S}$)** is a globally shared number that is used to break cycles in dependencies. It starts from one and is always set to be larger than all the sequence numbers of the interfering commands. Due to concurrency, it is possible that interfering commands originating from different command-leaders are assigned the same sequence number. In such cases, the replica identifiers are used to break ties.

Protocol Properties.

EZBFT provides the following properties of a BFT protocol [41]:

1. **Nontriviality.** Any request committed and executed by a replica must have been proposed by a client.

2. **Stability.** For any replica, the set of committed requests at any time is a subset of the committed requests at any later time. If at time t_1, a replica R_i has a request L committed at some instance I_L, then R_i will have L committed in I_L at any later time $t_2 > t_1$.

3. **Consistency.** Two replicas can never have different requests committed for the same instance.

4. **Liveness.** Requests will eventually be committed by every non-faulty replica, as long as at least $2f + 1$ replicas are correct.

4.3 Protocol Design

In this section, we present ezBFT in detail, along with an informal proof of its properties. We have also developed a TLA+ specification of ezBFT and model-checked the protocol correctness; this can be found in Appendix A.

A client command may either take a *fast path* or a *slow path*. The fast path consists of three communication steps, and is taken under no contention, no Byzantine failures, and during synchronous communication periods. The *slow path* is taken otherwise to guarantee the final commit order for commands, and incurs two additional steps.

4.3.1 System Model

We consider a set of nodes (replicas and clients), in an asynchronous system, that communicate via message passing. The replica nodes have identifiers in the set $\{R_0, ..., R_{N-1}\}$. We assume the Byzantine fault model in which nodes can behave arbitrarily. We also assume a strong adversary model in which faulty nodes can coordinate to take down the entire system. Every node, however, is capable of producing cryptographic signatures [80] that faulty nodes cannot break. A message m signed using R_i's private key is denoted as $\langle m \rangle_{R_i}$. The network is fully connected and quasi-reliable: if nodes R_1 and R_2 are correct, then p_2 receives a message from R_1 exactly as many times R_1 sends it.

To preserve safety and liveness, ezBFT requires at least $N = 3f + 1$ replica nodes in order to tolerate f Byzantine faults. ezBFT uses two kinds of quorums: a fast quorum with $3f + 1$ replicas, and a slow quorum with $2f + 1$ replicas. Safety is guaranteed as long as only up to f replicas fail. Liveness is guaranteed during periods of synchronous communication. Furthermore, there is no assumption on the number of faulty clients.

4.3.2 The Fast Path Protocol

The fast path consists of three communication steps in the critical path and one communication step in the non-critical path of the protocol. Only the communication steps in the critical path contribute to the client-side latency. The fast path works as follows.

1. Client sends a request to a replica.

The client c requests a command L to be executed on the replicated state by sending a message $\langle \text{REQUEST}, L, t, c \rangle_{\sigma_c}$ to a EZBFT replica. The closest replica may be chosen to achieve the optimal latency. The client includes a timestamp t to ensure exactly-once execution.

2. Replica receives a request, assigns an instance number, collects dependencies and assigns a sequence number, and forwards the request to other replicas.

When a replica R_i receives the message $m = \langle \text{REQUEST}, L, t, c \rangle_{\sigma_c}$, it becomes the command-leader for L. It assigns c to the lowest available instance number I_L in its instance space and collects a dependency set $\mathcal{D}$ using the command interference relation that was previously described. A sequence number $\mathcal{S}$ assigned for c is calculated as the maximum of sequence numbers of all commands in the dependency set. This information is relayed to all other replicas in a message $\langle \langle \text{SPECORDER}, O_{Ri}, I_L, \mathcal{D}, \mathcal{S}, h, d \rangle_{\sigma_{Ri}}, m \rangle$, where $d = H(m)$, h is the digest of R_i's instance space, and O_{Ri} is its owner number.

Nitpick. Before taking the above actions, R_i ensures that the timestamp $t > t_c$, where t_c is the highest time-stamped request seen by R_i thus far. If not, the message is dropped.

3. Other replicas receive the SPECORDER message, speculatively executes the command according to its local snapshot of dependencies and sequence number, and replies back to the client with the result and an updated set of dependencies and sequence number, as necessary.

When replica R_j receives a message $\langle \langle \text{SPECORDER}, O_{Ri}, I_L, \mathcal{D}, \mathcal{S}, h, d \rangle_{\sigma_{Ri}}, m \rangle$ from replica R_i, it ensures that m is a valid REQUEST message and that $I_L = maxI_{Ri} + 1$, where $maxI_{Ri}$ is the largest occupied slot number in R_i's instance space. Upon successful validation, R_j updates the dependencies and sequence number according to its log, speculatively executes the command, and replies back to the client. A reply to the client consists of a message $\langle \langle \text{SPECREPLY}, O_{Ri}, I_L, \mathcal{D}', \mathcal{S}', d, c, t \rangle_{\sigma_{Ri}}, R_j, rep, SO \rangle$, where rep is the result, and $SO = \langle \text{SPECORDER}, O_{Ri}, I_L, \mathcal{D}, \mathcal{S}, h, d \rangle_{\sigma_{Ri}}$.

4. The client receives the speculative replies and dependency metadata.

The client receives messages $\langle \langle \text{SPECREPLY}, O, I_L, \mathcal{D}', \mathcal{S}', d, c, t \rangle_{\sigma_{Rk}}, R_k, rep, SO \rangle$, where R_k is the sending replica. The messages from different replicas are said to match if they have identical O, I_L, $\mathcal{D}'$, $\mathcal{S}'$, c, t, and rep fields. The number of matched responses decides the fast path or slow path decision for command L.

Nitpick. Any two dependency sets are equal if both the sets have the same elements (commands).

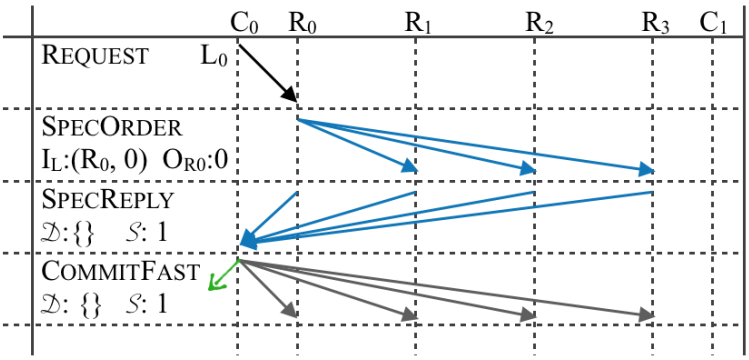


Figure 4.1: An example of a fast path execution.

4.1 The client receives $3f + 1$ matching responses.

The receipt of $3f + 1$ matching responses from the replicas constitutes a fast path decision for command L. This happens in the absence of faults, network partitions, and contention. The client returns reply *rep* to the application and then asynchronously sends a message $\langle \text{CommitFast}, c, I_L, CC \rangle$, where CC is the commit certificate consisting of $3f + 1$ matching SpecReply responses, and returns.

5. The replicas receive either a CommitFast or a Commit message.

5.1 The replicas receive a CommitFast message.

Upon receipt of a $\langle \text{CommitFast}, c, I_L, CC \rangle$ message, the replica R_i marks the state of L as committed in its local log and enqueues the command for final execution. The replica does not reply back to the client.

Example. Figure 4.1 shows an example case. The client sends a signed Request message to replica R_0 to execute a command L_0 on the replicated service. Replica R_0 assigns the lowest available instance number in its instance space to L_0. Assuming that no instance number was used previously, the instance number assigned to L_0 is $I_{L_0} = \langle r_0, 0 \rangle$. Then, R_0 collects dependencies and assigns a sequence number to L_0. As the first command, there exists no dependencies, so the dependency set $\mathcal{D} = \{\}$. Thus, the sequence number is $\mathcal{S} = 1$.

A signed SpecOrder message is sent to other replicas in the system with the command and compiled metadata. Other replicas – R_1 through R_3 – receive this message, add the command to their log, and start amassing dependencies from their log that are not present in $\mathcal{D}$. No other replica received any other command, thus they produce an empty dependency set as well, and the sequence number remains the same. Since there are no dependencies, all replicas immediately execute the command, speculatively, on their copy of the application state. The result of execution, unchanged dependency set, sequence number, and the digest

of log are sent in a SPECREPLY message to the client. The client checks for identical replies and returns the result to the application. The replies are identical in this case because no other command conflicts with L_0 in any of the replicas and the replicas are benign.

4.3.3 Execution Protocol

EZBFT uses *speculative execution* as a means to reply to the client quickly. However, the protocol must ensure that every correct replica has identical copies of the state. This means that, when necessary (as described in Section 4.3.4), the speculative state must be rolled back and the command must be re-executed correctly; this is called *final execution*.

Moreover, differently from existing BFT solutions, EZBFT collects command dependencies that form a directed graph with potential cycles. The graph must be processed to remove cycles and obtain the execution order for a command.

Each replica takes the following steps:

1. Waits for the command to be enqueued for execution. For final execution, wait for the dependencies to be committed and enqueued for final execution as well.

2. A dependency graph is constructed by including R and all its dependencies in $\mathcal{D}$ as nodes and adding edges between nodes indicating the dependencies. The procedure is repeated recursively for each dependency.

3. Strongly connected components are identified and sorted topologically.

4. Starting from the inverse topological order, every strongly connected component is identified, and all the requests within the component are sorted in the sequence number order. The requests are then executed in the sequence number order, breaking ties using replica identifiers. During speculative execution, the execution is marked *speculative* on the shared state. During final execution, the speculative results are invalidated, command re-executed, and marked *final*.

Note that speculative execution can happen in either the *speculative* state or in the *final* version of the state, which ever is the latest. However, for final execution, commands are executed only on the previous *final* version.

4.3.4 The Slow Path Protocol

The slow path is triggered whenever a client receives either unequal and/or insufficient SPECREPLY messages that is necessary to guarantee a fast path. The client will receive

unequal replies if the replicas have different perspectives of the command dependencies, possibly due to contention or due to the presence of Byzantine replicas. The case of insufficient replies happen due to network partitions or Byzantine replicas.

The steps to commit a command in the slow path are as follows.

4.2 The client receives at least $2f + 1$ possibly unequal responses.

The client c sets a timer as soon as a Request is issued. When the timer expires, if c has received at least $2f + 1$ $\langle\langle \text{SpecReply}, O_{Ri}, I_L, \mathcal{D}, \mathcal{S}, d, c, t\rangle_{\sigma_{Ri}}, R_j, rep, SO\rangle$ messages, it produces the final dependency set and sequence number for L. The dependency sets from a known set of $2f+1$ replicas are combined to form $\mathcal{D}'$. A new sequence number $\mathcal{S}'$ is generated if the individual dependency sets were not equal. c sends a $\langle \text{Commit}, c, I_L, \mathcal{D}', \mathcal{S}', CC\rangle_{\sigma_c}$ message to all the replicas, where CC is the commit certificate containing $2f+1$ SpecReply messages that are used to produce the final dependency set.

Nitpick. Each command-leader specifies a known set of $2f+1$ replicas that will form the slow path quorum, which is used by the client to combine dependencies when more than $2f + 1$ reply messages are received. This information is relayed to the clients by the respective command-leaders and is cached at the clients.

5.2 The replicas receive a Commit message.

Upon receipt of a $\langle \text{Commit}, c, I_L, \mathcal{D}', \mathcal{S}', CC\rangle_{\sigma_c}$ message, replica r updates command L's metadata with the received dependency set $\mathcal{D}'$ and sequence number $\mathcal{S}'$. The state produced after the speculative execution of L is invalidated, and L is enqueued for final execution. The result of final execution, *rep* is sent back to the client in a $\langle \text{CommitReply}, L, rep\rangle$ message.

6.2 The client receives $2f + 1$ CommitReply messages and returns to the application.

The client returns *rep* to the application upon receipt of $2f + 1$ $\langle \text{CommitReply}, L, rep\rangle$ messages. At this point, execution of command L is guaranteed to be safe in the system, while tolerating upto f Byzantine failures. Moreover, even after recovering from failures, all correct replicas will always execute L at this same point in their history to produce the same result.

Example

Figure 4.2 shows an example of a slow path. Two clients c_0 and c_1 send signed Request messages to replicas R_0 and R_3, respectively, to execute commands L_1 and L_2, respectively, on the replicated service. Assume that L_1 and L_2 conflict. Replica R_0 assigns the lowest available instance number of $\langle R_0, 0\rangle$ to L_1. Thus, R_0 collects a dependency set $\mathcal{D}_{L_1} = \{\}$ and assigns a sequence number $\mathcal{S}_{L_1} = 1$ to L_1. Meanwhile, R_3 assigns the instance number

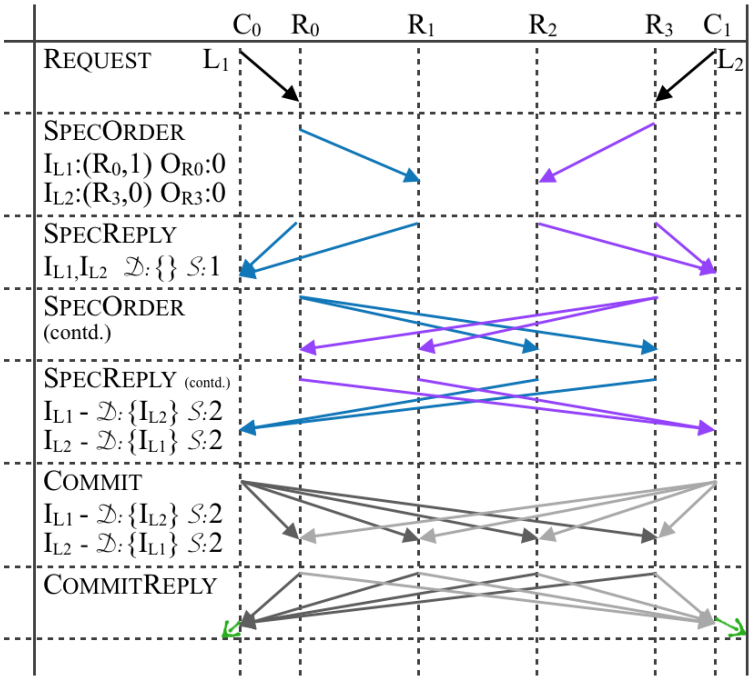


Figure 4.2: EZBFT: An example of a slow path execution.

$\langle R_3, 0\rangle$ to L_1; the dependency set is $\mathcal{D}_{L_2} = \{\}$, and sequence number is $\mathcal{S}_{L_2} = 1$. Replicas R_0 and R_3 send SPECORDER messages with their respective commands and their metadata to other replicas. Let's assume that R_0 and R_1 receive L_1 before L_2, while R_2 and R_3 receive L_2 before L_1. The dependency set and sequence number will remain unchanged for L_1 at R_0 and R_1, because the dependency set in the SPECORDER message received is the latest. However, the dependency set and sequence number for L_2 will update to $\mathcal{D}'_{L_2} = \{L_1\}$ and $\mathcal{S}'_{L_2} = 2$, respectively. Similarly, the dependency set and sequence number will remain unchanged for L_2 at R_0 and R_1, but for L_1, $\mathcal{D}'_{L_1} = \{L_2\}$ and $\mathcal{S}'_{L_1} = 2$, respectively. Replicas send SPECREPLY messages for both L_1 and L_2 with the new metadata to the respective clients c_0 and c_1.

Let's assume that the slow quorum replicas are R_0, R_1, and R_2 for R_0, and R_1, R_2, and R_3 for R_3. Since client c_0 observes unequal responses, it combines the dependencies for L_1 from the slow quorum and selects the highest sequence number to produce the final dependency set $\mathcal{D}_{L_1} = \{L_2\}$ and sequence number $\mathcal{S}_{L_1} = 2$. This metadata is sent to the replicas in a COMMIT message. Similarly, c_1 produces the final dependency set $\mathcal{D}_{L_2} = \{L_1\}$ and sequence number $\mathcal{S}_{L_2} = 2$ for L_2, and sends a COMMIT message to the replicas. The replicas update the dependency set and sequence number of the commands upon receipt of the respective COMMIT messages, and the commands are queued for execution. The replicas wait for the receipt of the COMMIT messages of all commands in the dependency set before processing them.

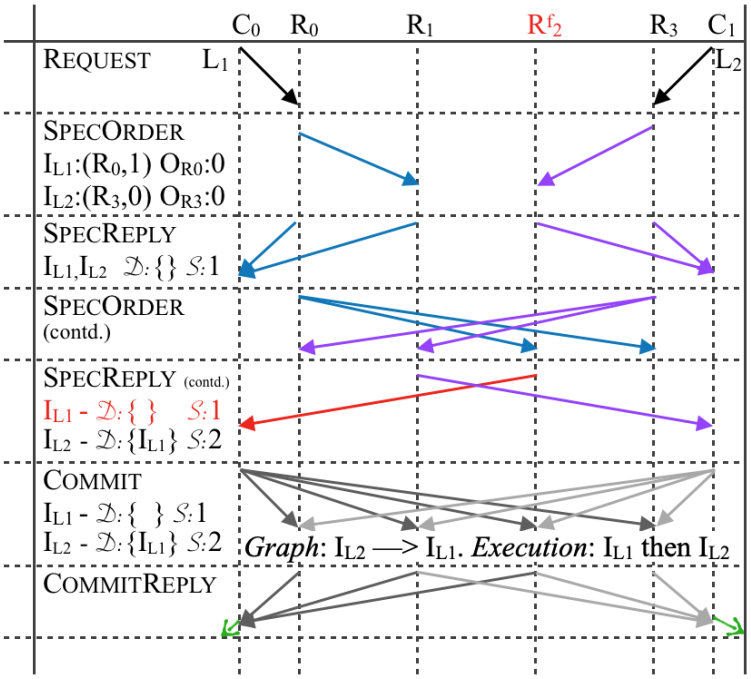


Figure 4.3: An example of a slow path with a faulty replica.

After the construction of the graph and the inverse topological sorting, there will exist commands L_1 and L_2 with a cyclic dependency between them. Since the sequence numbers for both the commands are the same and thus cannot break the dependency, the replica IDs are used in this case. Thus, L_1 gets precedence over L_2. L_1 is executed first, followed by L_2. The result of the executions are sent back to the clients. The clients collect $2f + 1$ reply messages and return the result to the application.

Example with a faulty replica

Figure 4.3 shows an example of the slow path that is very similar to that of Figure 4.2, but with a faulty replica R_2 that misbehaves. Notice that the Request and SpecOrder steps (first four rows) remain the same. Upon receipt of the SpecOrder message from R_0 and R_3 for L_1 and L_2, respectively, the replicas collect the dependency set and update the sequence number, and send back SpecReply messages to the client. For L_1, R_0 and R_1 send $\mathcal{D}'_{L_1} = \{\}$ and $\mathcal{S}'_{L_1} = 1$; however, R_2 misbehaves and sends $\mathcal{D}'_{L_1} = \{\}$ and $\mathcal{S}'_{L_1} = 1$, even though it received L_2 before L_1. For L_2, R_2 and R_3 send $\mathcal{D}'_{L_2} = \{\}$ and $\mathcal{S}'_{L_2} = 1$; R_1 sends $\mathcal{D}'_{L_2} = \{L_1\}$ and $\mathcal{S}'_{L_2} = 2$. c_0 receives a quorum of $2f + 1$ SpecReply messages, and sends a Commit message with an empty dependency set and a sequence number 1. On the other hand, the correct R_1 that participated in command L_1's quorum sends back the correct dependency set and sequence number. Therefore, the final commit message for L_2 will have L_1 in its dependency set. Thus, even though replicas immediately execute L_1 since L_1's final

dependency set is empty, they cannot do so for L_2. Correct replicas must wait until L_1 is *committed* before constructing the graph, at which point L_1 will be executed first due to the smallest sequence number, followed by L_2.

4.3.5 Triggering Owner Changes

ezBFT employs a mechanism at the clients to monitor the replicas and take actions to restore the service when progress is not being made. Although the slow path helps overcome the effects of a participant Byzantine replica, it does ensure progress of a command when its command-leader, the replica that proposed that command, is Byzantine. From the client-side, two events can be observed to identify misbehaving command-leaders.

4.3 The client times-out waiting for reply from the replicas.

After the client sends a request with command L, it starts another timer, in addition to the one for slow-path, waiting for responses. If the client receives zero or fewer than $2f + 1$ responses within the timeout, it sends the $\langle \text{REQUEST}, L, t, c, R_i \rangle_{\sigma_c}$ message to all the replicas, where R_i is the original recipient of the message.

When replica R_j receives the message, it takes one of the following two actions. If the request matches or has a lower timestamp t than the currently cached timestamp for c, then the cached response is returned to c. Otherwise, the replica sends a $\langle \text{RESENDREQ}, m, R_j \rangle$ message where $m = \langle \text{REQUEST}, L, t, c, R_i \rangle_{\sigma_c}$ to R_i and starts a timer. If the timer expires before the receipt of a SPECORDER message, R_j initiates an ownership change.

4.4 The client receives responses indicating inconsistent ordering by the command-leader and sends a proof of misbehavior to the replicas to initiate an ownership change for the command-leader's instance space.

Even though a client may receive prompt replies from the replicas, it must check for inconsistencies leading to a *proof of misbehavior* against the command-leader. The SPECREPLY message of the form $\langle \langle \text{SPECREPLY}, O_{Ri}, I_L, \mathcal{D}', \mathcal{S}', d, c, t \rangle_{\sigma_{Ri}}, R_k, rep, SO \rangle$ from different replicas are said to match if they have identical O_{Ri}, I, $\mathcal{D}$, $\mathcal{S}$, c, t, and *rep* fields, but the contention may affect the equality of the dependency set and sequence number fields. Thus, the command-leader is said to misbehave if it sends SPECORDER messages with different instance numbers to different replicas (i.e., if the I field varies between the replicas). The client c can identify by inspecting SPECORDER SO message embedded in the SPECREPLY message received from the replicas. In this case, the client collects a pair of such messages to construct a $\langle \text{POM}, O_{Ri}, POM \rangle$ message, where POM is a pair of SPECREPLY messages, proving misbehavior by the command-leader R_i of L.

4.3.6 The Owner Change Protocol

An ownership change is triggered for an instance space if its original owner is *faulty*. However, to initiate an ownership change, there must exist either a proof of misbehavior against the owner, or enough replicas must have timed out waiting for a reply from the owner.

A replica R_j commits to an ownership change by sending a $\langle$StartOwnerChange, R_i, $O_{Ri}\rangle$ message to other replicas, where R_i is the suspected replica and O_{Ri} is its owner number.

When another replica R_k receives at least $f+1$ StartOwnerChange messages for R_i, it commits to an ownership change. From this point forward, R_k will not participate in R_i's instance space. The new owner number is calculated as $O'_{Ri} = O_{Ri} + 1$, and the new command-leader is identified using O'_{Ri} *mod* N (henceforth R_l). Replicas that have committed to an owner-change send $\langle$OwnerChange$\rangle$ messages to the new leader. Once the new command-leader R_l receives $f+1$ OwnerChange messages, it becomes the new owner of R_i's instance space and finalizes its history.

Each replica sends an OwnerChange message containing its view of R_i's instance space, i.e., the instances (speculative) executed or committed since the last checkpoint, and the commit-certificate with the highest owner number that it had previously responded to with a commit message, if any. The new owner collects a set P of OwnerChange messages and selects only the one that satisfies one of the following conditions. For clarity, we label the sequence of instances in each OwnerChange message as P_i, P_j, and so on.

There exists a sequence P_i that is the longest and satisfies one of the following conditions.

Condition 1 P_i has Commit messages with the highest owner number to prove its entries.

Condition 2 P_i has at least $f+1$ SpecReply messages with the highest owner number to prove its entries.

If there exists a sequence P_j that extends a P_i satisfying any of the above conditions, then P_j is a valid extension of P_i if one of the following conditions hold:

1. P_i satisfies Condition 1, and for every command L in P_j not in P_i, L has at least $f+1$ SpecReply messages with the same highest order number as P_i.
2. P_i satisfies Condition 2, and for every command L in P_j not in P_i, L has a signed Commit message with the same highest order number as P_i.

The new owner sends a NewOwner message to all the replicas. The message includes the new owner number O'_{R0}, the set P of OwnerChange messages that the owner collected as a proof, and the set of safe instances G produced using Condition 1 and Condition 2. A replica accepts a NewOwner message if it is valid, and applies the instances from G in R_i's instance space. If necessary, it rolls-back the speculatively executed requests and re-executes them again.

At this point, R_i's instance space is frozen. No new commands are ordered in the instance space, because each replica has its own instance space that it can use to propose its command. The owner change is used to ensure the safety of commands proposed by the faulty replicas.

4.3.7 Correctness

We formally specified EZBFT in TLA+ and model-checked using the TLC model checker. The actions of Byzantine replicas were modelled after inspirations from the BFT Paxos specification in [93]. The TLA+ specification is provided in Appendix A. In this section, we provide an intuition of how EZBFT achieves its properties.

Nontriviality. Since clients must sign the requests they send, a malicious primary replica cannot modify them without being suspected. Thus, replicas only execute requests proposed by clients.

Consistency. To prove consistency, we need to show that if a replica R_j commits L at instance I, then for any replica R_k that commits L' at I, L and L' must be the same command.

To prove this, consider the following. If R_j commits L at $I = \langle R_i, - \rangle$, then an order change should have been executed for replica R_i's instance space. If R_j is correct, then it would have determined that L was executed at I using the commit certificate in the form of SPECORDER or COMMIT messages embedded within the CHANGEOWNER messages. Thus, L and L' must be the same. If R_j is malicious, then the correct replicas will detect it using the invalid progress-certificate received. This will cause an ownership change.

In addition, we need to also show that conflicting requests L and L' are committed and executed in the same order across all correct replicas. Assume that L commits with $\mathcal{D}$ and $\mathcal{S}$, while L' commits with $\mathcal{D}'$ and $\mathcal{S}'$. If L and L' conflict, then at least one correct replica must have responded to each other in the dependency set among a quorum of $2f + 1$ replies received by the client. Thus, L will be in L''s dependency set and/or viceversa. The execution algorithm is deterministic, and it uses the sequence number to break ties. Thus, conflicting requests will be executed in the same order across all correct replicas.

Stability. Since only f replicas can be Byzantine, there must exist at least $2f+1$ replicas with the correct history. During an ownership change, $2f + 1$ replicas should send their history to the new owner which then validates it. Thus, if a request is committed at some instance, it will be extracted from history after any subsequent owner changes and committed at same instance at all correct replicas.

Liveness. Liveness is guaranteed as long as fewer than f replicas crash. Each primary replica attempts to take the fast path with a quorum of $3f + 1$ replicas. When faults occur and a quorum of $3f + 1$ replicas is not possible, the client pursues the slow path with $2f + 1$ replicas and terminates in two additional communication steps.

Consistency Guarantees. EZBFT, by default, provides per-object linearizability [73] for

transitive operations, or specifically those that target a single object. For multi-object operations, the equivalent property is *strict-serializability*, which ezBFT provides since, each command is enqueued for final execution only after the commands in its dependency set are committed [117].

4.4 Evaluation

We implemented ezBFT, and its state-of-the-art competitors PBFT, FaB, and Zyzzyva in Go, version 1.10. In order to evaluate all systems in a common framework, we used `gRPC` [7] for communication and `protobuf` [67] for message serialization. We used the HMAC [85] and ECDSA [80] algorithms in Go's *crypto* package to authenticate the messages exchanged by the clients and the replicas. The systems were deployed in different Amazon Web Service (AWS) regions using the EC2 infrastructure. The VM instance used was m4.2xlarge with 8 vCPUs and 32GB of memory, running Ubuntu 16.04. We implemented a replicated key-value store to evaluate the protocols. Note that, for Zyzzyva and ezBFT, the client process implements the logic for the client portion of the respective protocols.

The key-value store is in-memory mapping string keys to byte array values. In our experiment, the keys and values were 8 bytes and 16 bytes, respectively. The store was initialized with 10,000 keys and random values.

Among the protocols evaluated, only ezBFT is affected by contention. Contention, in the context of a replicated key-value store, is defined as the percentage of requests that concurrently access the same key. Prior work [117] has shown that, in practice, contention is usually between 0% and 2%. Thus, a 2% contention means that roughly 2% of the requests issued by clients target the same key, and the remaining requests target clients' own (non-overlapping) set of keys. However, we evaluate ezBFT at higher contention levels for completeness.

4.4.1 Client-side Latency

To understand ezBFT's effectiveness in achieving optimal latency at each geographical region, we devised two experiments to measure the latency experienced by clients located at each region for each of the protocols.

Experiment 1

We deployed the protocols with four replica nodes in the AWS regions: US-East-1 (Virginia), India, Australia, and Japan. At each node, we also co-located a client that sends requests to the replicas. For single primary-based protocols (PBFT, FaB, Zyzzyva), the primary was

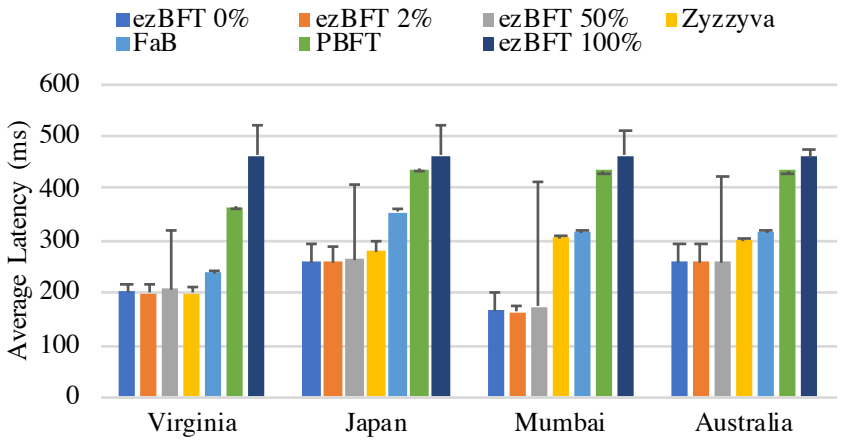


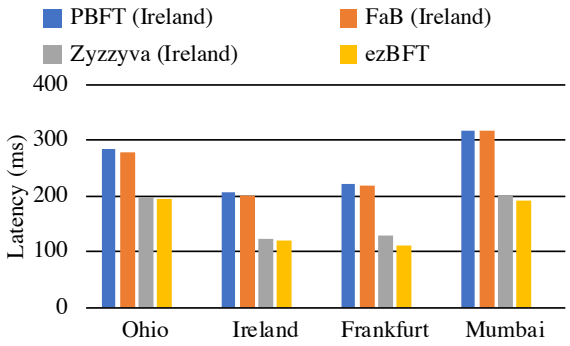
Figure 4.4: Median latencies for Experiment 1. All primaries are in US-East-1 region. The latency is shown per region as recorded by the clients in that region. Error bars indicate 99%ile.

set to US-East replica; thus, clients in other replicas send their requests to the primary. For EZBFT, the client sends its requests to the nearest replica (which is in the same region). The clients send requests in closed-loop, meaning that a client will wait for a reply to its previous request before sending another one.

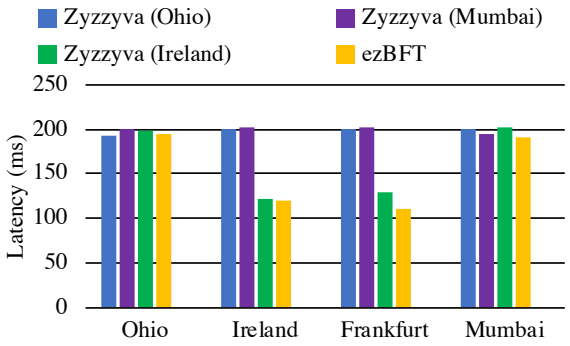
Figure 4.4 shows the median latency (in milliseconds) observed by the clients located in their respective regions (shown on x-axis) for each of the four protocols. For EZBFT, the latency was measured at different contention levels: 0%, 2%, 50%, and 100%; the suffix in the legend indicates the contention. Among primary-based protocols, PBFT suffers the highest latency, because it takes five communication steps to deliver a request. FaB performs better than PBFT with four communication steps, but Zyzzyva performs the best among primary-based protocols using only three communication steps. Overall, EZBFT performs as good as or better than Zyzzyva, for up to 50% contention. In the US-East-1 region, both Zyzzyva and EZBFT have about the same latency because they have the same number of communication steps and their primaries are located in the same region. However, for the remaining regions, Zyzzva clients must forward their requests to US-East-1, while EZBFT clients simply send their requests to their local replica, which orders them. At 100% contention, five communication steps required for total-order increases EZBFT's latency close to that of PBFT's.

Experiment 2

To better understand Zyzzyva's best and worst-case performances and how they fare against EZBFT, we identified another set of AWS regions: US-East-2 (Ohio), Ireland, Frankfurt, and India. This experiment was run similar to that of Figure 4.4. The primary was placed



(a) Median latencies for Experiment 2.
.



(b) Median latencies for Experiment 2. Zyzzva's primary is at different regions. .

Figure 4.5: Best case for Zyzzyva and the effect of moving primary to different regions. Experiments reveal ezBFT's effectiveness. Legend entries show primary's location in parenthesis.

in Ireland. The results are shown in Figure 4.5. Figure 4.5a shows the median latencies as observed by the clients in each deployed region for each of the four protocols. The choice of Ireland as the primary region represents the best case for Zyzzyva. Hence, ezBFT performs very similar to Zyzzyva.

In Experiment 1, the regions mostly had non-overlapping paths between them, and thus the first communication step of sending the request to the leader can be seen clearly (notice Mumbai in Figure 4.4. On the other hand, in Experiment 2, connections between the regions have overlapping paths. For example, sending a request from Ohio to Mumbai for ezBFT will take about the same time as sending a request from Ohio to Mumbai via the primary at Ireland for Zyzzyva.

Figure 4.5b shows the effect of moving the primary to different regions. We disregard PBFT and FaB in this case, as their performance do not improve. For Zyzzyva, moving the primary to US-East-2 or India substantially increases its overall latency. In such cases, ezBFT's latency is up to 45% lower than Zyzzyva's. This data-point is particularly important as it reveals how the primary's placement affects the latency.

To curb the negative effects of Byzantine primary replicas, in [42], the authors propose to frequently move the primary (this strategy is adopted by other protocols including Zyzzyva). From Figure 4.5b, we can extrapolate that such frequent movements can negatively impact latencies over time. Given these challenges, we argue that ezBFT's leaderless nature is a better fit for geo-distributed deployments.

Effect of Byzantine Replicas. While a Byzantine primary, in primary-based approaches, can cease progress, ezBFT enables non-Byzantine replicas to continue to make progress with the correct replicas using slow path, effectively isolating the impact of Byzantine replicas.

Therefore, the client-side latency observed for a four-replica system will be equivalent to that observed at 100% contention with a small change: the requests at the affected node will timeout until before the view change. After the view change, the new owner will complete any pending timed-out requests from the clients of affected replica. The new requests will be sent to the closest node. In our evaluation, with a timeout period of 3 seconds, the view change procedure completed in about 4 seconds. For more details, refer to the technical report [23].

4.4.2 Client Scalability

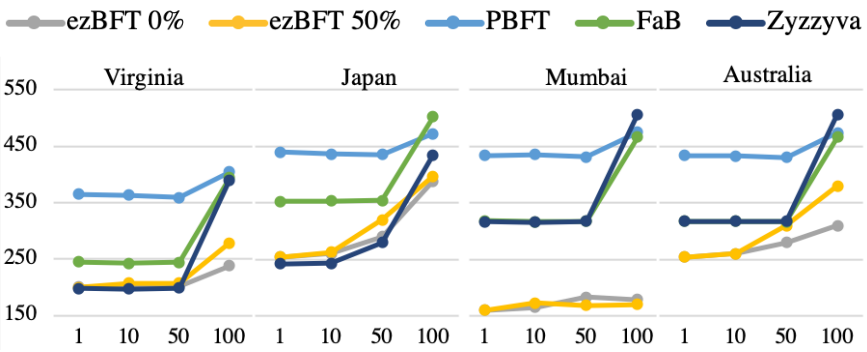


Figure 4.6: Latency per location while varying the number of connected clients (1 – 100) per region.

Another important aspect of EZBFT is its ability to maintain low client-side latency even as the number of connected clients increases. For this experiment, we deployed the protocols in Virginia, Japan, Mumbai, and Australia, and measured client-side latency per region by varying the number of connected clients. Figure 4.6 shows the results. Notice that as Zyzzyva approaches 100 connected clients per region, it suffers from an exponential increase in latency. However, EZBFT, even at 50% contention, is able to scale better with the number of clients. Particularly, in Mumbai, EZBFT maintains a stable latency even at 100 clients per region, while Zyzzyva's latency shoots up.

4.4.3 Server-side Throughput

We also measured the peak throughput of the protocols. For this experiment, we deployed the protocols in five AWS regions: US-East-1 (Virginia), India, Australia, and Japan. We co-located ten clients with the primary replica in US-East-1. Unlike the experiments so far, here the clients send requests in an open-loop, meaning that they continuously and asynchronously send requests before receiving replies.

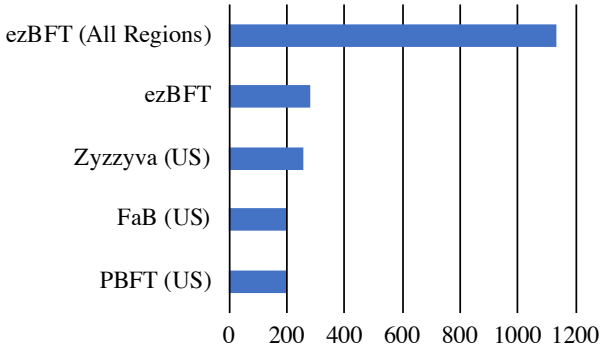


Figure 4.7: Throughput of ezBFT and competitor BFT protocols.

Figure 4.7 shows the results. For ezBFT, we carried out two experiments: i) clients are placed only at US-East-1 (labelled ezBFT in the figure); and ii) clients are placed at every region (labelled "ezBFT (All Regions)" in the figure). Due to the leaderless characteristic, each of the replicas can feed requests into the system, increasing the overall throughput. The contention was set to 0%, and no batching was done.

Observe that when clients are placed only at US-East-1, ezBFT performs at par or slightly better than others. On the other hand, when clients are placed in all the other regions, which does not yield any benefit for other protocols, ezBFT's throughput increases by as much as four times, as all ezBFT replicas are able to process and deliver requests concurrently.

4.5 Conclusions

State-of-the-art BFT protocols are not able to provide optimal request processing latencies in geo-distributed deployments – an increasingly ubiquitous scenario for many distributed applications, particularly blockchain-based applications.

This chapter presented ezBFT, a leaderless BFT protocol that provides three-step consensus in the common case, while essentially nullifying the latency of the first communication step. ezBFT provides the classic properties of BFT protocols including nontriviality, consistency, stability, and liveness. Our experimental evaluation reveals that ezBFT reduces latency by up to 40% compared to Zyzzyva.

Chapter 5

Leaderless Trust-based BFT with Dester

This chapter presents Dester, a leaderless, hybrid fault-tolerant state machine replication protocol. Dester has no designated leader. Each replica processes client commands by only relying on a set of geographically-nearest replicas, thereby avoiding potential leader bottlenecks, resulting in low client-side latencies and better system throughput. Dester is, in part, made possible by a novel trusted subsystem called TruDep that provides the necessary trust required to ensure secure, leaderless operation. TruDep is designed specifically for trusted execution environments (such as Intel SGX) with the goal of minimizing the number of lines of trusted code.

This chapter analyzes Dester, establishes its correctness, and shows that it commits client commands in two communication steps under favorable conditions (without any failures and command conflicts). We implemented Dester, conducted experimental evaluations in the Azure cloud platform, and compared it against state-of-the-art hybrid protocols (Hybster [32]), leader-based BFT protocols (PBFT [41], SBFT [64], Hotstuff [140]), and leaderless BFT protocols (ezBFT [23]). Our experimental evaluations reveal that Dester provides 60% reduction in latency and up to 2.4× better throughput than competitors in a 7-node geographically-replicated setting. Furthermore, the results reveal that Dester is better positioned for smaller deployments of up to $N = 19$ replicas. Nevertheless, Dester performs at par with competitors at larger deployments ($N = 49$ replicas).

Dester lies at the intersection of leaderless BFT/CFT protocols and hybrid BFT protocols. Leaderless protocols have been investigated in the CFT model as a way to overcome the previously mentioned disadvantages of leader-based protocols [21, 22, 116]. That body of work has shown that it is possible to construct an ordering of client commands without relying on a single leader, yielding optimal latencies for clients regardless of their geographic location. In the BFT model, hybrid protocols have recently received significant attention, in part due to the increasing commodity-scale availability of TEEs, which can be leveraged to reduce replication resources [32, 82, 98, 135]. However, existing hybrid protocols are leader-based, which increases latencies for geographically-remote clients while providing low latencies for local clients. Chapter 4 presents a leaderless BFT protocol that extends the body of work on leaderless CFT protocols to the BFT space, but in a non-hybrid setting. However, ezBFT's replication resources are higher ($N = 3f + 1$). Dester is the first leaderless hybrid protocol.

This chapter makes the following contributions:

1. A solution to the problem of low performance in geographical deployments of state-of-the-art hybrid protocols by proposing a novel leaderless protocol.
2. The notion of fast decisions and fast quorums in the hybrid fault model to enable consensus in two communication steps under favorable scenarios.
3. The design for a new trusted subsystem TruDep that attests messages and validates dependencies to enable construction of leaderless hybrid protocols.
4. Dester, a leaderless hybrid fault-tolerant consensus protocol that uses resources similar to CFT protocols but tolerates Byzantine faults with the help of the TruDep subsystem.

Thus, the paper's central contribution is Dester, a leaderless consensus protocol for the hybrid fault model that consistently provides low latency in geographical deployments and provides comparable throughput to existing throughput-oriented protocols at scale.

The rest of the chapter is organized as follows. Section 5.1 describes the hybrid fault model and overviews Dester. Section 5.2 presents the system model and discusses the quorum requirements for Dester. Section 5.3 describes the interfaces and implementation of the TruDep subsystem, presents the Dester algorithm in detail, and analyzes Dester's correctness. Section 5.4 evaluates Dester. Section 5.5 concludes the chapter.

5.1 Overview

We begin by briefly describing the hybrid fault model, followed by an overview of Dester.

5.1.1 Background

We begin by highlighting the most important practical failure models and describe the expectations of BFT and Hybrid protocols designed for these models.

Broadly, Byzantine failures can be classified into *Omission* faults and *Commission* faults [45]. Under omission faults, a replica may fail to send one or more messages specified by the protocol, but sends no incorrect messages. Crash faults are a subset of omission faults. Commission faults include all other non-omission faults in which a replica may send a message not specified by the protocol. This includes *equivocation*, i.e., the ability of a replica to make conflicting statements without being detected as having an intent to compromise consistency.

In the hybrid fault model, the ability of the replicas to equivocate is stripped away using a trusted subsystem [44, 98]. A trusted subsystem certifies the message sent by the replicas to ensure that a malicious replica cannot cause different correct replicas to execute different operations as their *i-th* operation. For example, in *Hybster* [32], replicas attest messages with *monotonically increasing counter certificates* using the `TrInX` trusted subsystem. Therefore, replicas cannot sign different messages using the same counter value without being detected.

Thus, additional resources that were required to detect equivocation in the BFT model are unnecessary in the hybrid model. One fundamental change is that the number of replicas required to tolerate f failures reduces from $3f + 1$ to $2f + 1$. In the hybrid model, at most f replicas may behave arbitrarily except the trusted subsystem, which can only fail by crashing.

5.1.2 Intuition and Contributions

DESTER has been designed specifically to take advantage of the recent hardware and cloud trends – such as trusted execution environments and geo-distributed deployments – to achieve a low latency system with an optimal replication factor and a practically sufficient fault tolerance model. The hybrid fault model adopted by DESTER can defend systems from most Byzantine behaviors, such as software bugs and hardware errors, as long as the trusted subsystem is secure. Moreover, a substantial reduction in the number of nodes to $2f + 1$ (from $3f + 1$ in the BFT model) and the use of CPU-based trusted execution environments instead of external devices [44, 82, 135], help boost performance and resilience to a larger extent [32].

DESTER's hybrid fault model is made possible by its trusted subsystem, TRUDEP. The subsystem has been designed specifically to accommodate DESTER's requirements and take unique advantage of flexibility provided by CPU-based TEEs. Every replica has an instance of the TRUDEP subsystem, and its purpose is to ensure that the messages exchanged by DESTER replicas are correct and that malicious replicas cannot equivocate without detection. In DESTER, equivocation means that a replica can propose different commands to different replicas for the same ordering position or may withhold dependency information (described below) for commands from other replicas. TRUDEP's purpose is to prevent such equivocation by certifying messages sent and verifying the messages received by replicas.

Furthermore, DESTER's leaderless operation is key in unlocking low latency and high performance in geographic deployments. Leaderless operation enables every replica to propose and commit commands without relying on designated replicas. As we show in our evaluation in Section 5.4, the use of leader, or *primary* replicas has many disadvantages. Firstly, clients from non-leader sites experience very high latencies by forwarding commands to a (distant) leader. Secondly, very high load at the leader causes resource bottlenecks leading to low throughput. Finally, unavailability is inevitable under a faulty leader.

DESTER exploits the notion of *command interference* to allow concurrent ordering and execution of commands originating at different replicas. The technique, adopted from the Generalized Consensus [89] definition in the CFT model, involves executing concurrent commands in any order as long as they do not interfere. This enables DESTER replicas to commit client commands in two communication steps from receipt. Furthermore, DESTER applies caution when ordering interfering commands concurrently, thus an additional communication step may be required depending on the outcome of prior steps. We also use an alternate term *conflicting* to denote command interference.

To establish a relationship between interfering commands in different instance spaces, DESTER collects command dependencies. Two commands are dependent, if they interfere and thus, a deterministic execution order is required to ensure system consistency. Replicas only execute a command, when at least a quorum of replicas observe or agree to the same set of dependencies for that command. Furthermore, the execution of command involves respecting these dependencies deterministically across replicas.

5.2 Preliminaries

5.2.1 System Model

We consider a set of nodes (replicas and clients), in an asynchronous system, that communicate via message passing. The replica nodes have identifiers in the set $\{r_0, ..., r_{N-1}\}$. We assume the hybrid fault model in which nodes can behave arbitrarily, except the trusted subsystem that can only fail by crashing. Every node, however, is capable of producing cryptographic signatures [80] that faulty nodes cannot break.

To preserve safety and liveness, DESTER requires at least $N = 2f + 1$ replica nodes in order to tolerate f arbitrary faults. Safety is guaranteed as long as only up to f replicas fail. Liveness is guaranteed during periods of synchronous communication.

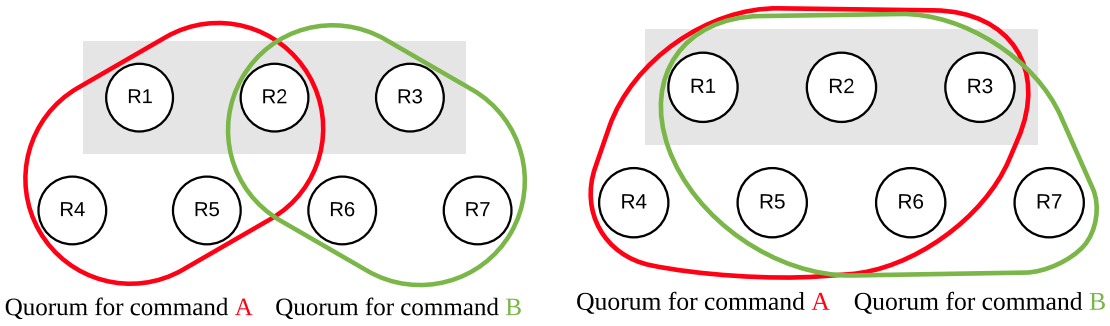
We consider an adversary that controls all the system software including the operating system. Thus, the adversary can schedule multiple instances of the same enclave, offer the latest and previous versions of sealed data, and block, delay, and read and modify all messages sent by the enclaves. However, the adversary cannot read or modify the enclave runtime memory or learn any information about the secrets held in enclave data. Furthermore, the enclave is capable of generating cryptographic operations that the adversary cannot break. We also assume that the adversary cannot compromise the SGX protections on participating nodes (e.g. via physical attacks).

5.2.2 Fast Decisions and Quorums

DESTER can solve consensus *efficiently* in two communication steps without a global leader under the common case. It has been shown in crash and Byzantine fault models [90, 108, 109] that a larger size of quorums, called *fast quorums* can be exploited to reduce the communication steps under favorable conditions yielding *fast decisions*. However, the definition of fast decisions diverge between the crash and Byzantine fault models. Lamport reduced the number of communication steps for CFT consensus by alleviating the leader role when different nodes propose commands during non-overlapping intervals. The leader steps in to enforce global order when nodes propose concurrently and cannot reach quorum agreement.

On the other hand, Martin and Alvisi [108] reduced the number of communication steps for BFT consensus under the presence of leader. BFT consensus requires one communication step more than that required in the CFT model. Thus, reducing communication in this case was under the presence of leader.

DESTER uses the former definition. Due to use of TEE, hybrid protocols have only one step after the initial leader broadcast step (e.g. Prepare [32]), so it is optimal in that sense. Thus, DESTER uses fast decision to provide leaderless operation. Note that Fast Paxos [90] uses a global leader as a fallback, while DESTER does not rely on any specific leader role at all. A fast decision requires a fast quorum to ensure that enough nodes have information to aid in view changes.



(a) Majority Quorum.

(b) DESTER Fast Quorum .

Figure 5.1: Classic Majority Quorum vs. DESTER Fast Quorum. Despite loss of f (shaded) nodes, the remaining nodes constitutes a majority in a fast quorum but not so in a classic quorum.

Fast decisions require fast quorums. The reason a classic majority quorum won't work is that there is only one node at the intersection of any two quorums. For instance, if two interfering commands are proposed in two different quorums concurrently, the node in the intersection is the only one to be aware of both the commands and decides their order. The loss of this node due to a fault will jeopardize the decision taken for those commands.

Therefore, we adopt a larger quorum definition from Fast Paxos. An intuition why this works is shown in Figure 5.1. It can be observed in Figure 5.1b that despite the loss of f nodes in a fast quorum, the remaining fast quorum nodes will constitute a majority among the rest of the nodes. This will ensure that there will be at least one node available always that participated in the quorums for any pair of interfering commands, and that node will ensure that consistent decisions are taken during view changes.

However, a fast quorum may not always be reachable due to the possibility of faults. In addition, DESTER nodes may not agree on the dependencies for a command to yield a fast decision. Thus, a slow decision procedure that uses a classic quorum is required. The general rule is that two fast quorums and a classic quorum must have a non-empty intersection [90].

If any two interfering commands from different nodes are unable to reach a fast decision, a slow decision can be safely reached only if the slow quorum consists of a node that was in the intersection of the two fast quorums. Dester uses two kinds of quorums: a fast quorum $\mathcal{FQ}$ with $f + \left\lfloor \frac{N}{2} \right\rfloor$ replicas, and a slow quorum $\mathcal{SQ}$ with $f + 1$ replicas. A fast quorum is required for guaranteeing responses in two communication steps under Dester's leaderless operation. Slow quorum is used for the third communication step and during the recovery procedure.

5.3 Protocol Design

This section presents the Dester protocol. We begin with a description of the trusted subsystems and then explain the core Dester algorithm along with the pseudocode.

5.3.1 The TruDep Subsystem

The role of a trusted component in a hybrid fault tolerance system is to ensure that replicas cannot equivocate without detection. In that regard, the TruDep subsystem solves two important challenges for Dester. First, by attesting every message with a *monotonically increasing counter*, it ensures that the protocol cannot send contradicting messages to different replicas. For instance, if a replica r_0 has already sends a message m with counter value i to r_1, then r_0 cannot send another message m' with the same counter value i to r_2, because the subsystem only attests a message with a counter value once. Second, by providing dependency validation, the subsystem ensures that replicas cannot equivocate on the set of dependencies for client commands. Before a message for a client command is sent to other replicas, TruDep includes the actual dependency set for the command before attesting it with a counter value. Moreover, the receiving replica's TruDep system checks the attestation and extracts the dependencies from the message and updates its internal data structure to aid in future validations.

The TruDep provides the following properties:

1. Counter Uniqueness: The certificate returned for any two create interface calls are not assigned the same counter value.
2. Counter Monotonicity: The next assigned counter value is always greater than the previous one.
3. Counter Sequentiality: The counter value is always incremented by one.
4. Dependency Set Validity: The subsystem neither hides nor overstates any dependency for the command. The dependency set provided for a command c only includes all the identifiers whose commands conflict with c as observed by the trusted subsystem.

The TRUDEP subsystem provides four interfaces: two for creating certificates and two for verifying certificates. Each replica has an instance of the TRUDEP subsystem.

- `CreateTDI(`i`, `c`)`: Given a replica identifier i and a command c, this interface assigns an identifier $id = \langle i, ctrs[i] \rangle$, computes the dependency set dep for c, and attach a MAC certificate that proves the validity of identifier id and dependency set dep. The dependency set is the set of commands conflict with c as observed by that particular instance of the TRUDEP subsystem. $ctrs$ is the array of counter values maintained by the subsystem to aid in assigning monotonic counter values to certificates. Each generated certificate is assigned a unique, non-decreasing counter value to ensure that any two certificates are not assigned the same counter value.
- `VerifyTDI(`i`, `c`, `id`, `dep`, `$cert$`)`: This interface is used to verify the validity of the certificate produced by another TRUDEP instance. The instance verifies whether the $cert$ is valid MAC certificate created for that particular command c with identifer id and dependency set dep and returns a boolean value indicating the outcome. Furthermore, the verifying instance also learns any new dependency information from the dep set received from another instance. This learning aids in future `CreateTDI` calls to provide the most up to date dependency information.
- `CreateTDVC(`i`, `m`)`: This interface is used to create a certificate $cert$ that represents the state of replica i's counter $ctrs[i]$. The purpose of such a certificate is to indicate to correct replicas the number of identifiers that have been generated by incrementing $ctrs[i]$ and computing a MAC certificate over the message m and $ctrs[i]$.
- `VerifyTDVC(`i`, `m`, `id`, `$cert$`)`: This interface verifies the TDVC certificate by validating that the $cert$ is the proper MAC certificate for m and id.

Every TRUDEP instance has the following variables, in addition to the secret key used for creating certificates. $ctrs$ is an array of numbers, one for each replica in the system. A counter is used to ensure the first three properties of the subsystem stated above. $cmds$ is a map of instances, whose entry holds information about a command interference relation. It is used to compute the dependencies for a command before certification.

5.3.2 USIG Subsystem

DESTER uses another trusted subsystem to aid with the slow phase of the protocol. This subsystem is the *Universal Sequential Identifier Generator* (USIG) subsystem used by the MinBFT protocol [135]. The USIG subsystem provides the first three properties of the TRUDEP subsystem, namely uniqueness, monotonicity, and sequentiality. The subsystem attaches a unique counter value to a message and generates a MAC certificate. This ensures that any two different messages always have different counter values. To create and verify certificates, the USIG provides two interfaces:

- `CreateUI(`m`)` creates a signed certificate UI_i for message m with the next value from the monotonic counter. The certificate is computed using the secret key of the USIG instance. $UI_i = \langle ctr, H(m) \rangle_p$, where ctr is the counter value and $H(m)$ is the hash of the message.

- `VerifyUI(`UI_i`,` m`)` uses the USIG instance i's public key and verifies whether the certificate UI_i was computed for message m.

5.3.3 Overview

The main goal of DESTER is to ensure that conflicting commands execute in the same order at all replicas. Given two commands c and c' that conflict, we use the notation $c \mapsto_i c'$ [59] to indicate that c executes before c' at replica r_i. We use the notation $c \rightsquigarrow c'$ to denote a real-time order in which c was executed before c' is submitted at some replica. To simplify, we define $\mapsto = \rightsquigarrow \cup (\cup_{i \in N} \mapsto_i)$.

Protocol Properties. DESTER ensures the following properties [59, 117]:

1. **Nontriviality.** Any request committed and executed by a correct replica must have been proposed by a client.
2. **Consistency.** Two replicas can never have different commands committed for the same instance.
3. **Ordering.** Given two conflicting commands, they will be executed in the same deterministic order by every replica.

5.3.4 Protocol Details

Each replica r_i has its own set of sequence numbers with identifiers in the set $\mathcal{I} = \{\langle i, 1 \rangle, \langle i, 2 \rangle, ...\}$ where i is the replica identifier. In addition, each replica maintains the following data structures: `cmd` is a mapping from an unique command identifer to a command's payload; `status` is a mapping that tracks the progress of a command by its identifier.

Propose phase

A client submits a command c to a DESTER replica by invoking the `submit(`c`)` function. The replica becomes the command's initial coordinator and invokes the `CreateTDI` function to generate a triple or a 3-tuple (line 2). The triple consists of the unique identifier for the command $id = \langle i, l \rangle$, where i is the replica identifier and l is the next available sequence number; dependencies for the command dep; and a signature $cert$. Then, the coordinator starts the PROPOSE phase and sends a `Propose` message (line 4) with the identifier, command, dependencies, and signature to replicas in a fast quorum $Q \subseteq \mathcal{FQ}$.

Upon receiving the `Propose` message from the coordinator (line 5), a replica in the fast quorum invokes the `VerifyTDI` interface to assert that the identifer and dependencies included for the command are valid (line 6). The replica then begins the Fast Commit phase and attempts to commit the command in the next communication step.

Fast Commit phase

After verifying the certificate received in the `Propose` message, a replica in quorum Q invokes the `CreateTDI` interface to generate its share of dependencies along with a certificate (line 7). The command, dependencies, and quorum information are recorded by the replica. The status of the command is set to PROPOSE. The replica sends a `FastCommit` message to all replicas (line 10). The `FastCommit` message includes the command identifier, dependencies, and the signature that attests the dependencies.

Another replica that receives a `FastCommit` message verifies its validity using the attached certificate by invoking the `VerifyTDI` function. After a replica receives a quorum of `FastCommit` messages, it checks if the received $deps_j$ from each of the Q replicas are all the same (line 16). If the dependencies match, it means that at least a fast quorum of the replicas agree on a common dependency set and that this set can be retrieved from at least one correct replica that is at the intersection of a slow quorum. Furthermore, replicas make commit decisions independently using only the `FastCommit` messages they receive. Replicas satisfy the following invariant.

Invariant 1. If replicas r_0 and r_1 commit commands with $(id,\ c,\ D)$ and $(id,\ c',\ D')$ respectively, where id is the command identifier, c is the command payload and D is the dependency set, then $c = c'$ and $D = D'$.

The TRUDEP subsystem produces a unique identifier for each command, and attaches a dependency set that reflects the conflicts of the command as seen by the replica when the interface is invoked. The dependency set is used to ensure that the conflicting commands are executed in the same order across all replicas and allow non-conflicting commands to be executed in any order. Thus, given two conflicting commands, at least one command must be in the dependency set of the other, which is captured by the following invariant.

Invariant 2. Assume two commands with $(id,\ c,\ D)$ and $(id',\ c',\ D')$ have been committed. If $id \neq id'$ and c and c' conflict, then either $id' \in D$ or $id \in D'$, or both.

Slow Commit phase

When the dependency sets from the Q `FastCommit` messages do not match, then a replica cannot decide a final common order for the command and must take additional steps to enforce a common decision. Moreover, if the initial coordinator does not make timely progress,

another replica must take over the commands and ensure a consistent decision across views. Otherwise, if replicas do not agree on the dependencies, later coordinators may try to enforce a different dependency set that could violate consistency guarantees. In order to guarantee a consistent decision across views, replicas must agree on the dependency set before committing the command. The slow commit phase serves this purpose.

The slow commit phase is akin to the Commit phase of the two-phase MinBFT protocol [135]. The Prepare phase is implicit from the outcome of the FastCommit phase as every replica takes the same action using the same fast quorum of `FastCommit` messages.

The slow commit phase takes one communication step. Each fast quorum replica accumulates the $deps_j$ received from the fast quorum of replicas to produce the dependency set D. Now, replicas must reach consensus on the dependency set D for the command. It generates a USIG identifier UI by calling the `CreateUI` interface. The replica sends a `SlowCommit` message containing both the USIG and TruDep identifiers along with the dependency set D to a slow quorum of replicas. Note that to verify the *cert*, replicas also need the `FastCommit` messages. We assume that $Q' \subseteq Q$; thus, replicas in Q' should already have the `FastCommit` message. However, any replica can obtain the `FastCommit` messages by requesting for it from other replicas.

When a replica receives $Q' \in \mathcal{SQ}$ `SlowCommit` messages, it verifies both the USIG and TruDep identifiers, ensures that the dependency set D is properly computed from the fast quorum Q. Finally, the replica updates its dep mapping for id with the received dependency set and sets the command status to COMMIT.

The following theorem implies that accumulation of dependency sets to generate the `SlowCommit` message maintains Invariant 2.

Theorem 5.1. *Given two conflicting commands with identifiers id and id′ and dependencies D and D′ computed as in line 16 over fast quorums, either id′ ∈ D or id ∈ D′, or both.*

Proof. Proof by contradiction. Assume there are two conflicting commands with identifiers id and id' and dependencies dep and dep' such that $id' \notin dep$ and $id \notin dep'$. Both dep and dep' are computed by accumulating the dependency set of a majority of replicas, and any two majority quorums Q and Q' must intersect. If $id' \notin dep$, then Q observed id before id'. Similarly, if $id \notin dep'$, then Q' observed id' before id. This is a contradiction as since the quorums intersect at least one replica must have seen that id before id' or vice versa and will include one in the other's dependency set.

□

Algorithm 1: DESTER: Normal case protocol at Replica i.

cmd[id] $\leftarrow$ $nil \in C$	`Command`
status[id] $\leftarrow$ START	`Command Status`
dep[id] $\leftarrow \emptyset \subseteq \mathcal{I}$	`Dependency Set`
quorum[id] $\leftarrow \emptyset \subseteq \mathcal{P}$	`Fast Quorum`
view[id] $\leftarrow id.1$	`View`
msgs $\leftarrow \emptyset$	`Set of sent messages`

1 **function** *Submit*(c)
2 $\langle id, dep, cert\rangle \leftarrow$ `CreateTDI`(i, c)
3 $Q \leftarrow$ `fast_quorum`(i)
4 **send** `Propose`(id, c, dep, Q, $cert$) to Q

5 **receive** *Propose*(id, c, dep, Q, $cert$) *from* j
6 **assert** `VerifyTDI`(j, c, id, dep, $cert$)
7 $\langle id', dep', cert'\rangle \leftarrow$ `CreateTDI`(i, c)
8 cmd[id] $\leftarrow c$; quorum[id] $\leftarrow Q$
9 dep[id] $\leftarrow dep'$; status[id] $\leftarrow$ PROPOSE
10 **send** `FastCommit`(id, id', dep', $cert'$) to Q

11 **receive** *FastCommit*(id, id', dep_j, $cert'$) from all $j \in Q$
12 **assert** `VerifyTDI`(j, cmd[id], id', dep_j, $cert'$) $\wedge$ status[id] = PROPOSE $\wedge$
13 **if** $\forall k, l \in Q : dep_k = dep_l$ **then**
14 dep[id] $\leftarrow dep_j$; status[id] $\leftarrow$ COMMIT
15 **else**
16 $D \leftarrow \cup_Q dep_j$; $v \leftarrow$ view[id]
17 $UI_i \leftarrow$ `CreateUI`($\langle id$, cmd[id], $D\rangle$)
18 $Q' \leftarrow$ `slow_quorum`($id.1$)
19 **send** `SlowCommit`(id, D, v, UI_i, $cert'$) to Q'
20 **end**

21 **receive** *SlowCommit*(id, D_j, v, UI_j, $cert'$) from all $j \in Q$
22 **assert** `VerifyTDI`(j, cmd[id], id', dep_j, $cert'$) $\wedge$ `VerifyUI`($\langle$ id, cmd[id], D_j $\rangle$, UI_j) $\wedge$
$\forall k, l \in Q : D_k = D_l$
23 dep[id] $\leftarrow D_j$; status[id] $\leftarrow$ COMMIT

Algorithm 2: DESTER: Recovery protocol.

24 **function** *RequestViewChange() of coordinator k*
25 $vcs \leftarrow \{(v, v+1) : v = view[id] \wedge v \mod N = k\}$
26 **send** `ReqViewChange`(r_i, r_k, vcs) to all $r_j \in \mathcal{R}$

27 **receive** *ReqViewChange(r_i, r_k, rec) from $f+1$*
28 $O \leftarrow \{m \in msgs : m.1 = r_k\}$
29 $UI \leftarrow$ `CreateUI`(O)
30 $cert \leftarrow$ `CreateTDVC`()
31 **send** `ViewChange`(r_i, v', O, UI, $cert$)

32 **receive** *ViewChange(r_j, v_j, O_j, UI_j, $cert_j$) from all $j \in Q$*
33 **assert** `VerifyTDVC`($cert_j$) $\wedge$ `VerifyUI`(O_j, UI_j)
34 $VCQ \leftarrow$ ViewChange from all $j \in Q$
35 $S \leftarrow$ `ComputeNextViewState`()
36 $UI \leftarrow$ `CreateUI`(S)
37 **send** `NewView`(r_i, v', VCQ, S, UI) to all

38 **receive** *NewView(r, v', VCQ, S, UI) from j*
39 **assert** `VerifyUI`(S, UI) `ValidateNewViewState`
40 **update** deps, cmd, quorum, view, status for each id

Command Recovery and View Change

When the initial coordinator of the command fails to make timely progress, a different replica takes over the command and takes steps to commit it. This entails a view change, in which a replica r_i attempts to take over the commands that were coordinated by a failed replica r_f and establishes a new view in which replica r_i attempts to commit r_f's commands. To aid this, each command identifier is associated with a view number. The initial view of each command identifier is i, where r_i is the initial coordinator. For each identifier, the view numbers are allocated to replicas in round-robin order. The view change procedure establishes a new view together for all the commands that were coordinated by the failed replica.

The View Change protocol is akin to MinBFT's View Change protocol and is triggered if at least $f+1$ replicas observe that a replica is not making timely progress. When a replica r_i observes that replica r_k is not making timely progress in its coordinator role, it expresses its concern with a `ReqViewChange` message (line 25). Since malicious replicas can spuriously trigger view changes stalling liveness, a correct replica only commits to a view change only if it receives $f+1$ `ReqViewChange` messages.

After receiving $f+1$ `ReqViewChange` messages, a replica commits to changing the view for all the commands coordinated by r_i and instating a new replica to coordinate those commands.

Note that the new coordinator is picked in round-robin order using the new view number. Given the initial coordinator is i, the next coordinator is $i + 1 \mod N$.

After committing for a view change, a replica sends a `ViewChange` message to all replicas. To construct the `ViewChange` message, a replica generates both the TRUDEP view change and the USIG certificates. The TRUDEP certificate is created by invoking `CreateTDVC` and the USIG certificate is created by invoking `CreateUI`. By attaching the USIG certificate, the replica is forced to reveal all the messages with counter values preceding the `ViewChange` message's *UI* certificate. With this information, the new coordinator can enforce a valid view and consequently, other replicas can also verify that the new coordinator is enforcing a valid view.

Once the new coordinator receives $f + 1$ `ViewChange` messages (line 32), it verifies both the TRUDEP and USIG certificates, and constructs the new view state. For each command identifier in the message logs, the state of the command in the new view is computed using the following rules.

- If at least one replica has sent the `SlowCommit` message for command c with identifier id and dependencies D, then id is included in the new state S with dependencies D from the `SlowCommit` message.
- If there is a replica that sent the `FastCommit` message, the dependencies are accumulated from $Q \cap Q_k$ replicas, where Q_k is the original quorum for the `FastCommit` message.
- A *noOp* is proposed, since there is no sufficient information to either prove that a fast path or a slow path commit has taken place.

Once the new view state S is computed, a `NewView` message is sent along with a UI certificate. Each replica that receives the `NewView` message will validate the certificate and verify that the state S is valid by performing the same computation as the new coordinator. If the state is invalid, it requests to change to view $v + 2$. Otherwise, it applies the state and executes the committed commands.

Execution

DESTER's execution protocol is similar to Atlas' execution protocol [59] and is presented in Algorithm 3. At each replica, the execution protocol runs in an infinite loop and attempts to execute committed commands. A command can only be executed if both the command and its dependencies are either committed or executed. Moreover, the execution protocol gathers a set S of committed commands whose dependencies are either committed and in set S, or are already executed. Once S is computed, the elements of the set are ordered deterministically by the identifiers and executed one by one.

Algorithm 3: DESTER: Execution protocol.

```
41 loop
42     commit ← {id ∈ I | status[id] = COMMIT}
43     execute ← {id ∈ I | status[id] = EXECUTE}
44     S ← {id ∈ commit | dep[id] ⊆ S ∪ execute}
45     for id ∈ S ordered by < do
46         execute(cmd[id])
47         status[id] ← EXECUTED
48     end
49 end
```

The execution protocol ensures that if two commands c and c' with identifiers id and id' are in two different sets S and S', respectively, and S is executed before S', then $id' \notin deps[id]$.

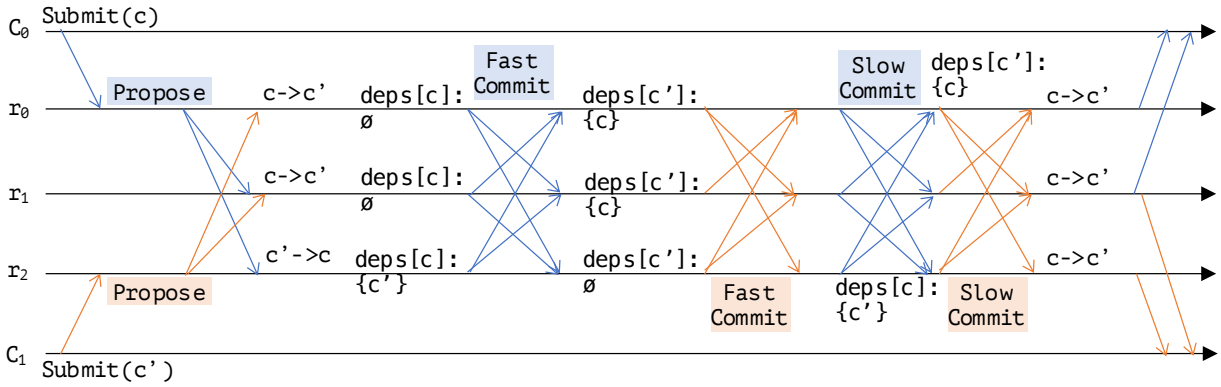


Figure 5.2: DESTER: An example execution.

Example. Figure 5.2 shows an example execution of DESTER, highlighting both fast commit and slow commit phases. The example consists of three replicas r_0 through r_2 and two clients C_0 and C_1. Since $N = 3$ replicas, the size of the fast quorum is three and the slow quorum is two.

Two clients C_0 and C_1 submit their signed commands c and c', respectively, to replicas r_0 and r_2, to be executed on the replicated service. Assume that c and c' conflict. Replica r_0's TRUDEP assigns identifier $id = \langle 0, 0 \rangle$ to c, computes dependency set $dep = \emptyset$, and provides a certificate $cert$. Then, r_0 sends a PROPOSE message with its commands and metadata to a fast quorum of replicas, which in this case includes all replicas. Similarly, Replica r_2's TRUDEP assigns identifier $id = \langle 2, 0 \rangle$ to c', computes dependency set $dep' = \emptyset$, and provides a certificate $cert$. Then, r_2 sends a PROPOSE message with its commands and metadata to a fast quorum of replicas. Let r_0 and r_1 receive the proposal for c before c', while r_2 receives the proposal for c' before c. Thus, r_0 and r_1 compute dependency sets $deps[c] = \emptyset$ and

$deps[c'] = \{c\}$, while r_2 computes dependency sets $deps[c'] = \emptyset$ and $deps[c] = \{c'\}$. Replicas then share the updated dependency sets by sending `FastCommit` messages for commands c and c' respectively.

Each replica that receives the `FastCommit` messages for both c and c' conclude that a slow phase is necessary since different replicas observe different dependency sets. Therefore, replicas send the `SlowCommit` messages for both c and c' with the aggregated dependency sets $deps[c] = \{c'\}$ and $deps[c'] = \{c\}$. The replicas mark the commands c and c' as committed after receiving a slow quorum of `SlowCommit` messages.

After committing the commands, the replicas run the execution algorithm to execute both c and c' in the same loop of the execution algorithm. Let $c < c'$, thus c will be executed before c'. Clients accepts the result after receiving the reply from $f + 1$ replicas.

Remarks

DESTER logs all the messages and command metadata to aid during retransmissions and dependency computations. In order to garbage collect executed commands, their metadata, and related messages, replicas periodically run the checkpointing protocol. This protocol is similar to those in existing protocols [135]. Each replica sends a `Checkpoint` message indicating the last identifier that it has executed, the hash of its state, and a certificate obtained by calling `CreateUI` interface. A replica that collects $f + 1$ `Checkpoint` messages with the same information and valid UIs, called a checkpoint certificate, concludes that the state of all commands before the identifier id included in the checkpoint can be safely garbage collected. The replica thus discards all the commands and their metadata with identifier less than and equal to id.

A lagging replica may not be able to send and receive messages for identifiers that have been garbage collected. In this case, the replica must perform a state transfer to reach an up-to-date state. The state transfer protocol is identical to the one presented in [41, 42].

5.3.5 Correctness

In this section, we present the proof of correctness of DESTER's properties.

Invariant 1. If two replicas commit commands with (id, c, D) and (id, c', D') respectively, where id is the command identifier, c is the command payload and D is the dependency set, then $c = c'$ and $D = D'$.

Proof. A correct replica marks a command as committed at lines 14, 23, and 40. We can assume that the two commits happened at different replicas. We have the following cases:

Case 1: Assume replicas r_0 and r_1 both commit at Line 14. A command c is assigned a unique identifier id by the TruDep of the initial coordinator and provided a certificate. This ensures that the coordinator cannot generate a certificate for another command c' with the same identifier id. Thus, $c = c'$. In addition, the trusted subsystem also provides a dependency set D. The identifier id is included in the `FastCommit` messages collected and thus the command for id will be c at correct replicas. In addition, correct replicas will generate a `FastCommit` message only if the id is valid, and only commit a command in fast path if they collect a fast quorum of messages with the same dependency D. Since the dependency sets in the `FastCommit` messages are produced by the TruDep along with a certificate, the dependency sets must be the same. Thus, we have that $D' = D$.

Case 2: Assume replicas r_0 and r_1 both commit at Line 23. This means that the command undergoes the fast commit phase and is unable to collect a quorum with same dependency sets, or did not receive the fast quorum messages in time. Case 1 above ensures that $c = c'$. The dependencies in each `SlowCommit` message will be the same since each replica participating in the slow quorum accumulated the dependencies from the fast quorum Q. Thus, the dependency set must be the same i.e. $D' = D$.

Case 3: Assume that one replica r_0 committed at line 14 and another replica r_1 committed at line 23. This is possible if r_0 was able to collect a fast quorum of `FastCommit` messages, but r_1 could not collect a fast quorum within a time period and instead takes the slow commit phase. The dependency sets are accumulated from a majority of `FastCommit` messages. Note that every replica uses the same fast quorum as mandated by the initial coordinator in its `Propose` message (Line 5). Moreover, after the timeout, the dependency set for the slow commit phase is aggregated from the subset of `FastCommit` messages. TruDep ensures that replicas send the same valid dependency set for a command to other replicas. Thus, given r_0 was able to collect a fast quorum of `FastCommit` messages with the same dependency sets, r_1's majority quorum of `FastCommit` messages should all have the same dependency set since the majority quorum is a subset of the fast quorum. Therefore, when r_1 collects majority `SlowQuorum` messages and commits the command, its dependency set will match with the dependency set at r_0. Hence, we have $D = D'$. □

Invariant 2. Assume two commands with (id, c, D) and (id', c', D') have been committed. If $id \neq id'$ and c and c' conflict, then either $id' \in D$ or $id \in D'$, or both.

Lemma 5.2. *Conflicting commands must be executed in the same order by the replicas.*

Proof. The proof is by contradiction. Assume that given two correct replicas r_i and r_j and two conflicting commands c and c' with identifiers id and id' respectively, r_i executes c before c' while r_j executes c' before c.

Case 1: Assume both c and c' are in the same execution set S at replica r_i. Since correct replicas agree on the dependencies of each command before committing, the dependency sets for a command at any two correct replicas must be the same. Thus, both c and c' must be in the same execution set at r_j, Since the commands within an execution set are ordered

deterministically using the command identifiers at each replica, c and c' must be executed is the same order by both the replicas r_i and r_j. This is a contradiction.

Case 2: Assume c and c' are in different execution sets S and S' at r_i. Similarly, c and c' must be in different sets at r_j. This can only happen if $id' \notin dep[id]$ at r_i and $id \notin dep[id']$ at r_j. However, this contradicts Invariant 2, which says that given two conflicting commands, one must be in the dependency set of the another. □

Lemma 5.3. *Given an execution order $c_1 \mapsto ... \mapsto c_n$ for $n \geq 2$, whenever r_i executes c_n, some replica has already executed c_1.*

Proof. By Induction on n. For $n = 2$, the definition of $\mapsto$ implies the required. For $n \neq 3$, assume $c_1 \mapsto ... \mapsto c_{n-1} \mapsto c_n$. When a replica r_i executes c_n, we must show that c_1 has already executed at some replica. There are two cases to consider: $c_{n-1} \rightsquigarrow c_n$ or $c_{n-1} \mapsto_j c_n$.

Case 1: If $c_{n-1} \rightsquigarrow c_n$, then c_{n-1} is executed at some replica r_j before c_n is submitted, and consequently executed, at r_i. By induction, it can be shown that c_1 executes before c_{n-1} is executed at r_i. Thus, c_1 executes before c_n.

Case 2: When $c_{n-1} \mapsto_j c_n$, it is possible that either $c_{n-1} \mapsto_i c_n$ or $c_n \mapsto_i c_{n-1}$ have taken place. Since the latter case contradicts Lemma 5.2, it should be that $c_{n-1} \mapsto_i c_n$ at replica r_i. By induction, it can be shown that c_1 is executed before c_{n-1} is executed at i. Hence, c_1 executes before c_n. □

Theorem 5.4. *The execution order $c_1 \mapsto ... \mapsto c_n$ is acyclic [59].*

Proof. By contradiction. Assume that there exists an execution order $c_1 \mapsto ... \mapsto c_n \mapsto c_1$ for $n \geq 2$. This means that c_1 is executed before and also after c_n. By Lemma 5.3, c_1 should have been executed at some process before c_n is executed at some replica. Thus, c_1 will not be part of the execution group by Line 43. This is a contradiction. □

5.4 Evaluation

In this section, we evaluate DESTER to show that it achieves the following goals: 1. near-optimal latency to geographically located clients; 2. throughput higher than or comparable to existing hybrid and BFT protocols, as it allows every replica to order and execute commands concurrently; and 3. availability during replica failures unlike primary-based protocols (e.g. Hybster) that loses availability temporarily when primary fails.

We compare DESTER to a representative subset of well-known BFT protocols focused on performance and scalability:

1. Hybster [32] is the state-of-the-art leader-based Hybrid protocol known for providing high-throughput and low-latency in cluster deployments.

2. ezBFT [23] is a recent leaderless BFT protocol tuned for providing low-latency in geographic deployments similar to Dester.

3. SBFT [64] is a leader-based BFT protocol that provides consistently high throughput at scale in geographic deployments by reducing the number of communication messages per commit using threshold signatures and a fast-commit mode.

4. Hotstuff [140] provides a single commit-mode, but pipelines communication steps to provide a scalable high-throughput protocol. Hotstuff switches the leader regularly to limit the influence of faulty replicas.

5. PBFT-MAC [41] is a single commit-mode leader-based BFT protocol that uses Message Authentication Codes instead of signatures. It can provide high throughput at smaller system sizes.

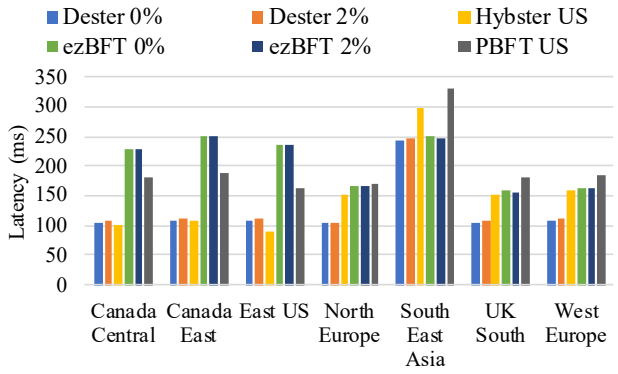
We implemented each of the protocols in Golang under a common framework. The TruDep, USIG and Hybster's TrInx trusted components were implemented in C using Intel SGX SDK and interfaced with the Go application using the cgo interface.

We implemented a key-value store to benchmark the protocols as it serves as a good abstraction for building higher level systems ranging from databases [3] to ledgers [64]. Among the protocols evaluated, Dester and ezBFT are affected by contention. Contention, in the context of a replicated key-value store, is defined as the percentage of requests that concurrently access the same key. Prior work [117] has shown that, in practice, contention is usually between 0% and 2%. Thus, a 2% contention means that roughly 2% of the requests issued by clients target the same key, and the remaining requests target clients' own (non-overlapping) set of keys. However, we also evaluate Dester at 100% contention for completeness.

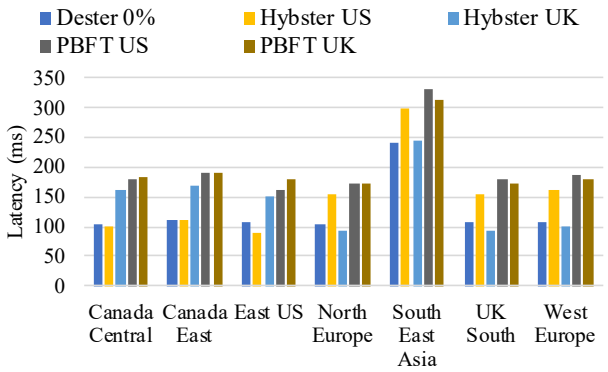
Deployment. We evaluated each of the protocol by deploying in 7 regions of the Microsoft Azure cloud platform. The regions were East US, Canada Central, Canada East, UK South, North Europe, West Europe and South East Asia. In each region, we used the SGX-enabled DC8v2 virtual machine to deploy our protocols.

5.4.1 Client-side Latency

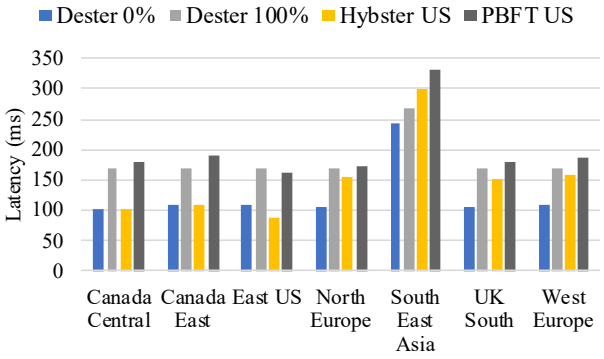
To understand Dester's effectiveness in achieving optimal latency at each geographical region, we devised an experiment to measure the average latency experienced by clients located at each region for each of the protocols. At each node, we also co-located 20 clients that sends requests to the replicas. For leader-based protocols (Hybster and PBFT), the leader was set to a particular region (as indicated in the figure); thus, clients in other regions send their requests to the leader. For Dester and ezBFT, the client sends its requests to the nearest replica, which is in the same region. The clients send requests in closed-loop, meaning that a client will wait for a reply to its previous request before sending another one.



(a) Latency under the base-case scenario for all protocols.



(b) Latency advantage of DESTER over primary-based protocols.



(c) DESTER at 100% contention vs. primary-based protocols.

Figure 5.3: Client-side latency at each of the seven regions.

Figure 5.3 shows the average latency (in milliseconds) observed by the clients located in their respective regions (shown on x-axis) for each of the protocols. For Dester, the latency was measured at different contention levels: 0%, 2%, and 100%; the suffix in the legend indicates the contention. We perform the analysis under three cases. Figure 5.3a shows the best case of all the protocols. The best-case latency for the Dester and ezBFT protocols is exhibited under low contention (0-2%). For the primary-based protocols, the best-case is exhibited when the primary is one of the closely located quorum of replicas. Dester provides latencies similar to Hybster for the replicas in the North American continent, when Hybster's primary is location. In other regions, Dester provides 30% lower latency than Hybster. This is despite Dester requiring a larger quorum with two additional replicas than Hybster's quorum. Furthermore, Dester outperforms both the BFT protocols in all the regions (except ezBFT in Asia) by providing up to 60% lower latency.

Figure 5.3b highlights Dester's ability to provide optimal latency at each region compared to primary-based protocols. For primary-based protocols, Hybster and PBFT, we measured latencies by placing the primary in two regions: East US and UK South. Notice that the primary's location influences the latencies at different regions. The UK primary is advantageous for non-US replicas, while the US primary is advantageous for US replicas. This is an important observation since protocols such as AWARE [34] provide a primary placement oracle atop of a BFT protocol to determine a location that can provide the best latencies to all the clients. However, as shown in the Figure, a leader placement oracle cannot provide optimal latencies to clients in all configurations. In contrast, Dester leaderless nature enables it to provide near-optimal latencies to its clients.

Figure 5.3c shows the effect of contention on Dester's performance. The best case for Dester is exhibited under low contention workloads. Under heavy contention (100%), the need for the slow phase communication steps for all commands increases the latencies by up to 1.5× from the low contention case. However, note that Dester performs at par with Hybster in non-US regions when the Hybster's primary is placed in US.

Note that SBFT and Hotstuff are both throughput-oriented protocols focused on scalability. These protocols incur higher latencies due to their inherent design. Thus, we only evaluate SBFT and Hotstuff in Section 5.4.3.

5.4.2 Server-side Throughput

During the latency experiment, we also measured the peak throughput of the protocols. This is useful in observing the maximum attainable throughput of the protocols when they provide their best possible latencies. The requests consist of 16-byte keys and 256-byte values.

Figure 5.4 shows the results. Due to the leaderless nature, both ezBFT and Dester outperform the primary-based protocols. However, Dester provides 1.25× better throughput than ezBFT. Compared to Hybster and PBFT, Dester performs 1.9× and 2.7× better,

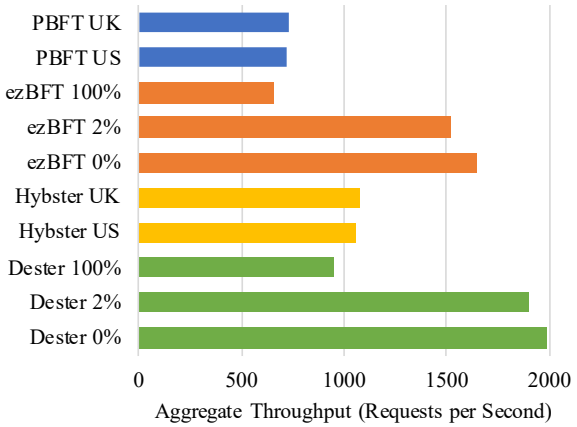


Figure 5.4: Throughput of DESTER and competitor BFT protocols.

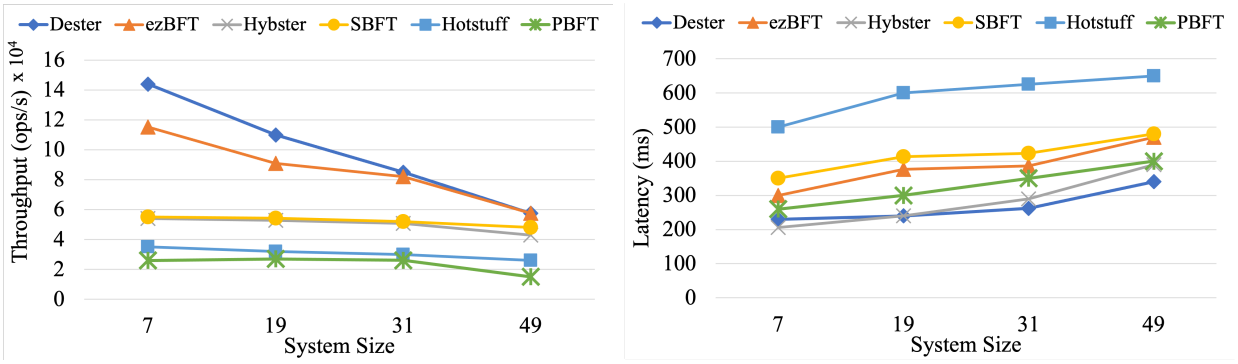
respectively. As contention increases, DESTER's performance falls by 50% as more slow paths are taken. However, DESTER at 100% contention performs at par with Hybster.

5.4.3 Scalability

In the scalability experiment, we measured the impact of increasing the system size on the performance of the protocols. For this experiment, we used the same set of seven regions as with other experiments, but varied the system size between 7 and 49 replicas. For this experiment, we enabled server-side batching of commands as is common in scalable deployments [71]. Replica(s) batch 100 commands before proposing them. Furthermore, we also compare with Hotstuff and SBFT as both are throughput-oriented scalable protocols. Moreover, since we only target DESTER for low contention scenarios, we set the contention level to 2% for this experiment. Figure 5.5 shows the results.

At 7 and 19 replicas, DESTER provides higher throughput than all other protocols. It performs $\approx 1.25\times$ better than EZBFT. However, as we increase the system size to 49 replicas, DESTER's performs only as good as EZBFT and the scalable protocol, SBFT. Due to the quadratic communication, DESTER's performance degrades as system size scales. EZBFT takes advantage of its linear communication and is able to scale better, but its large quorum size and additional slow path to handle the contention degrades its performance at larger system sizes.

On the other hand, the scalable protocols Hotstuff and SBFT maintain their performance at scale despite their low absolute throughput. Note that Hotstuff's rotating primary mechanism forces it to incur lower overall throughput even though the protocol is able to maintain the throughput at scale. While at 7 replicas, DESTER performs $\approx 2.5\times$ better than SBFT in terms of throughput, at 49 nodes, DESTER's performance is only at par with SBFT's



(a) Throughput.

(b) Latency.

Figure 5.5: Scalability of Protocols.

performance.

It should be noted that Dester provides low latency even at higher scales despite suffering a lower throughput. This is due to the ability of multiple replicas to process client commands concurrently.

5.4.4 Fault Tolerance

One of the crucial aspects of Dester is its ability to provide better availability guarantees than primary-based systems. Unavailability is inevitable when primary is faulty. To show this, we devised an experiment in which we terminate the leader primary in Hybster and a random replica in Dester, about 20 seconds through the experiment. The result is shown in Figure 5.6. For Dester, only clients from the crashed region timeout and reconnect to another region, while for Hybster, a leader change phase is started. Thus, there is only slight decrease in throughput for few seconds for Dester, but a total loss of throughput in Hybster during that time.

Furthermore, the performance of Dester when there are f faulty replicas can be inferred from the 100% contention case of Figure 5.4. Since Dester takes the slow path when replicas collect inconsistent dependencies and/or unable to collect a fast quorum (possible due to faulty replicas), the performance captured by the 100% contention case is representative of performance under f faults as well.

5.4.5 Summary of Results

Our experimental results reveal that Dester is highly effective in providing low latency for geographically located clients. Furthermore, the performance of the protocol is highly affected by contention. It provides better performance than existing protocols at low con-

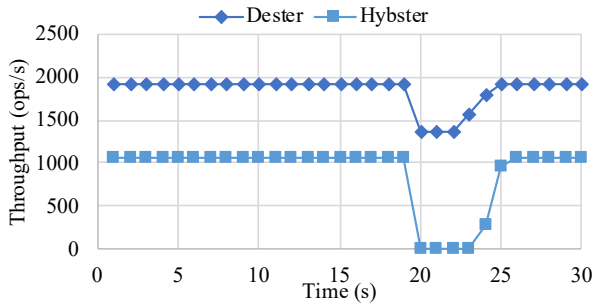


Figure 5.6: DESTER Fault Tolerance.

tention (< 2%). In addition, DESTER's performance degrades as system size increases. In summary, DESTER is better suited for smaller deployments of up to $N = 19$ replicas and for applications with a lower conflict rate between 0–2%.

5.5 Conclusions

CPU-based Trusted Execution Environments have made state machine replication solutions based on hybrid fault model more appealing than ever. However, existing hybrid protocols provide poor performance and availability guarantees in geographic deployments. DESTER is a leaderless hybrid fault-tolerant state machine replication protocol that provides near-optimal latencies for geographically located clients, better or comparable system throughput to existing solutions, and better availability guarantees when a single replica failure does not affect the operation of the system.

Chapter 6

Fault Transformations with Bumblebee

Byzantine Fault-Tolerant (BFT) State Machine Replication (SMR) solutions [8, 16, 35, 41, 45, 46, 53, 68, 83] can defend applications from arbitrary faults. These include crash (benign) faults, non-crash non-malicious faults, and malicious faults. However, for many replicated systems, such as those maintained by a single enterprise-class organization within a private network, malicious attacks, where the adversary can take control of faulty machines as well as the network in a coordinated way, are essentially non-existent [103]. This makes BFT protocols superfluous in such cases.

Prior work has proposed special hardening processes [48] and fault-recovery mechanisms [14] to detect or tolerate non-crash, non-malicious failures. However, this adds unnecessary complexity to layers outside the core consensus layer. Instead, BFT protocols that incorporate trusted components (refer to Chapter 2 for more information) can provide tolerance beyond crash faults without the additional replication overheads of traditional BFT protocols. For these protocols that follow the hybrid fault model, the number of replicas required for replication by trust-based Hybrid protocols matches the CFT requirement, i.e., $2f + 1$ replicas. For these protocols, a software-based subsystem can be safe-guarded using the trusted execution environments (TEE) in commodity hardware (e.g., Intel SGX [52]) providing the required level of trust.

Hybrid (BFT/TEE) protocols have been proposed either as a "ground-up" construction [32] or as a transformation of existing BFT protocols [135]. The former technique involves a complex design and verification process, while the latter technique requires that stronger protocols are first built for Byzantine settings and then relaxed for the Hybrid model. Hence, such approaches will likely hinder the large-scale adoption of Hybrid protocols.

On the other hand, significant research investment has been made for the CFT model and a rich body of CFT protocols exist (e.g., [12, 21, 89, 90, 95, 107, 117, 120, 122]). CFT protocols have been well understood (e.g., Raft [120]) as well as widely implemented (e.g., Paxos [95], Raft, Zab [79]). Due to the stronger failure model, CFT protocols are typically less complex to design and build than BFT protocols. This raises the question: is it possible to transform CFT protocols to tolerate arbitrary faults under partial synchrony using a trusted component, via a general methodology? If that can be done, then it would pave the way for a large number of hybrid protocols that tolerate arbitrary faults to be easily rolled out, leveraging the rich body of literature on CFT protocols. Many existing SMR systems in the datacenter (e.g., [29, 79]) already implement some CFT protocol, thus hybridization

is more likely, owing to the benefits of such a transformation.

To affirmatively answer the question, this chapter presents BUMBLEBEE, a three-step methodology to manually transform CFT protocols to Hybrid protocols. We identify the fundamental design differences between CFT and Hybrid protocols, and develop key insights that are necessary for a CFT-to-Hybrid transformation. We present a parameterized Generic Algorithm that allows instantiating consensus algorithms in both the CFT and hybrid fault models. From the Generic representation of a CFT protocol, we show that it is possible to produce a Hybrid protocol by changing the parameters and using a communication mechanism that uses trusted components. We manually apply BUMBLEBEE technique for transforming popular CFT protocols, Raft, Paxos, and M^2Paxos [122] into their hybrid counterparts.

Generic algorithm construction to construct protocols under different fault models has been investigated before [124]. Such construction techniques limited their scope to benign failures and/or Byzantine failures [115]. Our work focuses on generic construction of benign and hybrid protocols. To aid in the transformation, we also present communication primitives using trusted components to replace the benign communication primitives.

To understand BUMBLEBEE's overheads, we developed implementations. We realized the USIG subsystem using Intel SGX SDK, and implemented the hybrid versions of Raft, Paxos, and M^2Paxos, and evaluated them on SGX-capable virtual machines in the Azure cloud infrastructure [1]. We also modified Raft's (CFT) implementation in etcd [5], a widely used key-value store, and implemented our transformed Hybrid Raft protocol. Our experimental evaluations reveal that the overheads due to hybridization in the context of a system such as etcd is less than 30% in terms of system throughput.

The rest of the chapter is organized as follows. Section 6.1 presents the system model and the predicates and properties that underpin BUMBLEBEE. Section 6.2 describes the transformation methodology for transforming a benign algorithm to the hybrid model. Section 6.3 discusses application of the transformation methodology on some well-known benign protocols. Section 6.4 evaluates the transformed protocols and compares them to their benign counterparts and relevant BFT protocols. Section 6.5 concludes the chapter.

6.1 Preliminaries

6.1.1 Model

We assume the partial synchrony timing model [57] in which the system alternates between synchronous (good) and asynchronous (bad) periods. A process is considered honest if it correctly executes the algorithm, otherwise it is Byzantine. A Byzantine process is allowed to exhibit any arbitrary behavior. An honest process that crashes is considered faulty, otherwise

it is correct.

For Hybrid protocols, we assume the existence of a trusted subsystem within each process that can fail only by crashing. We assume the adversary controls all the system software including the operating system. Thus, the adversary can instantiate multiple enclave instances within the same node, offer stale version of sealed data to the enclave, and arbitrarily delay or withhold the messages send by the enclaves. However, we assume that the adversary cannot tamper with the enclave runtime memory or extract any part of the data within the enclave including secret keys. Furthermore, the enclave is capable of generating cryptographic operations that the adversary cannot break. We also assume that the adversary cannot compromise the SGX protections on participating nodes (e.g. via physical attacks).

The system consists of n processes. There can be at most b Byzantine processes, and at most f (honest) crash faulty processes. The set of all processes is denoted by Π, the set of honest processes is denoted by $\mathcal{H}$, and the set of correct processes is denoted by $\mathcal{C}$.

Round Model. Similar to prior works [114, 115, 124], we express our algorithm in terms of a round model. Each round r consists of a sending function S_p^r and a transition function T_p^r. A process p invokes the sending function S_p^r to compute and send messages to other processes. The transition function T_p^r is invoked at the end of the round when processes have finished receiving messages from other processes. The T_p^r takes as input the messages received and the current process state, and computes the new state of the process. Rounds are closed in that the messages sent in a round r are received in the same round r. However, this does not require lock-step execution by replicas.

Furthermore, a Byzantine process cannot send messages on behalf of a honest process. If an honest process receives v from p in round r, and p is honest, then v was sent by p in round r. s_p^r represents the state of process p in round r, $S_p^r(s_p^r)$ indicates the message sent by honest process, and $\vec{\mu}_p^r$ represents the vector of messages received in round r by process p. Each index q of the vector $\vec{\mu}_p^r$ represents the message from process q.

6.1.2 Communication Predicates

During bad periods, $\mathcal{P}_{int}$ is the best guarantee that can be provided [114]. $\mathcal{P}_{int}$ captures that, in any round r, if an honest process sends v, then every honest process receives v or nothing. It is expressed as follows (the $\perp$ indicates nothing).

$$\mathcal{P}_{int}(r) \equiv \forall p, q \in \mathcal{H} : (\vec{\mu}_p^r[q] = S_q^r(s_q^r)) \vee (\vec{\mu}_p^r[q] = \perp)$$

However, under the partial synchrony model, during sufficiently good periods, it is possible to ensure that a correct process receives every message sent by a correct process. This is captured by the following predicate:

$$\mathcal{P}_{good}(r) \equiv \forall p, q \in \mathcal{C} : (\vec{\mu}_p^r[q] = S_q^r(s_q^r))$$

Furthermore, it is possible to ensure another predicate $\mathcal{P}_{cons}$ that in addition to $\mathcal{P}_{good}$ ensures that each correct process receives the same set of messages.

$$\mathcal{P}_{cons}(r) \equiv \mathcal{P}_{good}(r) \wedge \forall p, q \in \mathcal{C} : (\vec{\mu}_p^r = \vec{\mu}_q^r)$$

Prior works [57, 114] have provided implementations of $\mathcal{P}_{good}$ and $\mathcal{P}_{cons}$ for benign and Byzantine fault models. In this chapter, we provide respective implementations under the hybrid fault model.

6.1.3 Consensus Problem

It has been shown that State Machine Replication protocols can be constructed from consensus protocols [127]. The consensus problem is formally specified by the following properties [115]:

- **Validity**: Given all processes are honest, an honest process only decides v if it is the initial value of some process.
- **Agreement**: No two honest processes decide differently.
- **Termination**: All correct processes eventually decide.
- **Unanimity**: If all honest processes start from the same initial value v, then an honest process can only decide v.

The unanimity property above is only suited for Byzantine processes. Thus, it is not present in the properties defined in Section 2.2.

6.2 Transformation Methodology

Our proposed technique, BUMBLEBEE, for transforming benign consensus protocols to tolerate Byzantine failures with the help of the trusted components involves the following steps:

1. **Lifting Abstraction:** The benign algorithm is represented in terms of the Generic Algorithm (Section 6.2.1).
2. **Replacing Parameters:** The benign model parameters of the Generic Algorithm are replaced with the hybrid model parameters (Section 6.2.3)
3. **Upgrading Communication:** The benign implementations of $\mathcal{P}_{good}$ and $\mathcal{P}_{cons}$ are replaced with the hybrid counterparts (Section 6.2.4)

6.2.1 Lifting Abstractions: Generic Algorithm

We describe a parameterized Generic Algorithm that allows the instantiation of consensus algorithms in the crash and hybrid fault models. The algorithm works in phases, where each phase ϕ is composed of three rounds: Selection round, Validation round, and Decision round. It uses two abstract functions *FLV* and *Validator*, whose instantiations decide whether the algorithm is crash fault-tolerant or hybrid fault-tolerant. The instantiations are provided in 6.2.3 The algorithm is presented in Algorithm 4.

State at process p

- $vote_p$ stores the value that will be considered for decision in the Decision Round. Initially, $vote_p$ is set containing $init_p$, the initial value of process p.
- ts_p stores the phase at which the vote in $vote_p$ was validated by process p in the Validation round. The initial value is zero.
- $history_p$ store the tuple of $vote_p$ and ts_p values. It tracks the values that were considered for decision at different phases during the execution of the algorithm.

Selection Round $(3\phi - 1)$

This round involves processes choosing a value that will be considered for decision in the Decision Round. To choose a value, each process sends its state $\langle vote_p, ts_p, history_p \rangle$ to all processes. At the end of the round, honest processes invoke the $FLV(\vec{\mu}_p^r)$ function to compute the chosen value. If the *FLV* function allows choosing any value (i.e., it returns ?), then honest processes pick a value from the received ones in $\vec{\mu}_p^r$ deterministically. If the chosen value $select_p$ is null, then the $vote_p$ and $history_p$ variables are not updated, since no values exist to choose from.

The *FLV* functions is used to find the locked value from the set of received messages. We use the notion of locked value defined in [124]. A value v is said to be locked in round r if there is an honest process that decided v in round $r' < r$ or all honest processes have the same initial value v. The Selection Round and the *FLV* function ensure that if some value is locked in some round r, then the process will set its $vote_p$ to that value v. The *FLV* function should satisfy the following properties [124]:

- FLV-validity: If all processes are honest and $FLV(\vec{\mu}_p^r)$ returns v such that $v \neq ?$ and $v \neq null$, then $\exists$ process q such that $v = \vec{\mu}_p^r[q].vote$.
- FLV-agreement: If value v is locked in round r , only v or *null* can be returned.
- FLV-liveness: If $\forall q \in \mathcal{C} : \vec{\mu}_p^r \neq \perp$, then *null* cannot be returned.

Algorithm 4: Generic Algorithm.

Initialization:
- $vote_p := init_p$
- $ts_p := 0$
- $history_p := \{(init_p, 0)\}$

Selection Round $r = 3\phi - 2$**:**
- S_p^r :
  - send $\langle vote_p, ts_p, history_p \rangle$ to all
- T_p^r :
  - $select_p \leftarrow \boxed{FLV(\vec{\mu}_p^r)}$
  - **if** $select_p = ?$ **then**
    - $select_p \leftarrow$ choose deterministically $v : \langle v, -, -, - \rangle \in \vec{\mu}_p^r$
  - **end**
  - **if** $select_p \neq null$ **then**
    - $vote_p \leftarrow select_p$
    - $history_p \leftarrow history_p \cup \{(vote_p, \phi)\}$
  - **end**

Validation Round $r = 3\phi - 1$**:**
- S_p^r :
  - **if** $p \in \boxed{Validators(p, \phi)}$ *and* $select_p \neq null$ **then**
    - send $\langle select_p \rangle$ to all
  - **end**
- T_p^r :
  - **if** $\exists v : |\{q \in Validators(p, \phi) : \vec{\mu}_p^r[q] = \langle v, - \rangle\}| > \frac{|Validators(p,\phi)|}{2}$ *or* $\langle v, \phi \rangle \in history_p \wedge q \in Validators(p, \phi) : \vec{\mu}_p^r[q] = v$ **then**
    - $vote_p \leftarrow v$
    - $ts_p \leftarrow \phi$
  - **end**

Decision Round $r = 3\phi$**:**
- S_p^r :
  - send $\langle vote_p, ts_p \rangle$ to all
- T_p^r :
  - **if** *received more than* $\frac{n}{2}$ *messages with* $\langle v, \phi \rangle$ **then**
    - DECIDE v
  - **end**

Validation Round

The validation round ensures that the chosen value is validated before the processes vote on the value during the phase. A voted value is validated if an honest process is sure that only at most one value will be voted for in a given phase by honest processes. For this purpose, the processes that belong to the set returned by the $Validator(p, \phi)$ function send their chosen value from the Selection Round $\langle select_p \rangle$ to all processes. At the end of the round, each process validates the value. A value v received from another process q is validated, if at least a majority of validators have chosen the same value v, or if in the previous Selection round, $\langle v, \phi \rangle$ was added to the $history_p$ set by process p.

The Validator function returns a set of processes that satisfy the following properties [124]:

- Validator-validity: $|Validator(p, \phi)| > 0$. That is, the validator function returns at least one process.
- Validator-agreement: $\forall p, q \in \mathcal{H}$ and $\forall \phi$, if $|Validator(p, \phi)| > 0$ and $|Validator(q, \phi)| > 0$, then $Validator(p, \phi) = Validator(q, \phi)$. That is, invoking the $Validator$ function at any two honest processes p and q during a given round ϕ returns the same set of processes.
- Validator-liveness: There exists a good phase ϕ_0 such that $\forall p \in \mathcal{C} : |Validator(p, \phi_0) \cap \mathcal{C}| > \frac{|Validator(p, \phi_0)| + b}{2}$.

Decision Round

The Decision Round is responsible for deciding a value v if and when certain conditions hold. Each process sends its vote and timestamp to other processes. If a process receives a majority of messages with the same voted value and timestamp $\langle v, \phi \rangle$, then v is decided.

6.2.2 Correctness of Generic Algorithm

This section presents a proof of correctness of the Generic Algorithm. Note that the proof is heavily inspired from [115].

Lemma 6.1. *If, in every phase ϕ of the Generic Algorithm, (i) Validator-validity holds (ii) Validator-agreement holds (iii) FLV-validity holds (iv) an honest process p sets $vote_p = v$ and $ts_p = \phi$, and (v) another honest process q sets $vote_q = v'$ and $ts_q = \phi$ (lines 24-25), then $v = v'$.*

Proof. For contradiction, assume that in some phase ϕ, two honest processes p and q have respectively $vote_p = v$ and $ts_p = \phi_0$, and $vote_q = v'$ and $ts_q = \phi_0$. This means that in round

$3\phi_0 - 1$, at least $x - b$ honest processes, where $x > \frac{|Validator(p,\phi_0)|}{2}$ send message $\langle v, i\rangle$ and at least $y - b$ honest processes send message $\langle v', -\rangle$, where $y > \frac{|Validator(q,\phi_0)|}{2}$. By Validator-agreement, we have that $Validator(p, \phi_0) = Validator(q, \phi_0)$. Therefore, $x - b + y - b > |Validator(p, \phi_0)|$. Since Validator-validity holds, at least one honest process h must have sent $\langle v, -\rangle$ to one process and $\langle v', -\rangle$ to another process. This contradicts the fact that h is an honest process.

Given that FLV-Validity holds, lines 14 and 15 should have been executed. This means that there is a $\langle vote_p, \phi\rangle \in history_p$. Since FLV-Validity holds, honest replicas will have the same $vote_p$ for a given ϕ in their $history_p$. Thus, if the $q \in Validators(p, \phi)$ is honest, it will send to all $select_p = vote_p$, that exists in the $history_p$ of honest replicas. Thus, lines 24-25 is executed, and $vote_q$ is set to v and ts_q is set to ϕ, and not any other value. □

Theorem 6.2. *If (i) function $FLV(\vec{\mu}_p^r)$ satisfies FLV-validity and FLV-agreement, (ii) function $Validator(p, \phi)$ satisfies Validator-validity and Validator-agreement, then Algorithm 4 ensures weak validity, strong unanimity, and agreement.*

Termination holds if (iii) function $FLV(\vec{\mu}_p^r)$ satisfies FLV-liveness, and (iv) there is a good phase ϕ_0 in which Validator-liveness holds.

Proof. (a) Agreement: We prove by contradiction. Assume that process p decides v in round $r = 3\phi$, and process p' decides $v' \neq v$ in round $r' = 3\phi'$. At least $\frac{n}{2}$ processes, which must include at least one honest process, should have sent $\langle v, \phi\rangle$ in round $r = 3\phi$. Similarly, another set of $\frac{n}{2}$ processes containing one honest process should have sent $\langle v', \phi'\rangle$ in round $r' = 3\phi'$ – (**). There are two cases to consider: $\phi = \phi'$ and $\phi > \phi'$.

- Consider $\phi = \phi'$: From the statement marked (**) above, an honest process has sent $\langle v, \phi\rangle$ and another honest process has sent $\langle v', \phi'\rangle$. There is an honest process h that sets $vote_h = v$ and $ts_h = \phi$, and another honest process h' that sets $vote_{h'} = v'$ and $ts_{h'} = \phi$ at lines 21-22 of round $r = 3\phi - 1$. This contradicts with Lemma 6.1 and the Validator-agreement property.
- Consider $\phi > \phi'$: Let ϕ' be the smallest phase that is larger than ϕ. Assume that a honest process decides $v' \neq v$ in ϕ'. v is locked in all phases $> \phi$ by definition of a locked value. There is an honest process h that sets $vote_{h'} = v'$ and $ts'_h = \phi$) at lines 21-22 of round $r' = 3\phi' - 1$. By statement (**) above, $\frac{N}{2} - b$ honest processes sent $\langle v', \phi\rangle$. By Validator-validity, Validator-agreement, and FLV-validity, there is an honest process h' that sent $\langle v', -\rangle$ at line 18. Therefore, the function $FLV(\vec{\mu}_p^r)$ returns v' or ? at line 9 on process h'. This contradicts the FLV-agreement property and the fact that v is locked.

(b) Weak Validity: The FLV-validity property ensures that given all processes are honest and the FLV function only returns some v such that $v \neq ?$ or $v \neq null$, then some process q must vote for that value v. Assuming that all processes are honest, together with the FLV-validity property, the Weak validity property holds.

(c) Strong Unanimity: Since FLV-validity and FLV-agreement properties hold, only a valid value v is returned at all honest processes. Consequently, since Validator-validity and Validator-agreement holds, all honest process will receive the same v from the validators. Thus, honest processes will decide the same value. Since all processes are honest by assumption, Strong Unanimity holds.

(d) Termination: Assume a good phase ϕ_0 and that Validator-liveness holds. Due to the $\mathcal{P}_{cons}(3\phi-2)$ predicate in defined Section 6.2.4, all correct processes should receive identical message sets in round $3\phi-2$ since the assumption is that all the processes are honest. Therefore, all honest processes must select the same value. The value selected by honest process cannot be *null* by the FLV-liveness property and the predicate $\mathcal{P}_{cons}(3\phi-2)$. Thus, all correct processes have the same value v for $select_p$, and $vote_p$ when round $r = 3\phi_0 - 2$ ends.

Since Validator-liveness and Validator-agreement hold, all correct processes p consider the same set $Validator(p, \phi_0)$ at line 20. Because Validator-liveness holds, the set returned by the $Validator(p, \phi_0)$ must contain more than $\frac{|Validator(p,\phi_0)|}{2}$ correct processes. Consequently, since $\mathcal{P}_{good}(3\phi-1)$ holds, all correct processes set $vote_p = v$ and $ts_p = \phi_0$ in lines 24-25. Since $\mathcal{P}_{good}(3\phi-1)$ holds in round $r = 3\phi_0$, all correct processes receive at least $\frac{n}{2}$ messages $\langle v, \phi_0 \rangle$. Thus, they decide v.

□

6.2.3 Instantiations of Parameters

Validators(p, ϕ)

To tolerate Byzantine faults, a validator function that returns all the processes Π is enough to satisfy the validity, agreement, and liveness properties [124]. However, since we use a trusted component that ensures that equivocating messages are not sent, it is enough for the validator function to return a single process. For simplicity, this can be round-robin primary selector used in numerous BFT protocols.

To tolerate benign faults, it is enough for the validator function to return a single process, since all the processes are honest and will not deviate from the algorithm. However, the *Validators* function must pick different processes according to some heuristic to avoid crashed processes and ensure liveness. For this purpose, we use a Leader-election mechanism [95]. In each phase, processes vote for a validator q to be the leader in that phase, and if a validator q receives a majority of votes, then the return value of the *Validators* function is q.

$FLV(\vec{\mu}_p^r)$

Since the FLV function finds the locked value, it is important that the function depends on majority quorums used in the decision phase. The FLV function for the benign and hybrid models are in Algorithms 5 and 6, respectively. Note that the FLV function for the benign model is adopted from [115]. The FLV function for the hybrid model is a modified version of the Byzantine FLV function in [115].

Algorithm 5: Benign Fault FLV.

$possibleVotes_p \leftarrow \{(vote, ts, -) \in \vec{\mu}_p^r\} : |\{(vote', ts', -) \in \vec{\mu}_p^r : vote = vote' \vee ts > ts'\}| > \frac{n}{2}$
if $|possibleVotes| = 1$ **then**
  return $v : (v, -, -) \in possibleVotes$
else if $|\vec{\mu}_p^r| > \frac{n}{2}$ **then**
  return ?
else
  return $\perp$
end

Algorithm 6: Hybrid Fault FLV.

$possibleVotes_p \leftarrow \{(vote, ts, -) \in \vec{\mu}_p^r\} : |\{(vote', ts', -) \in \vec{\mu}_p^r : vote = vote' \vee ts > ts'\}| > b$
$correctVotes_p \leftarrow \{v : (v, ts, -) \in possibleVotes_p \wedge |\{(vote', ts', history') \in \vec{\mu}_p^r : (v, ts) \in history'\}| > b$
if $|correctVotes| = 1$ **then**
  return $v : (v, -, -) \in correctVotes$
else if $|correctVotes_p| > 1 \vee |\{(vote, ts, -) \in \vec{\mu}_p^r : ts = 0\}| > b$ **then**
  return ?
else
  return *null*
end

6.2.4 $\mathcal{P}_{good}$ and $\mathcal{P}_{cons}$ Implementations

To tolerate Byzantine failures with only $2b+1$ processes, we depend on the trusted component in each process to ensure correct operation of the protocols. Our trusted component of choice is Unique Sequential Identifier Generator (USIG) [135]. The USIG component exists at every process and provides the following properties: 1. two different messages are never assigned

the same identifier; 2. an identifier generated is always larger than the previous one; 3. the generated identifier is one more than the last generated identifier.

The USIG provides two interfaces: `CreateUI` and `VerifyUI` for creating and verifying an identifier *UI* for a particular message m. Each *UI* is a tuple of the identifier itself and a Message Authentication Code (MAC) produced by the trusted component over the message and the identifier. By assigning a unique, incrementing identifier to each message using the USIG, honest processes can detect whenever a Byzantine processes tries to send equivocating messages to different processes, because a pair of equivocating messages will have different *UI* identifiers.

In Algorithm 7, we present the implementation of $\mathcal{P}_{good}$ under the hybrid fault model. To prove the validity of messages, every message sent to other processes is attested with the unique identifier from the *USIG*. When a message is received by a process, it verifies the *UI* identifier and ensures that equivocating messages are not received. For simplicity, we assume that the identifier generated by the *USIG* correspond to the round numbers r. This way, if a Byzantine process sent two messages in the same round, it will have a higher identifier than the current round number. This can be easily realized in practical implementations [32].

Algorithm 7: $\mathcal{P}_{good}$ implementation using USIG.

Round r**:**
- S^r :
  - $UI_p := \langle ctr, \sigma \rangle := \texttt{CreateUI}(m)$
  - send $\langle UI_p, m \rangle$ to all
- T_p^r :
  - **if** $\texttt{VerifyUI}(m, UI_q) : \langle UI_q, m \rangle \in \vec{\mu}_p^r[q] \wedge \langle ctr, \sigma \rangle = UI_q : ctr = r$ **then**
    - $\vec{M}_p[q] \leftarrow m$
  - **end**

Under synchrony, $\mathcal{P}_{good}$ will hold, and the messages sent by processes in round r will be delivered in the same round. For the rounds $3\phi - 1$ and 3ϕ of the Generic algorithm, it is enough to ensure this guarantee, as the validation and decision rounds only vote and decide on the value selected in the previous round. The selection round, however, requires stronger guarantees for message delivery, because the correctness of the *FLV* function relies on the fact that every process receive the same set of messages in that round. To ensure this, we rely on the implementation of the $\mathcal{P}_{cons}$ predicate provided in Algorithm 8.

We achieve $\mathcal{P}_{cons}$ by performing a 2-round translation of predicate $\mathcal{P}_{good}$ in Algorithm 8. To achieve the translation, we convert a round r of an algorithm into two micro-rounds $\langle r, 1 \rangle$ and $\langle r, 2 \rangle$ while satisfying the following [115]: (i) $\mathcal{P}_{good}$ holds for each round $\langle r, i \rangle$; (ii) the message m_p sent by process p in round $\langle r, 1 \rangle$ is the message sent by p in round r; (iii) the messages received by p in round r is computed at the end of round $\langle r, 2 \rangle$; and (iv) $\mathcal{P}_{cons}$ holds for round r.

The translation works as follows. In the first round $\langle r, 1\rangle$, processes send the message m to all processes. Similar to Algorithm 7, the messages are attested using the *USIG* identifer *UI*. $coord(r)$ returns the coordinator of round r. For simplicity, this can be set to the process returned by the *Validator* function.

Algorithm 8: $\mathcal{P}_{cons}$ implementation using USIG.

Initialization:
  $\forall q \in \Pi : received_p[q] = \perp$
Round $\rho = \langle r, 1\rangle$**:**
  S_p^ρ :
    $UI_p := \langle ctr, \sigma\rangle := \texttt{CreateUI}(m)$
    send $\langle UI_p, m\rangle$ to all
  T_p^ρ :
    forall $q \in \Pi$ **do**
      if $\texttt{VerifyUI}(m, UI_q) : \langle UI_q, m\rangle \in \vec{\mu}_p^r[q] \wedge \langle ctr, \sigma\rangle = UI_q : ctr = r$ **then**
        $received_p[q] \leftarrow m$
      end
    end
Round $\rho = \langle r, 2\rangle$**:**
  S_p^ρ :
    if $p = coord(r)$ **then**
      $UI_p := \langle ctr, \sigma\rangle := \texttt{CreateUI}(received_p)$
      send $\langle UI_p, received_p\rangle$ to all
    end
  T_p^ρ :
    if $\texttt{VerifyUI}(m, UI_q) : \langle UI_q, m\rangle \in \vec{\mu}_p^r[q] \wedge \langle ctr, \sigma\rangle = UI_q : ctr = r$ **then**
      $\vec{M}_p[q] \leftarrow m$
    end

6.2.5 Transformation Steps

Given the aforementioned mechanisms, transforming a benign consensus protocols to tolerate Byzantine failures with the help of the trusted components involves the following steps:

1. **Lifting Abstraction:** The benign algorithm should be represented in terms of the Generic Algorithm. In the next section, we provide representations of some of the CFT algorithms in literature. We show that the generic representations are straightforward for most leader-based Paxos-style protocols.

2. **Replacing Parameters:** This step involves substituting the benign *FLV* and *Validator* functions with the hybrid counterparts. Specifically, the leader-election *Validator* function is replaced with the rotating coordinator function. The benign *FLV* function is replaced by the Hybrid variant in Algorithm 6.

3. **Upgrading Communication:** All the communication between processes are replaced by the respective hybrid tolerant variants of $\mathcal{P}_{good}$ and $\mathcal{P}_{bad}$ from Algorithms 7 and 8.

6.3 Examples

In this section, we apply the transformation technique to some popular benign fault-tolerant protocols, namely Paxos [95], Raft [120], M^2Paxos [122]. We note that our technique is general and can be applied to any benign consensus protocol that can be represented in terms of the Generic Algorithm presented in Section 6.2.1. Numerous Paxos variants exist that provide different performance characteristics. The methodology we describe here will largely apply to most Paxos protocols that employ a leader election mechanism. Transforming all the known Paxos variants is beyond the scope of this paper.

6.3.1 Paxos Transformation

In this section, we show how Paxos, a benign consensus protocol, can be represented in terms of the Generic Algorithm, and apply the required transformations to produce the Byzantine fault-tolerant version of the protocol using the trusted component, we call *Hybrid Paxos*.

The Prepare phase 1b of Paxos corresponds to the Selection round $(3\phi - 2)$ of the Generic algorithm. The Accept phase 2a corresponds to the Validation round $(3\phi - 1)$, while the phase 2b corresponds to the Decision round (3ϕ). The *Validator* function is the Leader-election mechanism that encapsulates both 1a and 1b of the Prepare phase. However, as in Paxos, the *Validator* function can be simplified by having the leader send a 1a message with intent to be the leader and waiting to receive enough leader votes along with the values during the selection round.

From the Generic representation of Paxos, the transformations described in Section 6.2.5 is applied to obtain the representation of Hybrid Paxos. This algorithm bears similarities to the MinBFT [135] protocol. The View Change protocol of the MinBFT protocol is similar to the Selection round with the $\mathcal{P}_{cons}$ predicate translation. The Validation round correspond to the Prepare step, while the Decision round corresponds to the Commit step.

Note that BFT protocols only invoke the View Change protocol only when the coordinator must be replaced. In such cases, the Selection phase can be suppressed. The same holds for Paxos as well.

6.3.2 Raft Transformation

In this section, we show how the Raft benign consensus protocol can be represented in terms of the Generic Algorithm, and apply the required transformations to produce the Byzantine fault-tolerant version of the protocol, we call Hybrid Raft.

The Raft protocols has two major modes of operation: Log Replication and Leader Election. The Log Replication component corresponds to the rounds $3\phi - 1$ and 3ϕ of the Generic Algorithm. The leader sends, in Raft's terms `AppendEntries` to all processes in round $3\phi - 1$. The processes exchange their votes by sending `AppendEntriesResponse` to all processes. The only difference between the description of the Raft in [120] and ours is that the `AppendEntriesResponse` messages are sent only to the leader, which in turn piggybacks it in the next `AppendEntries` message.

The Leader Election mode is responsible electing a new leader while ensuring that there are no discrepancies between the logs at different processes. Raft's approaches the problem by making sure that a process becomes a leader only if its logs are up to date. This is different from the description of the Generic Algorithm for Paxos-like approaches that allow any process to be elected leader and ensuring that the latest logs are sent to the leader. Regardless, by electing a up-to-date leader and following the Selection round of the Generic algorithm, the resulting protocol will still achieve the same result as Raft, albeit with additional messages [77]. The Leader Election is provided by the *Validators* function.

From the Generic representation of benign Raft, the transformations described in Section 6.2.5 is applied to obtain the representation of Hybrid Raft.

6.3.3 M^2Paxos Transformation

We now present Hybrid M^2Paxos, a hybrid fault-tolerant protocol obtained by transforming M^2Paxos [122] using the Generic Algorithm. M^2Paxos is a multi-leader consensus protocol that uses an object-specific ownership mechanism to provide low latency and high throughput in wide area network (WAN) deployments. The core novelties of M^2Paxos are (i) an object-specific ownership mechanism wherein a replica can propose an order for commands if it owns the accessed objects; and (ii) fast adaptation to changing access patterns, particularly common in geographic deployments, by stealing object ownership from other replicas.

Since M^2Paxos uses an instantiation of the Paxos protocol [95] per object, it is enough to replace each instantiation of Paxos with the Hybrid Paxos protocol. However, the resulting protocol under our technique provides different characteristics than the M^2Paxos protocol aside from the fault-tolerant guarantees. The Byzantine tolerant nature of the resulting protocol prevents stealing the ownership of objects. It cannot be deduced properly whether a request to steal the ownership is from an honest or a malicious process. Moreover, due to the use of the rotating coordinator selection for the *Validation* function, it is not possible

to pick a random coordinator during a view change. This is a limitation of the generic transformation methodology, however, manual improvements can be made to the resulting algorithm to achieve the intended characteristics.

6.4 Evaluation

Our motivation behind Bumblebee is to show that existing systems that employ CFT protocols can benefit from a transformation methodology to aid in the transition to the hybrid fault model without the additional design and engineering overhead of a "ground-up" protocol. As such, the evaluation aims to answer the following questions:

1. What is the additional overhead induced by using the trusted component USIG?
2. What is the performance overhead as a result of the transformation?
3. Do the transformed Hybrid protocols scale as well as their original CFT versions?
4. How does the transformation affect the performance of a real-world system?

We implemented the proposed hybrid solutions, Hybrid Paxos, Hybrid Raft, and Hybrid M^2Paxos (and also MinBFT) in Go. The trusted subsystem USIG was implemented in C using Intel SGX SDK and was interfaced with the Go application using `cgo` [65]. All protocols, except Hybrid Raft, was implemented in a common framework. We implemented Hybrid Raft by applying the transformations to etcd's Raft implementation [5].

We deployed all the systems on a cluster of three virtual machines in the Azure infrastructure [1]. Each VM belong to the DCsv2-series and has a 3.7GHz Intel Xeon E-2176G Processor with 8GB RAM. The OS used was Ubuntu 16.04 with Intel SGX SDK v2.8.1 and Go v1.14 installed.

6.4.1 USIG

Hybrid protocols use of trusted components to generate cryptographic certificates, inducing performance bottlenecks. Therefore, we first evaluate the performance of the USIG subsystem in order to understand the overhead of SGX calls and cgo interfaces. We measured the throughput of creating the *UI* certificates for different message sizes under three different setups: (i) calling the SGX function from a C application, (ii) calling the SGX function from a Go application using the cgo interface, and (iii) generating counter certificates using untrusted Go code.

Figure 6.1 shows the results. The x-axis shows the message sizes and the y-axis shows the throughput in terms of megabytes of messages certified per second. We can observe that,

for smaller messages (under 8KB), creating the USIG certificates are more expensive than untrusted certification. In some cases, the difference is as much as 2.5x (for 1KB). However, the difference zeros out for 16KB messages and beyond. Also, note that the cgo interface adds negligible overheads.

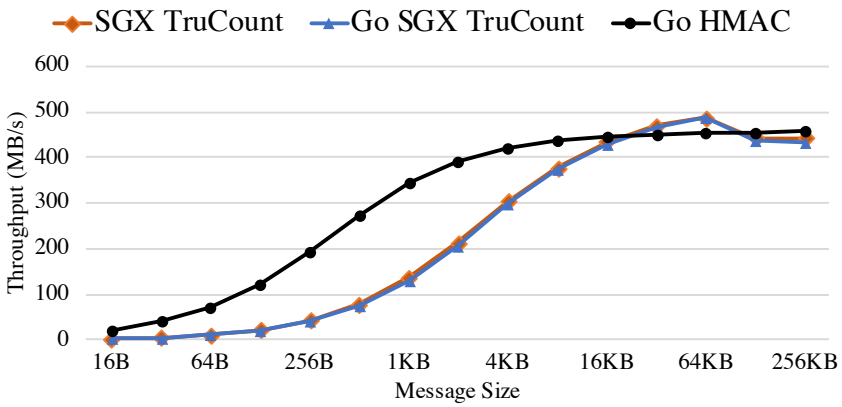


Figure 6.1: Throughput of SGX certifier vs Go-based certifier.

6.4.2 Performance of Hybrids

We compare the performance of Hybrid protocols to their CFT counterparts to understand the performance penalties of migrating to a different fault model. We also compare the protocols with MinBFT [135], a "ground-up" transformed version of the PBFT protocol. For this experiment, we varied the number of clients between 1 and 250, and injected requests into the system. We note that Hybster and Hybrid Paxos perform similar, as they are equivalent protocols. On the other hand, Hybrid M^2Paxos has the advantage of using per-object Hybrid Paxos. Thus, commands on different objects can now be executed in parallel, thereby improving the overall throughput. In our experiment, the ownership acquisition was turned off, so Hybrid M^2Paxos replicas forward commands to the appropriate leader.

Figure 6.2 shows the throughput versus latency of the original and the transformed variants of the CFT protocols. It can be observed that the hybrid variants incur a latency penalty in the range of 30% (for Hybrid M^2Paxos) to 45% (for Hybrid Paxos). This is due to the increase in the number of messages exchanged as well as the usage of USIG to create attestations. Moreover, Raft and its hybrid variant has the best performance among single-leader protocols despite similarity. We attribute this to etcd's original high-performance implementation, which Hybrid Raft also exploits resulting in a performance penalty of only about 30%. Specifically, unlike Paxos that replicates a request to a quorum of nodes, Raft replicates requests on a per node basis and waits until a quorum has acknowledged. This enables replicas to be independently replicated and thus improve overall performance. Fi-

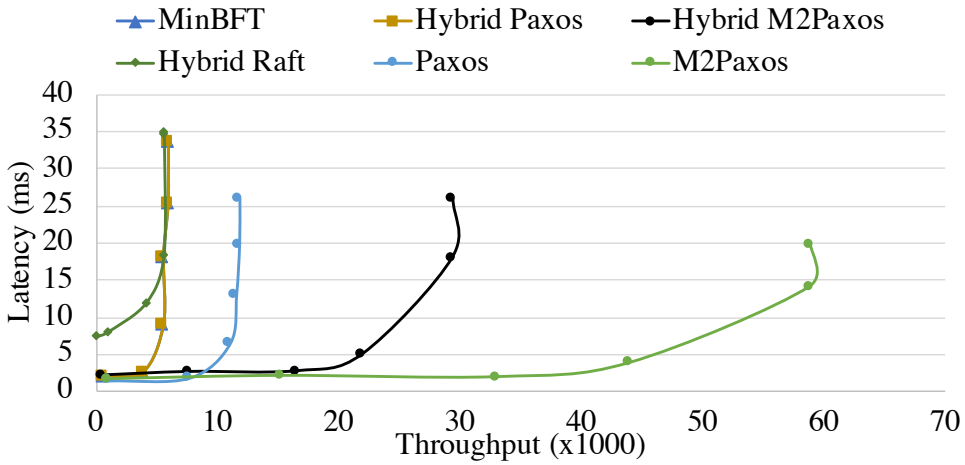


Figure 6.2: Throughput vs latency of Hybrid protocols.

nally, we believe such penalties are acceptable given the security and integrity benefits of the transformed protocols.

6.4.3 Scalability of Hybrids

In our next experiment, we analyze the scalability of the Hybrid protocols and compare with the CFT protocols. We repeated the throughput versus latency experiment (in Section 6.4.2) for 3, 5, 7, and 9-node setups.

Figure 6.3 compares the scalability of Hybrid Raft and original Raft as the number of replicas increase. It can be observed that Hybrid Raft's performance penalty is about 30% in the 3-node setup and grows to about 50% in the 9-node setup. This can be attributed to the exponential increase (n^2) in the number of messages exchanged as well as the overhead of interfacing with the SGX enclave and creating attestations. However, in practice, this overhead can be minimized by batching protocol messages sent over the network as well as interfacing with the SGX for a batch of messages (due to low SGX overheads for large messages as shown in Figure 6.1).

6.4.4 Fault Tolerance of Hybrids

Our hybrid protocols behave similar to their CFT counterparts under normal conditions (non-faulty nodes and network). As soon as a non-leader is faulty, the throughput decreases by about 5–6% in a 3-node setup and by about one percent in a 7-node setup. In case of 3 nodes, the loss of a node increases the load on the remaining two nodes since they must bear all the responsibilities including being part of the quorum and responding to the clients.

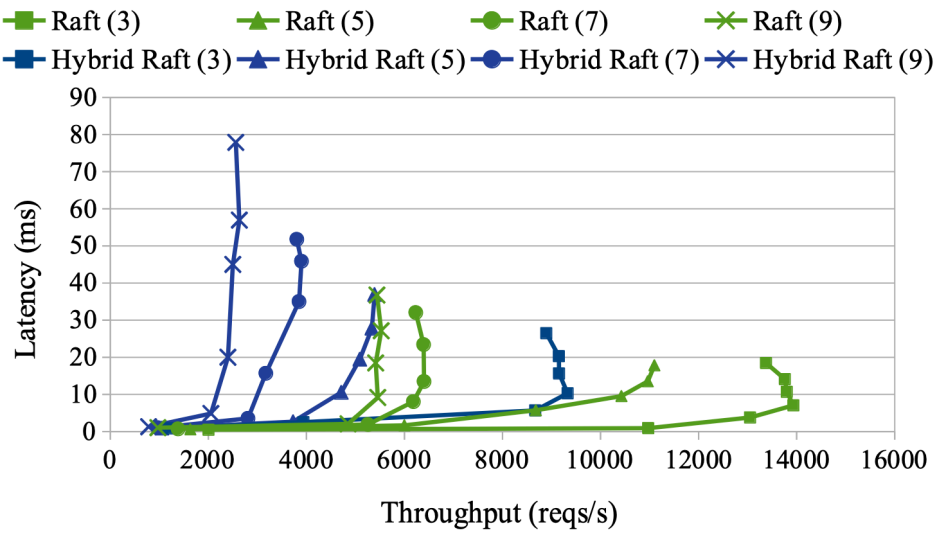


Figure 6.3: Throughput vs latency of Hybrid Raft vs Original Raft as number of nodes is varied. In the legend, the number inside the parenthesis indicate the number of nodes in that experiment.

However, at 7 nodes, the loss of a node still leaves three additional nodes that can share the workload.

6.4.5 etcd

etcd [5] is a distributed key-value store that is widely used in numerous production systems [6]. It uses the Raft consensus protocol to manage its replicated log. We transformed etcd's Raft implementation and benchmarked it with etcd's benchmarking tool. We modified the tool to collect $f + 1$ replies for each request and added the capability for all replicas to reply to the client. We compared our transformed etcd against the original version.

For this experiment, we deployed three replicas of etcd, and instantiated 500 clients to send requests to the replicas. We varied the payload size and measured the throughput by sending 100K requests. Figure 6.4 shows the results. It can be observed that the maximum performance penalty of the hybrid approach is between 20–30%. etcd performs several optimizations including batching of entries, which creates large enough message sizes that nullify the impact of SGX calls (recall Section 6.4.1). This shows that the proposed BUMBLEBEE approach is viable in today's systems, causing minimal performance degradation, while tolerating non-crash faults including hardware and software failures.

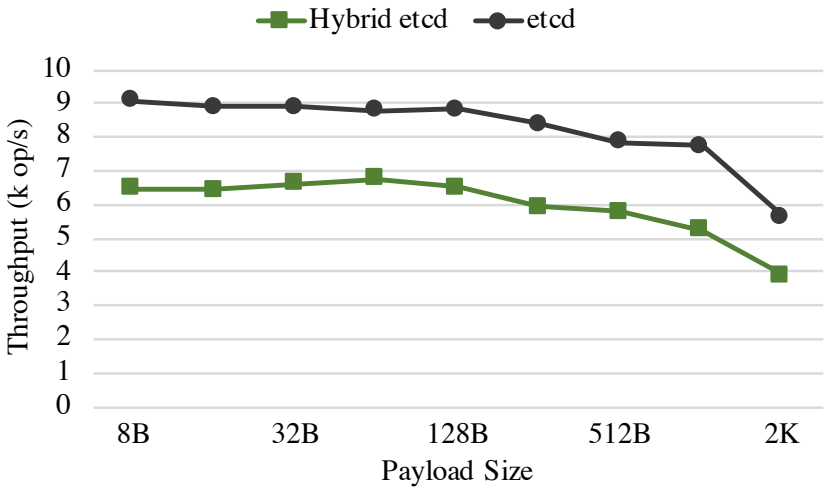


Figure 6.4: Original etcd vs Hybrid Raft etcd.

6.5 Conclusions and Limitations

This chapter presents a general methodology to transform existing crash fault-tolerant protocols into those that tolerate arbitrary faults. We achieve this using a trusted subsystem that is available in today's commodity processors. We showed that CFT protocols such as Raft and Paxos, which are widely in production use today, can be easily transformed into their hybrid variants, significantly improving their fault tolerance capabilities with limited performance overheads.

Our methodology currently focuses only on single leader-based CFT protocols, except M^2Paxos (which resembles single leader protocols to a large extent). Multi-leader protocols [21, 117] in the CFT model are inspired from Paxos, but our current methodology do not support transforming such protocols. As shown in Chapters 4 and 5, leaderless protocols use special mechanisms such as dependency graphs, fast and slow paths, that are not captured in our current Generic algorithm. Their hybridization is an interesting direction.

Chapter 7

Partial Decentralization with DQBFT

7.1 Introduction

In Chapter 4 and Chapter 5, we empirically showed that BFT protocols that adopt the primary-backup approach [41, 53, 64, 83] have significant performance bottlenecks. We further showed that leaderless protocols, ezBFT and Dester, exhibit better performance by addressing the four bottlenecks of primary-backup approach: a) load imbalance among primary and backup nodes, b) under utilization of resources at backup nodes, c) high WAN latencies for remote clients to send requests to the primary than clients that are local to the primary, and d) poor tolerance to primary failures [70]. By exploiting the commutativity property of client commands and the use of dependency tracking to order conflicting commands, leaderless protocols require additional coordination while processing concurrent conflicting commands. Thus, their performance degrades as workload contention increases. Consequently, these protocols do not scale their performance well to larger systems.

Other BFT solutions to overcome the primary-backup approach have drawbacks. Specifically, the rotating primary approach [133, 134, 140] does not take into account many aspects of modern geographically distributed systems including variations in node hardware, network bandwidth, and available resources. In such settings, a slow node can quickly degrade the overall performance.

7.1.1 Towards Partial Decentralization

To address these aspects, this chapter presents DQBFT[1], a paradigm for designing highly scalable consensus protocols by partially decentralizing the core consensus process into two distinct and concurrent steps that may be handled at potentially different replicas. Rather than adopting a completely decentralized approach where individual replicas replicate commands and also coordinate to find a *total order*, DQBFT divides the task of consensus protocols into two: 1) durable replication of client commands without a global order at correct replicas, and 2) ordering of the commands to guarantee a *total order*. Durable replication is carried out by each individual replicas for the commands it receives from its clients, while ordering is performed by a dedicated sequencer. Ordering involves assigning a

[1]Destiny stands for Divide and conQuer BFT

global order to a replica that has proposed a command. Thus, unlike the rotating primary technique [133, 140], this approach can better accommodate variations in node hardware, network bandwidth, and available resources.

Our approach is unique in that it allows for concurrent progression of the two stages. DQBFT uses separate instances of consensus protocols at individual replicas to carry out replication, while another consensus protocol is responsible for assigning the global order to individual replicas. This simultaneous replication and ordering allows DQBFT to significantly improve the performance of compared to past approaches [17].

7.2 Protocol Design and Description

We propose DQBFT, a paradigm for building BFT protocols by decentralizing the responsibility of the primary to tackle the performance and scalability challenges caused by load imbalance and resource under-utilization in existing primary-backup BFT protocols. Our motivation is based on the observation that any consensus protocol must perform two important actions to commit a command: request dissemination and ordering. Request Dissemination is a decentralized operation and does not require replicas to coordinate, but only acknowledge receipt. In contrast, ordering requires replicas to coordinate to ensure that everyone has a single view of the sequence of operations. To simplify this process, a replica can be elected to propose ordering for the commands.

In DQBFT, clients can send commands to any replica. In DQBFT, any replica can individually disseminate and order client commands using an instance of a consensus protocol. While this ensures that the commands are replicated to correct replicas, the order produced itself is local to the replica and not global. This constitutes a partial order. The *primary* replica produces a global order among the partial orders produced by the individual replicas.

Such a *semi-decentralized* approach has the following benefits. First, the decentralized process of dissemination distributes load evenly across replicas and enables clients to connect to a nearest replica in geo-distributed deployments. Second, for ordering, the global view of all commands enables the sequencer to order commands optimally, i.e., each newly proposed command can be dynamically assigned to the first unused global sequence number (unlike EBAWA [134], Spinning [133], and Hotstuff [140] which could slow-down due to slow nodes). Moreover, there is no need for additional coordination in case of conflicts (unlike ezBFT [23], Aliph [68]).

7.2.1 Design

At a high level, the DQBFT paradigm is composed of two sub-protocols: the dissemination protocol and the ordering protocol. The dissemination protocol employs multiple instances

of consensus, called D-instances, to enable every replica to disseminate and partially order its client commands. Meanwhile, the ordering protocol uses a single instance of consensus, called the O-instance, that agrees on the global order among the partial orders produced by the D-instances. There are as many D-instances as there are replicas and every replica is the coordinator of at least one D-instance. [2] A replica proposes commands in a series of sequence numbers belonging to its own D-instance to produce its partial order. The primary proposes D-instance sequence numbers in O-instance's sequence number space to effectively produce a global order using the replica-specific partial orders.

To tolerate Byzantine faults, both the dissemination and ordering should be based on a BFT consensus protocol. Primary-based protocols that satisfy the following properties [135] can be used for instantiating protocols under the DQBFT paradigm.

(P1) If a correct replica executes a command α at sequence number $\mathcal{S}$ in view v, no correct replica will execute $c' \neq c$ with sequence number $\mathcal{S}$.
(P2) If a correct replica executed a command α at sequence number $\mathcal{S}$ in view v, no correct replica will execute α with sequence number $\mathcal{S}' > \mathcal{S}$ in any view $v' > v$.
(P3) During a stable view where the communication between correct replicas are synchronous, a proposed client command is committed by a correct replica.
(P4) A view v will eventually transition to a new view $v' > v$ if enough correct replicas request for it.

Every replica replicates its client commands using its D-instance protocol. Before dissemination, the requests are assigned a replica-specific sequence number to generate a partial ordering among other requests originating at the same replica. The primary uses the ordering protocol to find a global order for the client requests disseminated by the replicas.

In DQBFT, the O-instance primary replica performs a fixed amount of work. Since the task of disseminating the client requests is offloaded to other replicas, the primary is only responsible for totally ordering the requests. An important aspect of DQBFT is that the ordering subprotocol does not order client requests, but only the replica-specific sequence numbers assigned to the client requests by the replicas during dissemination.

Futhermore, DQBFT allows the dissemination and the ordering protocols to proceed simultaneously. While request dissemination is in progress, the ordering protocol *optimistically* proposes a global ordering for the request. This allows the communication for both the protocols to overlap and effectively reduce the overall number of communication steps. Note that such concurrent processing does not strain the communication channels. Since the ordering protocol is only ordering the sequence numbers, the message sizes are constant and are only a few bytes. The dissemination protocol carries a larger variable payload containing the client requests.

While optimistic ordering enables substantial reduction in the effective number of communication steps improving latency, the faulty behavior of the replicas can causes holes in the

[2] A replica may lead multiple D-instances during failures.

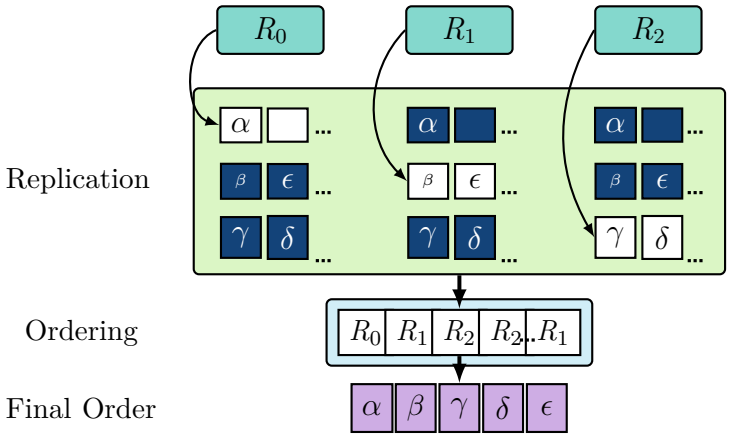


Figure 7.1: DQBFT's separation of replication and ordering.

sequence number space generated by the replicas and the primary. It is possible that a global sequence number is assigned without correct replicas ever receiving the disseminated request. In such cases, the holes must be *reclaimed and filled* in order for further sequence numbers to be executed.

7.2.2 Protocol Description

Figure 7.1 illustrates DQBFT's separation of replication and ordering steps. The algorithm is shown in Figure 7.2.

Client

A client can submit its command to any replica. A client command contains an operation and a monotonically increasing timestamp. Every replica maintains a data structure containing the last executed timestamp and the corresponding reply for each client. This is used to ensure that the replicas do not execute duplicate operations and to provide a reply to the client when required. After sending the command, the client sets up a timer and waits for the replicas to respond or the timer expires.

If the client timer expires before the replicas respond, the client sends the command to all the replicas and indicates its initial target replica, which serves to indicate the client's original recipient. Replicas will respond to the client if they have a reply. Otherwise, if the command timestamp has not been seen before, then correct replicas forward the command to the target replica and observe whether it makes progress in time. The view change protocol is triggered if the target replica does not make any progress.

Since any replica can process client commands in DQBFT, faulty clients can send the same

A Destiny replica executes the following sub-protocols:
Dissemination Protocol
N instances of PBFT protocol are used for dissemination. Each replica "owns" one instance and replicates its client requests with that instance. The prefix "D-" and the replica identifier embedded in the messages helps to identify the protocol instance.

(1) **D-PrePrepare.** A replica R_n receives a client request for an operation o and sends a $\langle \texttt{D-PrePrepare}, v_n, R_n, \mathcal{S}_i^n, o \rangle$ message to all replicas. v_n is the view number and $\mathcal{S}_i^n$ is the lowest available sequence number.

(2) **D-Prepare.** A replica R_m that receives a PrePrepare message $\langle \texttt{D-PrePrepare}, v_n, R_n, \mathcal{S}_i^n, o \rangle$ ensures the validity of the view and sequence numbers. Consequently, R_m sends a $\langle \texttt{D-Prepare}, v_n, R_n, \mathcal{S}_i^n, Hash(o) \rangle$ message to the collectors.

(3) **D-Commit.** A replica collects $2f + 1$ `D-Prepare` messages, aggregates the signature shares into a single signature, and sends the signature along with the $\langle \texttt{D-Commit}, v_n, R_n, \mathcal{S}_i^n, Hash(o) \rangle$ message. A replica that receives $2f + 1$ Commit messages marks the operation as *disseminated*.

Ordering Protocol

(1) **O-PrePrepare.** The primary R_p of the ordering protocol receives the $\langle \texttt{D-PrePrepare}, v_n, R_n, \mathcal{S}_i^n, o \rangle$ from R_n and begins to optimistically assign a global ordering number for $\mathcal{S}_j^p$. Primary R_p sends the $\langle \texttt{O-PrePrepare}, v_p, R_p, \mathcal{S}_j^p, R_n, \mathcal{S}_i^n \rangle$ message to all replicas.

(2) **O-Prepare.** A replica R_q that receives a PrePrepare message $\langle \texttt{O-PrePrepare}, v_p, R_p, \mathcal{S}_j^p, R_n, \mathcal{S}_i^n \rangle$ ensures the validity of the view and sequence numbers and the fact that the sequence number is assigned to a valid client request. Consequently, R_q sends a $\langle \texttt{O-Prepare}, v_p, R_p, \mathcal{S}_j^p \rangle$ message to the collectors.

(3) **O-Commit.** A replica collects $2f + 1$ `O-Prepare` messages, and sends the signature along with the $\langle \texttt{O-Commit}, v_p, R_p, \mathcal{S}_j^p \rangle$ message. A replica that receives the Commit message verifies the signature and follows the execution procedure.

Figure 7.2: DQBFT Protocol Execution using the PBFT protocol.

command with the same timestamp to different replicas specifying each of them as the initial target. Since correct replicas maintain a data structure containing the last executed timestamp per client, Correct replicas will skip execution of the command if it has already been executed. Hence, this will not cause multiple executions of the same command.

Agreement Protocol

A replica R_n that receives the client command, say α, becomes the command's coordinator. The coordinator is responsible for enforcing a partial ordering for α with respect to the commands previously coordinated by it. The coordinator uses its D-instance protocol and assigns a sequence number $\mathcal{S}_i^n$ before replicating the command to other replicas. Concurrently, the O-instance primary must be notified to globally order the D-instance sequence number $\mathcal{S}_i^n$. Note that the O-instance protocol only orders the D-instance sequence numbers, and not the commands. The O-instance primary uses the initial message (e.g. PrePrepare) sent by the D-instance primary as the request for finding the global order for $\mathcal{S}_i^n$. The O-instance primary R_p assigns a sequence number $\mathcal{S}_j^p$ to the C-instance number $\mathcal{S}_i^n$ and sends protocol messages to replicas to produce a global order for α.

Execution

A command α proposed by replica R_n is *decided* at a replica when it has been committed under a D-instance sequence number $\mathcal{S}_i^n$ and the D-instance sequence number $\mathcal{S}_i^n$ has been committed under a O-instance sequence number $\mathcal{S}_i$. However, the command cannot be executed yet. The command is considered *ready* for execution only after all the corresponding commands mapped to the O-instance sequence numbers up to $\mathcal{S}_i$ are committed and executed. Replicas execute the command and respond to the client.

Checkpoint and State-transfer Protocols

As described earlier, the consensus protocols use the checkpoint mechanism to reduce the memory footprint at the replicas by garbage collecting logs for previously executed commands. This procedure also aids in bringing any lagging replicas to the latest state, by allowing up-to-date nodes exchange the checkpoint data and recent logs via the state-transfer protocol. Note that replicas can be lagging as a result of primary's intentions. This has implications for DQBFT.

Example 7.1. Consider a DQBFT instantiation of PBFT with $N = 4$ ($f = 1$) replicas. Let R_0 be the O-instance primary and act Byzantine. Let R_1 and R_2 use their D-instances to commit two commands, say α and β using quorum replicas R_0, R_1, and R_2. Let R_0, the O-instance primary, order the D-instances using quorum replicas R_0, R_1, and R_3. Now, R_1 is the only replica that can execute the commands and respond to the client. Neither R_2 nor

R_3 have all the necessary O-instance and D-instance messages, respectively, to execute the commands. Thus, these replicas must transfer up-to-date state from other correct replicas before execution.

Below, we discuss measures to prevent malicious replicas from stalling progress of other replicas.

Controlling Byzantine Behavior

While the optimistic ordering mechanism reduces the number of effective communication steps, it enables Byzantine replicas to elongate the latency at each step thereby increasing the overall request latency. We rely on two techniques to prevent such behavior. A Byzantine replica can send the `D-Prepare` message only to the O-instance primary that causes the global sequence number to be assigned without the request being disseminated. A common technique to prevent this behavior is by flooding the `D-Prepare` message. When a correct replica receives a `D-Prepare` message, it will multicast the message to all other replicas. This will ensure that the `D-Prepare` message will be received by other replicas in one communication step after initially received by a correct replica. To reduce the large number of messages, alternatively, a combination of collectors and a gossip protocol can perform this function.

Despite, a replica can still collude with the primary and cause a global ordering slot to be committed without disseminating the request. To curb such behavior from reducing the overall performance of the system through view changes, we fall back to a more pessimistic approach on a per R-instance basis. After an R-instance view change due to lack of progress, correct replicas will now require the O-instance primary to wait until the requests are properly disseminated. If the R-instance appears to make behave properly for a time interval, this requirement is relaxed and the optimistic approach is followed.

View Change Protocol

There are cases where the primaries do not make progress, either deliberately or due to other non-Byzantine causes (e.g. network disruptions). The view change protocol is used to reinstate progress whenever the primaries of the D-instances or the O-instance fail to do so. One of the important characteristics of DQBFT is that it adds no additional complexity in terms of additional steps over the existing view change procedure used by the underlying consensus protocols. Recall that, a replica triggers a view change if it does not receive timely messages from the primary or if it receives a proof that the leader is faulty (either via a publicly verifiable contradiction from the client or when $f + 1$ replicas complain).

Since, in DQBFT, replicas can serve multiple roles (e.g. primary, backup) at the same time, it is possible that a replica makes progress on a subset of roles while ceasing progress on other

roles. By using the view change procedures of the respective D-instances and the O-instance, we ensure that only those primary roles that do not make progress are replaced, without affecting the other roles.

D-instance View Change. When a client does not receive $f + 1$ responses for its command in time, it forwards the command to all replicas and indicates its *initial target* primary. Correct replicas that have not yet seen the D-instance sequence number assigned for the command will forward the command to the target primary, reset a timer, and wait for the primary to assign a sequence number under its D-instance and send the initial message. If the replica timers expire without a message from the primary, correct replicas will invoke the view-change procedure for the D-instance.

The D-instance primary will propose a special *no-op* command for sequence numbers that do not have a command associated but were committed in the O-instance. This can happen if the O-instance primary receives the message for D-instance sequence number $\mathcal{S}_i^n$, but the sequence number did not commit before the view change, and no correct replica is aware of the command in that sequence number.

O-instance View Change. On the other hand, a client can also time-out if the O-instance primary fails to make timely progress. In this case, it is the responsibility of the correct replicas to monitor the O-instance primary and start the view change. Correct replicas start the timer for the O-instance primary after they have received the client command from the respective D-instance primary.

7.2.3 Correctness

DQBFT guarantees the following properties of a consensus protocol:

- **Safety.** Any two correct replicas execute the same sequence of client requests.
- **Liveness.** A client request proposed by a replica will eventually be executed by every correct replica.

Lemma 7.2. *If a correct replica executes a command α whose D-instance sequence number $\mathcal{S}_i^n$ is mapped to O-instance sequence number $\mathcal{S}_j$ in view v, no correct replica will execute $\beta \neq \alpha$ at O-instance sequence number $\mathcal{S}_j$ in view v.*

Proof. The D-instance and O-instance protocols satisfy property P1 (Section 7.2). Consequently, α is committed by replica R_n's D-instance at sequence number $\mathcal{S}_i^n$ by correct replicas. Furthermore, $\mathcal{S}_i^n$ is the value committed at O-instance number $\mathcal{S}_j^o$. Assume a correct replica R_m executes β at $\mathcal{S}_j^o$. This would entail that either (i) β was committed at $\mathcal{S}_i^n$ by correct replicas or (ii) some $\mathcal{S}_k^n$ assigned to β was was committed at O-instance $\mathcal{S}_j^o$ instead of $\mathcal{S}_i^n$. This contradicts property P1. □

Lemma 7.3. *If a correct replica executes a command α whose D-instance sequence number $\mathcal{S}_i^n$ is mapped to O-instance sequence number $\mathcal{S}_j^o$ in view v, no correct replica will execute $\beta \neq \alpha$ at O-instance sequence number $\mathcal{S}_j^o$ in any view $v' > v$.*

Proof. The individual consensus instances satisfy Property P2. The property ensures that if a command is chosen at a sequence number, it will remain chosen at that sequence number at all higher views. Thus, α remains chosen at $\mathcal{S}_i^n$ at all higher views. Similarly, $\mathcal{S}_i^n$ is mapped to $\mathcal{S}_j^o$ at all higher views. Suppose a correct replica executes β at view $v' > v$, then either (i) β is assigned to $\mathcal{S}_i^n$ by correct replicas or (ii) some $\mathcal{S}_k^n$ assigned to *beta* was committed at O-instance $\mathcal{S}_j^o$ instead of $\mathcal{S}_i^n$. Both the conditions contradict property P2. □

The Lemmas 7.2 and 7.3 satisfy the following theorem for safety:

Theorem 7.4. *Any two correct replicas commit the same sequence of operations.*

Lemma 7.5. *During a stable view of the O-instance and the D-instance, a proposed client command is executed by a correct replica.*

Proof. In a stable view, the correct primary will propose client requests in a timely fashion to the replicas (Property P3). Thus, the D-instance primary will ensure dissemination of the client requests. Since there are at most f faulty replicas, there will remain $N - f$ correct ones that will respond to primary's messages. Thus, clients commands will be committed during the view by correct replicas after receiving from a correct D-instance primary. The O-instance primary, being one of the correct replicas, will receive the D-instance primary's PREPREPARE [3] with the client command and sequence number. The correct O-instance primary will send the D-instance sequence number as its command to correct replicas and it will be committed in the view. Thus, the client command will be assigned a global order by correct replicas mapping the D-instance sequence number to the corresponding O-instance. □

Lemma 7.6. *A view v will eventually transitioned to a new view $v' > v$ if at least $N - f$ replicas request for it.*

Proof. The proof follows directly from Property P4 applied to both the D-instances and the O-instance. □

Theorem 7.7. *A command sent by a correct client is eventually executed by correct replicas.*

Proof. During a stable view, Lemma 7.5 shows that the proposed command is learnt by correct replicas. When the view is unstable and the replica timers expire properly, $f + 1$ correct replicas will request a view change. By Lemma 7.6, a new view v' will be installed.

[3] BFT primaries send a PrePrepare message, while Hybrid primaries send a Prepare message

However, if less than $f+1$ replicas request the view change, then the remaining replicas that do not request the view change will follow the protocol properly. Thus, the system will stay in view v and the replicas will continue to commit requests in the view. When proposals are not committed in time or when more than f replicas request a view change, then all correct replicas will request a view change and it will be processed as in Lemma 7.5.

Even after a view change, the new view v' may not necessarily be stable. If the new primary deviates from the algorithm or does not process messages in time, this will cause correct replicas to request another view change and move to the next view. Since there can only be at most f faulty replicas, after at most $f+1$ view changes, a stable view will be installed. Furthermore, if the faulty primary follows the algorithm enough such that a view change cannot be triggered, by Lemma 7.5, replicas will continue to commit the blocks. □

The individual consensus protocols satisfy linearizability [42]. The below theorem proves that a command executed after committing via a D-instance and an O-instance satisfy linearizability.

Theorem 7.8. Linearizability*: If α and β are commands, and the request for β arrives after α is ready, then α will be executed before β.*

Proof. When α is ready, there must be at least i O-instances belonging to R_n. We prove this by contradiction. Assume there are less than i O-instances for R_n but α is ready. This can happen only because there is a view change, and correct replicas observe less than i instances. However, since α was ready for execution before the view change, there is at least one correct replica that will ensure that the primary of the new view enforces no less than i instances. Thus, this is a contradiction.

When β is received after α is ready, there should be at least i O-instances committed belonging to R_n. Case (i): If the O-instance primary is non-faulty, it will only assign sequence numbers in monotonically increasing order, so there will be no empty slots. Case (ii): After a O-instance view change, correct replicas will observe at least i O-instances belonging to R_n since α is ready, and they will ensure that the new primary enforces the i O-instances for R_n.

□

7.3 Conclusions

We presented DQBFT, a paradigm for designing highly scalable BFT consensus protocols. DQBFT's key insight is the observation that consensus requires two actions for committing: replication and ordering, and these can be performed independently. While replication does not require coordination and can therefore be decentralized, ordering requires coordination

to ensure a single consistent view of the operation sequence and can be efficiently performed using a sequencer.

In the next chapter, we leverage DQBFT to build a highly scalable BFT protocol and also provide an empirical evaluation of various instantiations of DQBFT.

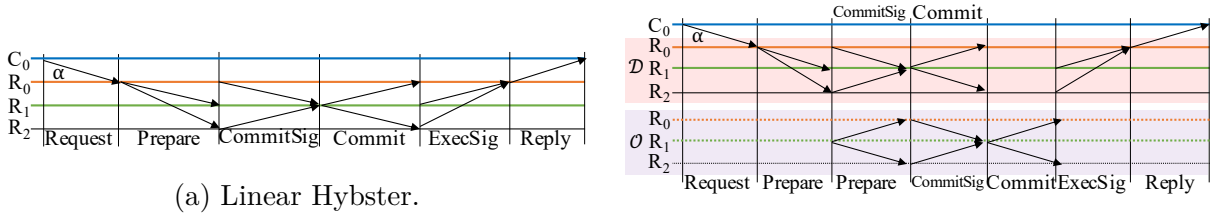
Chapter 8

Scaling Partial Decentralization with Destiny

While the DQBFT paradigm can be adopted by most BFT protocols, this chapter identifies and presents a combination of techniques that elevate the impact of the partial decentralization approach. The result is Destiny, a highly scalable, partially decentralized BFT consensus protocol. Destiny leverages the DQBFT paradigm and enhances it with three techniques, each of which has been designed to achieve high performance while scaling to hundreds of replicas. Briefly, the techniques include: (1) using a hardware-assisted trusted subsystem to increase the fault-tolerance capability and decrease quorum sizes; (2) linear communication; (3) using cryptography to allow a single-message acknowledgement.

Traditional BFT consensus protocols perform multiple rounds of communication among two-thirds of the replicas to reach agreement which quickly inflates the latency in geo-distributed deployments. Prior works have shown that the lower bound on the number of nodes required to reach Byzantine consensus can be reduced from $3f+1$ to $2f+1$, where f is the number of tolerable faults, by utilizing a trusted subsystem [32, 135]. Such protocols are called *Hybrid Fault-Tolerant* consensus protocols. These protocols can provide better performance than classical BFT protocols, but as we show in Section 8.2, the techniques used are not directly applicable at the scale of hundreds of nodes and require some careful considerations. Specifically, we show that protocols that use linear communication at least during the common case scale better than non-linear protocols. With this insight, we adopted and linearized the common case communication of Hybster [32], producing Linear Hybster. Both the replication and ordering steps of Destiny use instances of the *Linear Hybster* protocol.

By performing an empirical evaluation, we show that, while the DQBFT technique is generally applicable to many primary-based BFT protocols, a careful design of the individual components can vastly improve performance and scalability beyond a plug-and-play solution. We implemented and evaluated three protocols under the DQBFT paradigm including Destiny in an geo-distributed setting with various system sizes ranging from 19 nodes up to 294 nodes, withstanding between $f = 9$ and $f = 147$ Byzantine failures (Section 8.2). Our evaluations reveal that the proposed techniques are highly effective. The decentralized versions of SBFT and Destiny outperform their vanilla counterparts: $\approx 3\times$ better throughput. Furthermore, Destiny provides up to 50% lower latency than the next best protocol Linear Hybster.



(a) Linear Hybster.

(b) Destiny.

Figure 8.1: Normal communication steps in Linear Hybster and Destiny.

The rest of this chapter is organized as follows. Section 8.1 proposes Destiny, a trust-based BFT protocol under the DQBFT paradigm that scales to hundreds of geo-replicated nodes. Section 8.2 presents the empirical evaluation of Destiny, DQBFT protocols, and other state-of-the-art solutions. Section 8.3 concludes the chapter.

8.1 Protocol Design

We now present Destiny, a instantiation of the DQBFT paradigm designed for scaling to hundreds of replicas, and provide consistently high throughput and low latency even under high loads. We note that for deployments that cannot depend on the trusted subsystem, a traditional BFT instantiation should provide better performance, and we evaluate them in Section 8.2. We begin with a description of various components and techniques underpinning Destiny.

Destiny follows a hybrid fault model due to its ability to tolerate more faults and use smaller size quorums. Destiny leverages a custom variant of Hybster [32], called Linear Hybster, to achieve higher performance and greater scalability.

Linear Hybster improves Hybster's normal-case communication complexity from quadratic to linear using threshold signatures and specialized collector roles. The collector aggregates messages from replicas and re-broadcasts to all replicas. Since the messages are cryptographically signed, threshold signatures [39, 126, 130] can be used to reduce the number of outgoing collector messages from linear to constant.

The same mechanism is employed for responding to the client. Clients wait for a single aggregated reply from a collector replica, instead of waiting for replies from $f + 1$ replicas. The collector replica collects the signatures from $f + 1$ replicas and sends a single response and signature to the client.

8.1.1 System Model and Cryptography

We consider a set of nodes, which includes replicas as well as clients, in an asynchronous system, that communicate via message passing. We assume the hybrid fault model – the Byzantine fault model augmented with trusted hardware – in which nodes can behave arbitrarily, except the trusted subsystem, which can only fail by crashing. Every replica, however, is capable of producing cryptographic signatures [80] that faulty nodes cannot break. To preserve safety and liveness, Destiny requires at least $N = 2f + 1$ replica nodes to tolerate f arbitrary faults. Safety is guaranteed as long as only up to f replicas fail. Liveness is guaranteed during periods of synchronous communication.

We consider an adversary that controls all the system software including the operating system. However, the adversary cannot read or modify the trusted environment's memory at run-time or learn any information about the secrets held in the subsystem's data. Furthermore, the trusted subsystem is capable of generating cryptographic operations that the adversary cannot break. We also assume that the adversary cannot compromise the trusted subsystem's protections on participating nodes (e.g., via physical attacks). We acknowledge that the trusted environments are prone to rollback attacks. For simplicity of prose, we assume that such attacks can be prevented using existing techniques [110].

Destiny uses threshold signatures to aggregate signatures at the collector. The threshold signature with a threshold parameter t allows any subset t from a total of n signers to produce a valid signature on any message. It also ensures that no subset less than size t can produce a valid signature. For this purpose, each signer holds a distinct private signing key that can be used to generate the corresponding signature share. The signature shares of a signed message can be combined into a single signature that can be verified using a single public key. We use a threshold signature scheme based on Boneh-Lynn-Shacham (BLS) signatures [105]. Such signatures are constructed using pairing over elliptic curves. We use the BLS12-381 [30] signature scheme that produces 192-byte signature shares. The aggregate signatures are also 192 bytes long.

We assume a computationally bounded adversary that cannot do better than known attacks. he communication between replicas and clients is authenticated using public key infrastructures (PKI) such as TLS.

8.1.2 The ThreshSign Subsystem

The ThreshSign subsystem is a local service that exists on every node. It allows for creating and verifying different types of threshold signatures for a message m using a specified counter tc and a corresponding counter value tv. By hosting part of ThreshSign in a trusted subsystem (described later), ThreshSign guarantees a set of properties even if the replica is malicious.

ThreshSign provides the following functions:

- **Independent Counter Signature Shares** with input (m, tc, tv'). ThreshSign generates such a signature for a message m if the provided new value tv' for counter tc is *greater than* its current value tv. It updates the counter tc's value to tv' and computes a signature share using the subsystem's instance ID, counter tc's ID, its new value tv', current value tv, and the message m.
- **Aggregate Signature Shares**. It returns a single signature by aggregating at least t valid threshold signatures.
- **Verify Signature**. It verifies the aggregated signature *sig* using the public key and returns true if message m was indeed accepted by at least $f + 1$ nodes and is assigned the counter value tv of counter tc.
- **Continuing Counter Certificates** with input (m, tc, tv, tv'). ThreshSign generates a message authentication code (MAC) certificate for a message m if the submitted new value tv' for counter tc is *greater than or equal to* its current value tv. It updates the counter tc's value to tv' and computes a signature share using its private key share, the subsystem's instance ID, counter tc's ID, its new value tv', current value tv, and the message m.
- **Verify Certificate** with input (m, mac, tc, tv, tv'). It verifies the MAC certificate mac using the secret key and returns true if message m is assigned a continuing certificate that transitions the counter tc from tv to tv'.

In addition, ThreshSign also provides the capabilities of TrInX [32], the original Hybster's trusted component to aid in view change and state transfer mechanisms. We implement the ability to instantiate ThreshSign's multiple instances within a single trusted subsystem. Every instance can host a variable number of counters as needed by the protocol. For instance, Hybster requires certificates using at least three different counters for different protocol phases (e.g., checkpoints, view changes). Furthermore, the signing and the certifying functions must be hosted securely along with the private keys inside the trusted subsystem, while the signature aggregation and verification functions may be hosted outside as they only deal with public keys. To initialize the trusted subsystem, we rely on attestation services provided by hardware vendors to verify that the secure code is running inside the enclave and perform any initialization steps.

8.1.3 Linear Hybster

We now discuss the modifications to Hybster [32], to achieve linear communication in the common case to create Linear Hybster.

Figure 8.1a shows Linear Hybster's execution steps in the normal case. Hybster commits a command in two steps and requires clients to wait for $f + 1$ replies. A quadratic number of *Commit* messages are exchanged by replicas in an all-to-all communication, which bottlenecks throughput. We use a collector and an additional communication step to reduce this

A Destiny replica executes the following sub-protocols:

Dissemination Protocol

N instances of the Linear Hybster protocol are used for dissemination. Each replica "owns" one instance and replicates its client requests with that instance. The prefix "D-" and the replica identifier embedded in the messages helps to identify the protocol instance.

(1) **D-Prepare.** A replica R_n receives a client request for an operation o and sends a $\langle$`D-Prepare`$, v_n, R_n, \mathcal{S}_i^n, o\rangle$ message to all replicas. v_n is the view number and $\mathcal{S}_i^n$ is the lowest available sequence number.

(2) **D-CommitSig.** A replica R_m that receives a Prepare message $\langle$`D-Prepare`$, v_n, R_n, \mathcal{S}_i^n, o\rangle$ ensures the validity of the view and sequence numbers. Consequently, R_m sends a $\langle$`D-CommitSig`$, v_n, R_n, \mathcal{S}_i^n, \mathit{Hash}(o)\rangle$ message to the collectors. The CommitSig message is attached an independent counter signature share obtained by invoking the trusted ThreshSign service with $(v_n|\mathcal{S}_i^n)$ as the counter value.

(3) **D-Commit.** A collector collects $f + 1$ `D-CommitSig` messages, aggregates the signature shares into a single signature, and sends the signature along with the $\langle$`D-Commit`$, v_n, R_n, \mathcal{S}_i^n, \mathit{Hash}(o)\rangle$ message. A replica that receives the Commit message verifies the signature and marks the operation as *disseminated*.

Ordering Protocol

(1) **O-Prepare.** The primary R_p of the ordering protocol receives the $\langle$`D-Prepare`$, v_n, R_n, \mathcal{S}_i^n, o\rangle$ from R_n and begins to optimistically assign a global ordering number for $\mathcal{S}_j^p$. Primary R_p sends the $\langle$`O-Prepare`$, v_p, R_p, \mathcal{S}_j^p, R_n, \mathcal{S}_i^n\rangle$ message to all replicas.

(2) **O-CommitSig.** A replica R_q that receives a Prepare message $\langle$`O-Prepare`$, v_p, R_p, \mathcal{S}_j^p, R_n, \mathcal{S}_i^n\rangle$ ensures the validity of the view and sequence numbers and the fact that the sequence number is assigned to a valid client request. Consequently, R_q sends a $\langle$`O-CommitSig`$, v_p, R_p, \mathcal{S}_j^p\rangle$ message to the collectors. The CommitSig message is attached an independent counter signature share obtained by invoking the trusted ThreshSign service with $(v_p|\mathcal{S}_j^p)$ as the counter value.

(3) **O-Commit.** A collector collects $f + 1$ `O-CommitSig` messages, aggregates the signature shares into a single signature, and sends the signature along with the $\langle$`D-Commit`$, v_p, R_p, \mathcal{S}_j^p\rangle$ message. A replica that receives the Commit message verifies the signature and follows the execution procedure.

Figure 8.2: Destiny Protocol Execution.

quadratic communication to linear. In *Linear Hybster*, replicas send the *Commit* messages to a collector (up to $f + 1$ can be used for fault-tolerance), which aggregates at least $f + 1$ messages and sends them to other replicas. Hybster uses *TrInX*, a *trusted* MAC provider that ensures authenticity and non-repudiation of messages. This requires any pair of replicas to use unique secret keys to exchange messages between them. We replace *TrInX* with a ThreshSign subsystem instance σ. ThreshSign is configured with a threshold of $f + 1$ out of $N = 2f + 1$ total replicas.

Furthermore, Hybster (and most BFT protocols) require that the clients wait for equivalent replies from at least $f + 1$ replicas to defend against incorrect responses from malicious replicas. Linear Hybster, in contrast, reduces this $f + 1$ communication to one single message using threshold signatures. For this purpose, Linear Hybster uses another instance of threshold signatures, π, with threshold $f + 1$. Now, once the client command is executed at each replica, the result of execution is signed using π and sent to the collector in a EXECSIG message. The collector collects and aggregates signature shares from $f + 1$ valid EXECSIG messages and generates an EXECPROOF message. This message is sent to the replicas as well as the client along with the result of execution. The client validates the aggregated signature, accepts the result, and returns.

View Change. A replica triggers a view change if it does not receive timely messages from the leader or if it receives a proof that the leader is faulty (either via a publicly verifiable contradiction from the client or when $f + 1$ replicas complain).

Replica R_m supports a new primary R'_p of a view $v+1$ by sending a VIEWCHANGE message with the PREPAREs for all order numbers in its current ordering window in view v. A continuing counter certificate is attached to the message to ensure that even if replica R_i is faulty, it includes all the PREPAREs it is aware of up to the current order number. After sending the VIEWCHANGE message for $v + 1$, replica R_m is prohibited from participating in view v. Due to the use of continuing counter certificates, a new leader R'_p can determine all the proposals of the former primary R_p by collecting only a quorum of VIEWCHANGE messages.

Once a correct leader l' collects at least $f+1$ VIEWCHANGE messages, it begins constructing the new view. It is possible that the new leader is lagging behind the current ordering window, in which case the new leader invokes the state-transfer protocol to request the checkpoint messages and the service state from an up-to-date replica. A replica cannot establish as a new leader until its ordering window matches with the VIEWCHANGE messages. Since only f replicas can be faulty at most, there is at least one correct replica that contains the adequate information to help the new primary move to the new ordering window.

Unlike the agreement protocol, the view change mechanism makes use of continuing counter certificates provided by ThreshSign. For a view change, replicas individually must announce their current view and their intended view, unlike normal case execution where replicas jointly accept a proposed command. Continuing counter certificates serve this purpose well allowing replicas to individually prove their log state to other replicas and the new primary.

8.1.4 Protocol

We begin our discussion with an example of Destiny's agreement protocol followed by a description of the different sub-protocols.

Example 8.1. Figure 8.1b illustrates with an example how Destiny commits a command using the D- and O-instance protocols.

Assume that R_1 serves the primary role in the O-instance protocol. A client submits command α to replica R_0. R_0 becomes the *initial* coordinator of α. We call it initial because if it fails, some other replica will take over. We also assume that R_0 and R_1 will play the collector roles for the D-instance and O-instance respectively. A replica playing the collector role is responsible for collecting signature shares, aggregating them, and multicasting the combined signature. R_0 selects the lowest unused sequence number in its D-instance space, assigns it to α, and disseminates the command by multicasting a `D-Prepare` message.

R_1 receives the `D-Prepare` message and triggers the O-instance protocol for replica R_0. R_1 proposes R_0's ID and α's sequence number to the next available O-instance sequence number (say j) and sends a `O-Prepare` message to give R_0 an O-instance to commit its command.

Replicas accept either of the prepare messages and send the corresponding `D-CommitSig` and `O-CommitSig` messages, respectively, to the commit collectors, R_0 and R_1. The `D-CommitSig` and the `O-CommitSig` messages are signed by the σ_0 and τ ThreshSign instances respectively. The respective commit collectors wait for at least $f + 1$ valid `D-CommitSig` (respectively `O-CommitSig`) messages and invoke the ThreshSign subsystem to aggregate the signature shares in the commit messages into a single signature. The aggregated signatures are sent via the corresponding `D-Commit` and `O-Commit` messages.

Replicas receive the valid `O-Commit` and `D-Commit` messages, commit the command, and mark it for execution. After executing the command at the global order number j, each replica signs the resulting state and sends a signed `ExecSig` message to the *execution* collector. The execution messages are signed using a ThreshSign instance that is different from the ones used during commit. This ThreshSign instance does not require trust and is kept outside the trusted subsystem.

The execution collector R_0 collects at least $f + 1$ valid `ExecSig` messages, aggregates the signatures into a single `ExecProof` message, and sends the message to all the replicas. It also sends this message to the client along with the result of execution. The client verifies the signature, accepts the result, and returns.

Agreement Protocol

The normal-case protocol is presented in Figure 8.2. We assume that a primary for the O-instance is elected beforehand. We say that a command is *decided* when the command leader can safely respond to the client. For a command to be ready, the D-instance must be

committed and the corresponding O-instance belonging to R_i's sequence number must be committed.

Destiny commits both D-instances and O-instances strictly using the Linear Hybster protocol after being accepted by a majority, requiring a total of four communications steps. Note that the O-instance starts after the first communication step of the D-instance. Verification of execution results takes an additional two communication steps. The messages in the normal phase protocol are signed by invoking the *Independent Counter Signature Shares* function of the corresponding ThreshSign instance. This ensures the following properties: (i) *Uniqueness*: the same counter value is not assigned to two different messages and (ii) *Monotonicity*: the counter value assigned to a message will always be greater than the previous counter value.

Execution and Acknowledgement

Replicas execute commands as they become *ready* for execution. After execution, as in Linear Hybster, replicas forward the signed result to a collector, which then aggregates $f + 1$ signatures. The collector sends this signature back to the replicas and to the client, indicating that the client's command was executed. Note that this step does not require the use of the trusted subsystem.

Checkpoint, State-tranfer and View change Protocols

Destiny uses the respective checkpoint, state-transfer, and view change algorithms of underlying Linear Hybster protocol.

8.2 Evaluation

We implemented Destiny and DQBFT versions of the SBFT (DQ-SBFT) and Hybster (DQ-Hybster) protocols, and conducted an experimental evaluation to answer the following questions:

1. How does the throughput and latency of the protocols under the DQBFT paradigm compare to state-of-the-art protocols in a geo-distributed deployment?
2. What is performance and scalability improvement, if any, due to partially decentralized consensus? In other words, how does Destiny and the other DQBFT protocols compare to Linear Hybster and SBFT?
3. What is the impact of batching on the performance of protocols?
4. How do the DQBFT protocols scale from 10s to 100s of replicas?

5. How do the DQBFT protocols handle O-instance primary failures, compared to primary failures in primary-based protocols?

We evaluated Hybster [32], SBFT [64], Hotstuff [140], and Prime [17] alongside Destiny, DQ-SBFT, and DQ-Hybster. Hybster serves as the state-of-the-art hybrid protocol, while Hotstuff and SBFT serve as the state-of-the-art Byzantine protocols. We also consider Linear Hybster to quantify the effect of Destiny's partially decentralized consensus.

SBFT [64] is a scalable variant of PBFT that uses two commit modes: a fast and a slow path. Despite its normal-case linear communication complexity and a fast path commit sub-protocol, SBFT's performance is limited by the single-leader protocol and a fast quorum size of $N = 3f + 1$, when fast path redundancy c is zero. HotStuff [140] is a leader-based protocol that achieves linear communication complexity and uses a three-phase commit mechanism that yields a simpler view change protocol.

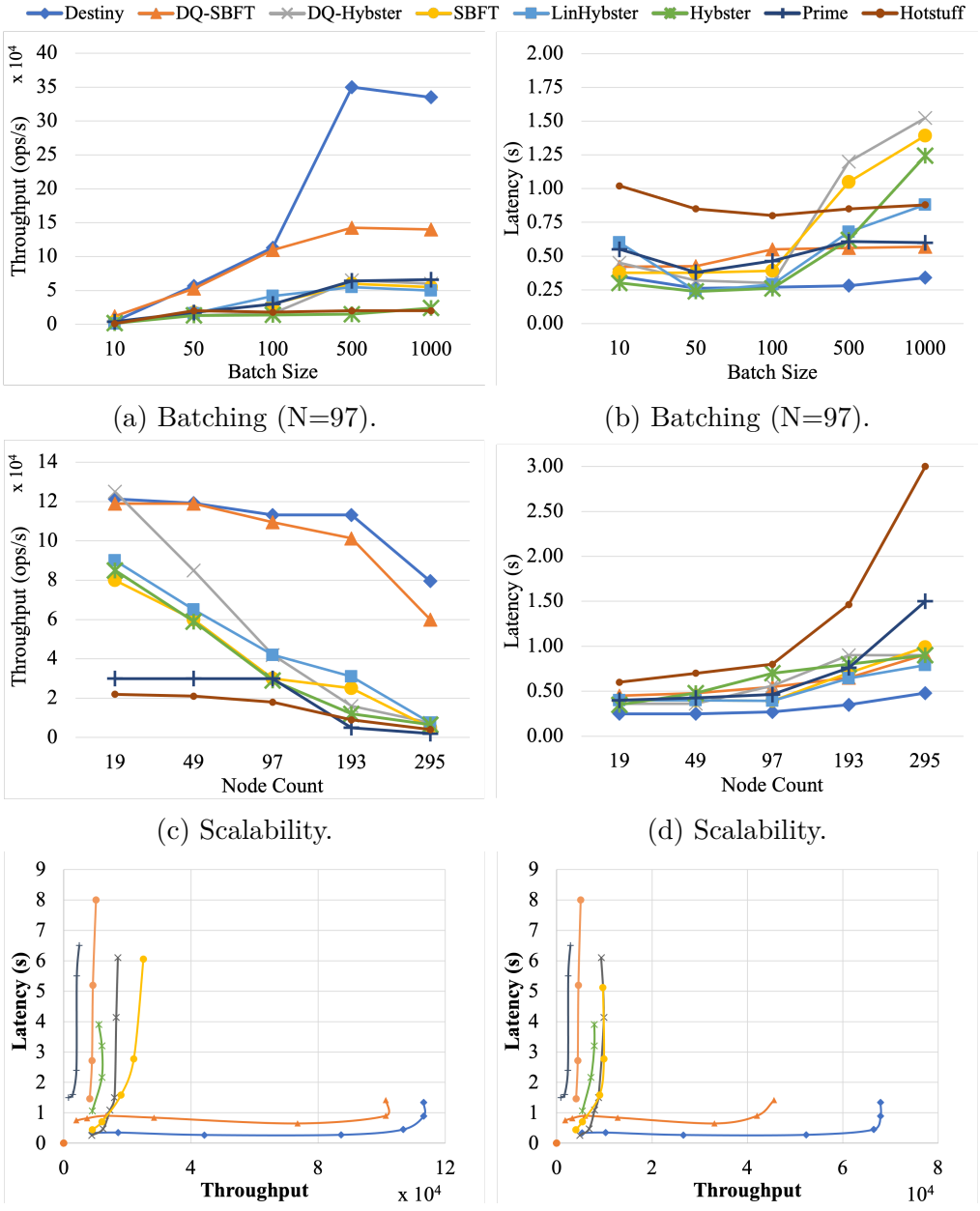
Implementation. We implemented all the protocols within a common framework in Go. The ThreshSign subsystem's trusted part and Hybster's trusted counter TrInx were implemented in C++ using Intel SGX SDK [52] and interfaced with the Go application using `cgo` [65]. For a fair comparison, we evaluated all systems using a common framework: we used `gRPC` [7] for communication and `protobuf` [67] for message serialization. We used the ECDSA [80] algorithms in Go's *crypto* package to authenticate the messages exchanged by the clients and the replicas. Our implementations of BFT protocols perform out-of-order processing of commands, except Hotstuff, which does not support out-of-order processing because it rotates the primary role regularly. For trust-based BFT protocols, the out-of-order processing is limited due to the use of counter-based trusted components. Creation of signatures using the trusted components happen in order, whereas all other message processing happen out-of-order.

Experimental setup. We used SGX-enabled VMs from the Azure Confidential Computing [112] platform. We obtained VMs from ten different datacenter regions: six in North America, three in Europe, and one in South East Asia. We deployed the protocols in each of these regions on multiple VMs.

The number of replicas and the number of VMs depend on the experiment. Each VM consists of 8 vCPUs and 32GB of memory. The VMs were part of a Kubernetes [66] cluster and the protocol replicas and clients were deployed as *pods*. We placed one replica pod per VM and placed the clients on different VMs. We designated a replica in East US to serve the primary role.

Table 8.1: Maximum tolerated failures for different N..

$N =$	19	49	97	193	295
SBFT, Hotstuff, Prime	6	16	32	64	98
Hybster, Destiny	9	24	48	96	147



(a) Batching (N=97).

(b) Batching (N=97).

(c) Scalability.

(d) Scalability.

(e) Performance (N=193).

(f) Performance with f failures.

Figure 8.3: Evaluation of DQBFT and other consensus protocols.

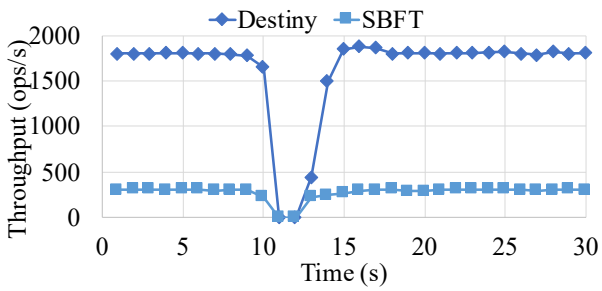


Figure 8.4: Throughput under leader failure.

We carried out experiments for five different values of N (the number of replicas) between 19 and 295. Table 8.1 shows the number of tolerated failures by the protocols for these values, for ease of exposition. Up to 5k clients were placed in each of the three regions depending on the experiment. Clients send requests in a closed-loop, meaning they wait for the result of the previous request before sending the next one. The request payload size was set at 256 bytes. Unless otherwise stated, we use a batch size of 100 client commands per batch.

8.2.1 Performance

We first measure the protocols' performance in terms of throughput and latency. For this experiment, we set $N = 193$. The results are shown in Figure 8.3e.

Performance of Hybster, Linear Hybster, and SBFT is limited by the primary. The performance of primary-backup protocols is limited by the primary's ability to disseminate and order client requests. The DQBFT protocols, DQ-SBFT and Destiny, provide 4× and 5× higher throughput over their original primary-based counterparts respectively. It should be noted that DQ-Hybster performs poorly than other DQBFT protocols. This is due to the quadratic number of messages that are processed by each D-instance and the quadratic number of trusted MACs that each replica must produce as well as verify. DQ-SBFT and Destiny, in contrast, perform better, particularly in geo-distributed, due to their underlying protocols that provide linear communication in the common case.

8.2.2 Scalability

In our first scalability experiment, we measured the impact of adding nodes on protocol performance. For this experiment, we varied between 19 and 295 replicas, and injected enough client requests to achieve the peak throughput without causing a latency spike for each protocol.

Table 8.1 shows the five different system sizes in which we evaluated the protocols. Figures 8.3c and 8.3d show the results.

The single-leader protocols – Hybster, Linear Hybster, and SBFT – show a similar downward trend in throughput as the system size grows. Hotstuff maintains its throughput up to 49 nodes after which the larger BFT quorums and primary rotation significantly degrade its performance. While Destiny and DQ-SBFT perform $\approx 1.5\times$ better than Hybster and SBFT at 19 replicas, the difference grows as system size increases, reacing a peak of $3\times$ at 193 replicas. DQ-Hybster sharply declines in performance due to its quadratic communication mechanism.

In Figure 8.3d, the latency of single-leader protocols only doubles between 19 and 295 nodes, while HotStuff show multi-fold increases. Destiny provides consistently lower throughput than other protocols. Note that while Hotstuff's latencies exceeds a second, Destiny provides 80% lower latency in the order of few hundred milliseconds at 295 replicas.

We observe that, at 295 replicas, Destiny and DQ-SBFT observe a sharp dip in performance. We attribute this decline partly to the lack of compute resources for the processes at this particular configuration. While for up to 193 replicas, we place each replica process in its own VM, at 295 replicas, we had to binpack 295 replicas into 193 VMs, due to Azure capacity restrictions. This limits the computation resources available for each replica process, thus reducing the throughput.

8.2.3 Batching

We also measured the impact of batching on the system performance. For this experiment, we deployed the protocols onto a 97-replica setup and varied the command batch size between 10 and 1000. The results are in Figures 8.3a and 8.3b.

It can be observed that the DQBFT variants, Destiny and DQ-SBFT are able to scale better with the batch size. This is because they do not saturate the resources of any particular replica unlike other primary-based protocols. Furthermore, it can be observed the Destiny performs better than DQ-SBFT at higher batch sizes of 500 and 1000. Up to batch size of 100, both Destiny and DQ-SBFT protocols are limited by the performance of threshold signatures mechanism. While both the protocols overcome this bottleneck at batch size of 500, Destiny is able to scale better as it uses fewer resources (in terms of replicas) to commit commands, due to its quorum size. DQ-SBFT on the other hand uses all the replicas for its quorum that saturates resources at higher batch sizes.

8.2.4 Availability under Failures

To measure the impact of failures on protocol availability and performance, we device two kinds of experiments. The first measures the impact on performance when only a quorum of replicas is available (otherwise there are f simultaneous failures). For this experiment, we instantiated each protocol with only a quorum of processes up and running. The quorum

size differs for each protocol and is based on the number of tolerated failures, f: $2f + 1$ for Hotstuff and Prime, $3f + c + 1$ for SBFT, and $f + 1$ for Hybster and Destiny.

Figure 8.3f shows the results. All protocols witness throughput reductions. Hybster and Destiny, now, must wait for responses from the additional regions in order to collect their quorum, while SBFT and DQ-SBFT is now forced to take the slow-path protocol with two additional communication steps. Hotstuff's latency and throughput are only slightly affected because the number of regions it needs to reach a quorum as well as the number of communication steps remain the same.

The second experiment measures the impact of leader failures during protocol execution. Destiny has a leader for each of the D-instances and the O-instance, but the highest impact is caused by the O-instance leader failure. Thus, for Destiny, we kill the O-instance leader.

Figure 8.4 shows the results. SBFT's and Destiny's throughput drop to zero as soon as their leaders are killed and the systems execute their respective view change protocols. Once a stable leader is reinstated, their throughput restores.

An important property of Destiny is its ability to make *delayed decisions*. While the O-instance was undergoing the view change, D-instances of the live replicas are still able to replicate the commands. Once a new O-instance leader is reinstated, the ordering of pending D-instances is batched together in fewer O-instance order numbers.

8.2.5 Summary of Results

We observe that applying the DQBFT paradigm is effective in providing multi-fold throughput improvements over the original protocols. However, when scaling to hundreds of replicas, we observe that protocols that provide linear communication are more effective candidates for DQBFT approach. Furthermore, when the compute bottleneck is alleviated using large batch sizes, we observe that Destiny can provide multi-fold throughput increase than DQ-SBFT. This is because Destiny's quorum size leads to lower resource utilization per command batch, and the available resources is then used for driving consensus of additional command batches.

8.3 Conclusions

Destiny implements DQBFT to perform partially decentralized consensus. Additionally, it linearizes all-to-all communication through message aggregation and threshold signatures, powered by trusted hardware. We presented Destiny's design, analysis, and evaluation, which reveals its superiority: 3× over prior art. Trust-based approaches such as Destiny require special (trusted) hardware. But their ubiquity and wide availability at the commodity-scale makes them increasingly viable.

Chapter 9

Flexible Trust-based Byzantine Fault Tolerance

In this chapter, we revisit the quorum intersection in Hybrid protocols. We show that the quorum requirements used by existing Hybrid protocols are too conservative. These existing quorums systems require that every quorum interest with all other quorums within a given view as well as across different views. However, we show that only quorums across views must intersect to ensure safety, while quorums within the same view need not intersect. Consequently, Hybrid protocols using the flexible quorum technique can opt for using smaller non-majority quorums during normal executions, in exchange for using much larger than majority quorums during view changes. With this intuition, we present *Flexible* MinBFT, a modified version of the state-of-the-art hybrid fault-tolerant protocol MinBFT [135], using the new *Flexible Hybrid Quorum* intersection technique. Our technique separates the relationship between the number of replicas N and the number of faults tolerated f that is typical in traditional hybrid protocols, and instead, enables adopting different values of f for the same N. *Flexible* MinBFT uses normal commit quorums of size at least $f + 1$ and view change quorums of size at least $N - f$.

The rest of the chapter is organized as follows. Section 9.1 revisits MinBFT and its majority quorums. Section 9.2 introduces the flexible quorum technique to produce *Flexible* MinBFT. Section 9.3 provides a proof of correctness for *Flexible* MinBFT.

9.1 Revisiting MinBFT

MinBFT [135] is a hybrid fault tolerant protocol that uses a trusted service to require only $N = 2f + 1$ replicas to tolerate f Byzantine faults. The trusted service prevents malicious replicas from equivocating to correct replicas, providing an efficient solution to the consensus problem under the hybrid fault model.

9.1.1 USIG Trusted Service

The protocol uses a trusted service called the Universal Sequential Identifier Generator (USIG) that is present in each replica and provides two interfaces: one for signing and

A MinBFT replica executed the following protocol.

(1) **Prepare.** The primary assigns a sequence number to the client request and sends a `Prepare` message to all replicas.

(2) **Commit.** Each replica receives the `Prepare` message and broadcasts a `Commit` message.

- A replica accepts a request if it collects a commit certificate consisting of $f+1$ `Commit` messages.
- If a replica does not hear back from the primary in time, it will send a `ReqViewChange` message.
- If a replica receives $f+1$ `ReqViewChange` messages, it transitions to the next view and sends the `ViewChange` message.

Figure 9.1: MinBFT Normal Execution.

another for verifying messages. The USIG service assigns monotonically increasing counter values to messages and signs them. The service provides the following properties: (i) Uniqueness: no two messages are assigned the same identifier; (ii) Monotonicity: a message is never assigned an identifer smaller than the previous one; and (iii) Sequentiality: the next counter value generated is always one more than the last generated value.

To access its service, USIG provides two interfaces:

- `CreateUI(`m`)` creates a signed certificate UI_i for message m with the next value from the monotonic counter. The certificate is computed using the private key of the USIG instance r. $UI_r = \langle ctr, H(m) \rangle_p$, where ctr is the counter value and $H(m)$ is the hash of the message.
- `VerifyUI(`UI_i`,` m`)` uses the USIG instance i's public key and verifies whether the certificate UI_i was computed for message m.

9.1.2 The MinBFT Protocol

The MinBFT protocol proceeds in a sequence of views. The primary for each view is replica r_i where $i = v \mod N$, v is the view number and N is the system size. The primary is responsible for handling client requests, assigning sequence number to those requests, and forwarding the requests to the replicas. The sequence numbers the primary assigns to requests is generated by the USIG service instance within the primary. The replicas accept

the request and execute it once they collect a commit certificate. A commit certificate indicates that a majority of replicas have observed the same message from the primary.

When the primary receives a client request m with operation o, it assigns a sequence number to the request. The sequence number is the one generated by the USIG service. The primary r_i sends the request in a message $\langle \texttt{Prepare}, v, r_i, m, UI_i \rangle$, where UI_i contains the unique sequence number and the signature obtained from the USIG module. Each replica r_j in turn sends the $\langle \texttt{Commit}, v, r_j, r_i, m, UI_i, UI_j \rangle$ message to all other replicas. A client request is accepted at a replica if it receives $f + 1$ valid `Commit` messages, called a *commit certificate*.

Correct replicas only responds to the primary's `Prepare` message if the following conditions hold: v is the current view number and the sender of the message is the primary of v; the USIG signature is valid; and that the messages are received in sequential order of the USIG counter value. To prevent a faulty replica from executing the same operation twice, each replica maintains a V_{req} to store the request identifier of the latest operation executed for each client. The messages are always processed in the order of the USIG sequence number to prevent duplicity of operations and holes in the sequence number space. Replicas only execute an operation if it has not been executed already.

The view change protocol is triggered if the current primary fails to make timely progress. Replica sends a $\langle \texttt{ReqViewChange}, r_i, v, v' \rangle$ message to other replicas if it times out waiting for messages from the primary. A replica moves into a new view if it receives $f+1$ `ReqViewChange` messages and consequently broadcasts a $\langle \texttt{ViewChange}, r_i, v', C_l, O, UI_i \rangle$ message. C_l is the last stable checkpoint certificate, O is the set of generated messages since the last checkpoint. The new primary computes and sends a $\langle \texttt{NewView}, r_i, v', V_{vc}, S, UI_i \rangle$, where V_{vc} is the *new view certificate* that contains the set of $f + 1$ `ViewChange` messages used to construct the new view and S is the set of prepared or committed requests since the last checkpoint. Replicas validate the received `NewView` message, update its sequence of operations to match S, executes the pending operations, and starts accepting messages in the new view v'.

For conciseness, we defer the explanation of the checkpoint and state transfer procedures to the original paper [135].

9.2 MinBFT with Flexible Quorums

In this section, we introduce the notion of Flexible Hybrid Quorums. First, we show that not all quorums need to intersect and consequently show that the system size N need not be a function of f. Specifically, we show that only quorums of different kinds must intersect. Thus, sizes of commit quorums Q_c can be reduced at the cost of increasing the sizes of the view change quorums Q_{vc}. We apply this technique to MinBFT and call the resulting protocol as *Flexible* MinBFT.

MinBFT uses simple majority quorums for both the commit and the new view certificates.

Thus, every quorum intersects with every other quorum. Consequently, commit quorums Q_c intersect with other commit quorums. However, this is excessive. In MinBFT, the replica at the intersection of any two commit quorums, ensures that an operation is assigned only one *UI* certificate. Note that the primary's USIG service already ensures that an *UI* is assigned only once. If the primary and the intersecting replica are malicious, the replica may still vote for the same operation at two different *UI*s. Correct replicas will handle this using the V_{req} data structure. Since they process the messages in *UI* order, they will observe that an operation has already been executed and not execute it again. This makes the intersection replica redundant. Thus, we relax the assumption that the different commit quorums intersect with each other. At the same time, to tolerate f failures, the commit quorums should consists of more than f replicas. Thus, we have that $|Q_c| > f$.

On the other hand, any view change quorum Q_{vc} must intersect with any commit and view change quorums to ensure that the decisions made within a view are safely transitioned to future views. Hence, we have that $|Q_{vc}| + |Q_c| > N$.

The Flexible Hybrid Quorum requirement is captured by the following equations:

$$|Q_c| > f \tag{9.1}$$

$$|Q_{vc}| + |Q_c| > N \tag{9.2}$$

By setting Q_c to the smallest possible value i.e. $|Q_c| = f + 1$, we can observe that $|Q_{vc}|$ should equal $N - f$, to satisfy Equation 9.2. Consequently, the system size N need not be a function of f.

We applied the Flexible Hybrid Quorums to MinBFT. The resulting algorithm, *Flexible* MinBFT remains largely the same, except for the quorums they use. First, the commit quorums size still remains the same $Q_c = f + 1$, except that the variable f, the number of tolerated faults, is independent of N, the system size. Second, the size of view change quorums Q_{vc} now equals $N - f$ instead of $f + 1$. The protocol does not require any other changes.

The safety and liveness guarantees provided by Flexible MinBFT are given below. We first present the intuition and the related lemmas, then proceed with a complete proof in Section 9.3.

Safety within a view.

This is ensured by the trusted subsystem and the commit quorums. The trusted subsystem ensures non-equivocation, so a Byzantine replica cannot send conflicting proposals. Correct replicas will only vote on the proposed operation if the proposal is valid and it has not voted for the same operation before. The following Lemma formalizes this notion.

Lemma 9.1. *In a view v, if a correct replica executes an operation o with sequence number i, no correct replica will execute o with sequence number $i' \neq i$.*

Safety across views

This is ensured by the trusted subsystem and the view change quorums. The intersection of the commit and the view change quorum consists of at least one replica. Thanks to the trusted component, the replica in the intersection cannot equivocate, and must reveal the correct sequence of operations executed, as otherwise there will be holes that correct replicas can detect. Thus, a correct primary will gather the correct sequence and apply it in the next view.

Lemma 9.2. *If a correct replica executes an operation o with sequence number i in a view v, no correct replica will execute o with sequence number $i' \neq i$ in any view $v' > v$.*

Quorum availability.

A non-faulty primary will always receive responses from a quorum of $f + 1$ replicas and this quorum will contain at least one honest replica.

Lemma 9.3. *During a stable view, an operation requested by a correct client completes.*

Lemma 9.4. *A view v eventually will be changed to a new view $v' > v$ if at least $N - f$ correct replicas request its change.*

9.3 Correctness of Flexible MinBFT

Note that our proof structure overlaps with MinBFT's original proof for ease of exposition.

Lemma 9.1. In a view v, if a correct replica executes an operation o with sequence number i, no correct replica will execute o with sequence number $i' \neq i$.

Proof. If r executes o with sequence number i, then it will have $f+1$ valid `Commit` messages for $\langle o, i \rangle$ from a quorum Q_c.

We prove by contradiction. Suppose another correct server r' execute o with sequence number $i' > i$. This can happen if s' received $f + 1$ valid `Commit` messages for $\langle o, i' \rangle$ from quorum Q'_c. Note that $\exists Q_c, Q'_c : Q_c \cap Q'_c = \emptyset$ i.e. any two $\mathcal{Q}_c$ quorums need not intersect. There are two cases to consider depending on the primary:

1. *Primary is correct:* This is trivial since a correct primary will not generate two UIs for the same operation o.

2. *Primary is faulty:* Lets say r sends $\langle \texttt{Commit}, v, r, s, UI_p, UI_r \rangle$ to s for $\langle o, i \rangle$ and say r' sends $\langle \texttt{Commit}, v, r', s', UI_p, UI_r \rangle$ to s for $\langle o, i' \rangle$.

   (a) s' executed some operation at i: Primary cannot generate two messages with the same sequence number. Thus, i must have been o. Since $o.seq \leq V_{req}[c]$, o will not be executed again.

   (b) s' did not execute i: Replicas can only execute in sequence number order. Since $i < i'$, s' must wait to execute i. Once its executes i, executing i' fall under previous case.

Thus, it is not possible for any two replicas to execute the same operation with different sequence number in view v. □

Lemma 9.2. If a correct replica executes an operation o with sequence number i in a view v, no correct replica will execute o with sequence number $i' \neq i$ in any view $v' > v$.

Proof. If a correct replica s executes o with sequence number i in view v it must have received at least $f + 1$ valid `Commit` messages for $\langle o, i, v \rangle$ from quorum Q_c replicas.

Proof by contradiction: Let us suppose that another correct replica s' executes o at sequence number $i' > i$ in $v' > v$, then s' would have received $f+1$ valid `Commit` messages for $\langle o, i', v' \rangle$ from quorum Q'_c replicas.

Note that we have that any two commit quorums need not intersect i.e. $Q_c \cap Q'_c = \emptyset$.

We first deal with the case where $v' = v + 1$ and generalize later for arbitrary values of $v' > v$.

Let p be the primary of new view v'. First, we show that the primary p in view v' cannot deny the fact that o was accepted/executed before v'. Then, we show that no correct replica will execute o with $i' \neq i$ in v' and prove the contradiction.

The `NewView` message sent by the new primary includes the new view certificate V_{vc} that contains $N-f$ `ViewChange` messages from a quorum Q_{vc} that contains of at least one correct replica, say $r \in Q_{vc}$ that will send the correct `ViewChange` message to the new primary. Given this, we consider the following cases:

1. *Primary is correct and replica $r \in Q_{vc}$ is correct*: If primary p is correct, it inserts $N-f$ `ViewChange` messages into V_{vc} including the one from r. There are two possibilities.

   (a) o was executed after checkpoint: r is correct, so O contains `Commit` that r sent for o, therefore V_{vc} and S in the `NewView` message assert O was executed explicitly.

   (b) o was executed before the checkpoint: The latest checkpoint C_l shared by r implies that o was executed implicitly. Since the V_{vc} sent in the `NewView` contains the C_l, it implies that o was executed.

2. the primary p is correct, but there is no correct replica in Q_{vc} that executed o: There should exist at least a faulty replica $r \in Q'_c$ that accepted o because $Q'_c \cap Q'_{vc} \neq \emptyset$ ($q_{cmt} + q_{vc} > 1$).

   (a) o was executed after checkpoint: r might be tempted to not include o's messages in O, but if it did, p being correct would not put r's `ViewChange` message in V_{vc} as p can detect its invalid. There are two possible ways a detection will happen: (i) if r executed a request o' after o, r might put the `Commit` message for o' but not o leaving a hole in the log that p will detect. (ii) if r sent `Commit` for o with a USIG value cv, it might leave out all commit after o with $cv' > cv$ from the O log. But, this log will also be considered invalid by p since r must sign the `ViewChange` message containing O before sending it. The USIG will sign with a $cv'' > cv + 1$ that will allow a correct p to detect an incomplete O. Thus, r must include all commits and Case 1 above will apply.

   (b) o was executed before the stable checkpoint: One way this can happen is when r includes an older checkpoint message but p will detect the invalid `ViewChange` message because the USIG value of the message will disclose that there are messages since the checkpoint message that r failed to disclose.

3. Primary is faulty but $r \in Q_{vc}$ is correct and executed o. In this case, the faulty primary p may attempt to modify the contents of O that it receives from r before inserting into V_{vc}. However, this will leave a hole and other correct replicas will detect this misbehavior since they run the same procedure the primary runs for computing the `NewView` message. If p removes o and all further operations after o, correct replicas can also detect it because the USIG value of the r's `ViewChange` message inside V_{vc} will indicate the missing messages (as in Case 2). Similarly, if the primary tries to add an older checkpoint certificate, correct replicas will detect it from the holes in the USIG values. Therefore, a fault primary cannot tamper with a `ViewChange` message without detection. Thus, Case 1 will happen.

4. Primary is faulty and no correct $r \in Q_{vc}$ has executed o. A faulty $r \in Q_{vc}$ may exist. Given $|Q_c| + |Q_{vc}| > N$, r cannot successfully convince the primary to behave as if it did not execute o. Even if the primary being faulty uses r's `ViewChange` message in the V_{vc}, other replicas will detect the missing sequence number and the corresponding commit message for o. Thus, we will fall back to Cases 2 and 3 above.

The above four cases show that p in view v' must assert O was executed before v' in the certificate V_{vc}. Now, we show no correct replicas will execute o with $i' \neq i$ in v'. There are two cases:

1. *Primary is correct:* A correct primary p will never generate a second USIG certificate for the same operation and correct replicas will not send a commit message o with i' in view v'.

2. *Primary is faulty:* It is possible for a faulty primary to create a new `Prepare` message for o and successfully create a new USIG $UI'_p = \langle i, H(o)\rangle_p$ and send it to a replica r. However, every replica maintains the V_{req} that holds the last executed operation identifier seq for each client. Thus, r will discover that o was already executed since $o.seq \leq V_{req}[c]$. Thus, o will not be executed again.

This proves that if a correct replica executed o at sequence number i in view v, then no correct replica will execute o at sequence number $i' \neq i$ in view $v' = v + 1$.

We now generalize for arbitrary values of $v' > v$. There are two cases:

1. *$v' = v + k$ but no request was accepted in view v'' such that $v' < v'' < v + k$:* This case is trivial and falls under the case of $v' = v + 1$ since only view change related messages are sent in v'' which mirrors the v to v' transition.

2. *$v' = v + k$ but requests were prepared/accepted in view v'' such that $v' < v'' < v + k$:* At each view change, replicas must propagate information about operations from one view to its consecutive view (e.g. v to $v + 1$, and so on). This is done either via the checkpoint certificate or the via the O log set. Thus, each transition becomes the case of $v' = v + 1$ above.

□

Theorem 9.5. *Let s be a correct replica that executed more operations of all correct replicas up to a certain instant. If s executed the sequence of operations $S = \langle o_1, ...o_i\rangle$, then all other correct replicas executed this same sequence of operations or a prefix of it.*

Proof. Let $prefix(S, k)$ be a function that gets the prefix of sequence S containing the first k operations, with $prefix(S, 0)$ being the empty sequence. Let $\bullet$ be an operation that concatenates sequences.

We prove by contradiction. Assume the theorem is false, i.e there exists a correct replica r' that executed some sequence of operations S' that is not a prefix of S. Let $prefix(S, i) = prefix(S', i-1) \bullet \langle o_i\rangle$ and $prefix(S', i) = prefix(S, i-1) \bullet \langle o'_i\rangle$ such that $o_i \neq o'_i$ In this case, o_i was executed as the ith operation by replica r and o'_i was executed as the ith operation by replica r'. Assume o_i was executed in view v and o'_i was executed in view v'. Setting $v = v'$ will contradict Lemma 9.1 and setting $v' \neq v$ will contradict Lemma 9.2.

□

Lemma 9.3. During a stable view, an operation requested by a correct client completes.

Proof. A correct client c will send an operation o with an identifier larger than any previous identifiers to the replicas. The primary p being correct, will construct a valid `Prepare`

message with a valid USIG certificate $UI_p = \langle i, H(o) \rangle_p$ and send it to all replicas. At least $f + 1$ correct replicas will validate the `Prepare` message, verify the UI, and send a corresponding `Commit` message. Since there can be only f faults, there should exist at least $N - f$ correct replicas, out of which $f + 1$ (q_{cmt}), should successfully produce these `Commit` messages. When a correct replica receives q_{cmt} valid `Commit` messages, o will be executed and replied to the client c. Since q_{cmt} correct replicas exist, a correct client will receive $f + 1$ same replies indicating that operation o was properly executed at sequence number i. □

Lemma 9.4. A view v eventually will be changed to a new view $v' > v$ if at least $N - f$ correct replicas request its change.

Proof. A correct replica r sends a $\langle ReqViewChange, r, v, v' \rangle$ message requesting a view change to all replicas. However, at least $N - f$ correct replicas must send such a message to actually trigger a view change. Say a set of $N - f$ correct replicas request a view change from v to $v + 1$ by sending the `ReqViewChange` message. The primary for the new view is $p = (v + 1) \mod N$. Consider the two cases:

1. *the new view is stable*: correct replicas will receive the `ReqViewChange` messages. Consequently, correct replicas that receive at least $N - f$ `ReqViewChange` messages will enter the new view v' and send a `ViewChange` message to all replicas. The primary p, being stable, for view $v + 1$ will send a valid `NewView` message in time. Thus, correct replica that receive the message will transition to new view $v' = v + 1$.

2. *the new view is not stable*: We consider two cases:

   (a) *the primary p is faulty and does not send the `NewView` message in time, or p is faulty and sends an invalid `NewView` message*, or p is not faulty but the network delays p's message indefinitely. In all these cases, the timer on other correct replicas that sent the `ViewChange` message will expire waiting for the new view message. These replicas will trigger another view change to view $v + 2$.

   (b) *the primary p is faulty and sends the NewView message to only a quorum Q_{vc} of $N - f$ replicas but less than $N - f$ replicas are correct, or p is correct but there are communication delays.* The replicas in quorum Q_{vc} may enter the new view and process requests in time. However, the correct replicas that does not receive the NewView message will timeout and request change to view $v + 2$. However, there will be less than $N - f$ replicas, so a successful view change trigger will not happen. If the faulty replicas deviate from the algorithm, other correct replicas will join to change the view.

□

Theorem 9.6. *An operation requested by a correct client eventually completes.*

Proof. The proof follows from the Lemmas 9.3 and 9.4.

When the view is stable, Lemma 9.3 shows that the client operations are properly committed. However, when the view v is not stable, there are two possibilities:

1. *at least $f+1$ replicas timeout waiting for messages and request a view change:* Lemma 9.4 handles this case and ensures that a stable view $v' > v$ is established.

2. *less than $f+1$ replicas request a view change:* There should exist at least a quorum Q of $f+1$ replicas that are in the current view v. As long as these replicas continue to follow the algorithm, they will continue to stay in view v and client requests will be committed in time. However, if the replicas are not timely, then the correct replica from Q in view v will send the `ReqViewChange` message. With this message, a successful view change is triggered and the previous case takes happens.

If the new view v' is not stable, another view change will be triggered depending on whether Cases 1 or 2 above holds. However, this process will not continue forever. Since there are only f Byzantine replicas and due to the assumption that the network delays do not grow indefinitely, eventually there should exist a view v'' that is stable such that the primary responds in a timely manner and follows the algorithm.

□

Chapter 10

Dual Fault Tolerance with DuoBFT

10.1 Introduction

This dissertation has thus far presented two protocol designs that leverage trusted execution environments (TEEs) in modern CPUs, namely Dester (Chapter 5) and Destiny (Chapter 8), and a technique, Bumblebee (Chapter 6), to construct high-performance and scalable protocols under the hybrid fault model. The use of TEEs raises some significant challenges. First, it greatly reduces the choice of hardware used to deploy such protocols. At the time of this writing, Intel SGX [52] and ARM TrustZone [123], were the only available capabilities for commodity processors. However, Byzantine protocols implicitly/explicitly require diversity in the deployment stack to reduce the number of correlated failures [61]. Second, security vulnerabilities have been discovered in trusted execution environments recently [131, 132]. Although active research in the area aims to solve these problems, the impact of undiscovered vulnerabilities raises concerns. In addition, the fact that there are only a few vendors for such TEEs exemplify the problem. This raises questions on their applicability to BFT and blockchain systems.

Despite the challenges, the hybrid fault model has some unique properties that can be appealing for certain applications. The model tolerates more faulty replicas in the partial synchrony model, that can only be matched under the synchronous timing model [9]. To cope with the challenges, we propose reducing the reliance on trusted execution environments, and further, offer mechanisms to fallback to a safer fault model during drastic situations.

10.1.1 Towards providing a dual fault model

This chapter proposes DuoBFT, a BFT protocol that solves the aforementioned problems by offering two unique features: Cheap Resilience and Dual Fault Assumption under the partially synchronous timing model.

Cheap Resilience. We introduce a technique that allows DuoBFT to make consensus decisions with only a quorum of little over 1/3 replicas. DuoBFT incorporates the Flexible Hybrid Quorums technique, discussed in Chapter 9, within a traditional BFT protocol and

enables hybrid quorums decisions with only 1/3 *hybrid* replicas[1]. There is no complete reliance on trusted execution environments since DuoBFT only needs a third of replicas to have the trusted hardware capability. In the common case, the fraction of *hybrid* replicas are enough to make commit decisions in two communication steps under the hybrid fault model. Alternatively, since DuoBFT is also a BFT protocol, commit decisions under the BFT model with a 2/3 quorum and three communication steps is also possible. This is useful when hybrid quorums are not possible or feasible.

Dual Learner Assumptions under the Partial-Synchrony model. DuoBFT exposes the dual fault model outside of the replicas by separating the commit rules from the protocol. This allows different learners to make commit decisions by choosing between the BFT and the hybrid fault models. In doing so, DuoBFT provides an unique trade-off to the learners: quick decisions made possible by hybrid replicas versus uncompromising resilience to malicious behavior. Furthermore, our solution allows different learners to adapt their fault assumptions dynamically without depending on the replicas.

By letting the learners dynamically choose the fault model, DuoBFT empowers them to tailor end-user experiences at a finer granularity, depending on the operation. For instance, in a payment system, a $10 transaction may require faster and cheaper guarantees than a $1M transaction [106]. Thus, learners processing the $10 transaction can take advantage of the faster response under the hybrid fault model to offer better end-user experience, while learners processing the $1M transaction can spend the additional communication steps to decide under the BFT model and provide stronger resilience guarantees to the end-user.

In DuoBFT, the replicas propose and vote on blocks that contain client transactions or operations, and the learners make commit decisions on the blocks depending on their adopted fault model. Replicas internally collect two types of quorums to help the learners make their commit decisions. The Hybrid commit quorum consists of votes from the hybrid replicas, and the BFT commit quorum consists of votes from any 2/3 replicas. Clients can collect either of these quorums to make the respective commit decisions. However, under certain system conditions, learners may need to update their assumption to continue to make proper commit decisions. The hybrid replicas may be unreachable and since there are only about 1/3 such replicas, learners adopting the hybrid model need to switch to the BFT model to make progress. Similarly, a BFT quorum may not attainable in time, possibly due to a network partition, in which case learners can temporarily adopt the hybrid model. DuoBFT's dual fault model provides such flexibility. Furthermore, if severe security vulnerabilities were discovered in the trusted execution environments, the learners can simply switch to using the BFT model for added safety while the vulnerabilities are being fixed.

Furthermore, the flexibility provided by the DuoBFT's dual fault model is better than speculation[64, 84], or tentative[40] execution capabilities provided by other known protocols. While speculation requires 50% larger quorums than PBFT [41] and the tentative execution only reduces the execution overhead by overlapping the last communication step with ex-

[1] *Hybrid* replicas are those that have trusted components

ecution, the hybrid model uses 50% smaller quorums than PBFT and reduces one overall communication step. At the same time, DuoBFT does not incur any more overall communication steps than that required to commit under the BFT model, unlike [64, 83].

This contribution of this chapter is **DuoBFT**, a protocol that provides Cheap Resilience and Dual Fault Assumption under the partial synchrony timing model. DuoBFT can make Hybrid commit decisions with 1/3 replicas and also make traditional BFT commit decisions. The protocol supports learners under two fault models – Hybrid and BFT – to provide a unique trade-off: decisions made possible by hybrid replicas versus uncompromising resilience to malicious behavior. We also show that DuoBFT provides the guarantees of both fault models within the same protocol.

By letting the learners to choose their commit rules, DuoBFT falls into a class of similar protocols that include Flexible BFT [106] and Bitcoin [118]. However, DuoBFT provides a completely different set of properties than these protocols. For instance, the trust-based assumption that DuoBFT adopts results in an efficient algorithm that lets learners decide quickly and with strong resilience. Furthermore, as we will discuss later, the a-b-c fault model adopted in Flexible BFT can, in fact, be seen as complementary to the guarantees provided by DuoBFT.

The rest of the chapter is organized as follows. Section 10.2 presents the terminology for DuoBFT. Section 10.3 presents the DuoBFT protocol, explanation of its properties along with proofs, and some optimizations. A discussion on some unique features of DuoBFT with respect to existing solutions is presented in Section 10.4. Section 10.5 evaluates the protocols. Section 10.6 concludes the paper.

10.2 Preliminaries

We consider a system that consists of a set of nodes, called *replicas* that communicate via message passing. These replicas implement a replicated service that receive requests from client and ensure that the same sequence of totally ordered requests is learnt by the client. The goal of the consensus protocol is to ensure agreement on the replicated state among replicas withstanding a number of faulty servers.

Per Lamport's terminology [95], a consensus protocol consists of three roles: *proposers* proposes values to add to the sequence, *acceptors* vote to add these values to the sequence, and *learners* learn the sequence of values and execute them. In a replicated service, the clients act as both proposers and learners, while the replicas act as the acceptors. A consensus protocol provides the following two guarantees:

- **Safety.** Any two learners learn the same sequence of values.

- **Liveness.** A value proposed by the proposer will eventually be executed by every

learner.

Most consensus protocols offer a single fault model that the learners use to learn the sequence of values. In contrast, DuoBFT offers a dual fault model and supports learners with fault assumptions. Thus, based on the assumptions, learners may learn values differently. To support learners with different assumptions, DuoBFT provides the following guarantees similar to [106]:

- **Safety.** Any two learners with correct but potentially different assumptions learn the same sequence of values.

- **Liveness.** A value proposed by the proposer will be eventually executed by every learner with a correct assumption.

Fault Model. DuoBFT supports two fault models: the Byzantine fault model and the hybrid fault model. In both the models, a replica is correct if it strictly follows the algorithm, otherwise it is faulty. In addition, faulty replicas can collude to cause harm to correct replicas.

In the Hybrid model, we assume the existence of a trusted execution environment in each *hybrid* replica that hosts the protocol's trusted code. We use the term *hybrid* replica to refer to those replicas that have the trusted component and participate in the hybrid fault model. In MinBFT, every replica is a hybrid replica, while in DuoBFT only a subset of replicas are hybrid ones. Despite some replicas being faulty, the trusted execution environment in each hybrid replica is assumed to be tamperproof and the code it executes strictly follows the algorithm. We will revisit this requirement later in Section 10.4. Lastly, the trusted component can fail by crashing.

Timing Model. We assume the partially synchronous timing model [38]. Eventually, there exists a time during which correct replicas communicate synchronously and messages are timely. We, further, assume that the network can drop, reorder, and duplicate messages. To ensure reliable delivery of messages, we rely on generic retransmission techniques that use a buffer to store outgoing messages and retransmits them periodically. Furthermore, we do not assume any bounds on processing and communication delays except that such delays do not grow indefinitely.

Cryptography. We assume that the adversary cannot break cryptographic computation such as hashes and signatures. In addition, the hashing algorithms are collision resistant. Every replica is aware of other replicas' public keys. Each replica can verify the messages they receive using the corresponding replica's public key.

10.3 Protocol Design

DuoBFT requires $N = 3f + 1$ replicas to tolerate f faults. Learners can choose to tolerate either f Byzantine faults or f hybrid faults by assuming the respective fault model. For ease of exposition, we use f_B to denote Byzantine faults and f_H to denote hybrid faults, but $f_B = f_H = f$. Learners with different assumptions can coexist at the same time and they can also choose to update their assumptions at any time. To tolerate hybrid faults, at least $f_H + 1$ replicas must have the USIG trusted subsystem. We call these *hybrid replicas*. Learners can commit in the hybrid fault model by collecting votes from hybrid replicas only. However, to commit in the BFT model, learners can collect votes from any quorum of replicas including hybrid replicas. This flexible scheme allows learners assuming the hybrid model to make progress even when a trusted quorum is not possible by simply adjusting their fault assumptions. This enables DuoBFT to use as few hybrid replicas as possible to reach a hybrid quorum.

In the rest of this section, we will present the DuoBFT, overview its properties, and discuss its guarantees. Similar to other recent works such as Casper [37], Hotstuff [140], and Flexible BFT [106], we explain DuoBFT in terms of a blockchain protocol where the votes are pipelined. First, we present the terminologies.

10.3.1 Preliminaries

Blockchain

As presented in the previous sections, in classical BFT protocols, the agreement happens on a sequence of client requests or more generally values. Similarly, in a blockchain protocol, the agreement happens on a chain of blocks. We use the term *block* to refer to a value in the chain (or sequence). Each block has a reference to a predecessor block except the first block in the chain, also called the genesis block, which has no predecessors. Every block has a height parameter that indicates its position from the genesis block. A block B_k at height k has the following format: $B_k := (b_k, h_{k-1})$ where b_k refers to the block's value and $h_{k-1} = H(B_{k-1})$, the hash of the predecessor block in the chain. For the genesis block, the predecessor hash is null, thus $B_1 := (b_1, \perp)$. Note that only the genesis block can have a null predecessor hash; other block must specify a valid hash.

Block Prefix and Equivocation

Let S be a sequence of blocks in increasing height order. The prefix of a Block B_k at height k in sequence S, denoted $prefix(S, k)$, is the prefix of the sequence S containing the first k blocks from the genesis block. Equivocation happens when the sequence of blocks diverges.

Given two blocks B_i and B_j, we say those blocks diverge when B_i is not the ancestor of B_j or vice versa.

Block Certificates

Replicas vote on the blocks by signing on the hash of the block $H(B_k)$. A quorum of these votes form a quorum certificate. In DuoBFT, replicas collect two kinds of certificates for a block: a hybrid quorum certificate $\mathcal{C}_H(B_k)$ and a BFT quorum certificate $\mathcal{C}_B(B_k)$.

10.3.2 Protocol

A DuoBFT replica executes the following protocol:

(1) **Propose.** The primary P creates a block and sends it in a $\langle \texttt{Propose}, v, P, B_k, s_P \rangle$ message to all replicas. It attests the block with a USIG identifier or a signature depending on its capability. The primary collects votes for the blocks from the replicas. The votes are either attested with a *UI* certificate or a signature. The primary sends the next block when it receives a quorum certificate for its previous block.

(2) **Vote.** A replica R receives the propose message $\langle \texttt{Propose}, v, P, B_k, s_P \rangle$ from the primary, validates if it extends the last proposed block, and votes for it. The vote is sent in a $\langle \texttt{Vote}, v, R, B_k, s_P, s_R \rangle$ message to other replicas. A *UI* certificate or a signature is attached depending on the replica.

In addition, Replica R records the following information for a block:

- $q_v(B_k)$: The votes received for block B_k by replica R from any other replica in view v.
- $\mathcal{C}_H(B_k)$: A set of $f + 1$ votes with *UI* certificates form a hybrid quorum certificate for block B_k.
- $\mathcal{C}_B(B_k)$: A set of $2f + 1$ votes form a BFT quorum certificate for block B_k.

Figure 10.1: DuoBFT Normal Protocol.

The DuoBFT protocol proceeds in a view by view fashion. For simplicity of prose, the primary of each view is decided using the formula $v \mod N$ i.e. primary roles are assigned to replicas in round-robin order. The primary of the view is responsible for proposing blocks that other replicas vote on. While the primary can be chosen using any known techniques [63], care should be taken to ensure that hybrid replicas are preferred to enable commit

under both fault models. If the primary is not a hybrid replica, hybrid commits are not possible. We discuss methods for primary selection in Section 10.4.

(1) **View Change Request** A replica requests a view change if it does not receive proposals from the replicas in a timely manner, or if it observes equivocating blocks either via the proposal or the vote messages.

- Replica R sends a $\langle$`ReqViewChange`, $R, v, v'\rangle$ to request a view change from v to $v' = v + 1$.
- A replica that receives $f + 1$ `ReqViewChange` messages transitions to the new view and multicasts the `ViewChange` message to other replicas.
- A replica that receives $f + 1$ `ViewChange` message also transitions to the new view and sends its `ViewChange` message to other replicas. The View Change message consists of all the blocks that the replicas have a quorum certificate for.

(2) **New View.** The new primary P' collects $2f+1$ `ViewChange` messages and computes the sequence of blocks in the new view v'. It sends a $\langle$`NewView`$\rangle$ message to all replicas.

(3) **New View Install.** A replica that receives the new primary P''s `NewView` messages, validates it, and installs S, the block sequence in the new view.

Figure 10.2: DuoBFT View Change Protocol.

At a high level, DuoBFT works as follows. The primary proposes a block to replicas. Replicas vote on the block if it is safe to do so. A quorum of such votes on a block make a quorum certificate. After collecting a quorum certificate for a block, the primary moves on to propose the next block extending the previous one. We will discuss how the client commits later since they are the ones that make the actual commit decisions. Replicas use the view change protocol to install a new view if they are unable to make progress in the current view. The view change protocol begin only if $f + 1$ replicas request a view change.

Normal protocol

The normal protocol is executed when a view is stable. In a stable view v, the primary and $N - f$ replicas behave correctly and exchange messages in a timely manner. The primary creates a new block B_k that extends the highest block in the chain it is aware of. If the primary is a hybrid replica i.e. it hosts the USIG component, it creates an UI certificate for the block. Otherwise, it simply signs it. The primary P sends the block to the replicas in

a $\langle$`Propose`, $v, P, B_k, s\rangle$ message, where v is the current view and s is either the UI or the signature.

A replica R that receives the `Propose` message for block B_k, votes on the block if it extends the previously proposed block in the view. Similar to MinBFT, DuoBFT replicas only process blocks in increasing height order. When blocks are received out of order, replicas wait to receive the predecessor blocks to B_k and validates the blocks, casts its vote in the height order. If R is a hybrid replica, it creates an UI certificate for its vote; otherwise, it simply signs the vote. Replica R sends its vote in a $\langle$`Vote`, $v, R, B_k, s_P, s_R\rangle$ message to other replicas.

The votes collected by a replica can form up to two kinds of quorum certificates for the block B_k depending on whether or not the primary is a hybrid replica. Every replica records the following information for a block:

- $\mathcal{C}_H(B_k)$: A set of $f+1$ votes with UI certificates form a hybrid quorum certificate for block B_k. Note that this certificate can be collected only if the primary is a hybrid replica.

- $\mathcal{C}_B(B_k)$: A set of $2f+1$ votes form a BFT quorum certificate for block B_k. Since there are $f+1$ hybrid replicas, there will be at least one hybrid replica in any BFT quorum and that replica will produce a UI certificate. Thus, the BFT quorum certificate will contain both UIs and signatures.

View Change

If a replica detects equivocation or lack of progress by the primary, it will start the view change procedure to move from the current view v to the next stable view v'. A replica requests a view change by sending a $\langle$`ReqViewChange`, $R, v, v'\rangle$ message to other replicas. When a replica receives at least $f+1$ `ReqViewChange` messages, it starts the view transition and sends a $\langle$`ViewChange`, $R, v', O, UI_i\rangle$ message, where O contains the sequence of blocks for which R has collected any quorum certificate. The primary P' of the new view v' will collect $2f+1$ valid `ViewChange` messages to form the new view certificate. P' will use this certificate to compute the set of blocks S for which quorum certificates exist. P' sends a $\langle$`NewView`, $r_i, v', V_{vc}, S, UI_i\rangle$, where V_{vc} is the *new view certificate* that contains the set of $f+1$ `ViewChange` messages used to construct the new view and S is the set of prepared or committed requests. Replicas verify the validity of the S by performing the same computation as the new primary using the new view certificate. Then, replicas adjust their local state according to S and start voting in the new view v'.

The new view computation performed by the replicas and the primary is similar to the one presented in PBFT [40]. We omit the details for conciseness.

1. **Hybrid Commit Rule:** A learner assuming the Hybrid Commit Rule commits a block B_k iff it collects Hybrid quorum certificate for block B_k i.e. $\mathcal{C}_H(B_k)$.
2. **BFT Commit Rule:** A learner assuming the BFT Commit Rule commits a block B_k iff it collects BFT quorum certificates for the blocks B_k and B_{k+1} i.e. $\mathcal{C}_B(B_k)$ and $\mathcal{C}_B(B_{k+1})$, and B_{k+1} extends B_k.

Figure 10.3: DuoBFT Commit Rules.

10.3.3 Learners

DuoBFT allows different learners to decide differently on the commitment status of the block based on how they perceive the system. The protocol supports two kinds of commit rules depending on whether the learner chooses the Byzantine fault model or the hybrid fault model.

Hybrid Commit Rule

A learner can choose to commit blocks adopting the hybrid fault model. In this case, a learner commits a block when it receives at least $f + 1$ votes from hybrid replicas. Note that the number of hybrid replicas can be capped at $f + 1$ for a system with $N = 3f + 1$ total replicas. Moreover, note that a learner can only decide to commit on a block only if there are enough hybrid replicas in the system. Otherwise, such a learner will never terminate and must recalibrate its assumptions and use the Byzantine fault model instead.

Under the Hybrid Commit Rule, the protocol provides the same safety guarantees as Flexible MinBFT.

BFT Commit Rule

Alternatively, a learner can choose to commit blocks adopting the Byzantine fault model. In such a case, the learner commits a block when it receives at least $2f + 1$ votes for the block B_k and the consecutive block.

Under the BFT Commit Rule, the protocol provides the same safety guarantees as PBFT.

10.3.4 Proof

Lemma 10.1. *If a learner commits a block B_l in a view v, then no learner with the same assumptions will commit B'_l that does not equal B_l in view v.*

Proof. **BFT Commit Rule:** We prove by contradiction. Say a learner commits block B_l. It will have q_c votes for B_l and its immediate successor. Suppose another learner commits block B'_l then it will have q_c votes for B'_l and its immediate successor. However, the intersection of two q_c quorums will have at least one correct replica that will not vote for two blocks at the same height. This is a contradiction.

Hybrid Commit Rule: We prove by contradiction. Say a learner commits block B_l. It will have $f+1$ USIG votes for B_l. Suppose another learner commits block B'_l then it will have $f+1$ USIG votes for B'_l. However, a primary cannot sign two messages with the same USIG identifier. Thus, there is no way there can exist two blocks at the same height l. This is a contradiction.

Thus, it is not possible for any two learners to commit different blocks at the same height in view v. □

Lemma 10.2. *If a learner commits a block B_l in a view v, no learner with the same assumptions will commit block B'_l that does not equal B_l at the same height l in any view $v' > v$.*

Proof. **BFT Commit Rule:** We prove by contradiction. A learner commits block B_l in view v then it will have q_c votes for B_l and its immediate successor. Suppose another learner commits block B'_l in view $v' > v$ then it should have q_c votes for B'_l and its immediate successor.

Since B_l is committed in view v, there should exist quorum certificates for blocks B_l and B_{l+1}. In the new view $v' = v+1$, the `NewView` message sent by the new primary includes a `NewView` certificate with q_r `ViewChange` messages that contains at least one correct replica. The `ViewChange` message from the correct replica will have the correct certificate for B_l and B_{l+1} Thus, the new primary must enforce blocks B_l and B_{l+1} in the new view $v+1$. It will receive votes only for B_l in the new view $v+1$. For B'_l to be committed, q_c replicas must votes for it, which cannot happen since there is at least one correct replica in the intersection of q_r and q_c that received the NewView message with correct certificates. This is a contradiction. Thus, B'_l cannot have been committed in $v+1$.

Hybrid Commit Rule: We prove by contradiction. A learner commits block B_l in view v then it will have $f+1$ votes for B_l. Suppose another learner commits block B'_l in view $v' > v$ then it should have $f+1$ votes for B'_l.

Since B_l is committed in view v, there should exist a USIG quorum certificate for block B_l. In the new view $v' = v+1$, the `NewView` message sent by the new primary includes a new

view certificate with $N - f$ `ViewChange` messages that contains at least one correct replica. The correct replica's `ViewChange` message will have the correct quorum certificate for B_l. It might happen that the new primary might remove some block entries from the `ViewChange` message, but this will be detected as the USIG-signed `NewView` message will reveal the holes in the message log (See Lemma 9.4 for additional details.) Thus, the new primary must enforce blocks B_l and B_{l+1} in the new view $v + 1$. It will receive votes only for B_l in the new view $v + 1$. For B'_l to be committed, q_c replicas must votes for it, which cannot happen since there is at least one correct replica in the intersection of q_r and q_c that received the NewView message with correct certificates. This is a contradiction. Thus, B'_l cannot have been committed in $v + 1$.

For both commit rules above, the case for arbitrary $v' > v$ where $v' = v + k$ will fall under the case of $v + 1$, since at each view transition, the information from one view is propagated to the next view. □

Theorem 10.3. *Any two learners with the same commit rule commit the same sequence of blocks in the same order.*

Proof. To elaborate on the theorem, if a learner following a commit rule commits the sequence of blocks $S = \langle B_1 ... B_i \rangle$, then another learner that follows the same commit rule will commit the same sequence of blocks S or a prefix of it. We use $prefix(S, i)$ to represent the first i blocks of the sequence S. We use the $\bullet$ operator to concatenate any two sequences.

Assume the theorem is false i.e. there should exist two sequences S and S' committed by two learners that is not a prefix of each other. Assume the sequences conflict at i, such that $prefix(S, i) = prefix(S', i - 1) \bullet \langle B_i \rangle$ and $prefix(S', i) = prefix(S, i - 1) \bullet \langle B'_i \rangle$. Precisely, there exists two blocks B_i and B'_i at the same height i committed by two different learners with the same commit rule. Assume that block B_i was committed in view v and block B'_i was committed in view v'. If $v = v'$, then this will contradict Lemma 10.1. If $v' > v'$, then this will contradict Lemma 10.2. Hence, the theorem must hold. □

Lemma 10.4. *During a stable view, a proposed block is committed by a learner.*

Proof. In a stable view, the correct primary will propose blocks in a timely fashion. If the primary is hybrid, then it will generate an $UI = \langle i, H(b) \rangle_p$ for the block. Correct replicas that receive the proposal will vote for the it. Replicas that are hybrid will generate an UI for their votes. Since there are at most f faulty replicas, they will remain $N - f$ correct ones. For a hybrid quorum, at least $f + 1$ of these $N - f$ replicas will reply on time. Similarly, for a BFT quorum $N - f = 2f + 1$ replicas will reply on time. Thus, a learner will receive the votes on time and will commit the block using their commit rule. □

Lemma 10.5. *A view v will eventually transitioned to a new view $v' > v$ if at least $N - f$ replicas request for it.*

Proof. A replica R can request a view change by sending a $\langle \texttt{ReqViewChange}, R, v, v' \rangle$ message. The view change mechanism is triggered when replicas receive $f + 1$ ReqViewChange messages for the same view. Assume that replicas collect $f + 1$ messages for transitioning from v to $v + 1$. The primary for the new view is $(v + 1) \mod N$ by definition. Consider the two cases:

1. *the new view is stable*: correct replicas will receive the `ReqViewChange` messages. Consequently, correct replicas that receive at least $f + 1$ `ReqViewChange` messages will enter the new view v' and send a `ViewChange` message to all replicas. The primary p, being stable, for view $v + 1$ will send a valid `NewView` message in time. Thus, correct replica that receive the message will transition to new view $v' = v + 1$.

2. *the new view is not stable*: We consider two cases:

   (a) *the primary p is faulty and does not send the `NewView` message in time, or p is faulty and sends an invalid `NewView` message*, or p is not faulty but the network delays p's message indefinitely. In all these cases, the timer on other correct replicas that sent the `ViewChange` message will expire waiting for the NewView message. These replicas will trigger another view change to view $v + 2$.

   (b) *the primary p is faulty and sends the NewView message to only a quorum Q' of q_{vc} replicas but less than q_{vc} replicas are correct, or p is correct but there are communication delays.* The replicas in quorum Q' may enter the new view and process requests in time. However, the correct replicas that does not receive the NewView message will timeout and request change to view $v + 2$. However, there will be less than $f + 1$ replicas, so a successful view change trigger will not happen. If the faulty replicas deviate from the algorithm, other correct replicas will join to change the view.

□

Theorem 10.6. *A proposed block is eventually learnt by learners with correct commit rules.*

Proof. When the view is stable, Lemma 10.4 shows that the proposed block is learnt by the learners. When the view is not stable and the replica timers expire properly, $f + 1$ replicas will request a view change. By Lemma 10.5, a new view v' will be installed.

However, if less than $f + 1$ replicas request the view change, then the remaining replicas that do not request the view change will follow the protocol properly. Thus, the system will stay in view v and the learners will continue to commit blocks in the view. When proposals are not committed in time or when more than f replicas request a view change, then all correct replicas will request a view change and it will be processed as in Lemma 10.4.

Even after a view change, the new view v' may not necessarily be stable. If the new primary deviates from the algorithm or does not process messages in time, this will cause correct

replicas to request another view change and move to the next view. Since there can only be at most f faulty replicas, after at most $f + 1$ view changes, a stable view will be installed. Furthermore, if the faulty primary follows the algorithm enough such that a view change cannot be triggered, by Lemma 10.4, learners will continue to commit the blocks.

□

10.3.5 Optimizations

Reducing Message Complexity. Our presentation of DuoBFT closely follows the message pattern of the signature-based PBFT protocol [40]. Therefore, we assume that the replicas multicast their votes to all other replicas incurring a $O(n^2)$ normal case message complexity. The complexity can be reduced to $O(n)$ by modifying the replicas to send their votes only to the primary, and enabling the primary to collect the votes and share the quorum certificate with the replicas. The primary can also piggyback the quorum certificate along with the proposal for the next block. This is a known technique adopted by many existing protocols [37, 106].

Choice of Primary Replica. The hybrid commit rule requires that blocks are proposed and voted for by hybrid replicas. Since there are only $f + 1$ hybrid replicas out of $3f + 1$ replicas, the round-robin primary assignment can install a non-hybrid replica as the primary. In this case, commits under the hybrid commit rule is no longer possible until another view change is triggered and a hybrid primary replica is installed. To favor hybrid replicas, we propose using a blacklisting mechanism similar to the one presented in [133]. Every replica maintains a blacklist that initially consists of all non-hybrid replicas. The blacklist has a maximum size of $N - f$ and the items are arranged in FIFO order. When an item is added to a full blacklist, it removes the oldest item from it. When a correct replica requests a view change, it will choose the next view such that the new primary is not in the blacklist. However, to ensure progress at times when hybrid replicas cannot establish a stable view, we remove replicas from the blacklist. When a view change happens, the old primary is added to the blacklist and the least recently added replica is removed from the blacklist.

10.4 Discussion

Implications of Trusted Environment Compromises. As mentioned in the introduction, trusted execution environments are increasingly being scrutinized for security vulnerabilities. In the hybrid fault model, the compromise of the trusted component is enough to break the safety of the protocol. However, DuoBFT holds safety in such cases. Note that there are only $f + 1$ hybrid replicas. Even if the trusted component is compromised, per our assumption, this can only affect at most f hybrid replicas. Thus, the remaining $2f + 1$ replicas will follow the algorithm correctly. While the hybrid quorum certificates can become invalid, recall that

replicas also collect BFT quorum certificates. Thus, safety is still preserved for the sequence of blocks that have collected the BFT quorum certificates.

Comparison to FlexibleBFT. We now highlight the differences of our protocol from Flexible BFT [106], a recent protocol that provides diverse learner assumptions. The first important distinction is that Flexible BFT provides the a-b-c fault model in addition to the BFT model. The replicas under the a-b-c model are allowed to attack the safety of the system, but when they aren't able to attach safety, they will ensure liveness. However, the implication of using this fault model is that Flexible BFT quorums are much larger than our flexible hybrid quorums. In Flexible BFT, the commit quorums used by the client q_c should be at least as large as the view change quorum q_r used by the replicas, i.e $q_c \geq q_r$. In contrast, DuoBFT uses hybrid commit quorums that are smaller than the view change quorums, and the BFT commit quorums are as large as the view change quorums. That is, in DuoBFT, $q_c \leq q_{vc}$.

Furthermore, Flexible BFT uses the synchrony timing model as a means to provide commits using simple majority quorums, which are smaller than Byzantine quorums. The protocol also tolerate $< 1/2$ failures. On the other hand, DuoBFT tolerates only $1/3$ failures, but under the hybrid model, its commit quorum sizes are really efficient, only a little over $1/3$ replicas. Thus, Flexible BFT uses the timing model to reduce quorum sizes, while DuoBFT uses the trusted component to achieve a very similar purpose. Furthermore, partially synchrony model enables "network-speed" replicas those that do not need lock-step executions unlike in the synchrony model. Thus, assumptions such as globally synchronized clocks are not required in our case.

10.5 Evaluation

In this section, we evaluate DuoBFT to answer the following questions: (1) What is the performance of DuoBFT compared to state-of-the-art protocols? (2) What is the overhead incurred by DuoBFT for providing the dual fault model?

We compared Flexible MinBFT (Chapter 9) and DuoBFT with MinBFT [135], PBFT [41], and Hotstuff [140]. MinBFT and PBFT serve as the baselines for assessing the performance of Flexible MinBFT and DuoBFT. Hotstuff is a recent state-of-the-art BFT protocol. We omit Flexible BFT because its overall quorum size is larger than that of general BFT protocols, and thus tend to perform poorly than PBFT.

We implemented each of the protocols in Go under a common framework. The USIG trusted component was implemented in C using the Intel SGX SDK and interfaced with the Go application using the cgo interface. The signatures were computed using the ECDSA algorithm with secp256r1 curves, while the hashes were computed using SHA256. We evaluated the protocols using a key-value store benchmark because it serves as a good abstraction for building higher level systems [64]. The payload size is set at 256 bytes.

Deployment. We deployed the protocols using the SGX-enabled DC8v2 virtual machines (8 vCPUs and 32GB of memory) provided by the Microsoft Azure cloud platform. We utilized 49 virtual machines evenly spread across seven different geographical regions. The regions were East US, Canada Central, Canada East, UK South, North Europe, West Europe and South East Asia. With 49 nodes, PBFT and Hotstuff each can tolerate 16 failures; and MinBFT can tolerate 24 failures. For Flexible MinBFT, we set $f = 16$. Consequently, DuoBFT can tolerate 16 hybrid and/or BFT failures. We used up to 1800 clients collocated with the replicas to inject requests into the system. The leader role was assigned to a replica in the US East region.

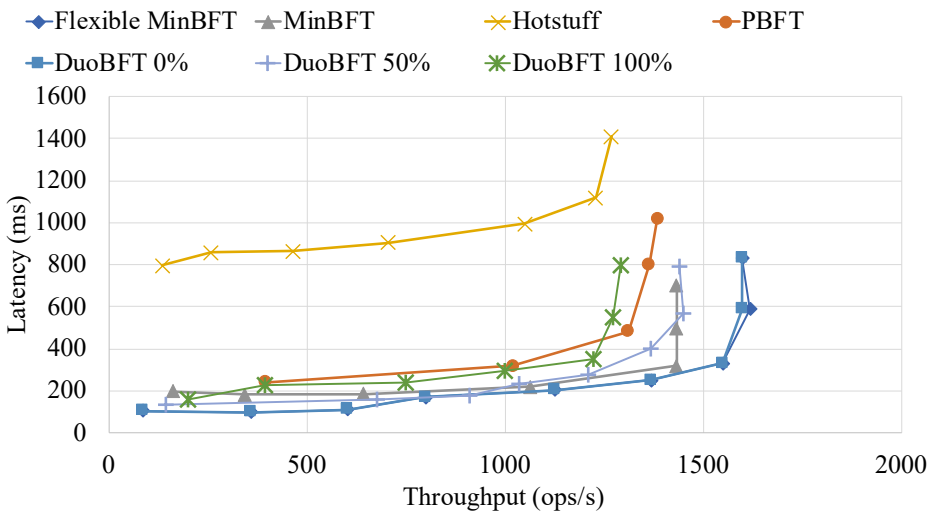


Figure 10.4: Latency vs. Throughput with Batch Size of 1.

By injecting requests using the clients in a closed-loop, we measured the throughput and latency of each of the protocols. Figure 10.4 presents the performance of each of the protocols when requests are processed without batching. We evaluated DuoBFT by varying the ratio of hybrid and BFT commits. The suffix in the legend indicates the number of BFT commits. Due to about 33% smaller quorums than MinBFT, Flexible MinBFT provides about 15% higher throughput and between 30-40% lower latency than MinBFT. Hotstuff and PBFT provide very similar throughput, but Hotstuff incurs higher latencies due to its linear message pattern and an additional communication round in addition to the two rounds in PBFT.

DuoBFT shadows Flexible MinBFT's performance when all of its requests are committed by collecting a hybrid quorum. Similarly, when all the requests are committed under the BFT quorum, the performance is similar to PBFT. Notice that DuoBFT takes a slight hit in performance due to the overhead of collecting both the BFT and Hybrid quorums. We observed the overhead to be between 4-9%. Furthermore, when half the requests are committed under the hybrid commit rule and the other half are committed under the BFT commit rule, DuoBFT performs similar to classic MinBFT.

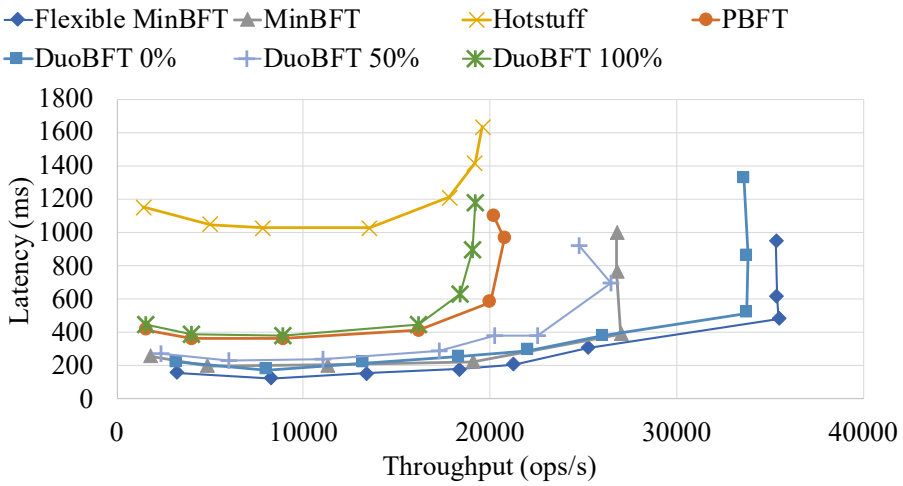


Figure 10.5: Latency vs. Throughput with Batch Size of 64.

Figure 10.5 presents the performance of each of the protocols when requests are processed in batches of 64. Flexible MinBFT provides 30% higher throughput and 20-30% lower latency than MinBFT. The overhead of DuoBFT is around 5-8% over the vanilla Flexible MinBFT and PBFT protocols. DuoBFT provides 8% lower throughput than Flexible MinBFT, and around 5% lower throughput than PBFT. In terms of latencies, DuoBFT with all hybrid commit requests provides around 5% lower latencies than MinBFT, while DuoBFT with all BFT commit requests incurs about 10% higher latencies than PBFT.

We conclude that DuoBFT provides the best of both Flexible MinBFT and PBFT protocols with very minimal overhead. Moreover, for systems that adopt a complete hybrid model, Flexible MinBFT provides the ability to set the number of failures independently from the system size thereby improving the performance of the system under normal cases.

10.6 Conclusions

This chapter presents DuoBFT, a protocol that provides two fault models under the partial synchrony setting and lets learners make commit decisions under either of them. We show that a fraction of hybrid replicas is enough to enable commit decisions under the hybrid fault model and still preserve safety guarantees. Furthermore, the incorporation of the flexible hybrid quorum technique is what makes the common view change protocol that supports both the fault models feasible. This leads to a simplified protocol that is capable of supporting two different fault models.

Chapter 11

Byzantine Ordered Consensus with Minimal Communication

The Byzantine Ordered Consensus [141] model incorporates the notion of ordering guarantees into consensus protocols. This prevents Byzantine replicas from unfairly delaying the consensus procedure of certain subset of commands over others. Pompe, the first protocol under this fault model, enforces the model by proposing a two phase architecture [141]. By separating the order assignment from the consensus phase, Pompe uses a pre-consensus phase that replicas use to agree on a total order timestamp to assign to any command seen by replicas. The consensus procedure is then simply tasked with agreeing on a set of commands whose timestamps are within a predetermined range such that older timestamps are not mistakenly added causing inconsistencies during monotonic execution based on increasing timestamps.

While Pompe achieves its goal of providing a protocol under the Byzantine Ordered Consensus model, the proposed solution incurs a higher overhead than a traditional BFT protocol. Particularly, the pre-consensus phase incurs at least six communication steps causing a significant increase in latency. Furthermore, the consensus phase should be carried out between carefully tuned intervals to ensure that a sufficiently large batch of commands are proposed in order to offset the throughput loss as a result of additional computation.

This chapter presents, Folly, a new Byzantine ordered consensus protocol, that overcomes the aforementioned drawbacks. Specifically, Folly's pre-consensus phase is only composed of three communication steps rather than six steps. Folly is able to achieve this gain using two techniques. First, the requirement that correct replicas must agree on the timestamps assigned to commands before consensus is relaxed. Instead, this decision is inferred from the result of the consensus phase. Second, the replicas are responsible for delivering the timestamped commands to the leader in time and also ensuring that the leader includes the commands sent by the correct replicas in the consensus procedure.

We implemented Folly in Golang and compared it against Pompe. We deployed four replicas of each of the protocols across three datacenters of the Azure compute infrastructure. We used Hotstuff [140] for the consensus phase in both Folly and Pompe, and measured the latency and throughput of the protocols. Our experimental results show that Folly is able to reduce latencies by $30 - 35\%$ compared to Pompe. Furthermore, our solution provides throughput similar to Pompe.

The main contribution of the chapter is Folly, a new Byzantine ordered consensus protocol that provides significant latency savings over the state-of-the-art protocol while maintaining the throughput. The rest of the paper is organized as follows. Section 11.1 provides a background on the motivation behind and guarantees of the Byzantine Ordered Consensus mechanism. Section 11.2 provides an overview of Folly's features. Section 11.3 discusses the protocol in depth and assess its correctness properties. Section 11.4 evaluates Section 11.5 concludes the paper.

11.1 Preliminaries

11.1.1 Ordering Guarantees in BFT protocols

The specification of Byzantine consensus does not enforce any ordering constraints among commands but only requires that commands are consistently ordered among replicas.

Impact of Leader. BFT consensus protocols typically use a specialized leader role that is responsible for proposing the order of the commands. However, a Byzantine leader is free to choose the next set of commands to order. Thus, such a leader can favor some commands over others, yet avoid detection by correct replicas.

Rotating Leader. Some protocols [133] propose changing the primary after each consensus instance in order to minimize any negative impact of Byzantine replicas. However, these protocols do not guarantee any ordering properties among commands submitted to the system.

11.1.2 Byzantine Ordered Consensus

The principle idea behind the Byzantine Ordered Consensus is to allow correct replicas to indicate their ordering preferences and expect that it will be respected by other correct replicas despite actions from Byzantine replicas to sway the preferences of correct nodes. The Byzantine Ordered Consensus model proposes three notions to capture the ordering guarantees that any protocol can have under the presence of Byzantine nodes.

Byzantine Oligarchy. A system is subject to Byzantine Oligarchy if it the actions of the Byzantine replicas can determine the ordering of any two commands c_1 and c_2 despite collecting ordering timestamps from correct replicas at all times.

Free will. A system is subject to Free will if all the commands proposed by replicas are eventually ordered, and that given any two correct replicas, it is possible that the ordering of the commands in one correct replica is different from the ordering of the same commands in a different replica. That is, given two commands c_1 and c_2, there can exist a trace where

one replica can orders c_1 before c_2, and another trace for another replica that orders c_2 before c_1.

Byzantine Democracy. A system is subject to a Byzantine Democracy if the actions of the Byzantine replica can determine the ordering of any two command c_1 and c_2 despite collecting ordering timestamps from correct replicas during certain times. Note that the important difference between a Byzantine Oligarchy and a Byzantine Democracy is that the under the former, commands are influenced by Byzantine replicas *all* of the time whereas under the later they are influenced *only some* of the time.

Give the above notions, the Byzantine Ordered Consensus model concludes that under Free will, it is impossible to curb the impacts of Byzantine replicas from swaying the ordering proposed by correct replicas.

To allow correct replicas to propose an ordering despite actions by Byzantine replicas, the Byzantine Ordered Consensus model proposes an ordering property, called *Ordering linearizability*. The Ordering linearizability property respects the ordering preferences from correct nodes and enforces an happens-before relationship among the timestamps collected from those nodes.

A Byzantine Ordered Consensus protocol offers the following properties [141]:

1. Consistency. Any two nodes agree on the same sequence of commands to execute.
2. Validity. If a correct node appends a command c to its log, then it has been previously proposed by some node in the ordering phase.
3. Ordering Linearizability. If the highest timestamp provided by any correct node for a command c_1 is lower than the lowest timestamp provided by any correct node for another command c_2 and if both c_1 and c_2 are committed, then c_1 will appear before c_2 in the totally-ordered ledgers constructed by correct nodes.
4. Termination. During sufficiently long periods of synchrony, a command c will eventually be included in the value committed by a unique consensus slot whose time interval includes ts.

11.1.3 Pompe

Pompe [141] is a protocol that enforces the Byzantine Ordered Consensus model by allowing replicas to propose ordering timestamps to commands and ensures ordering linearizability. Pompe achieves its goals by separating the ordering mechanism from the consensus mechanism and adopting a two-phase protocol. The first phase is responsible for assigning ordering timestamps to commands, while the second phase periodically accumulates timestamped commands and runs the consensus protocol.

Ordering Phase. The ordering phase takes two communication round-trips and begins when a replica wants to propose a command and assign a timestamp to it.

The first roundtrip involves the replica proposing a command c to all replicas and requesting a timestamp for it. Every replica assigns the next available monotonically increasing local timestamp to c. The proposing replica receives a supermajority $2f+1$ timestamps and picks the median among them as the ordering timestamp. Since there are f Byzantine replicas, choosing the median of the timestamps alleviates the impact that Byzantine replicas may have on the ordering.

The second roundtrip involves the ensuring that the median timestamp chosen for c is valid and will be included in the next consensus proposal. For this purpose, the proposing replica broadcasts the median timestamp ts assigned to c in the previous round and asks at least $2f+1$ replicas to accept it. Replicas will accept if the consensus slot for the time interval that includes timestamp ts has not begun yet. Otherwise, replicas will reject the timestamp.

Consensus Phase. The consensus phase in Pompe is responsible for collecting all the commands for a given time interval and proposing them using an off-the-shelf consensus protocol. Each slot in the consensus protocol orders commands whose timestamps belong to a predefined timestamp interval. The leader collects the commands within those timestamps from the replicas. Each replica waits for a certain duration before sending the commands it is aware of to the leader. The wait duration gives replicas sufficient time to ensure that all their commands are shared with the leader before the start of the consensus procedure.

Thus, in total, the pre-consensus step takes three communication roundtrips. This results is higher latencies for the ordering procedure that consequently impacts the end-to-end client perceived latencies.

11.2 Overview

In this section, we overview the key ideas behind Folly that allows it to assign ordering timestamps to commands in three communication steps and adds one additional communication step to ensure that the leader orders its commands. Thus, Folly, in total, saves one communication roundtrip from client-side latency computation compared to Pompe. First, we describe our system model.

11.2.1 System Model

We consider a system consisting of a set replicas that communicate via message passing. Each of these replicas implement the SMR specification; that is, they provide replicated service that receive requests from client and ensure that the same sequence of totally ordered requests is learnt by the client. The goal of the consensus protocol is to ensure agreement on

the replicated state among replicas withstanding a number of faulty servers. Furthermore, the specification of consensus we follow is Byzantine Ordered Consensus described in the previous section.

Under the fault model, a replica is correct if it strictly follows the algorithm, otherwise it is faulty. In addition, faulty replicas can collude to cause harm to correct replicas. The number of faulty replicas is bounded by f out of $N = 3f + 1$ total replicas in the system.

We assume the partially synchronous timing model. Eventually, there exists a time during which correct replicas communicate synchronously and messages are timely. We, further, assume that the network can drop, reorder, and duplicate messages. To ensure reliable delivery of messages, we rely on generic retransmission techniques that use a buffer to store outgoing messages and retransmits them periodically. Furthermore, we do not assume any bounds on processing and communication delays except that such delays do not grow indefinitely.

We assume that the adversary cannot break cryptographic computation such as hashes and signatures. In addition, the hashing algorithms are collision resistant. Every replica is aware of other replicas' public keys. Each replica can verify the messages they receive using the corresponding replica's public key.

11.2.2 Key Techniques

This section overviews Folly. We show how Folly achieves low-latency and high throughput while still ensuring the properties of the Byzantine Ordered Consensus specification.

Folly proposes a three phase architecture consisting of an ordering phase, a consensus phase and a post-consensus phase. Folly has pre-consensus and post-consensus phases, unlike Pompe that only has pre-consensus steps. By offsetting some of the pre-consensus communication steps to the post-consensus phase, Folly is able to reduce the number of effective communication steps required to reply back to the clients. This reduction in communication steps contributes to lower end-to-end request latencies and higher system throughput than Pompe.

Ordering Phase. The purpose of this phase is to ensure that commands are assigned valid timestamps. A replica r_i that proposes a command c and collects timestamps from $2f + 1$ replicas. The ordering timestamp ts for c is the median of the $2f + 1$ timestamps received from the replicas. The ordering timestamp indicates the total order among all the commands. Since at most f replicas are Byzantine, taking the median of the timestamps is enough to ensure that the timestamps belong to correct replicas. Replicas are notified of the assigned median timestamp ts for c.

Folly does not require locking the assigned timestamp to check whether the timestamped command will be included during the consensus phase for that range. Instead, replicas use outcome of the consensus phase to identify those commands that were included and those

that were not included.

Consensus Phase. Unlike Pompe, Folly does not propose consensus slots at lock-step intervals. It proposes commands for a given interval (ts, ts') as soon as it is sure that a supermajority of replicas have local timestamps that are higher than the maximum timestamp ts'. Individual replicas notify the leader when they are finished assigning timestamps that fall within the timestamp interval and monitor whether the leader properly proposes the timestamped commands in the consensus phase.

The consensus phase uses an off-the-shelf consensus protocol in order to agree on the sequence of slots. Each slot holds a set of commands with timestamps for a predetermined range. For instance, the first slot has commands with timestamps $[0, ts)$ and the second slot has commands with timestamps $[ts, ts')$ where $ts' > ts$ and so on.

Post-consensus phase The post-consensus phase is used to identify commands that did not make it to the consensus phase for $[ts, ts')$ despite having timestamp ts'', such that $ts < ts'' < ts'$. This phase is unique to Folly and is the result of reducing the number of communication steps during the ordering phase. However, note that this phase does not count towards the end-to-end client perceived latency during normal operation.

11.3 Protocol

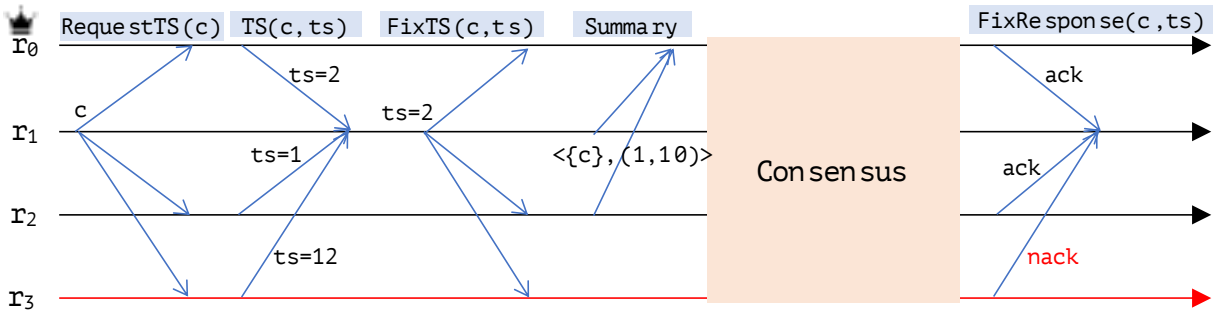


Figure 11.1: Folly example execution. Command c is proposed by r_1 to assign a timestamp. Despite r_3 returning a timestamp that is too large, picking the median value ensures that the timestamp returned by the correct replicas is assigned for c. Note that the ordering phase takes only four communication steps.

11.3.1 Pre-Consensus Ordering

Replica state. Each replica maintains the following data structures.

1. `localAcceptThresholdTS` is an integer that tracks what the latest possible timestamp of any stable command that r_j is aware of. Initially set to zero.

2. `localSequencedSet` is a set that contains all the commands that have been assigned a timestamp by enough correct nodes.

3. `highTS`, an n-sized vector of integers where `highTS[i]` stores the per replica r_i highest timestamp received so far; and

4. `highTSMsgs`, an n-sized vector of messages that stores the corresponding message for the highest timestamp in `highTS[i]`.

The protocol works as follows.

1. Node N_i broadcasts $\langle \text{REQUESTTS}, c \rangle_{\sigma_i}$ and waits for responses from $2f+1$ nodes. The σ_i indicates a signature by N_i over the message.

    - When Node N_j receives the REQUESTTS message, it assigns a timestamp ts to c and returns ts in a $\langle \text{TS}, c, ts \rangle_{\sigma_j}$ message.

2. N_i broadcasts a $\langle \text{FIXTS}, \text{C}, \text{TS} \rangle_{\sigma_i}$ message with the median final timestamp to the replicas. The ts is the median of the $2f+1$ timestamps that N_i collected via the TS message.

3. Periodically, every node sends a $\langle \text{SYNC}, \texttt{highTS}, \texttt{highTSMsgs} \rangle_{\sigma_i}$ message to all replicas.

    - When a replica receives the SYNC message, it will add all the commands whose timestamps are less than highTS of $2f+1$ replicas, to the `localSequencedSet` set.
    - After collecting SYNC message, each node will send the leader a SUMMARY message containing the commands within the given time interval that they know of.

11.3.2 Consensus

The consensus phase runs an off-the-shelf consensus protocol. Similar to Pompe, its values are set of commands with timestamps belonging to predefined ranges. The consensus phase for a slot is triggered when the leader receives $2f+1$ SUMMARY messages. The leader takes the union of all the commands in the SUMMARY messages, and runs the consensus protocol.

11.3.3 Post-Consensus Procedure and Execution

After receiving the set of commands as a result of the consensus phase for slot k, every replica will check whether all the commands whose timestamp must have been part of slot k are present in the set that was agreed during consensus. It is possible that the command could not be added to the set before the consensus procedure begun. Thus, replicas must send a FIXRESPONSE message to the original proposer informing whether the timestamp for c succeeded or not. Note that this communication step affects latency only if the timestamped command could be sent to the leader before the consensus phase began.

11.3.4 Correctness

Folly provides the same properties as Folly, and only reduces the communication step. Thus, Folly's theorem and lemma statements for correctness are same as Folly's. The proofs differ.

Theorem 11.1. *(Consistency). For every pair of correct replicas r_i and r_j with local logs L_i and L_j, the following holds: $L_i[k] = L_j[k]\ \forall k :: 0 \leq k \leq min(len(L_i), len(L_j))$, where $len()$ computes the number of entries in a log.*

Proof. The consensus protocol used in the consensus phase ensures that every correct replica agree on the same set of commands for each consensus slot. Furthermore, after the consensus decision on a set of commands, these commands are executed in the same deterministic order across all replicas. Hence, this shows that the log entries of any pair of correct replicas are equal if present. □

Theorem 11.2. *(Validity). If a correct replica appends a command c to its local totally-ordered log, then at least one replica in the system proposed c in the ordering phase.*

Proof. A command c is appended to the totally-ordered log only after the set containing it is agreed upon in a consensus slot, say k, via the consensus phase. This means that, when the consensus for slot k with time interval $[ts, ts')$ was proposed, the leader must have received SUMMARY messages from at least $2f + 1$ replicas for slot k. These summary messages must have included the commands with timestamps between ts and ts' including c. Furthermore, a command will be present in a SUMMARY message only if it has been proposed via the REQUESTTS message. Thus, at least one replica proposed c in the ordering phase. □

Lemma 11.3. *The assigned timestamp of a command will only be one within the interval of timestamps provided by correct nodes.*

Proof. Since there are at most f Byzantine replicas, at least $f+1$ out of the $2f+1$ timestamps received in the ordering phase for a command are from correct replicas. Among these $2f + 1$ timestamps, the median value is assigned to the command. The lowest f timestamps and

the highest f timestamps are discarded, hence the assigned timestamp for a command must be bounded by the timestamp from correct replicas. □

Theorem 11.4. *(Ordering linearizability). If the highest timestamp provided by any correct node for a command c_1 is lower than the lowest timestamp provided by any correct node for another command c_2 and if both c_1 and c_2 are committed, then c_1 will be applied before c_2 in the totally-ordered ledgers constructed by correct nodes.*

Proof. The previous lemma guarantees that the assigned timestamp of a command is unaffected by Byzantine nodes and thus its value will only be one within the interval of timestamps provided by correct nodes. Consequently, by the pre-condition in the theorem, c_1's timestamp will be smaller than c_2's. Note that nodes sort commands by their assigned timestamps. Therefore, when both c_1 and c_2 are committed, they will be applied in their timestamp order: c_1 will be applied before c_2 . □

Lemma 11.5. *During sufficiently long periods of synchrony, the pre-order phase ensures that if a correct node requests a timestamp for a command c, it will be assigned a timestamp ts by at least $2f + 1$ nodes and the command will be added to the `localSequencedSet` of at least $2f + 1$ nodes.*

Proof. Assume that a correct replica sends the REQUESTTS message and is able to successfully assign to command c, a timestamp of ts. During sufficiently long periods of synchrony, replicas will receive the FIXTS message and at least $2f + 1$ of them will add c to their `localSequencedSet`, before the consensus procedure kicks-off. This proves the stated lemma. □

Lemma 11.6. *If the `localSequencedSet` in at least $2f + 1$ nodes contain a command c with timestamp ts, then c will be part of the consensus slot k for time interval $[ts', ts'')$ such that $ts' \leq ts < ts''$.*

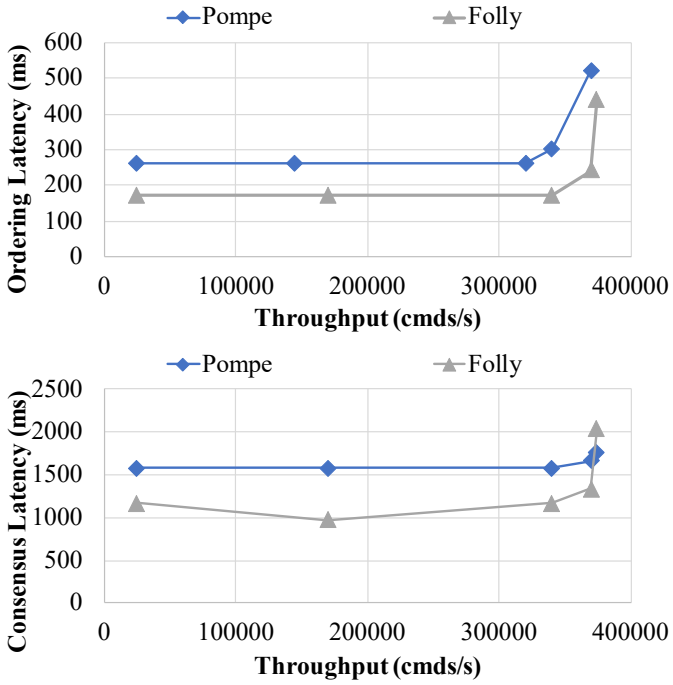
Proof. When correct replicas have local timestamps higher than the interval range for the consensus slot k, they will send a SUMMARY message to the primary. The FIXTS messages and the periodicity of the SYNC messages ensure that command c is present in the `localSequencedSet` of at least $2f + 1$ replicas. By assumption, since at most f of them are Byzantine, at least $f + 1$ replicas will include c in their SUMMARY messages for the consensus slot k. Note that the consensus phase is started by taking the union of the commands in $2f + 1$ summary messages. Thus, c must be included in the proposal for the consensus phase. The consensus phase will ensure that the slot containing the command gets committed during periods of synchrony. □

Theorem 11.7. *(Strong liveness). During synchronous periods, if a command c proposed by a correct node is assigned a timestamp, then it will eventually be included in the consensus phase and its total order will be determined by the assigned timestamp.*

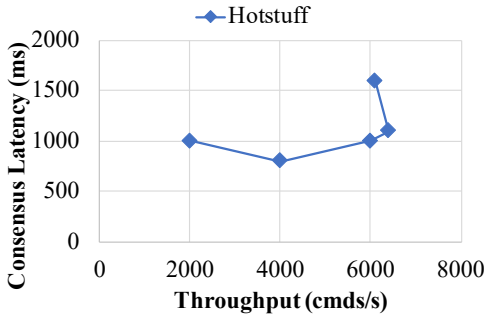
Proof. By the last two Lemmas and during sufficiently long periods of synchrony, c will eventually be among the set of commands committed in a consensus slot for the time interval that includes c's timestamp. After consensus, the set of commands agreed upon are totally ordered based on their timestamps at all the nodes. Thus, the position of c is determined by the assigned timestamp of c. □

11.4 Evaluation

We implemented Folly in Golang and evaluated it against Pompe-HS and Hotstuff. Pompe-HS is the Byzantine ordered consensus protocol that uses the Hotstuff [140] protocol for the consensus phase. Folly also uses Hotstuff as its consensus protocol. For Pompe's and Hotsuff's implementation, we used the original source code from the respective authors [140, 141].



(a) Latency versus Throughput for Folly and Pompe-HS. The top graph shows the average ordering performance incurred by Pompe and Folly. The bottom graph shows the average consensus performance of plain Hotstuff, and Pompe and Folly, respectively.



(b) Latency versus Throughput for Hotstuff.

Figure 11.2: Latency versus Throughput.

Deployment. We evaluated each of the protocols by deploying on the Azure cloud platform

spanning three datacenters: West US, South East Asia, and North Europe. We used the Standard D16s_v3 VMs with 16 vcpus and 64 GB memory. We placed $N = 4$ replicas in three datacenters, with one datacenter hosting both the replicas. We placed clients in each of the datacenters and increased the number of clients to increase the load on the replicas. The clients sent commands to the replicas in a closed-loop i.e., a client first waits for the reply to its previous command before sending the next command. Similar to [141], commands were 32 bytes long. Commands were batched for performance. For Hotstuff, we set the batch size to 800, and for Folly and Pompe, we set each node's ordering batch size to 200.

We measured client-perceived latency and throughput for each of the protocols. For Pompe and Folly, we report two latencies: ordering latency and consensus latency. The ordering latency captures the time taken by the ordering phase, while the consensus latency captures the time taken by the consensus phase and the ordering phase.

The results are in Figure 11.2. It can be observed that the ordering latencies for Folly are about 35% lower than Pompe's ordering latencies. There is no significant change in throughput between the two protocols. Similarly, the consensus latencies for Folly are about 30% lower than Pompe's latencies, while the throughput of both the protocols are very similar.

Thus, the reduction of two communication steps from the ordering phase results in better latencies for Folly while maintaining throughput at par with the state-of-the-art, Pompe.

11.5 Conclusions

In this chapter, we presented a new Byzantine Ordered Consensus protocol called Folly that reduces the number of communication steps required to assign a globally ordered timestamp to the commands. By performing an evaluation study, we show that our approach is effective in improving performance in terms of higher throughput as well as lower latency.

Chapter 12

Conclusions and Future Work

With the ever-increasing rate of cyberattacks, the ever-growing reliance on cloud computing resources, and the recent trends towards decentralized systems, the case for BFT systems is more appealing than ever. This dissertation has proposed several solutions for both existing applications such as databases and some newer classes of applications such as Blockchains. Each of these solutions provide unique trade-offs that make them a good fit for certain scenarios such as small deployments and low contention scenarios (ezBFT and Dester), versus larger deployments (DQBFT and Destiny). This dissertation presents protocols under four categories.

Leaderless Protocols. This dissertation proposes two leaderless consensus protocols, one each for the Byzantine and hybrid fault models. Existing Hybrid protocols provide poor performance and availability guarantees in geo-distributed deployments, even though CPU-based Trusted Execution Environments have made SMR solutions based on hybrid fault model more appealing than ever. Therefore, we present Dester, a leaderless hybrid fault-tolerant state machine replication protocol that provides optimal latencies for clients located at different geographical regions.

State-of-the-art BFT protocols are not able to provide optimal request processing latencies in geo-distributed deployments – an increasingly ubiquitous scenario for many distributed databases. ezBFT is a leaderless BFT protocol that provides three-step consensus in the common case, while essentially nullifying the latency of the first communication step.

Partially Decentralized Protocols. Typical deployments of Blockchain systems contain hundreds if not thousands of replicas. Dependency-based leaderless protocols cannot handle such scale since they need to perform additional steps to ensure all replicas observe dependencies consistently. Consequently, their performances are directly influenced by the amount of conflicting commands. This calls for simpler protocols and other techniques that can improve the performance of protocols at larger deployments.

DQBFT is a paradigm for designing highly scalable BFT consensus protocols. DQBFT's key insight is the observation that consensus requires two actions for committing: replication and ordering, and these can be performed independently. While replication does not require coordination and can therefore be decentralized, ordering requires coordination to ensure a single consistent view of the operation sequence and can be efficiently performed using a sequencer.

Destiny exploits this observation and performs partially decentralized consensus. Additionally, it linearizes all-to-all communication through message aggregation and threshold signatures, powered by trusted hardware. The result of these optimizations is a highly scalable protocol that can provide consistently high throughput at scale.

Byzantine Ordered Consensus. For certain Blockchain applications, it is important to ensure that client requests are treated fairly, but the traditional Byzantine Consensus does not specify any guarantee on request fairness. The Byzantine Ordered Consensus extends the original Byzantine Consensus properties by ensuring that client requests are treated fairly. Prior solutions sacrificed performance for fairness. This dissertation presents Folly, a new Byzantine Ordered Consensus protocol that reduces the number of communication steps required to assign a globally ordered timestamp to the commands. In doing so, Folly attempts to bring the performance of Byzantine Ordered protocols closer to their Byzantine counterparts.

Solutions for Practical Development and Deployment of BFT Protocols. The proponents of BFT protocols have long made the case against CFT solutions [84], but practitioners were discouraged because consensus protocols are hard to get right; even more cumbersome is adapting a totally new consensus scheme in a system that was built around the original protocol [35]. Therefore, practitioners have usually built *ad hoc* techniques to tolerate non-crash faults as they occur [28, 48, 111]. This approach creates unnecessary software bloat, increasing development and maintenance costs.

The BUMBLEBEE approach provides a three-step methodology to enhance existing CFT protocols to tolerate arbitrary faults. The Generic Algorithm we present is capable of representing most leader-based protocols in production use today (e.g. Paxos, Raft). Our intent is to provide system builders with a clear methodology to help them transform *in-house* consensus implementations to tolerate arbitrary faults. BUMBLEBEE should be seen akin to a specification tool (e.g. TLA+): It provides a *precise* way to design a Hybrid protocol starting from the specification of a CFT protocol. This would save system builder a lot of effort that would otherwise be spent in building and verifying new protocols from scratch.

Furthermore, Hybrid protocols rely on the integrity of the trusted execution environments (TEEs) to guarantee protocol safety. This total reliance is concerning since exploits have been discovered on CPU-based TEEs like Intel SGX [131]. Rather than completely depend on TEEs for both performance and safety, this dissertation proposes to use Hybrid protocols for performance and use a traditional BFT protocol to guarantee safety. DUOBFT is a protocol that provides two fault models under the partial synchrony setting and lets learners make commit decisions under either of them. We show that a fraction of hybrid replicas is enough to enable commit decisions under the hybrid fault model and still preserve safety guarantees. Furthermore, the incorporation of the flexible hybrid quorum technique is what makes the common view change protocol that supports both the fault models feasible. This leads to a simplified protocol that is capable of supporting two different fault models.

The protocols presented in this dissertation reaffirms that there is no one-size-fits-all solution

to building consensus protocols. Newer deployment settings and newer applications push the existing protocols to their limits triggering reanalyses of exiting techniques and the search for newer one. Furthermore, while certain techniques are transferable from one domain to another (e.g. CFT to BFT), target domain considerations and careful tuning can yield greater performance improvements.

From an implementation perspective, the choice of the Go language played an important role in development of the protocols discussed and evaluated in this dissertation. The simplicity of the language improved development velocity over a language such as Java that was used to build earlier prototypes and enables quick prototype implementations of the protocols. Furthermore, the ecosystem of libraries and tools around the language make it easy to build distributed applications. GRPC [7] and protobuf [67] greatly eased the networking implementation for communication between replicas. The deployment orchestration system Kubernetes [66] allowed for quick and repeatable deployments of systems of all sizes ranging from few nodes to over 300 nodes. The source code for the protocols discussed in this dissertation is available at https://github.com/ibalajiarun/go-consensus.

12.1 Future Work

There are many directions for future work for the solutions presented in this dissertation. Here, we present several future directions.

12.1.1 Additional Fault Models within DuoBFT

Currently, DuoBFT exposes the Trust-based (Hybrid) BFT model as well as the traditional BFT model as options to clients. The a-b-c fault model [106] distinguishes between Byzantine faults and non-Byzantine faults within the same protocol. This enables protocols adopting the a-b-c fault model to tolerate variable number of Byzantine and non-Byzantine faults. For certain applications, assuming a larger fault threshold f can help provide better guarantees to clients and enhance the quality of the service. For example, when processing a $100M transaction, it might be worthwhile to assume larger fault threshold to nullify any chance that a lower fault threshold is violated. Even though considering a larger fault threshold will significantly increase the transaction completion time, this is necessary to ensure the safety of high-profile transactions. DuoBFT currently uses a fixed f value. Introducing variable f to DuoBFT either by incorporating the a-b-c fault model or exploring other means remains interesting future work.

12.1.2 Byzantine Ordered Consensus with DQBFT paradigm

The Byzantine Ordered Consensus protocols, Pompe and Folly, both involve a two phase protocol. In the first phase, an ordering timestamp is assigned to commands, and in the second phase, the command undergoes consensus. While Folly reduces the communication steps in the ordering phase, the effective number of communication steps for both the phases in total is ten. In contrast, highly scalable protocols under the DQBFT paradigm have around half the number of communication steps despite being a two phase protocol. The reason is that DQBFT protocols allow the two phases to happen concurrently reducing the effective number of communication steps. Exploring such techniques for Byzantine Ordered Consensus is an interesting future direction.

12.1.3 Generic Algorithm for Leaderless Protocol Transformation

The BUMBLEBEE methodology currently focuses only on single leader-based CFT protocols, except M^2Paxos (which resembles single leader protocols to a large extent). Many Leaderless CFT protocols have been proposed in literature [21, 59, 117]. These protocols are inspired by Paxos to a large extent, yet due to additional elements in their design such as dependency collection, they do not fit well into the current Generic Algorithm. Even though EZBFT and DESTER protocols are deeply inspired from EPaxos, newer protocols such as Caesar [21] and Atlas [59] have the potential to provide performance improvements over the leaderless protocols presented in this dissertation. The current Generic Algorithm must be extended to accommodate these additional elements such as dependency processing. Exploring the feasibility of such techniques remain future work.

12.1.4 Automatic Bumblebee Transformations

The BUMBLEBEE approach is largely a mechanical technique. Thus, it may be possible to auto-generate the transformations. Given a specification of a CFT protocol in a formal language (e.g. TLA+ [86], Verdi [138]), an algorithm should first map the specification in terms of the Generic Algorithm and its parameters. From this mapping, the algorithm should produce a specification of the hybrid fault-tolerant version Given the complexities implicit in many consensus protocols, such an approach would minimize the time required for such transformations and minimize unnecessary bugs that could be induced with manual methods. However, such an approach requires that the algorithm identifies whether the input CFT specification is compatible with the BUMBLEBEE approach. This is because our technique does not apply to all existing CFT protocols particularly leaderless variants (e.g. EPaxos). We leave the exploration and analysis of such techniques for future work.

12.1.5 Flexible Leaderless Protocols

The leaderless protocols presented in this dissertation adopt larger quorums than primary-based protocols. The additional nodes in the quorum increases the chances of not observing the same set of dependencies at higher conflicts, leading to more slow decisions. This effectively reduces the performance of these protocols when the system size grows as the size of fast quorums witness a non-linear growth. The Flexible Quorums approach can allow for taming the fast quorum sizes as demonstrated by recent CFT protocols such as Atlas [59]. This dissertation describes how Flexible Quorums can be adopted to the hybrid model, and Flexible BFT quorums have already been introduced in the past [106]. Use of Flexible Quorums to design efficient leaderless protocols under the BFT and hybrid models that maintain their performance at scale remains interesting future work.

Bibliography

[1] [n.d.]. Azure. https://azure.microsoft.com/. Accessed: 2019-01-06.

[2] [n.d.]. Cloud Spanner - Google Cloud. https://cloud.google.com/spanner/. Accessed: 2018-11-06.

[3] [n.d.]. CockroachDB. https://github.com/cockroachdb/cockroach. Accessed: 2018-11-06.

[4] [n.d.]. CosmosDB - Azure. https://azure.microsoft.com/en-us/services/cosmos-db/. Accessed: 2018-11-06.

[5] [n.d.]. etcd - CoreOS. https://coreos.com/etcd/. Accessed: 2018-11-06.

[6] [n.d.]. etcd - Production Users. https://github.com/etcd-io/etcd/blob/master/Documentation/production-users.md. Accessed: 2019-01-05.

[7] [n.d.]. gRPC. https://github.com/grpc/.

[8] Michael Abd-El-Malek, Gregory R. Ganger, Garth R. Goodson, Michael K. Reiter, and Jay J. Wylie. 2005. Fault-scalable Byzantine Fault-tolerant Services. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles* (Brighton, United Kingdom) *(SOSP '05)*. ACM, New York, NY, USA, 59–74. https://doi.org/10.1145/1095810.1095817

[9] Ittai Abraham, Srinivas Devadas, Danny Dolev, Kartik Nayak, and Ling Ren. 2019. Synchronous Byzantine Agreement with Expected O(1) Rounds, Expected $O(n^2)$ Communication, and Optimal Resilience. In *Financial Cryptography and Data Security*, Ian Goldberg and Tyler Moore (Eds.). Springer International Publishing, 320–334.

[10] Ittai Abraham, Dahlia Malkhi, Kartik Nayak, Ling Ren, and Maofan Yin. 2019. Sync HotStuff: Simple and Practical Synchronous State Machine Replication. Cryptology ePrint Archive, Report 2019/270. https://eprint.iacr.org/2019/270.

[11] Atul Adya. 1999. *Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions*. Ph.D. MIT, Cambridge, MA, USA. Also as Technical Report MIT/LCS/TR-786.

[12] A. Ailijiang, A. Charapko, M. Demirbas, and T. Kosar. 2020. WPaxos: Wide Area Network Flexible Consensus. *IEEE Transactions on Parallel and Distributed Systems* 31, 1 (2020), 211–223. https://doi.org/10.1109/TPDS.2019.2929793

[13] Amitanand S. Aiyer, Lorenzo Alvisi, Allen Clement, Mike Dahlin, Jean-Philippe Martin, and Carl Porth. 2005. BAR Fault Tolerance for Cooperative Services. *SIGOPS Oper. Syst. Rev.* 39, 5 (Oct. 2005), 45–58. https://doi.org/10.1145/1095809.1095816

[14] Ramnatthan Alagappan, Aishwarya Ganesan, Eric Lee, Aws Albarghouthi, Vijay Chidambaram, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2018. Protocol-Aware Recovery for Consensus-Based Storage. In *16th USENIX Conference on File and Storage Technologies (FAST 18)* (Oakland, CA). USENIX Association, 15–32.

[15] Amazon Inc. [n.d.]. Elastic Compute Cloud. http://aws.amazon.com/ec2.

[16] Y. Amir, B. Coan, J. Kirsch, and J. Lane. 2008. Byzantine replication under attack. In *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*, Vol. 00. 197–206. https://doi.org/10.1109/DSN.2008.4630088

[17] Y. Amir, B. Coan, J. Kirsch, and J. Lane. 2011. Prime: Byzantine Replication under Attack. *IEEE Transactions on Dependable and Secure Computing* 8, 4 (July 2011), 564–577. https://doi.org/10.1109/TDSC.2010.70

[18] Y. Amir, C. Danilov, D. Dolev, J. Kirsch, J. Lane, C. Nita-Rotaru, J. Olsen, and D. Zage. 2010. Steward: Scaling Byzantine Fault-Tolerant Replication to Wide Area Networks. *IEEE Transactions on Dependable and Secure Computing* 7, 1 (Jan 2010), 80–93. https://doi.org/10.1109/TDSC.2008.53

[19] Y. Amir, C. Danilov, J. Kirsch, J. Lane, D. Dolev, C. Nita-Rotaru, J. Olsen, and D. Zage. 2006. Scaling Byzantine Fault-Tolerant Replication to Wide Area Networks. In *International Conference on Dependable Systems and Networks (DSN'06)*. 105–114. https://doi.org/10.1109/DSN.2006.63

[20] John Arquilla and Mark Guzdial. 2021. The SolarWinds Hack, and a Grand Challenge for CS Education. *Commun. ACM* 64, 4 (March 2021), 6–7. https://doi.org/10.1145/3449047

[21] Balaji Arun, Sebastiano Peluso, Roberto Palmieri, Giuliano Losa, and Binoy Ravindran. 2017. Speeding up Consensus by Chasing Fast Decisions. In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)* (2017-06). 49–60. https://doi.org/10.1109/DSN.2017.35

[22] B. Arun, S. Peluso, R. Palmieri, G. Losa, and B. Ravindran. 2020. Taming the Contention in Consensus-based Distributed Systems. *IEEE Transactions on Dependable and Secure Computing* (2020), 1–1. https://doi.org/10.1109/TDSC.2020.2970186

[23] B. Arun, S. Peluso, and B. Ravindran. 2019. ezBFT: Decentralizing Byzantine Fault-Tolerant State Machine Replication. In *2019 IEEE 39th International Conference*

on Distributed Computing Systems (ICDCS). 565–577. https://doi.org/10.1109/ICDCS.2019.00063

[24] Balaji Arun and Binoy Ravindran. 2020. DuoBFT: Resilience vs. Efficiency Trade-off in Byzantine Fault Tolerance. arXiv:2010.01387 [cs.DC]

[25] Hagit Attiya and Jennifer L. Welch. 1994. Sequential Consistency versus Linearizability. *ACM Trans. Comput. Syst.* 12, 2 (May 1994), 91–122. https://doi.org/10.1145/176575.176576

[26] Pierre-Louis Aublin, Rachid Guerraoui, Nikola Knežević, Vivien Quéma, and Marko Vukolić. 2015. The Next 700 BFT Protocols. *ACM Trans. Comput. Syst.* 32, 4 (2015), 12:1–12:45. https://doi.org/10.1145/2658994

[27] Amy Babay, John Schultz, Thomas Tantillo, Samuel Beckley, Eamon Jordan, Kevin Ruddell, Kevin Jordan, and Yair Amir. 2019. Deploying Intrusion-Tolerant SCADA for the Power Grid. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 328–335. https://doi.org/10.1109/DSN.2019.00043

[28] Peter Bailis and Kyle Kingsbury. 2014. The Network is Reliable. *Queue* 12, 7, Article 20 (July 2014), 13 pages. https://doi.org/10.1145/2639988.2655736

[29] Jason Baker, Chris Bond, James C. Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. 2011. Megastore: Providing Scalable, Highly Available Storage for Interactive Services. In *Proceedings of the Conference on Innovative Data system Research (CIDR)*. 223–234. http://www.cidrdb.org/cidr2011/Papers/CIDR11_Paper32.pdf

[30] Paulo S. L. M. Barreto, Ben Lynn, and Michael Scott. 2003. Constructing Elliptic Curves with Prescribed Embedding Degrees. In *Proceedings of the 3rd International Conference on Security in Communication Networks* (Amalfi, Italy) *(SCN'02)*. Springer-Verlag, Berlin, Heidelberg, 257–267. http://dl.acm.org/citation.cfm?id=1766811.1766837

[31] Rida Bazzi and Maurice Herlihy. [n.d.]. Clairvoyant State Machine Replications. In *Stabilization, Safety, and Security of Distributed Systems* (Cham, 2018), Taisuke Izumi and Petr Kuznetsov (Eds.). Springer International Publishing, 254–268.

[32] Johannes Behl, Tobias Distler, and Rüdiger Kapitza. 2017. Hybrids on Steroids: SGX-Based High Performance BFT. In *Proceedings of the Twelfth European Conference on Computer Systems* (Belgrade, Serbia) *(EuroSys '17)*. ACM, 222–237. https://doi.org/10.1145/3064176.3064213

[33] Michael Ben-Or. 1983. Another Advantage of Free Choice (Extended Abstract): Completely Asynchronous Agreement Protocols. In *Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing* (Montreal, Quebec, Canada) *(PODC '83)*. Association for Computing Machinery, New York, NY, USA, 27–30. https://doi.org/10.1145/800221.806707

[34] C. Berger, H. P. Reiser, J. Sousa, and A. Bessani. 2019. Resilient Wide-Area Byzantine Consensus Using Adaptive Weighted Replication. In *2019 38th Symposium on Reliable Distributed Systems (SRDS)*. 183–18309. https://doi.org/10.1109/SRDS47363.2019.00029

[35] Alysson Bessani, João Sousa, and Eduardo E. P. Alchieri. 2014. State Machine Replication for the Masses with BFT-SMART. In *Proceedings of the 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '14)*. IEEE Computer Society, Washington, DC, USA, 355–362. https://doi.org/10.1109/DSN.2014.43

[36] Sebastian Burckhardt. 2014. Principles of eventual consistency. (2014).

[37] Vitalik Buterin and Virgil Griffith. 2017. Casper the Friendly Finality Gadget. *CoRR* abs/1710.09437 (2017). arXiv:1710.09437 http://arxiv.org/abs/1710.09437

[38] Christian Cachin, Rachid Guerraoui, and Luís Rodrigues. 2011. *Introduction to Reliable and Secure Distributed Programming* (2. ed ed.). Springer.

[39] Christian Cachin, Klaus Kursawe, and Victor Shoup. 2005. Random Oracles in Constantinople: Practical Asynchronous Byzantine Agreement Using Cryptography. *J. Cryptol.* 18, 3 (July 2005), 219–246. https://doi.org/10.1007/s00145-005-0318-0

[40] Miguel Castro. 2001. *Practical Byzantine Fault Tolerance*. Ph.D. Dissertation. Massachusetts Institute of Technology.

[41] Miguel Castro and Barbara Liskov. 1999. Practical Byzantine Fault Tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation* (New Orleans, Louisiana, USA) *(OSDI '99)*. USENIX Association, Berkeley, CA, USA, 173–186. http://dl.acm.org/citation.cfm?id=296806.296824

[42] Miguel Castro and Barbara Liskov. 2002. Practical Byzantine Fault Tolerance and Proactive Recovery. *ACM Trans. Comput. Syst.* 20, 4 (Nov. 2002), 398–461. https://doi.org/10.1145/571637.571640

[43] Tushar Deepak Chandra and Sam Toueg. 1996. Unreliable Failure Detectors for Reliable Distributed Systems. *J. ACM* 43, 2 (March 1996), 225–267. https://doi.org/10.1145/226643.226647

[44] Byung-Gon Chun, Petros Maniatis, Scott Shenker, and John Kubiatowicz. 2007. Attested Append-only Memory: Making Adversaries Stick to Their Word. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles* (Stevenson, Washington, USA) *(SOSP '07)*. ACM, 189–204. https://doi.org/10.1145/1294261.1294280

[45] Allen Clement, Manos Kapritsos, Sangmin Lee, Yang Wang, Lorenzo Alvisi, Mike Dahlin, and Taylor Riche. 2009. Upright Cluster Services. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles* (Big Sky, Montana, USA) *(SOSP '09)*. ACM, 277–290. https://doi.org/10.1145/1629575.1629602

[46] Allen Clement, Edmund Wong, Lorenzo Alvisi, Mike Dahlin, and Mirco Marchetti. 2009. Making Byzantine Fault Tolerant Systems Tolerate Byzantine Faults. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation* (Boston, Massachusetts) *(NSDI'09)*. USENIX Association, Berkeley, CA, USA, 153–168. http://dl.acm.org/citation.cfm?id=1558977.1558988

[47] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. 2013. Spanner: Google's globally distributed database. 31, 3 (2013), 8.

[48] Miguel Correia, Daniel Gómez Ferro, Flavio P. Junqueira, and Marco Serafini. 2012. Practical Hardening of Crash-tolerant Systems. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference* (Boston, MA) *(USENIX ATC'12)*. USENIX Association, Berkeley, CA, USA, 41–41. http://dl.acm.org/citation.cfm?id=2342821.2342862

[49] M. Correia, N.F. Neves, and P. Verissimo. 2004. How to tolerate half less one Byzantine nodes in practical distributed systems. In *Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems, 2004.* 174–183. https://doi.org/10.1109/RELDIS.2004.1353018

[50] Miguel Correia, Giuliana S. Veronese, and Lau Cheuk Lung. 2010. Asynchronous Byzantine Consensus with 2f+1 Processes. In *Proceedings of the 2010 ACM Symposium on Applied Computing* (Sierre, Switzerland) *(SAC '10)*. Association for Computing Machinery, New York, NY, USA, 475–480. https://doi.org/10.1145/1774088.1774187

[51] Miguel Correia, Giuliana S. Veronese, and Lau Cheuk Lung. 2010. Asynchronous Byzantine consensus with 2f+1 processes. In *Proceedings of the 2010 ACM Symposium on Applied Computing (SAC '10)*. Association for Computing Machinery, 475–480. https://doi.org/10.1145/1774088.1774187

[52] Victor Costan and Srinivas Devadas. 2016. Intel SGX Explained. 2016, 086 (2016), 1–118.

[53] James Cowling, Daniel Myers, Barbara Liskov, Rodrigo Rodrigues, and Liuba Shrira. 2006. HQ Replication: A Hybrid Quorum Protocol for Byzantine Fault Tolerance. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation* (Seattle, Washington) *(OSDI '06)*. USENIX Association, 177–190.

[54] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. [n.d.]. Dynamo: Amazon's Highly Available Key-value Store. 41, 6 ([n. d.]), 205–220. https://doi.org/10.1145/1323293.1294281

[55] T. Distler, C. Cachin, and R. Kapitza. 2016. Resource-Efficient Byzantine Fault Tolerance. *IEEE Trans. Comput.* 65, 9 (Sep. 2016), 2807–2819. https://doi.org/10.1109/TC.2015.2495213

[56] Sisi Duan, Michael K. Reiter, and Haibin Zhang. 2018. BEAT: Asynchronous BFT Made Practical. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) *(CCS '18)*. Association for Computing Machinery, New York, NY, USA, 2028–2041. https://doi.org/10.1145/3243734.3243812

[57] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. 1988. Consensus in the Presence of Partial Synchrony. *J. ACM* 35, 2 (April 1988), 288–323. https://doi.org/10.1145/42282.42283

[58] M. Eischer and T. Distler. 2017. Scalable Byzantine Fault Tolerance on Heterogeneous Servers. In *2017 13th European Dependable Computing Conference (EDCC)*. 34–41. https://doi.org/10.1109/EDCC.2017.15

[59] Vitor Enes, Carlos Baquero, Tuanir França Rezende, Alexey Gotsman, Matthieu Perrin, and Pierre Sutra. 2020. State-Machine Replication for Planet-Scale Systems. In *Proceedings of the Fifteenth European Conference on Computer Systems* (Heraklion, Greece) *(EuroSys '20)*. Association for Computing Machinery, New York, NY, USA, Article 24, 15 pages. https://doi.org/10.1145/3342195.3387543

[60] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. 1985. Impossibility of Distributed Consensus with One Faulty Process. *J. ACM* 32, 2 (April 1985), 374–382. https://doi.org/10.1145/3149.214121

[61] Miguel Garcia, Alysson Bessani, and Nuno Neves. 2019. Lazarus: Automatic Management of Diversity in BFT Systems. In *Proceedings of the 20th International Middleware Conference* (Davis, CA, USA) *(Middleware '19)*. Association for Computing Machinery, New York, NY, USA, 241–254. https://doi.org/10.1145/3361525.3361550

[62] Mohit Garg, Sebastiano Peluso, Balaji Arun, and Binoy Ravindran. 2019. Generalized Consensus for Practical Fault Tolerance. In *Proceedings of the 20th International Middleware Conference* (Davis, CA, USA) *(Middleware '19)*. ACM, New York, NY, USA, 55–67. https://doi.org/10.1145/3361525.3361536

[63] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. 2017. Algorand: Scaling Byzantine Agreements for Cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 51–68. https://doi.org/10.1145/3132747.3132757

[64] G. Golan Gueta, I. Abraham, S. Grossman, D. Malkhi, B. Pinkas, M. Reiter, D. Seredinschi, O. Tamir, and A. Tomescu. 2019. SBFT: A Scalable and Decentralized Trust Infrastructure. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 568–580. https://doi.org/10.1109/DSN.2019.00063

[65] Google. [n.d.]. cgo - The Go programming language. http://golang.org/cgo. Accessed: 2019-01-10.

[66] Google. [n.d.]. Kubernetes. https://kubernetes.io/.

[67] Google. [n.d.]. Protocol Buffers. https://developers.google.com/protocol-buffers/.

[68] Rachid Guerraoui, Nikola Knežević, Vivien Quéma, and Marko Vukolić. 2010. The Next 700 BFT Protocols. In *Proceedings of the 5th European Conference on Computer Systems* (Paris, France) *(EuroSys '10)*. ACM, 363–376. https://doi.org/10.1145/1755913.1755950

[69] L. J. Gunn, J. Liu, B. Vavala, and N. Asokan. 2019. Making Speculative BFT Resilient with Trusted Monotonic Counters. In *2019 38th Symposium on Reliable Distributed Systems (SRDS)*. 133–13309. https://doi.org/10.1109/SRDS47363.2019.00024

[70] Suyash Gupta, Jelle Hellings, and Mohammad Sadoghi. 2021. RCC: Resilient Concurrent Consensus for High-Throughput Secure Transaction Processing. In *Int. Conf. on Data Engineering (ICDE)*.

[71] Suyash Gupta, Sajjad Rahnama, Jelle Hellings, and Mohammad Sadoghi. 2020. ResilientDB: Global Scale Resilient Blockchain Fabric. *Proc. VLDB Endow.* 13, 6 (Feb. 2020), 868–883. https://doi.org/10.14778/3380750.3380757

[72] Vassos Hadzilacos and Sam Toueg. 1994. *A Modular Approach to Fault-Tolerant Broadcasts and Related Problems*. Technical Report. USA.

[73] Maurice P. Herlihy and Jeannette M. Wing. [n.d.]. Linearizability: a correctness condition for concurrent objects. 12, 3 ([n. d.]), 463–492. https://doi.org/10.1145/78969.78972

[74] Chi Ho, Robbert van Renesse, Mark Bickford, and Danny Dolev. 2008. Nysiad: Practical Protocol Transformation to Tolerate Byzantine Failures. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation* (San Francisco, California) *(NSDI'08)*. USENIX Association, Berkeley, CA, USA, 175–188. http://dl.acm.org/citation.cfm?id=1387589.1387602

[75] Heidi Howard, Dahlia Malkhi, and Alexander Spiegelman. 2016. Flexible Paxos: Quorum Intersection Revisited. In *20th International Conference on Principles of Distributed Systems, OPODIS 2016, December 13-16, 2016, Madrid, Spain*. 25:1–25:14. https://doi.org/10.4230/LIPIcs.OPODIS.2016.25

[76] Heidi Howard, Dahlia Malkhi, and Alexander Spiegelman. 2017. Flexible Paxos: Quorum Intersection Revisited. In *20th International Conference on Principles of Distributed Systems (OPODIS 2016) (Leibniz International Proceedings in Informatics (LIPIcs))*, Panagiota Fatourou, Ernesto Jiménez, and Fernando Pedone (Eds.), Vol. 70. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 25:1–25:14. https://doi.org/10.4230/LIPIcs.OPODIS.2016.25

[77] Heidi Howard and Richard Mortier. 2020. Paxos vs Raft: Have We Reached Consensus on Distributed Consensus?. In *Proceedings of the 7th Workshop on Principles and Practice of Consistency for Distributed Data* (Heraklion, Greece) *(PaPoC '20)*. Association for Computing Machinery, New York, NY, USA, Article 8, 9 pages. https://doi.org/10.1145/3380787.3393681

[78] K. Hoyme and K. Driscoll. 1993. SAFEbus (for avionics). *IEEE Aerospace and Electronic Systems Magazine* 8, 3 (March 1993), 34–39. https://doi.org/10.1109/62.199819

[79] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. [n.d.]. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference* (Boston, MA, 2010) *(USENIX ATC'10)*. USENIX Association, 11–11.

[80] Don Johnson, Alfred Menezes, and Scott Vanstone. 2001. The Elliptic Curve Digital Signature Algorithm (ECDSA). *Int. J. Inf. Secur.* 1, 1 (Aug. 2001), 36–63. https://doi.org/10.1007/s102070100002

[81] Thomas A. Joseph and Kenneth P. Birman. 1988. *Reliable Broadcast Protocols*. Technical Report. USA.

[82] Rüdiger Kapitza, Johannes Behl, Christian Cachin, Tobias Distler, Simon Kuhnle, Seyed Vahid Mohammadi, Wolfgang Schröder-Preikschat, and Klaus Stengel. 2012. CheapBFT: Resource-efficient Byzantine Fault Tolerance. In *Proceedings of the 7th ACM European Conference on Computer Systems* (Bern, Switzerland) *(EuroSys '12)*. ACM, 295–308. https://doi.org/10.1145/2168836.2168866

[83] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. 2007. Zyzzyva: Speculative Byzantine Fault Tolerance. (2007), 45–58. https://doi.org/10.1145/1294261.1294267

[84] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. 2010. Zyzzyva: Speculative Byzantine Fault Tolerance. 27, 4 (2010), 7:1–7:39. https://doi.org/10.1145/1658357.1658358

[85] H. Krawczyk, M. Bellare, and R. Canetti. 1997. RFC2104: HMAC: Keyed-Hashing for Message Authentication.

[86] Leslie Lamport. [n.d.]. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Longman Publishing Co., Inc.

[87] Leslie Lamport. 1978. The implementation of reliable distributed multiprocess systems. *Computer Networks (1976)* 2, 2 (1978), 95–114. https://doi.org/10.1016/0376-5075(78)90045-4

[88] Leslie Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM* 21, 7 (July 1978), 558–565. https://doi.org/10.1145/359545.359563

[89] Leslie Lamport. 2005. *Generalized Consensus and Paxos*. Technical Report MSR-TR-2005-33. 60 pages. https://www.microsoft.com/en-us/research/publication/generalized-consensus-and-paxos/

[90] Leslie Lamport. 2006. Fast Paxos. *Distributed Computing* 19 (Oct 2006). https://www.microsoft.com/en-us/research/publication/fast-paxos/

[91] Leslie Lamport. 2011. Brief Announcement: Leaderless Byzantine Paxos. In *Distributed Computing*, David Peleg (Ed.). Springer Berlin Heidelberg, 141–142.

[92] Leslie Lamport. 2011. Byzantizing Paxos by Refinement. In *Proceedings of the 25th International Conference on Distributed Computing* (Rome, Italy) *(DISC'11)*. Springer-Verlag, 211–224.

[93] Leslie Lamport. 2011. Byzantizing Paxos by Refinement. In *Proceedings of the 25th International Conference on Distributed Computing* (Rome, Italy) *(DISC'11)*. Springer-Verlag, 211–224.

[94] Leslie Lamport. 2021. My Writings. http://lamport.azurewebsites.net/pubs/pubs.pdf.

[95] Leslie Lamport et al. 2001. *Paxos made simple*. Technical Report.

[96] Leslie Lamport, Robert Shostak, and Marshall Pease. 1982. The Byzantine Generals Problem. *ACM Trans. Program. Lang. Syst.* 4, 3 (July 1982), 382–401. https://doi.org/10.1145/357172.357176

[97] Leslie Lamport, Robert Shostak, and Marshall Pease. 1982. The Byzantine Generals Problem. *ACM Trans. Program. Lang. Syst.* 4, 3 (July 1982), 382–401. https://doi.org/10.1145/357172.357176

[98] Dave Levin, John R. Douceur, Jacob R. Lorch, and Thomas Moscibroda. 2009. TrInc: Small Trusted Hardware for Large Distributed Systems. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation* (Boston, Massachusetts) *(NSDI'09)*. USENIX Association, 1–14.

[99] B. Li, N. Weichbrodt, J. Behl, P. Aublin, T. Distler, and R. Kapitza. 2018. Troxy: Transparent Access to Byzantine Fault-Tolerant Systems. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 59–70. https://doi.org/10.1109/DSN.2018.00019

[100] ARM Limited. [n.d.]. ARM Security technology: building a secure system using TrustZone technology. http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf. ARM Technical White Paper, Accessed: 2018-11-06.

[101] Jian Liu, Wenting Li, Ghassan O. Karame, and N. Asokan. 2018. Toward Fairness of Cryptocurrency Payments. *IEEE Security and Privacy* 16, 3 (May 2018), 81–89. https://doi.org/10.1109/MSP.2018.2701163

[102] J. Liu, W. Li, G. O. Karame, and N. Asokan. 2019. Scalable Byzantine Consensus via Hardware-Assisted Secret Sharing. *IEEE Trans. Comput.* 68, 1 (Jan 2019), 139–151. https://doi.org/10.1109/TC.2018.2860009

[103] Shengyun Liu, Paolo Viotti, Christian Cachin, Vivien Quéma, and Marko Vukolic. 2016. XFT: Practical Fault Tolerance Beyond Crashes. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Savannah, GA, USA) *(OSDI'16)*. USENIX Association, 485–500.

[104] N.A. Lynch. 1996. *Distributed Algorithms*. Kaufmann. https://books.google.com/books?id=7C7oIV48RQQC

[105] Ben Lynn. 2007. *On the implementation of pairing-based cryptosystems*. Ph.D. Dissertation. Stanford University Stanford, California.

[106] Dahlia Malkhi, Kartik Nayak, and Ling Ren. 2019. Flexible Byzantine Fault Tolerance. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security* (London, United Kingdom) *(CCS '19)*. Association for Computing Machinery, New York, NY, USA, 1041–1053. https://doi.org/10.1145/3319535.3354225

[107] Yanhua Mao, Flavio P. Junqueira, and Keith Marzullo. 2008. Mencius: Building Efficient Replicated State Machines for WANs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (San Diego, California) *(OSDI'08)*. USENIX Association, USA, 369–384.

[108] J. P. Martin and L. Alvisi. 2005. Fast Byzantine consensus. In *2005 International Conference on Dependable Systems and Networks (DSN'05)*. 402–411. https://doi.org/10.1109/DSN.2005.48

[109] Jean-Philippe Martin and Lorenzo Alvisi. 2006. Fast Byzantine Consensus. *IEEE Trans. Dependable Secur. Comput.* 3, 3 (2006), 202–215. https://doi.org/10.1109/TDSC.2006.35

[110] Sinisa Matetic, Mansoor Ahmed, Kari Kostiainen, Aritra Dhar, David Sommer, Arthur Gervais, Ari Juels, and Srdjan Capkun. 2017. ROTE: Rollback Protection for Trusted Execution. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, Vancouver, BC, 1289–1306. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/matetic

[111] Vivek Menezes. [n.d.]. Trust, But Verify: How CockroachDB Checks Replication. https://www.cockroachlabs.com/blog/trust-but-verify-cockroachdb-checks-replication/, note = Accessed: 2018-11-06.

[112] Microsoft. [n.d.]. Azure Confidential Computing. https://azure.microsoft.com/en-us/solutions/confidential-compute/.

[113] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. 2016. The Honey Badger of BFT Protocols. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria) *(CCS '16)*. Association for Computing Machinery, New York, NY, USA, 31–42. https://doi.org/10.1145/2976749.2978399

[114] Zarko Milosevic, Martin Hutle, and André Schiper. 2009. Unifying Byzantine Consensus Algorithms with Weak Interactive Consistency. In *Principles of Distributed Systems*, Tarek Abdelzaher, Michel Raynal, and Nicola Santoro (Eds.). Springer Berlin Heidelberg, 300–314.

[115] Milošević, Žarko. 2013. *Abstractions for Solving Consensus and Related Problems with Byzantine Faults*. Ph.D. Dissertation. EPFL. https://doi.org/10.5075/epfl-thesis-5975

[116] Iulian Moraru. 2015. Egalitarian Distributed Consensus. http://www.pdl.cmu.edu/PDL-FTP/associated/CMU-CS-14-133.pdf.

[117] Iulian Moraru, David G. Andersen, and Michael Kaminsky. 2013. There is More Consensus in Egalitarian Parliaments. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farminton, Pennsylvania) *(SOSP '13)*. ACM, 358–372. https://doi.org/10.1145/2517349.2517350

[118] Satoshi Nakamoto. 2008. *Bitcoin: A peer-to-peer electronic cash system*. Technical Report.

[119] Brian M. Oki and Barbara H. Liskov. 1988. 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* (Toronto, Ontario, Canada) *(PODC '88)*. Association for Computing Machinery, New York, NY, USA, 8–17. https://doi.org/10.1145/62546.62549

[120] Diego Ongaro and John Ousterhout. 2014. In Search of an Understandable Consensus Algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)* (Philadelphia, PA). USENIX Association, 305–319.

[121] Rafael Pass and Elaine Shi. 2018. Thunderella: Blockchains with Optimistic Instant Confirmation. In *Advances in Cryptology – EUROCRYPT 2018 (Lecture Notes in Computer Science)*, Jesper Buus Nielsen and Vincent Rijmen (Eds.). Springer International Publishing, 3–33. https://doi.org/10.1007/978-3-319-78375-8_1

[122] Sebastiano Peluso, Alexandru Turcu, Roberto Palmieri, Giuliano Losa, and Binoy Ravindran. 2016. Making Fast Consensus Generally Faster. In *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)* (2016-06). 156–167. https://doi.org/10.1109/DSN.2016.23

[123] Sandro Pinto and Nuno Santos. 2019. Demystifying Arm TrustZone: A Comprehensive Survey. *ACM Comput. Surv.* 51, 6, Article 130 (Jan. 2019), 36 pages. https://doi.org/10.1145/3291047

[124] O. Rütti, Z. Milosevic, and A. Schiper. 2010. Generic construction of consensus algorithms for benign and Byzantine faults. In *2010 IEEE/IFIP International Conference on Dependable Systems Networks (DSN)*. 343–352. https://doi.org/10.1109/DSN.2010.5544299

[125] Fred B. Schneider. [n.d.]. The State Machine Approach: A Tutorial.

[126] Victor Shoup. 2000. Practical Threshold Signatures. In *Proceedings of the 19th International Conference on Theory and Application of Cryptographic Techniques* (Bruges, Belgium) *(EUROCRYPT'00)*. Springer-Verlag, Berlin, Heidelberg, 207–220. http://dl.acm.org/citation.cfm?id=1756169.1756190

[127] J. Sousa and A. Bessani. 2012. From Byzantine Consensus to BFT State Machine Replication: A Latency-Optimal Transformation. In *2012 Ninth European Dependable Computing Conference*. 37–48. https://doi.org/10.1109/EDCC.2012.32

[128] J. Sousa and A. Bessani. 2015. Separating the WHEAT from the Chaff: An Empirical Design for Geo-Replicated State Machines. In *2015 IEEE 34th Symposium on Reliable Distributed Systems (SRDS)*. 146–155. https://doi.org/10.1109/SRDS.2015.40

[129] Chrysoula Stathakopoulou, Tudor David, and Marko Vukolic. 2019. Mir-BFT: High-Throughput BFT for Blockchains. *CoRR* abs/1906.05552 (2019). arXiv:1906.05552 http://arxiv.org/abs/1906.05552

[130] C Stathakopoulous and Christian Cachin. 2017. Threshold signatures for blockchain systems. *Swiss Federal Institute of Technology* (2017).

[131] Stephan van Schaik, Andrew Kwong, Daniel Genkin, and Yuval Yarom. 2020. SGAxe: How SGX Fails in Practice. https://sgaxe.com/.

[132] Stephan van Schaik, Marina Minkin, Andrew Kwong, Daniel Genkin, and Yuval Yarom. 2020. CacheOut: Leaking Data on Intel CPUs via Cache Evictions. arXiv:2006.13353 [cs.CR]

[133] Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, and Lau Cheuk Lung. 2009. Spin One's Wheels? Byzantine Fault Tolerance with a Spinning Primary. In *Proceedings of the 2009 28th IEEE International Symposium on Reliable Distributed Systems (SRDS '09)*. IEEE Computer Society, Washington, DC, USA, 135–144. https://doi.org/10.1109/SRDS.2009.36

[134] Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, and Lau Cheuk Lung. 2010. EBAWA: Efficient Byzantine Agreement for Wide-Area Networks. In *Proceedings of the 2010 IEEE 12th International Symposium on High-Assurance Systems Engineering (HASE '10)*. IEEE Computer Society, Washington, DC, USA, 10–19. https://doi.org/10.1109/HASE.2010.19

[135] Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, Lau Cheuk Lung, and Paulo Verissimo. 2013. Efficient Byzantine Fault-Tolerance. *IEEE Trans. Comput.* 62, 1 (Jan. 2013), 16–30. https://doi.org/10.1109/TC.2011.221

[136] Gauthier Voron and Vincent Gramoli. 2019. Dispel: Byzantine SMR with Distributed Pipelining. *CoRR* abs/1912.10367 (2019). arXiv:1912.10367 http://arxiv.org/abs/1912.10367

[137] J.H. Wensley, L. Lamport, J. Goldberg, M.W. Green, K.N. Levitt, P.M. Melliar-Smith, R.E. Shostak, and C.B. Weinstock. 1978. SIFT: Design and analysis of a fault-tolerant computer for aircraft control. *Proc. IEEE* 66, 10 (Oct 1978), 1240–1255. https://doi.org/10.1109/PROC.1978.11114

[138] James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas Anderson. 2015. Verdi: A Framework for Implementing and Formally Verifying Distributed Systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA) *(PLDI '15)*. Association for Computing Machinery, New York, NY, USA, 357–368. https://doi.org/10.1145/2737924.2737958

[139] Gavin Wood et al. 2014. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper* 151, 2014 (2014), 1–32.

[140] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan Gueta, and Ittai Abraham. 2019. HotStuff: BFT Consensus with Linearity and Responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing* (Toronto ON, Canada) *(PODC '19)*. Association for Computing Machinery, New York, NY, USA, 347–356. https://doi.org/10.1145/3293611.3331591

[141] Yunhao Zhang, Srinath Setty, Qi Chen, Lidong Zhou, and Lorenzo Alvisi. 2020. Byzantine Ordered Consensus without Byzantine Oligarchy. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 633–649. https://www.usenix.org/conference/osdi20/presentation/zhang-yunhao

[142] Hanyu Zhao, Quanlu Zhang, Zhi Yang, Ming Wu, and Yafei Dai. 2018. SDPaxos: Building Efficient Semi-Decentralized Geo-Replicated State Machines. In *Proceedings of the ACM Symposium on Cloud Computing* (Carlsbad, CA, USA) *(SoCC '18)*. Association for Computing Machinery, New York, NY, USA, 68–81. https://doi.org/10.1145/3267809.3267837

Appendices

Appendix A

TLA+ Specification of ezBFT

MODULE *ezBFT*

EXTENDS *Naturals*, *FiniteSets*, *TLC*

$Max(S) \triangleq$ IF $S = \{\}$ THEN 0 ELSE CHOOSE $i \in S : \forall\, j \in S : j \leq i$

Constant parameters:
Commands: the set of all possible commands
Replicas: the set of all *EPaxos* replicas
FastQuorums(r): the set of all fast quorums where r is a command leader
SlowQuorums(r): the set of all slow quorums where r is a command leader

CONSTANTS *Commands*, *Clients*, *GoodReplicas*, *FakeReplicas*, *FastQuorums*(_),
SlowQuorums(_), *MaxBallot*

$Replicas \triangleq GoodReplicas \cup FakeReplicas$

ASSUME $IsFiniteSet(Replicas)$

Quorum conditions: (simplified)

ASSUME $\forall\, r \in Replicas$:

$\land\, SlowQuorums(r) \subseteq$ SUBSET $Replicas$

$\land\, \forall\, SQ \in SlowQuorums(r)$:

$\land\, r \in SQ$

$\land\, Cardinality(SQ) = ((Cardinality(Replicas) * 2) + 1) \div 3$

ASSUME $\forall\, r \in Replicas$:

$\land\, FastQuorums(r) \subseteq$ SUBSET $Replicas$

$\land \forall\, FQ \in FastQuorums(r) :$

$\land\ r \in FQ$

$\land\ Cardinality(FQ) = Cardinality(Replicas)$

Special none command

$none \triangleq \text{CHOOSE } c : c \notin Commands$

The instance space

$Instances \triangleq Replicas \times (1 \,..\, Cardinality(Commands))$

The possible status of a command in the log of a replica.

$Status \triangleq \{\text{“not-seen”}, \text{“spec-ordered”}, \text{“committed”}\}$

All possible protocol messages:

$SpecOrderMessage \triangleq$

$[type : \{\text{“spec-order”}\},\ src : Replicas,\ dst : Replicas,$

$inst : Instances,\ ballot : Nat \times Replicas,$

$cmd : Commands \cup \{none\},\ deps : \text{SUBSET } Instances,\ seq : Nat]$

$FixOrderMessage \triangleq$

$[type : \{\text{“fix-order”}\},\ src : Clients,\ dst : Replicas,$

$inst : Instances,\ ballot : Nat \times Replicas,$

$cmd : Commands \cup \{none\},\ deps : \text{SUBSET } Instances,\ seq : Nat]$

$SpecOrderReplyMessage \triangleq$

$[type : \{\text{“spec-order-reply”}\},\ src : Replicas,\ dst : Clients,$

$inst\ : Instances,\ ballot : Nat \times Replicas,$

$deps : \text{SUBSET } Instances,\ seq : Nat,\ committed : \text{SUBSET } Instances]$

$FixOrderReplyMessage \triangleq$

$[type : \{\text{“fix-order-reply”}\},\ src : Replicas,\ dst : Clients,$

$inst : Instances,\ ballot : Nat \times Replicas]$

$Message \triangleq$

$SpecOrderMessage$

$\cup$ $FixOrderMessage$

$\cup$ $SpecOrderReplyMessage$

$\cup$ $FixOrderReplyMessage$

Variables:

cmdLog = the commands log at each replica

proposed = command that have been proposed

executed = the log of executed commands at each replica

sentMsg = sent (but not yet received) messages

crtInst = the next instance available for a command leader

leaderOfInst = the set of instances each replica has started but not yet finalized

committed = maps commands to set of commit attributs tuples

ballots = largest ballot number used by any replica

VARIABLES $cmdLog,\ proposed,\ executed,\ sentMsg,\ crtInst,\ leaderOfInst,$

$committed,\ ballots$

$TypeOK \triangleq$

$\land$ $cmdLog \in [Replicas \to \text{SUBSET } [inst : Instances,$

$status : Status,$

$ballot : Nat \times Replicas,$

$cmd : Commands \cup \{none\},$

$deps : \text{SUBSET } Instances,$

$seq : Nat]]$

$\land$ $proposed \in \text{SUBSET } Commands$

$\land$ $executed \in [Replicas \to \text{SUBSET } (Nat \times Commands)]$

$\land$ $sentMsg \in \text{SUBSET } Message$

$\land$ $crtInst \in [Replicas \to Nat]$

$$
\begin{aligned}
&\land \quad leaderOfInst \in [Replicas \to \text{SUBSET } Instances] \\
&\land \quad committed \in [Instances \to \text{SUBSET } ((Commands \cup \{none\}) \times \\
&\qquad\qquad\qquad\qquad (\text{SUBSET } Instances) \times \\
&\qquad\qquad\qquad\qquad Nat)] \\
&\land \quad ballots \in Nat
\end{aligned}
$$

$$
\begin{aligned}
vars \triangleq\ & \langle cmdLog,\ proposed,\ executed,\ sentMsg,\ crtInst,\ leaderOfInst, \\
& committed,\ ballots\rangle
\end{aligned}
$$

Initial state predicate

$$
\begin{aligned}
&Init \triangleq \\
&\quad \land\ sentMsg = \{\} \\
&\quad \land\ cmdLog = [r \in Replicas \mapsto \{\}] \\
&\quad \land\ proposed = \{\} \\
&\quad \land\ executed = [r \in Replicas \mapsto \{\}] \\
&\quad \land\ crtInst = [r\ \in Replicas \mapsto 1] \\
&\quad \land\ leaderOfInst = [r \in Replicas \mapsto \{\}] \\
&\quad \land\ committed = [i \in Instances \mapsto \{\}] \\
&\quad \land\ ballots = 1
\end{aligned}
$$

Actions

$$
\begin{aligned}
&StartPhase1(C,\ client,\ cleader,\ Q,\ inst,\ ballot,\ oldMsg) \triangleq \\
&\quad \text{LET } newDeps \triangleq \{rec.inst : rec \in cmdLog[cleader]\} \\
&\qquad newSeq \triangleq 1 + Max(\{t.seq : t \in cmdLog[cleader]\}) \\
&\qquad oldRecs \triangleq \{rec \in cmdLog[cleader] : rec.inst = inst\} \\
&\qquad instCom \triangleq \{t.inst : t \in \{tt \in cmdLog[cleader] : \\
&\qquad\qquad\qquad\qquad tt.status \in \{\text{“committed”},\ \text{“executed”}\}\}\}\text{IN} \\
&\qquad \land\ cmdLog' = [cmdLog \text{ EXCEPT } ![cleader] = (@ \setminus oldRecs) \cup \\
&\qquad\qquad\qquad\qquad \{[inst \mapsto inst,
\end{aligned}
$$

$$
\begin{array}{l}
\quad\quad\quad

$$
\begin{array}{l}
\quad \land (\exists\, Q \in \mathit{FastQuorums}(\mathit{cleader}) : \\
\qquad \mathit{StartPhase}1(C,\ \mathit{client},\ \mathit{cleader},\ Q,\ \mathit{newInst},\ \mathit{newBallot},\ \{\})) \\
\quad \land\ \mathit{crtInst}' = [\mathit{crtInst}\ \text{EXCEPT}\ ![\mathit{cleader}] = @ + 1] \\
\quad \land\ \text{UNCHANGED}\ \langle \mathit{executed},\ \mathit{committed},\ \mathit{ballots}\rangle
\end{array}
$$

$$
\begin{array}{l}
\mathit{Phase}1\mathit{Reply}(\mathit{replica},\ Q,\ \mathit{client}) \triangleq \\
\land \exists\, \mathit{msg} \in \mathit{sentMsg} : \\
\quad \land\ \mathit{msg.type} = \text{“spec-accept”} \\
\quad \land\ \mathit{msg.dst} = \mathit{replica} \\
\quad \land\ \text{LET}\ \mathit{oldRec} \triangleq \{\mathit{rec} \in \mathit{cmdLog}[\mathit{replica}] : \mathit{rec.inst} = \mathit{msg.inst}\}\ \text{IN} \\
\qquad \land\ (\forall\, \mathit{rec} \in \mathit{oldRec} : \\
\qquad\quad (\mathit{rec.ballot} = \mathit{msg.ballot} \lor \mathit{rec.ballot}[1] < \mathit{msg.ballot}[1])) \\
\qquad \land\ \text{LET}\ \mathit{newDeps} \triangleq \mathit{msg.deps} \cup \\
\qquad\qquad\qquad (\{t.\mathit{inst} : t \in \mathit{cmdLog}[\mathit{replica}]\} \setminus \{\mathit{msg.inst}\}) \\
\qquad\quad \mathit{newSeq} \triangleq \mathit{Max}(\{\mathit{msg.seq}, \\
\qquad\qquad\qquad 1 + \mathit{Max}(\{t.\mathit{seq} : t \in \mathit{cmdLog}[\mathit{replica}]\})\}) \\
\qquad\quad \mathit{instCom} \triangleq \{t.\mathit{inst} : t \in \{\mathit{tt} \in \mathit{cmdLog}[\mathit{replica}] : \\
\qquad\qquad\qquad \mathit{tt.status} \in \{\text{“committed”}, \text{“executed”}\}\}\}\ \text{IN} \\
\qquad\quad \land\ \mathit{cmdLog}' = [\mathit{cmdLog}\ \text{EXCEPT}\ ![\mathit{replica}] = (@ \setminus \mathit{oldRec}) \cup \\
\qquad\qquad\qquad\qquad \{[\mathit{inst} \mapsto \mathit{msg.inst}, \\
\qquad\qquad\qquad\qquad\ \ \mathit{status} \mapsto \text{“spec-ordered”}, \\
\qquad\qquad\qquad\qquad\ \ \mathit{ballot} \mapsto \mathit{msg.ballot}, \\
\qquad\qquad\qquad\qquad\ \ \mathit{cmd} \mapsto \mathit{msg.cmd}, \\
\qquad\qquad\qquad\qquad\ \ \mathit{deps} \mapsto \mathit{newDeps}, \\
\qquad\qquad\qquad\qquad\ \ \mathit{seq} \mapsto \mathit{newSeq}]\}] \\
\qquad\quad \land\ \mathit{sentMsg}' = (\mathit{sentMsg} \setminus \{\mathit{msg}\}) \cup \\
\qquad\qquad\qquad\qquad \{[\mathit{type} \mapsto \text{“spec-order-reply”}, \\
\qquad\qquad\qquad\qquad\ \ \mathit{src} \mapsto \mathit{replica},
\end{array}
$$

$$
\begin{aligned}
&\quad dst \;\mapsto client,\\
&\quad inst \;\mapsto msg.inst,\\
&\quad ballot \mapsto msg.ballot,\\
&\quad deps \;\mapsto newDeps,\\
&\quad seq \;\mapsto newSeq,\\
&\quad committed \mapsto instCom]\}\\
&\wedge \text{UNCHANGED } \langle proposed,\ crtInst,\ executed,\ leaderOfInst,\\
&\qquad committed,\ ballots\rangle
\end{aligned}
$$

$$
\begin{aligned}
&Phase1Fast(client,\ cleader,\ i,\ Q) \triangleq\\
&\quad \wedge i \in leaderOfInst[cleader]\\
&\quad \wedge Q \in FastQuorums(cleader)\\
&\quad \wedge \text{LET } replies \triangleq \{msg \in sentMsg :\\
&\qquad\qquad \wedge msg.inst = i\\
&\qquad\qquad \wedge msg.type = \text{“spec-order-reply”}\\
&\qquad\qquad \wedge msg.dst = client\\
&\qquad\qquad \wedge msg.src \in Q\\
&\qquad\qquad \wedge msg.ballot = \langle 0,\ cleader\rangle\}\text{IN}\\
&\qquad \wedge\ (\forall\, replica \in (Q \setminus \{cleader\}) :\\
&\qquad\qquad \exists\, msg \in replies : msg.src = replica)\\
&\qquad \wedge\ (\forall\, r1,\ r2 \in replies :\\
&\qquad\qquad \wedge r1.deps = r2.deps\\
&\qquad\qquad \wedge r1.seq = r2.seq)\\
&\qquad \wedge\ \text{UNCHANGED } \langle proposed,\ executed,\ crtInst,\ ballots,\ sentMsg,\\
&\qquad\qquad leaderOfInst,\ committed\rangle
\end{aligned}
$$

$$
\begin{aligned}
&Phase1Slow(client,\ cleader,\ i,\ Q,\ ballot) \triangleq\\
&\quad \wedge i \in leaderOfInst[cleader]\\
&\quad \wedge Q \in SlowQuorums(cleader)
\end{aligned}
$$

$$
\begin{array}{l}
\quad \wedge \text{LET } \mathit{replies} \triangleq \{\mathit{msg} \in \mathit{sentMsg} : \\
\qquad\qquad \wedge \mathit{msg.inst} = i \\
\qquad\qquad \wedge \mathit{msg.type} = \text{“spec-order-reply”} \\
\qquad\qquad \wedge \mathit{msg.dst} = \mathit{cleader} \\
\qquad\qquad \wedge \mathit{msg.src} \in Q \\
\qquad\qquad \wedge \mathit{msg.ballot} = \mathit{ballot}\}\text{IN} \\
\quad\ \wedge\ (\forall\, \mathit{replica} \in (Q \setminus \{\mathit{cleader}\}) : \exists\, \mathit{msg} \in \mathit{replies} : \mathit{msg.src} = \mathit{replica}) \\
\quad\ \wedge\ \text{LET } \mathit{finalDeps} \triangleq \text{UNION } \{\mathit{msg.deps} : \mathit{msg} \in \mathit{replies}\} \\
\qquad\ \mathit{finalSeq} \triangleq \mathit{Max}(\{\mathit{msg.seq} : \mathit{msg} \in \mathit{replies}\}) \\
\qquad\ \mathit{rep} \triangleq \text{CHOOSE } \mathit{rep} \in \mathit{replies} \ : \text{TRUEIN} \\
\qquad \wedge\ \exists\, SQ \in \mathit{SlowQuorums}(\mathit{cleader}) : \\
\qquad\quad (\mathit{sentMsg}' = (\mathit{sentMsg} \setminus \mathit{replies}) \cup \\
\qquad\qquad\qquad [\mathit{type} : \{\text{“fix-order”}\}, \\
\qquad\qquad\qquad \mathit{src} : \{\mathit{cleader}\}, \\
\qquad\qquad\qquad \mathit{dst} : SQ \setminus \{\mathit{cleader}\}, \\
\qquad\qquad\qquad \mathit{inst} : \{i\}, \\
\qquad\qquad\qquad \mathit{ballot} : \{\mathit{ballot}\}, \\
\qquad\qquad\qquad \mathit{cmd} : \{\mathit{rep.cmd}\}, \\
\qquad\qquad\qquad \mathit{deps} : \{\mathit{finalDeps}\}, \\
\qquad\qquad\qquad \mathit{seq} : \{\mathit{finalSeq}\}]) \\
\qquad \wedge\ \text{UNCHANGED } \langle \mathit{cmdLog},\, \mathit{proposed},\, \mathit{executed},\, \mathit{crtInst},\, \mathit{leaderOfInst}, \\
\qquad\qquad\qquad\qquad \mathit{committed},\, \mathit{ballots}\rangle
\end{array}
$$

$$
\begin{array}{l}
\mathit{Phase2Reply}(\mathit{replica},\, \mathit{client}) \triangleq \\
\quad \exists\, \mathit{msg} \in \mathit{sentMsg} : \\
\qquad \wedge\ \mathit{msg.type} = \text{“fix-order”} \\
\qquad \wedge\ \mathit{msg.dst} = \mathit{replica} \\
\qquad \wedge\ \text{LET } \mathit{oldRec} \triangleq \{\mathit{rec} \in \mathit{cmdLog}[\mathit{replica}] : \mathit{rec.inst} = \mathit{msg.inst}\}\text{IN} \\
\qquad\quad \wedge\ (\forall\, \mathit{rec} \quad \in \mathit{oldRec} : (\mathit{rec.ballot} = \mathit{msg.ballot} \vee
\end{array}
$$

$$rec.ballot[1] < msg.ballot[1]))$$

$$\land \; cmdLog' = [cmdLog \text{ EXCEPT } ![replica] = (@ \setminus oldRec) \cup$$
$$\{[inst \;\mapsto msg.inst,$$
$$status \mapsto \text{“committed”},$$
$$ballot \mapsto msg.ballot,$$
$$cmd \;\mapsto msg.cmd,$$
$$deps \;\mapsto msg.deps,$$
$$seq \;\mapsto msg.seq]\}]$$

$$\land \; sentMsg' = (sentMsg \setminus \{msg\}) \cup$$
$$\{[type \;\mapsto \text{“fix-order-reply”},$$
$$src \;\mapsto replica,$$
$$dst \;\mapsto msg.src,$$
$$inst \;\mapsto msg.inst,$$
$$ballot \mapsto msg.ballot]\}$$

$$\land \; \text{UNCHANGED } \langle proposed,\ crtInst,\ executed,\ leaderOfInst,$$
$$committed,\ ballots\rangle$$

$$Phase2Finalize(client,\ cleader,\ i,\ Q,\ ballot) \triangleq$$

$$\land\, i \in leaderOfInst[cleader]$$

$$\land\, Q \in SlowQuorums(cleader)$$

$$\land \text{ LET } replies \triangleq \{msg \in sentMsg :$$
$$\land\, msg.inst = i$$
$$\land\, msg.type = \text{“fix-order-reply”}$$
$$\land\, msg.dst = cleader$$
$$\land\, msg.src \in Q$$
$$\land\, msg.ballot = ballot\} \text{ IN}$$

$$\land \; (\forall\, replica \in (Q \setminus \{cleader\}) : \exists\, msg \in replies :$$
$$msg.src = replica)$$

$\land$ LET $rep \triangleq$ CHOOSE $rep \in replies$: TRUE IN
$committed' = [committed$ EXCEPT $![cleader] = @ \cup$
$\{\langle rep.cmd,\ rep.deps,\ rep.seq\rangle\}]$
$\land\ leaderOfInst' = [leaderOfInst$ EXCEPT $![cleader] = @ \setminus \{i\}]$
$\land$ UNCHANGED $\langle cmdLog,\ sentMsg,\ proposed,\ executed,\ crtInst,\ ballots\rangle$

Action groups

$CommandLeaderAction \triangleq$
$\exists\, C \in (Commands \setminus proposed),\ client \in Clients,$
$cleader \in Replicas :$
$\land\ Propose(C,\ client,\ cleader)$

$ReplicaAction \triangleq$
$\exists\, replica \in Replicas : \exists\, client \in Clients :$
$\exists\, Q \in FastQuorums(replica) :$
$(\lor\ Phase1Reply(replica,\ Q,\ client)$
$\lor\ Phase2Reply(replica,\ client))$

$FakeReplicaAction \triangleq$
$\exists\, replica \in FakeReplicas :$
$\land \exists\, m \in \{mm \in SpecOrderReplyMessage \cup FixOrderReplyMessage :$
$mm.src = replica\} : sentMsg' = (sentMsg \cup \{m\})$
$\land$ UNCHANGED $\langle cmdLog,\ proposed,\ executed,\ crtInst,\ leaderOfInst,$
$committed,\ ballots\rangle$

$ClientAction \triangleq$
$\exists\, client \in Clients : \exists\, replica \in Replicas :$
$\exists\, inst \in leaderOfInst[replica] : \exists\, ballot \in \{0\ ..\ MaxBallot\} :$
$\lor (\exists\, Q \in FastQuorums(replica) :$
$Phase1Fast(client,\ replica,\ inst,\ Q))$
$\lor (\exists\, Q \in SlowQuorums(replica) :$

$$Phase1Slow(client,\ replica,\ inst,\ Q,\ \langle replica,\ ballot\rangle)))$$
$$\lor\ (\exists\, Q \in SlowQuorums(replica) :$$
$$Phase2Finalize(client,\ replica,\ inst,\ Q,\ \langle replica,\ ballot\rangle)))$$

Next action

$$Next \triangleq$$
$$\lor\ CommandLeaderAction$$
$$\lor\ ReplicaAction$$
$$\lor\ ClientAction$$
$$\lor\ FakeReplicaAction$$

The complete definition of the algorithm

$$Spec \triangleq Init \land \Box[Next]_{vars}$$

Theorems

$$Nontriviality \triangleq$$
$$\forall\, i \in Instances :$$
$$\Box(\forall\, C \in committed[i] : C \in proposed \lor C = none)$$

$$Stability \triangleq$$
$$\forall\, replica \in Replicas :$$
$$\forall\, i \in Instances :$$
$$\forall\, C \in Commands :$$
$$\Box((\exists\, rec1 \in cmdLog[replica] :$$
$$\land\ rec1.inst = i$$
$$\land\ rec1.cmd = C$$
$$\land\ rec1.status \in \{\text{“committed”},\ \text{“executed”}\}) \implies$$
$$\Box(\exists\, rec2 \in cmdLog[replica] :$$
$$\land\ rec2.inst = i$$
$$\land\ rec2.cmd = C$$

$\land\ rec2.status \in \{\text{“committed”}, \text{“executed”}\}))$

$Consistency \triangleq$

$\forall\, i \in Instances :$

$\Box(Cardinality(committed[i]) \leq 1)$

THEOREM $Spec \implies (\Box\, TypeOK) \land Nontriviality \land Stability \land Consistency$

\ * Modification History
\ * Last modified *Mon Oct* 15 16:37:08 *EDT* 2018 by *balaji*