Resilient and Scalable Architecture for Permissioned Blockchain Fabrics

By

SUYASH GUPTA
DISSERTATION

Submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

Computer Science

in the

OFFICE OF GRADUATE STUDIES

of the

UNIVERSITY OF CALIFORNIA

DAVIS

Approved:

2021

To my father,
Prof. Santosh Kumar,
who always dreamt of this day,
but could not see it getting fulfilled.

Contents

Abstract

Large-scale distributed databases are designed and deployed for handling commercial and cloud-based applications. The general expectation from these applications is to provide consistent and reliable services even in the presence of failures. This interest in fault-tolerant distributed applications has given rise to *blockchain* technology.

In this work, we study a *key* challenge for existing distributed and blockchain applications– agreement among the participating servers. We *first* look at the design of *commitment* protocols, which can handle failures of server nodes. These commitment protocols help in ensuring that either all the changes of a client request are applied or none of them exist. To ensure an efficient commitment process, the database community has mainly used the two-phase commit (2PC) protocol. However, the 2PC protocol is blocking under multiple failures. This necessitated the development of the non-blocking, three-phase commit (3PC) protocol. However, the database community is still reluctant to use the 3PC protocol, as it acts as a scalability bottleneck in the design of efficient transaction processing systems. In this work, we present EasyCommit protocol, which leverages the best of both worlds (2PC and 3PC). EC is non-blocking (like 3PC) and requires two phases (like 2PC). EasyCommit achieves these goals by ensuring two key observations: (i) first transmit and then commit, and (ii) message redundancy. We present the design of the EasyCommit protocol and prove that it guarantees both safety and liveness. We also present a detailed evaluation of the EC protocol and show that it is nearly as efficient as the 2PC protocol. To cater to the needs of geographically large scale distributed systems, we also design a topology-aware agreement protocol (Geo-scale EasyCommit) that is non-blocking, safe, live and outperforms 3PC protocol.

We next move beyond node failures and analyze systems where nodes can be byzantine. Prior works have employed a *replicated* system, where each participating server is a replica of the other, to handle byzantine failures. Our *second* work aims at designing a *byzantine fault-tolerant consensus* protocol that is both efficient and secure. Existing BFT algorithms face the following challenges: (i) they are communication expensive (require three phases of quadratic complexity), (ii) require a large number of replicas, (iii) depend on clients, and (iv) need trusted components.

To resolve these challenges, we present the Proof-of-Execution (PoE) consensus protocol. At the core of PoE are *out-of-order* processing and *speculative execution*, which allow PoE to execute

transactions before consensus is reached among the replicas. With these techniques, PoE manages to reduce the costs of BFT in normal cases, while guaranteeing reliable consensus for clients in all cases. We envision the use of PoE in high-throughput multi-party data-management and blockchain systems.

PoE and a majority of other BFT protocols adhere to a common design paradigm, the *primary-backup model*, which limits the throughput of these systems to the capabilities of a single replica (the primary). To push throughput beyond this single-replica limit, we propose *concurrent consensus.* In concurrent consensus, replicas independently propose transactions, thereby reducing the influence of any single replica on performance. To put this idea in practice, we propose our RCC paradigm that can turn any primary-backup consensus protocol into a *concurrent consensus protocol* by running many consensus instances concurrently. RCC is designed with performance in mind and requires minimal coordination between instances. Furthermore, RCC also promises increased resilience against failures.

To evaluate our scalable BFT protocols, we design ResilientDB, a high-throughput yielding permissioned blockchain fabric. ResilientDB is a result of a key intuition, *can a well-crafted system based on a classical* BFT *protocol outperform a modern protocol?* Our ResilientDB fabric proves that designing such a well-crafted system is possible, and even if such a system employs a three-phase protocol, it can outperform another system utilizing a single-phase protocol. This endeavor requires us to dissect existing permissioned blockchain systems and highlight different factors affecting their performance. ResilientDB fabric is based on these insights, employs multi-threaded deep pipelines to balance tasks at replicas, and provides guidelines for future designs.

Acknowledgments

At the conclusion of this enduring yet fruitful journey, I would like to take out some moments and thank those who helped me to reach this milestone. While I pen down this section, I remember several names and faces whose actions eased my doctoral ride. Although it is hard to mention everyone, I would like to earnestly thank every person who has ever touched my life for better or worse since I started this doctoral degree.

Foremost, I would like to extend my heartfelt gratitude to my advisor ***Dr. Mohammad Sadoghi***. Often students receive advisors who are good mentors or teaches, but in Dr. Sadoghi, I found a friend. I started my Ph.D. journey with Dr. Sadoghi at Purdue University, and I transferred to the University of California a few months after he moved to Davis. I had the privilege of being his first student at UC Davis, and since then, we have shared several moments of happiness and sadness. He guided me through all the difficult phases of a Ph.D., and his life advice and tips will be with me for years to follow.

Next, I would like to thank ***Dr. Dahlia Malkhi***, who advised me during my internship at Novi Research/Facebook. Dr. Malkhi's research knowledge is well-known among the academic community, and working with her at Novi helped me to gain necessary industrial experience. During my Ph.D., I was also fortunate to collaborate with ***Dr. Faisal Nawab*** and ***Dr. Natacha Crooks***. Their collaborations and guidance have helped in shaping several of my future research directions.

At this time, I would also like to thank ***Dr. Cindy Rubio González*** and ***Dr. Aditya Thakur*** for taking out time and reviewing my thesis. Their fruitful suggestions helped this thesis reach its current form. Further, I would like to extend my gratitude to my other Qualifying Examination Committee members: ***Dr. Jason Lowe-Power*** and ***Prof. Amr El Abbadi***. These committee members made my candidacy examination a learning experience, which gave me new insights into my research projects.

I would like to also to acknowledge the efforts by UC Davis Computer Science Graduate Chair ***Prof. Dipak Ghosal*** and Graduate Staff members ***Jessica Stoller***, ***Alyssa Bates***, and ***Noreen Johnson***. Their continuous efforts made my stay at UC Davis pleasant.

Next, I would like to thank ***Exploratory Systems Lab*** (ExpoLab) members, especially ***Sajjad Rahnama*** and ***Dr. Jelle Hellings***. Sajjad has been a good friend, and we have spent endless days and nights together working on different papers and our pet codebase ResilientDB. From Jelle, I have learned a lot about how to structure my research and formulate my ideas. At ExpoLab, we used to have intense long discussions where we often spent time arguing different ideas and building new research solutions. It is here we laid foundations to our startup ***MokaBlox***. At this point, I should also thank ***Dr. Thamir Qadah***, ***Shubham Pandey***, and ***Rohan Sogani*** for working with me on multiple research projects and bearing my inputs. I would also like to acknowledge around a dozen students that I got a chance to mentor at ExpoLab. Each day I worked with them, I learned new things that helped me develop into a better version of myself.

Last but not least, I would like to thank my friends and family members who have been my strength during this journey. I would like to remember my late father, ***Prof. Santosh Kumar***, who motivated me to start this journey. He dreamed of the day I earned a Ph.D. degree in Computer Science. He was my go-to person for any life-related decision, and his unprecedented departure left a void in my heart. During these trying times, my mother, ***Ms. Manju Gupta***, and my brother ***Shrey Gupta*** bulwarked my career. They shared my sorrows and boosted my happiness. There were moments when I wanted to drop out of the Ph.D., but their unselfish love and patience have helped me to bring this degree to its fulfillment. Finally, I would like to thank my good friend ***Dr. Rashmi Kumar*** who has remained true to the words *a friend in need is a friend indeed.*

CHAPTER 1

Introduction

Large scale distributed databases have been designed and deployed for handling commercial and cloud-based applications [11, 30, 37, 81, 98, 104, 108, 121, 123, 134]. Each such distributed system is a collection of multiple independently operating servers that together provide a single *service*. Such cooperation eases data sharing and improves data quality [39, 75, 118]. The common denominator across all these systems is the use of transactions. A transaction is a sequence of operations that either reads or modifies the data. These transactions are created by *clients* that submit their transactions to the distributed servers for processing.

The general expectation from any distributed application is that it will apply each client transaction in an ACID-compliant manner, such that the state of the underlying database will remain consistent [51]. Further, the database is expected to respect the atomicity boundaries. It does so by guaranteeing that either all the changes suggested by a client transaction will persist or none of the changes will take place. This atomicity guarantee acts as a contract and helps to establish trust among the communicating parties.

It is also common knowledge that distributed systems face node crashes and attacks [66, 109]. Recent failures have shown that the distributed systems are still miles away from achieving undeterred availability [43, 105, 106, 130]. As a result, there is a constant tussle in the systems community to find the appropriate degree of database consistency and availability necessary for achieving maximum system performance. To guarantee strong consistency, prior works have relied on semantics such as serializability [16] and linearizability [74]. However, these semantics have a causal effect on the latency and availability of the associated distributed application. Hence, a key desire is to design strongly consistent systems that are also efficient.

We envision multiple independent yet correlated avenues to realize such a dream. To fruition this dream, our *first* step aims at developing efficient *fault-tolerant agreement protocols* that are susceptible to node failures. Next, we take a step forward and design a secure and efficient protocol that guards against byzantine attacks. Finally, we realize the potential of full-scale parallelization and employ those principles to develop a protocol that achieves high throughput and low latency while guarding against byzantine attacks.

1.1. The Case for Efficient Agreement

A simple way to design a distributed database is to visualize it as a collection of *partitions*, where each partition stores a distinct set of data items. Such visualization is so prevalent in the database community that in the past three decades, several new designs have been proposed to improve consistency and availability of these *partitioned* databases [50, 68, 113, 114, 120, 125]. Moreover, prior works have shown that partitioning the data across servers helps in reducing data-access contention and achieving high system throughput [22, 81, 129]. As a result, several existing applications employ fast *agreement* protocols to guarantee an efficient, strongly consistent view to their clients [2, 11, 30, 50, 81, 96, 127].

A partitioned database receives client transactions that require access to a single partition or multiple partitions. Single partition transactions are easy to handle as they can be executed and committed by the receiving partition (server) without any coordination with other partitions. Multi-partition transactions require coordination or agreement between different partitions as they require access to data at several partitions. This coordination is achieved by employing *commit* protocols, which often bottleneck the maximum permissible system throughput [50, 125].

Existing commit protocols such as *two-phase commit* [50] and *three-phase commit* [125] are expected to tolerate simple node (partition) failures. In the case a participating node fails, the remaining participants take the necessary steps to preserve database consistency. One of the earliest and most popular commitment protocols is the *two-phase commit* (2PC) protocol [50]. As the name suggests, the 2PC protocol achieves agreement between the nodes in two phases. Prior works have illustrated that under specific scenarios, any partitioned database employing the 2PC protocol can lose its availability, that is, the nodes of the system may not progress or can get *blocked* [109, 125]. This led to the design of resilient *three-phase commit* (3PC) protocol [124, 126]. The 3PC protocol introduces an additional phase that makes it *non-blocking.* Evidently, this additional phase also acts as a major performance bottleneck. Further, if the participating nodes are at *geographically-distant* locations, an additional phase substantially increases latency.

We believe the design of a *hybrid* commit protocol, which leverages the best of both worlds (2PC and 3PC), is in order. As a result, we present the ***EasyCommit*** (EC) protocol, which requires two phases of communication and is non-blocking under node failures [56]. We associate two key insights with our design of the EasyCommit protocol. The first insight is to delay the commit (or abort) of the client transaction until the transmission of *final decision* to all the participating nodes. The second insight induces message redundancy in the network by requiring each participating node to forward the *final* decision to all the other participants.

As mentioned earlier, the participating nodes may be spread across geographically-distant locations. In such a case, the protocols like 2PC, 3PC, and EC would incur high latencies. Hence, we group the nodes located nearby in a *cluster* and design a *topology-aware* agreement protocol (henceforth referred as ***Geo-scale EasyCommit*** or GEC) to achieve agreement between the clusters of nodes [58]. Our evaluation illustrates that our GEC protocol is arguably more efficient than either of the 2PC or 3PC protocols.

1.2. The Case for Efficient Consensus

An efficient commit protocol ensures a common decision among the multiple partitions of a distributed database, even if some of the partitions suffer failures. However, these commit protocols cannot tolerate byzantine attacks or an unreliable network. It is common for nodes to act byzantine and work together to disrupt the agreement process. Further, the network could become unreliable, and messages could get delayed, duplicated, or lost. To handle such attacks, we need protocols that are more resilient than the traditional commit protocols.

The first step in guaranteeing resilience against byzantine attacks is *replication.* In a replicated database, each node holds a copy of the database, that is, each node is a replica of any other node in the system. As all the nodes are replicas, so there is no notion of multi-partition transactions. Each client transaction can be processed by any replica. However, to ensure database consistency, we need to ensure that all the replicas have the same state. As a result, each update to the database needs to be applied to all the replicas in the same order. To achieve this task, replicated databases run a consensus protocol, which tries to reach an agreement among the replicas. Further, as some of the replicas could be malicious, this consensus protocol should thwart any byzantine attack.

A majority of *byzantine fault-tolerant* (BFT) consensus protocols follow the *primary-backup* model [21, 57, 66, 87, 109]. In a *primary-backup* BFT consensus protocol, one replica acts as the *primary* and initiates the consensus, while other replicas act as backups and follow the protocol. PBFT is often credited for presenting the design of the first practical BFT consensus protocol [21]. PBFT achieves BFT consensus among the replicas in three phases, of which two require quadratic communication complexity. As a result, several works have presented exciting optimizations to reduce the costs associated with the PBFT protocol [1, 48, 66, 87, 133]. These rendered protocols have suggested: (i) increasing the number of replicas beyond $3f + 1$ [1], (ii) employing twin-paths or requiring clients to order queries [48, 87], and (iii) using additional trusted components [24, 133]. However, neither these optimizations sufficiently guard against attacks nor do they guarantee sustainable system throughput [10, 26].

To resolve this challenge, we design a novel consensus algorithm – Proof-of-Execution (PoE)‘[65]. PoE leverages speculative execution to reduce a quadratic communication phase from PBFT's design. Further, PoE provides a flexible design that helps to employ advanced cryptographic practices to yield a linear and resilient consensus. Our evaluation of PoE illustrates that PoE outperforms all the state-of-the-art BFT consensus protocols.

1.3. The Case for Parallel Consensus

All the primary-backup protocols share a common principle that the underlying consensus is led by the primary. This primary replica is responsible for initiating the consensus protocol for each incoming client request. For any client request, if the primary replica *fails* to ensure consensus, then all the backup replicas work together to *replace* this primary. This replacement process is necessary as it allows the non-faulty replicas to converge to a common state. Unfortunately, primary replacement is not cheap, as it requires pausing consensus on all outstanding requests until the primary is replaced.

A promising solution to all these problems is to make existing BFT consensus protocols primary agnostic. Such a solution would require us to give all replicas the power to act as a primary.

This brings us to the design of our *Resilient Concurrent Consensus* (RCC) paradigm [60, 67]. RCC takes as input a primary-backup BFT protocol and *parallelizes its consensus* by requiring each replica to run *multiple instances* of the input protocol in parallel. Further, RCC ensures that each instance is managed by a distinct replica. Hence, the throughput of the system is no longer dependent on one primary as there are multiple primaries. Our RCC paradigm guarantees that the failure of one instance does not affect processing of other instances. Further, RCC facilitates independent recovery of the state from a failed instance. Our evaluation of RCC ascertains the excellent scalability of our paradigm in comparison to the existing state-of-the-art protocols.

1.4. The Promise for Scalable Permissioned Blockchain

Until now, we have discussed the design of two scalable and efficient BFT protocols that can help replicated distributed databases achieve high throughputs. However, it is unclear what are the real-world use cases for these protocols. Recent interests in blockchain technology have illustrated that these BFT protocols lie at the core of any blockchain application [73, 90].

A blockchain in its simplest form is a tamper-proof linked list of blocks, where each block tracks some client transactions. This blockchain is replicated across all the replicas, and the transactions included in each block are decided through a BFT consensus protocol. The key reason blockchain technology has gauged our interest is its ability to act as a resolve to challenges in trade of valuable commodities such as art [20, 23, 112], food production [45], managing land property rights [111], managing identities [9, 20, 111], managing health care data [18, 49, 82] and energy production and energy trading [112].

Blockchain applications are often categorized as permissionless and permissioned [57]. Permissionless applications allow any replica to participate in the BFT consensus, and as a result, participating replicas can hide their identities. Hence, these permissionless applications need to reward replicas with incentives to promote non-byzantine behavior. In this thesis, we focus on the design and use of permissioned blockchain applications. Permissioned blockchain applications require identities of all the replicas to be known prior to the start of consensus. As a result, permissioned blockchain applications can employ traditional BFT consensus protocols to order their client transactions.

Several recent permissioned blockchain systems such as Hyperleder Fabric [8], MultiChain [46], and Tendermint [19] have presented exciting architectural details to achieve high system throughput. For instance, Hyperledger Fabric presents a distinct paradigm of executing transactions first and then ensuring they have a valid order, while Tendermint advocates reliance on synchronous setting to attain higher throughput. Despite these exciting principles, all of these works have missed the low-level system details that can help a permissioned blockchain system extract higher throughputs. We take into account these low-level details and design ResilientDB–a high-throughput yielding permissioned blockchain fabric [55, 59, 61, 62, 64, 115]. ResilientDB focuses on a system-centric design rather than a protocol-centric design and employs age-old database and distributed system principles to yield a scalable architecture. The effectiveness of our ResilientDB framework is evident from the fact that a slow consensus protocol like Pbft, when employed by ResilientDB replicas, outperforms a state-of-the-art permissioned blockchain employing a fast protocol like Zyzzyva.

1.5. The Commencement of Permissioned Blockchain Applications

The design of our efficient ResilientDB fabric and fast bft consensus protocols (PoE and RCC) allows us to apply these principles for designing efficient permissioned blockchain applications. In this thesis, we will illustrate in brief how our existing observations, help us to resolve the following challenges:

(1) **Reduced Replication through Trusted Subsystem.** Until now, all the bft protocols that we have studied allow less than one-third of the replicas to act byzantine. However, it is still desirable for a bft protocol to guard against a larger number of byzantine replicas. To resolve this challenge, prior works have introduced bft protocols that employ trusted components and allow less than half of the replicas to act byzantine [24, 92, 133]. However, these protocols make several assumptions in their design, which prevent their application to real-world setting. We illustrate how these protocols provide limited safety and liveness guarantees, and envision solutions that can eliminate these limitations.

(2) **Resilient Serverless-Edge Applications.** Edge applications have found prominent use cases with the emergence of the Internet of Things applications. These edge applications expect

time-critical response delivery. However, the key components for these edge applications are the edge devices, which have limited computing power. As a result, application developers have to deploy on-premise nodes to process client requests. This creates another challenge as on-premise nodes are hard to scale or maintain. We envision a new model for solving these challenges, the *serverless-edge* model. Existing serverless clouds provide developers access to massive computational resources while freeing them from the tasks of executing the code and managing the resources. However, neither the edge nodes nor the serverless cloud can be trusted by the client. Hence, we present the design of our ServerlessBFT protocol that facilitates the efficient processing of transactions in the serverless-edge infrastructure.

(3) **Efficient consensus of Geo-replicated Database.** It is common for replicated databases to have their replicas spread across a wide-area network [30]. This allows a replicated database to withstand failures that affect one location, while guaranteeing continuous service through replicas at other locations. The key issue for such geographically spread database deployments is that the communication between replicas becomes visibly expensive. In these systems, two or more replicas are often connected by networks that offer low bandwidth and high ping costs. To resolve this challenge, we envision the design of a BFT consensus protocol for geo-replicated databases [63].

CHAPTER 2

Efficient and non-blocking agreement protocols

An intuitive way to design a distributed database is to split its data across multiple servers, such that each server holds a unique partition of data. Such a distributed database is often considered scalable as it permits processing multiple client transactions in parallel by distinct partitions. If each client transaction requires data from only one partition, then there is no need for partitions to communicate with each other, and each partition can act as an individual database. However, in real-world applications, transactions require access to data in multiple partitions [30, 68, 131].

As databases are expected to remain consistent, these multi-partition transactions need to be handled carefully. Prior to deciding the fate of any multi-partition transaction (commit or abort), all the partitions need to reach a common decision. To reach a common decision among the paritions, prior works have presented the design of commit protocols [50, 125]. However, a key point in hindsight is that the use of a commit protocol should not cause an increase in communication latency in the corresponding distributed application.

Transaction commit protocols help in reaching an agreement among the participating nodes when a transaction has to be committed or aborted. To initiate such an agreement, each participating node is asked to *vote* its decision on the operations that accessed data in its partition. Each participating node can decide to either commit or abort the ongoing transaction. Depending on these votes, a common decision is reached.

One of the earliest and most popular commitment protocol is the *two-phase commit* (2PC) [50] Figure 2.1 presents the state diagram [109, 126] representation of the 2PC protocol. This figure shows the set of possible states (and transitions) that a coordinating node[1] and the participating nodes follow, in response to a transaction commit request. We use solid lines to represent the state transitions and dotted lines to represent the inputs/outputs to the system. For instance, the

[1]The coordinating node is the one which initiates the commit protocol, and in this work it is also the node which receives the client request to execute a transaction.

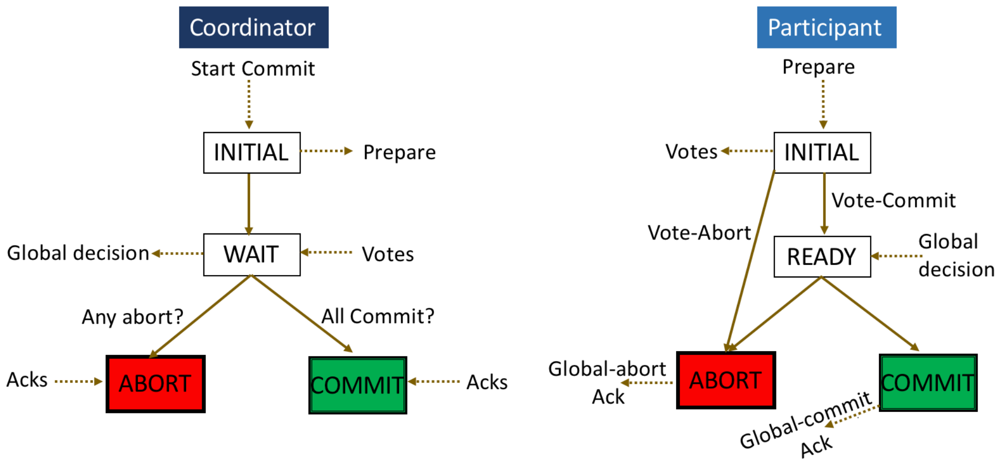


FIGURE 2.1. Two-Phase Commit Protocol

coordinator starts the commit protocol on transaction completion and requests all the participants to commence the same by transmitting *Prepare* messages. In case of multiple failures the two-phase commit protocol has been proved to be blocking [109, 125]. For example, if the coordinator and a participant fail, and if the remaining participants are in the `READY` state, then they cannot make progress (blocked!), as they are unaware about the state of the failed participant. This blocking characteristics of the 2PC protocol endangers database availability, and makes it unsuitable for use with the partitioned databases[2]. The inherent shortcomings of the 2PC protocol led towards the design of resilient *three-phase commit* [124, 126] – henceforth referred as 3PC protocol. The 3PC protocol introduces an additional `PRE-COMMIT` state between the `READY` and `COMMIT` states, which ensures there is no direct transition between the non-committable and committable states. This simple modification makes the 3PC protocol non-blocking under node failures.

However, the 3PC protocol acts as the major performance suppressant in the design of efficient distributed databases. It can be easily observed that the addition of the `PRE-COMMIT` state leads to an extra phase of communication among the nodes. This violates the need of an efficient commit protocol for geo-scale systems. Hence, the design of a *hybrid* commit protocol, which leverages the best of both worlds (2PC and 3PC), is in order. We present the *EasyCommit* (a.k.a EC) protocol, which requires two phases of communication and is non-blocking under node failures. We associate

[2]Partitioned database is the terminology used by the database community to refer to the shared-nothing distributed databases, and should not be intermixed with the term network partitioning.

two key insights with the design of EasyCommit protocol that allow us to achieve the non-blocking characteristic in two phases. The first insight is to delay the commitment of updates to the database until the transmission of global decision to all the participating nodes, and the second insight is to induce message redundancy in the network. EasyCommit protocol introduces message redundancy by ensuring that each participating node forwards the global decision to all the other participants (including the coordinator).

Prior works [4, 103] have illustrated the wide-scale application of geographically large scale systems. Such systems, adhering to the philosophy of partitioned databases, require complex agreement protocols that are both non-blocking and *topology-aware*. The key ingredient to these algorithms is their ability to take advantage to the geo-scale topology and present efficient results. It is important to understand that a simple geo-scale system consists of several clusters, and the communication across each clusters may be limited to few nodes. Hence, the design of 3PC (and even EC) may not reap benefit as it requires communication across all the participating nodes. This motivates us to learn from the design principles of EasyCommit protocol and construct a novel *topology-aware* agreement protocol for the geo-scale systems – *Geo-scale EasyCommit* (GEC).

Specifically, in this chapter, we make the following contributions.

- We present the design of a new two-phase commit protocol (EasyCommit) and show it is non-blocking under node-failures.
- We design an associated termination protocol, to be initiated by the active nodes, on failure of the coordinating node and/or participating nodes.
- We re-use the EasyCommit principles and present a novel agreement protocol that caters to the needs of geographically large scale systems.
- We extend ExpoDB [113, 114] framework to implement the EC protocol and its geo-scale variant. Our implementation can be used seamlessly with various concurrency control algorithms by replacing 2PC protocol with EC (and Geo-scale EasyCommit) to achieve efficient systems.
- We present a detailed evaluation of the EC protocol against the 2PC and 3PC protocol over two different OLTP benchmark suites: YCSB [29] and TPC-C [33], and scale the system upto 64 nodes, on the Microsoft Azure cloud.

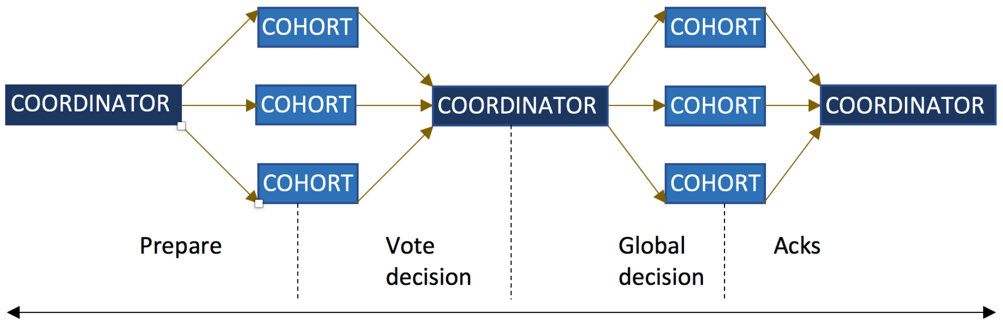


FIGURE 2.2. Time span of 2PC Protocol

- We also present an interesting evaluation of Geo-scale EasyCommit protocol against the 2PC and 3PC protocols when run across *four* geographically distant locations (across three continents). Our evaluation necessitates the need of an efficient and non-blocking geo-scale system.

2.1. Motivation and Background

The state diagram representation for the two-phase commit protocol is presented in Figure 2.1. In 2PC protocol, the coordinator and participating nodes require at most two transitions to traverse from `INITIAL` state to the `COMMIT` or `ABORT` states. We use Figure 2.2 to present the interaction between the coordinator and the participants, on a linear time scale. The 2PC protocol starts with the coordinator node transmitting a *Prepare* message to each of the cohorts [3] and adding a *begin_commit* entry in its log. When a cohort receives the *Prepare* message, it adds a *ready* entry in its log, sends its decision (*Vote-commit* or *Vote-abort*) to the coordinator. If a cohort decides to abort the transaction then it independently moves to the `ABORT` state, else it transits to the `READY` state. The coordinator waits for the decision from all the cohorts. On receiving all the responses, the coordinator analyzes all the votes. If there is a *Vote-abort* decision, then the coordinator adds an *abort* entry in the log, transmits the *Global-Abort* message to all the cohorts and moves to the `ABORT` state. If all the votes are to commit, then the coordinator transmits the *Global-Commit* message to all the cohorts, and moves to `COMMIT` state, after adding a *commit* entry to log. The cohorts on receiving the coordinator decision move to the `ABORT` or `COMMIT` state and add the *abort*

[3]The term cohort refers to a participating node in the transaction commit process. We use these terms interchangeably.

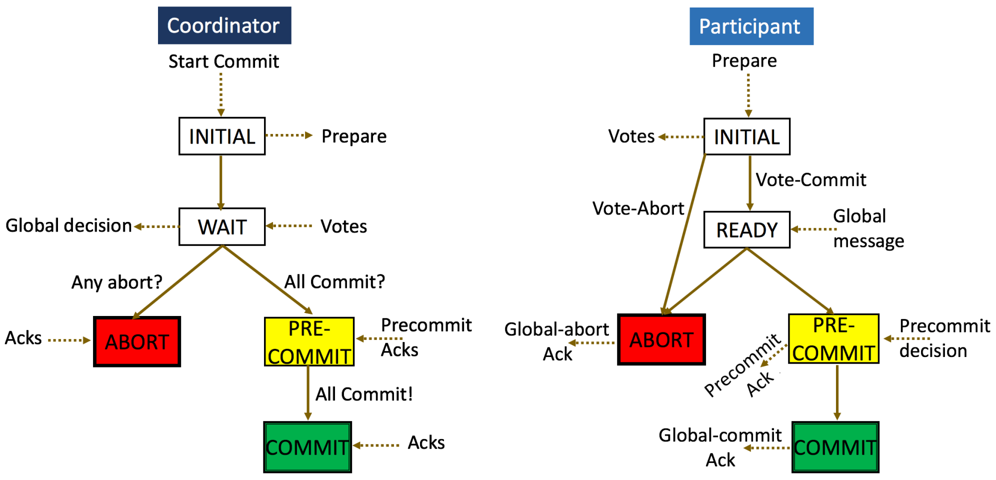


FIGURE 2.3. Three-Phase Commit Protocol

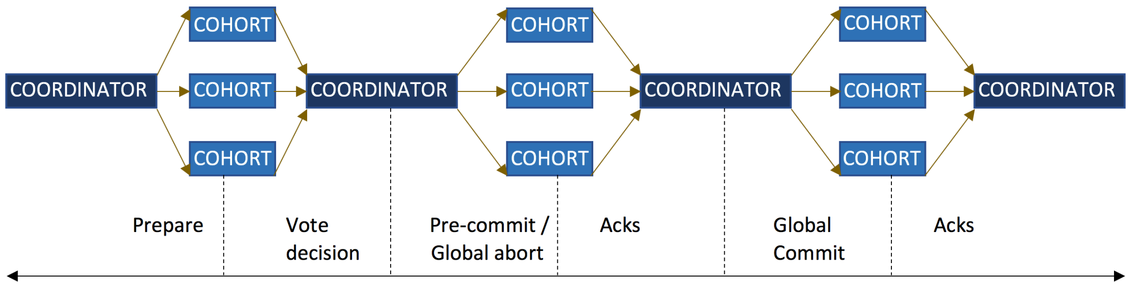


FIGURE 2.4. Time span of 3PC Protocol

or *commit* entry to the log, respectively. Finally, the cohorts acknowledge the global decision, which allows the coordinator to mark the completion of commit protocol.

The 2PC protocol has been proved to be blocking [109, 125] under multiple node failures. To illustrate this behavior let us consider a simple distributed database system with a coordinator C and three participants X, Y and Z. Now assume a snapshot of the system when C received *Vote-commit* from all the participants, and hence, it decides to send *Global-commit* message to all the participants. However, say C fails after transmitting *Global-commit* message to X, but before sending messages to Y and Z. The participant X on receiving the *Global-commit* message, commits the transaction. Now, assume X fails after committing the transaction. On the other hand, nodes Y and Z would *timeout* due to no response from the coordinator and would be blocked indefinitely, as they require node X to reach an agreement. They cannot make progress, as neither they have knowledge of the global decision nor they know the state of node X before failure. This situation can be prevented with the help of the three-phase commit protocol [124, 126].

Figure 2.3 presents the state transition diagram for the coordinator and cohort executing the three-phase commit protocol, while Figure 2.4 expands the 3PC protocol on the linear time scale. In the first phase, the coordinator and the cohorts, perform the same set of actions as in the 2PC protocol. Once the coordinator checks all the votes, it decides whether to abort or commit the transaction. If the decision is to abort, the remaining set of actions performed by the coordinator (and the cohorts) are similar to the 2PC protocol. However, if the coordinator decides to commit the transaction, then it first transmits a *Prepare-to-commit* message and then adds a *pre-commit* entry to the log. The cohorts on receiving the *Prepare-to-commit* message, move to the `PRE-COMMIT` state, add a corresponding *pre-commit* entry to the log and acknowledge the message reception to the coordinator. The coordinator then sends a *Global-commit* message to all the cohorts, and the remaining set of actions are similar to the 2PC protocol.

The key difference between the 2PC and 3PC protocol is the `PRE-COMMIT` state, which makes the latter non-blocking. The design of 3PC protocol is based on the [124]'s design of a non-blocking commit. In his work Skeen laid down two fundamental properties for the design of a non-blocking commit protocol: (i) no state should be adjacent to both the `ABORT` and `COMMIT` states, and (ii) no non-committable[4] state should be adjacent to the `COMMIT` state. These requirements motivated Skeen to introduce the notion of a new committable state (`PRE-COMMIT`) to the 2PC state transition diagram.

The existence of `PRE-COMMIT` state makes the 3PC protocol non-blocking. The aforementioned multi-node failure case does not indefinitely block the nodes Y and Z which are waiting in the `READY` state. The nodes Y and Z can make safe progress (by aborting the transaction) as they are assured that the node X could not have committed the transaction. Such a behavior is implied by the principle that no two nodes could be more than one state transition apart. The node X is guaranteed to be in one of the following states: `INITAL`, `READY`, `PRE-COMMIT` and `ABORT`, at the time of failure. This indicates that node X could not have committed the transaction, as nodes Y and Z are still in the `READY` state (It is important to note that in the 3PC protocol the coordinator sends

[4]`INITAL`, `READY` and `WAIT` states are considered as non-committable states.

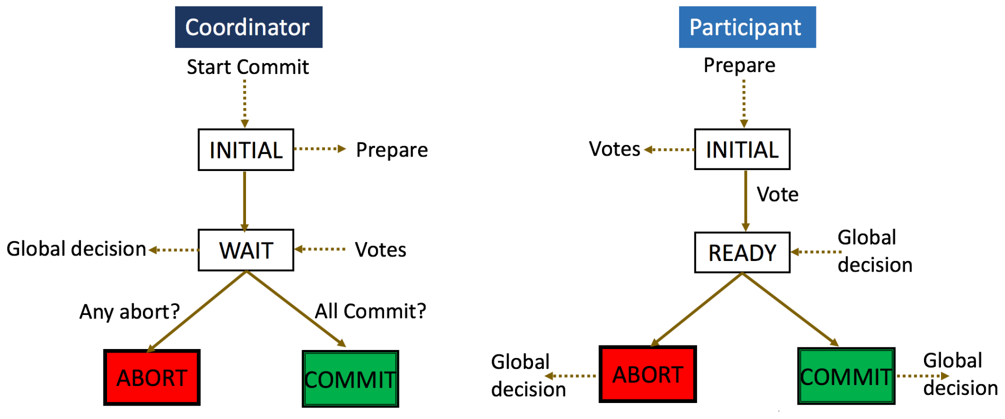


FIGURE 2.5. EasyCommit Protocol

the *Global-commit* message after it transmits the *Prepare-to-commit* message to all the nodes.). Interestingly, if either of nodes Y or Z are in the `PRE-COMMIT` state then they can actually commit the transaction. However, it can be easily observed that the non-blocking characteristic of the 3PC protocol comes at an additional cost, an extra round of handshaking.

2.2. Easy Commit

We now present the *EasyCommit* (*EC*) protocol. EC is a two-phase protocol, but unlike 2PC it exhibits non-blocking behavior. The EC protocol achieves these goals through two key insights: (i) first transmit and then commit, and (ii) message redundancy. EC ensures that each participating node forwards the global decision to all the other participants. To ensure non-blocking behavior, EC protocol also requires each node (coordinator and participants) to delay commit until it transmits the global decision to all the other nodes. Hence, the commit step subsumes message transmission to all the nodes.

2.2.1. Commitment Protocol. We present the EC protocol state transition diagram, and the coordinator and participant algorithms in Figures 2.5, 2.7 and 2.8 respectively. The EC protocol is initiated by the coordinator node. It sends the *Prepare* message to each of the cohorts and moves to the `READY` state. When a cohort receives the *Prepare* message it sends its decision to the coordinator and moves to the `READY` state. On receiving the responses from each of the cohorts, the coordinator first transmits the global decision to all the participants and then commits (or aborts) the transaction. Each of the cohorts, on receiving a response from the coordinator, first forward

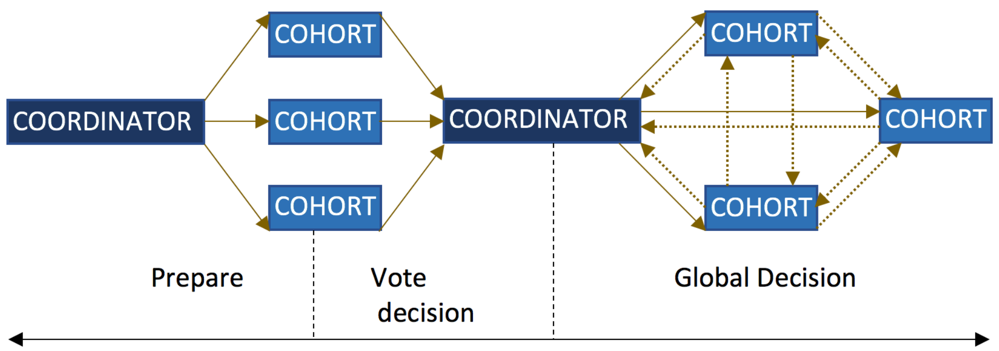


FIGURE 2.6. Time span of EC Protocol

```
1:  Send PREPARE to all participants.
2:  Add begin_commit to log.
3:  Wait for VOTE-COMMIT or VOTE-ABORT from all participants.

4:  event Coordinator timeouts do
5:      Run Termination Protocol

6:  event Coordinator receives all VOTE-COMMIT messages do
7:      Add global-commit-decision-reached in log.
8:      Send GLOBAL-COMMIT to all participants.
9:      Commit the transaction.
10:     Add transaction-commit to log.

11: event Coordinator receives one VOTE-ABORT message do
12:     Add global-abort-decision-reached in log.
13:     Send GLOBAL-ABORT to all participants.
14:     Abort the transaction.
15:     Add transaction-abort to log.
```

FIGURE 2.7. Coordinator's algorithm

the global decision to all the participants (and the coordinator) and then commit (or abort) the transaction locally.

We introduce multiple entries to the log to facilitate recovery during node failures. Note: the EC protocol allows the coordinator to commit as soon as it has communicated the global decision to all the other nodes. This implies that the coordinator need not wait for the acknowledgments. When a node `timeouts`, while waiting for a message, it executes the *termination protocol*. Some of the noteworthy observations are:

(I1) A participant node cannot make a direct transition from the `INITIAL` state to the `ABORT` state.

```
event Participant timeouts do
    Run Termination Protocol

event Participant receives PREPARE from the coordinator do
    Send decision VOTE-COMMIT or VOTE-ABORT to coordinator.
    Add ready to log.
    Wait for message from coordinator.

event Coordinator decision is GLOBAL-COMMIT do
    Add global-commit-received in log.
    Forward GLOBAL-COMMIT to all nodes.
    Commit the transaction.
    Add transaction-commit to log;

event Coordinator decision is GLOBAL-ABORT do
    Add global-abort-received in log.
    Forward GLOBAL-ABORT to all nodes.
    Abort the transaction.
    Add transaction-abort to log.
```

FIGURE 2.8. Participant's algorithm

(I2) The cohorts, irrespective of the global decision, always forward it to every participant.

(I3) The cohorts need not wait for message from the coordinator, if they receive global decision from other participants.

(I4) There exists some hidden states (a.k.a `TRANSMIT-A` and `TRANSMIT-C`), only after which a node aborts or commits the transaction (cf. discussed in Section 2.2.2).

In Figure 2.6, we also present the linear time scale model for the EasyCommit protocol. Here, in the second phase, we use solid lines to represent the global decision from the coordinator to the cohorts, and the dotted lines to represent message forwarding.

2.2.2. Termination Protocol. We now consider the correctness of the EC algorithm under *node-failures*. We want to ensure that the EC protocol exhibits both liveness and safety properties. A commit protocol is said to be *safe* if there isn't any instant during the execution of system under consideration when two or more nodes are in conflicting states (that is one node is in `COMMIT` state while other is in `ABORT`). A protocol is said to respect *liveness* if its execution causes none of the nodes to block.

During the execution of a commit protocol each node waits for a message for a specific amount of time before it *timeouts*. When a node timeouts, it concludes loss of communication with the sender node, which in our case corresponds to failure of the sender. A node is assumed to be blocked if it is unable to make progress on timeout. In case of such node failures, the active nodes execute the termination protocol to ensure system makes progress. We illustrate the termination protocol by stating the actions taken by the coordinator and participating nodes on timeout. The coordinator can timeout only in the `WAIT` state, while the cohorts can timeout in `INITIAL` and `READY` states.

(A1) **Coordinator Timeout in `WAIT` state** – If the coordinator timeouts in this state, then it implies that the coordinator didn't receive the vote from one of the cohorts. Hence, the coordinator first adds a log entry (*global-abort-decision-reached*), next transmits the *Global-abort* message to all the active participants and finally aborts the transaction.

(A2) **Cohort Timeout in `INITIAL` State** – If the cohort timeouts in this state, then it implies that it didn't receive the *Prepare* message from the coordinator. Hence, this cohort initiates communication with other active cohorts to reach a common decision.

(A3) **Cohort Timeout in `READY` State** – If the cohort timeouts in this state, then it implies that it didn't receive a *Global-Commit* (or *Global-Abort*) message from any node. Hence, it would consult the active participants to reach a decision common to all the participants.

Leader Election: In last two cases we force the cohorts to perform transactional commit or abort based on an agreement. This agreement requires selection of a new leader (or coordinator). The target of this leader is to ensure that all the active participants follow the same decision, that is, commit (or abort) the transaction. The selected leader can be in the `INITIAL` or the `WAIT` state. It consults all the nodes if any of them has received a copy of the global decision. If none of the nodes know the global decision, then the leader first adds a log entry (*global-abort-decision-reached*), next transmits the *Global-abort* message to all active participants and then aborts the transaction.

To prove correctness of EC protocol, Figure 2.9 expands the state transition diagram. We introduces two intermediate hidden states (a.k.a `TRANSMIT-A` and `TRANSMIT-C`). All the nodes are oblivious to these states, and the purpose of these states is to ensure message redundancy in the network. As a consequence, we categorize the states of the EC protocol under five heads:

- `UNDECIDED` – The state before reception of global decision (that is `INITIAL`, `READY` and `WAIT` states).
- `TRANSMIT-A` – The state on receiving the global abort.

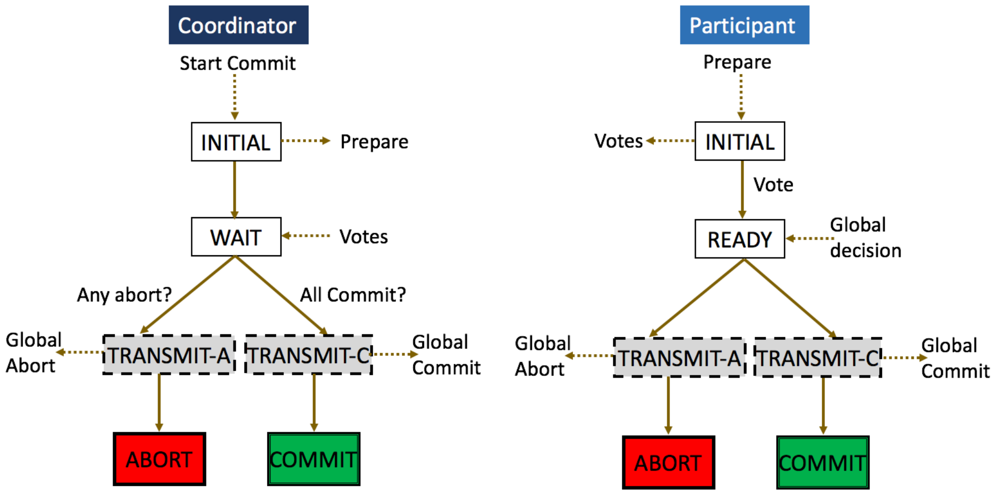


FIGURE 2.9. Logical expansion of EasyCommit Protocol.

	UNDECIDED	T-A	T-C	ABORT	COMMIT
UNDECIDED	Y	Y	Y	N	N
T-A	Y	Y	N	Y	N
T-C	Y	N	Y	N	Y
ABORT	N	Y	N	Y	N
COMMIT	N	N	Y	N	Y

FIGURE 2.10. Coexistent states in EC protocol (`T-A` refers to `TRANSMIT-A` and `T-C` refers to `TRANSMIT-C`).

- `TRANSMIT-C` – The state on receiving the global commit.
- `ABORT` – The state after transmitting *Global-Abort*.
- `COMMIT` – The state after transmitting *Global-Commit*.

Figure 2.10 illustrate whether two states can co-exist (Y) or they conflict (N). We derive this table on the basis of our observations: I - IV and cases A - C. We now have sufficient tools to prove the liveness and safety properties of the EasyCommit protocol.

THEOREM 2.2.1. *EasyCommit protocol is safe, that is, in the presence of only node failures, for a specific transaction, two nodes cannot be in both Aborted and Committed states, at any instant.*

PROOF. Let us assume the case that two nodes `p` and `q` are in the conflicting states (say `p` voted to abort the transaction and `q` voted to commit). This would imply that one of them received *Global-Commit* message while the other received *Global-Abort*. From (II) and (III) we can deduce that `p` and `q` should transmit the global decision to each other, but as they are in different states, it implies a contradiction. Also, from (I) we have the guarantee that `p` could not have directly transited to the `ABORT` state. This implies `p` and `q` would have received message from some other node. But, then they should have received the same global decision.

Hence, we assume that either of the nodes `p` or `q` first moved to a conflicting state and then failed. But, this violates property (IV) which states that a node needs to transmit its decision to all the other nodes before it can commit or abort the transaction. Also, once either of `p` or `q` fails, the rest of the system follows termination protocol (cases (A) to (C)) and reaches a safe state. It is important to see that the termination protocol is re-entrant. □

THEOREM 2.2.2. *EasyCommit protocol is live, that is, in the presence of only node failures, it does not block.*

PROOF. The proof for this theorem is a corollary of Theorem 3.1. The termination protocol cases (A) to (C) provide the guarantee that the nodes do not block and can make progress, in case of a node failure. □

2.2.3. Comparison with 2PC Protocol. We now draw out comparisons between the the 2PC and EC protocols. Although, EC protocol is non-blocking, it has a higher message complexity than 2PC. EC protocol's message complexity is $O(n^2)$, while the message complexity for 2PC is $O(n)$.

To illustrate the non-blocking property of EC protocol, we now tackle the motivational example of multiple failures. For the sake of completeness we restate the example here. Let us assume a distributed system with coordinator C and participants X, Y and Z. We also assume that C decides to transmit *Global-commit* message to all the nodes and fails just after transmitting message to the participant X. Say, the node X also fails after receiving the message from C. Thus, nodes Y and Z neither received messages from C nor from node X. In this setting, the nodes Y and Z would eventually timeout and run the termination protocol. From case (C) of termination protocol, it is evident that the nodes Y and Z would select a new leader among themselves and would safely transit to the `ABORT` state.

2.2.4. Comparison with 3PC protocol. Although, EC protocol looks similar to 3PC protocol, but it is a stricter and an efficient variant to 3PC protocol. It introduces the notion of a set of intermediate hidden states: `TRANSMIT-A` and `TRANSMIT-C`, which can be superimposed on the `ABORT` and `COMMIT` states, respectively. Also, in the EC protocol, the nodes do not expect any acknowledgements. So unlike the 3PC protocol, there are no inputs to the `TRANSMIT-A`, `TRANSMIT-C`, `ABORT` and `COMMIT` states. However, EC protocol has a higher message complexity than 3PC, which has a message complexity of $O(n)$.

2.3. Discussion

Until now, all our discussion assumed existence of only node failures. In Section 2.2 we prove that EC protocol is non-blocking under node failures. We now discuss the behavior of the 2PC, 3PC and EC protocols under communication failures that is message delay and message loss. Later in this section we also study the degree to which these protocols support independent recovery.

2.3.1. Message Delay and Loss. We now analyze the characteristics of 2PC, 3PC and EC protocols, under unexpected delays in message transmission. Message delays represent an unprecedented lag in the communication network. The presence of message delays can cause a node to timeout and act as if a node failure has occurred. This node may receive a message pertaining transaction commitment or abort, after the decision has been made. It is interesting to note that 2PC and 3PC protocols are not safe under message delays [15, 109]. Prior works [54, 109] have shown that it is impossible to design a non-blocking commitment protocol for unbounded asynchronous networks with even a single failure.

We illustrate the nature of 3PC protocol under message delay, as it is trivial to show that 2PC protocol is unsafe under message delays. The 3PC protocol state diagram does not provide any intuition about the transitions that two nodes should perform when both of them are active but unable to communicate. In fact, partial communication or unprecedented delay in communication can easily hamper the database consistency.

Let us consider a simple configuration with a coordinator C and the participants X, Y and Z. Assume that C receives *Vote-commit* message from all the cohorts. Hence, it decides to send the *Prepare-to-commit* message to all the cohorts. However, it is possible that the system starts facing unanticipated delays on all the communication links with C at one end. We can also assume that the paths to node X are also facing severe delays. In such a situation, the coordinator would proceed to *globally commit* the transaction (as it has moved to the `PRE-COMMIT` state), while the nodes X, Y and Z would abort the transaction (as from their perspective the system has undergone multiple failures). This implies that the 3PC termination protocol is not sound under message delays. Similarly, EC protocol is unsafe under message delays.

This situation can aggravate if the network undergoes message loss. Interestingly, message loss has been deemed to be true representation of the network partitioning [109]. Hence, no commit protocol is safe (or non-blocking) under message loss [15]. If the system is suffering from message loss then the participating nodes (and coordinator) would *timeout* and would run the associated terminating protocol that could make nodes transit to conflicting states. Thus, we conclude that 2PC, 3PC and EC protocols are also unsafe under message loss.

2.3.2. Independent Recovery. Independent recovery is one of the desired properties from the nodes in a distributed system. An independent recovery protocol lays down a set of rules that help a failed node to terminate (commit or abort) the transaction, which it was executing prior to its failure, without any help from other active participants. Interestingly, the 2PC and 3PC protocols support only partial independent recovery [15, 109].

It is easy to present a case where the 3PC protocol lacks independent recovery. Consider a cohort in the `READY` state that votes to commit the transaction and fails. On recovery this node needs to consult with the other nodes about the fate of the last transaction. This node cannot independently commit (or abort) the transaction, as it does not know the global decision, which could have been either commit or abort.

EC protocol supports independent recovery in following scenarios:

(i1) If a cohort fails before transmitting its vote, then on recovery it can simply abort the transaction.

(i2) If the coordinator fails before transmitting the global decision, then it aborts the transaction on recovery.

(i3) If either coordinator or participant fail after transmitting the global decision and writing the log, then on recovery they can use this entry to reach the consistent state.

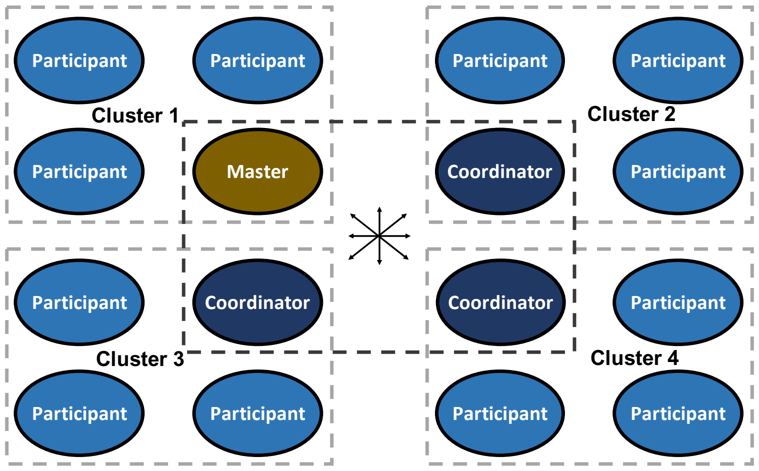


FIGURE 2.11. Geo-Scale system with four clusters.

2.4. Geo-Scale EasyCommit

In this section we design the EasyCommit protocol with regards to the geographically large-scale distributed database systems. A geographically large-scale (a.k.a geo-scale) system consists of multiple *clusters* of nodes, where each cluster is located at a geographically different location. Each cluster is structured akin to a distributed system with one node acting as the coordinator and rest of the cluster nodes acting as the participants.

Traditional agreement protocols do not directly cater to the needs of these systems due to existence of high communication costs. Moreover, an efficient geo-scale agreement protocol should take into consideration the proximity of nodes within (or outside) a cluster. Thus, arises the need for a *two-level* agreement protocol. It is important to understand that a *two-level* agreement protocol does not necessarily imply execution of one agreement protocol atop another. For instance, for a geo-scale system, neither is the design of a two-level 3PC protocol intuitive, nor simply appending two 3PC protocols guarantees correctness (non-blocking property).

Figure 2.11 illustrates a simple geo-scale system consisting of four clusters. Each cluster consists of one *local* coordinator and three participants. The local coordinators are responsible for facilitating agreement within their clusters. There also exists a *global* coordinator (henceforth referred as *master*) for ensuring agreement between the geographically distributed clusters. Note: the master may also act as the *local* coordinator for its own cluster.

2.4.1. Motivation for Geo-scale EasyCommit. A desirable geo-scale agreement protocol should be safe and benefit from the cluster topology. We know that 2PC protocol is blocking. Hence, it suffices to show that trivial extensions of 3PC protocol are unsafe for geo-scale systems. A simple design implies a two level execution, that is each cluster runs a 3PC protocol and all the local coordinators execute another 3PC protocol among themselves. Within each cluster the local coordinator is responsible for safe execution of 3PC protocol, while the master manages 3PC protocol run among the local coordinators. When a transaction is ready to be committed, the master requests all the coordinators to provide their decisions. These coordinators, in turn, request their cluster nodes to vote and transmit the agreed decision to the master. The master then follows the 3PC sends a *Prepare-to-commit* message and waits for acknowledgments. The coordinators also follow the similar protocol and forward the *Prepare-to-commit* acknowledgments to the master. Finally, the master sends a *Global-commit* message to all the coordinators, which they broadcast in their clusters.

Although, the aforementioned protocol is neat, it can be shown to be blocking. Consider a case where the master transmits the *Prepare-to-commit* message to all but one local coordinators (cluster 4 coordinator failed). Hence, it *timeouts* (waiting for acknowledgment) and transmits *Global-commit* to all the local coordinators. These local coordinators would transmit the decision within their clusters. Furthermore, assume all the clusters, except cluster 4, have failed. Meanwhile, the participants in cluster 4 would *timeout*, select a new leader, reach a common decision and try to communicate with other clusters. However, as all the clusters are dead they are unsure of the global decision and are blocked. The key idea is that when only one cluster is alive and the top level communication is restricted among the coordinators, then system can block. A simple solution is to have system-wide communication (execute original 3PC algorithm), but such a solution is not scalable (communication expensive) when nodes are at geographically large distances.

2.4.2. Geo-scale EC Commitment Protocol. Figure 2.12 presents the state diagram for Geo-scale EasyCommit protocol (henceforth referred as GEC). We refer to the configuration identical to Figure 2.11 for the ensuing discussion. Figures 2.13, 2.14 and 2.15 present the algorithm to be executed at the master node, coordinators and the participants. The geo-scale EasyCommit protocol also employs the same twin principles: (i) first transmit then commit, and (ii) message

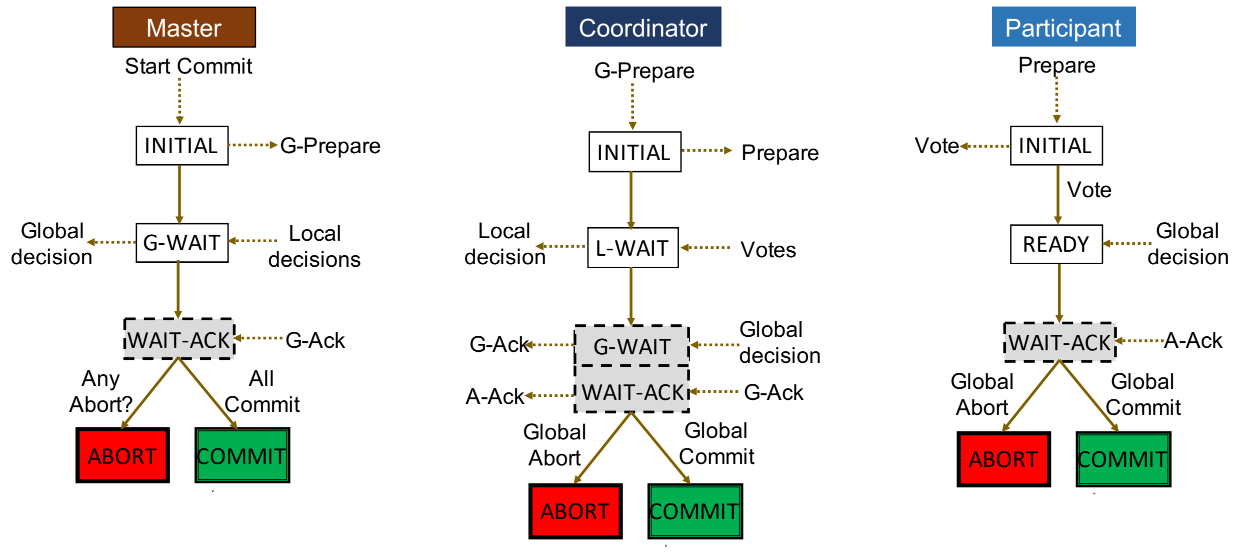


FIGURE 2.12. Geo-Scale EasyCommit Protocol with state diagrams for master coordinator, local coordinator and participant.

redundancy. These principles allow the protocol to attain safety and liveness. Additionally, we restructure the `WAIT` state. GEC introduces a new `G-WAIT` state at the master and coordinator nodes. Furthermore, it includes a `L-WAIT` state at the coordinators. GEC state machine also requires a `WAIT-ACK` state across all the nodes. The rendered state diagram encompasses a set of *half* states: (i) `WAIT-ACK` state at the master, and (ii) `READY` and `WAIT-ACK` states at a participant. These states are referred to as *half* states as a node on these states never transmits any message. GEC also supports a *concurrent* state at the coordinator. This concurrent state arises from the merge of `G-WAIT` and `WAIT-ACK` state at the coordinator.

These changes could lead to an inadvertent interpretation that GEC requires up to *four* phases. However, the existence of a *concurrent* state permits a coordinator to receive messages of multiple types. Note: the master node can also act as the local coordinator. This implies that it would undergo two concurrent state machines, which could be trivially managed by employing a multi-threaded system.

GEC state machine uses the new `WAIT-ACK` state to achieve non-blocking guarantee. This state allows each node to deterministically commit (or abort) the transaction. The GEC agreement protocol starts when the transaction execution is completed. The master node sends out the *G-Prepare* message to all the coordinators and moves to the `G-WAIT` state. Each coordinator on

```
event Initiate Commit Protocol do
    Send G-PREPARE to all coordinators.
    Add *begin_commit* to log.
    Wait for LOCAL-COMMIT or LOCAL-ABORT) from all coordinators.

event Master-node timeouts do
    Run **Termination Protocol**.

event All messages from local-coordinators are LOCAL-COMMIT do
    Add *global-commit-decision-reached* in log.
    Send GLOBAL-COMMIT to all participants.

event Received one LOCAL-ABORT do
    Add *global-abort-decision-reached* in log.
    Send GLOBAL-ABORT to all participants.

event Received G-ACK from all coordinators do
    if Decision was GLOBAL-COMMIT then
        Commit the transaction.
        Add *transaction-commit* to log.
    else
        Abort the transaction.
        Add *transaction-abort* to log.
```

FIGURE 2.13. Master Node's Algorithm

receiving the *G-Prepare* message transmits a *Prepare* message to its cluster participants and moves to the `L-WAIT` state. When a participant receives a *Prepare* message, it decides to *Vote-commit* or *Vote-abort* the transaction. It transmits its decision to the coordinator and moves to the `READY` state. If a coordinator receives at least one *Vote-abort* message, it sends a *Local-abort* message to the master, otherwise it transmits a *Local-commit* message. Next, the coordinator moves to the `G-WAIT` state and waits for the global decision from the master. The master node aggregates all the responses from the coordinators and generates the global decision. It sends the *Global-commit* decision to all the coordinators, if it received all the *Local-commit* responses, otherwise it transmits the *Global-abort* decision. Ensuing this transmission, the master node moves to `WAIT-ACK` state and waits for acknowledgment messages (*G-Ack*) from all the coordinators.

When a coordinator receives the global decision, it forwards the global decision to its cluster participants. Once the coordinator has sent the global decision, it creates an acknowledgment message (*G-Ack*) and transmits the same to all the coordinators (including the master). Next, the coordinator transits to the `WAIT-ACK` state and waits for the *G-Ack* messages from other coordinators. Each participant on receiving the global decision moves to the `WAIT-ACK` state and waits

```
event Received G-PREPARE from the master do
    Send PREPARE to all cluster participants.
    Add *begin_commit* to log.

event Local-coordinator timeouts do
    Run **Termination Protocol**.

event Received all VOTE-COMMIT messages from cluster participants do
    Add *local-commit-decision-reached* to log.
    Send *Local-commit* to master.

event Received all VOTE-ABORT messages from cluster participants do
    Add *local-abort-decision-reached* to log.
    Send *Local-abort* to master.

event Received GLOBAL-COMMIT from master do
    Add *global-commit-decision-reached* to log.
    Forward *Global-commit* to all cluster participants.
    Send *G-Ack* to master and all coordinators.

event Received GLOBAL-ABORT from master do
    Add *Global-abort-decision-reached* to log.
    Forward *Global-abort* to all cluster participants.
    Send *G-Ack* to master and all coordinators.

event Received G-ACK from all coordinators do
    Aggregate all G-ACK messages as an A-ACK message.
    Forward an *A-Ack* to all cluster participants.
    if Decision was GLOBAL-COMMIT then
        Commit the transaction.
        Add *transaction-commit* to log.
    else
        Abort the transaction.
        Add *transaction-abort* to log.
```

FIGURE 2.14. Local Coordinator's Algorithm

for an *aggregated* acknowledgment message from its coordinator. When a coordinator receives all the required *G-Ack* messages, it aggregates them into one message (*A-Ack*), transmits that message to all its cluster participants and decides to commit (or abort) the transaction. Finally, the participants on receiving the *A-Ack* message, follow the global decision and commit (or abort) the transaction.

It is important to understand that a coordinator can receive *G-Ack* from some node before it receives global decision from the master. Hence, the coordinator keeps track of number of *G-Ack* messages it has received. This implies that the coordinator can switch between the `G-WAIT` and

```
event Received PREPARE from the coordinator do
    Send decision VOTE-COMMIT or VOTE-ABORT to coordinator.
    Add ready to log.

event Participant timeouts do
    Run Termination Protocol.

event Received GLOBAL-COMMIT decision do
    Add global-commit-received in log.
event Received GLOBAL-ABORT decision do
    Add global-abort-received in log.

event Received A-ACK message from the local-coordinator do
    if Decision was GLOBAL-COMMIT then
        Commit the transaction.
        Add transaction-commit to log.
    else
        Abort the transaction.
        Add transaction-abort to log.
```

FIGURE 2.15. Participant's Algorithm

WAIT-ACK states. Thus, at coordinator, these states exist concurrently.

The preceding discussion permits following observations:

(I1) No node has a direct transition from INITIAL state to ABORT state.

(I2) Each coordinator after successfully transmitting the global decision in its cluster, transmits a *G-Ack* message to all other coordinators.

(I3) Each participant commits (or aborts) only after receiving an *A-Ack*.

(I4) Each coordinator commits after sending an *A-Ack* to its participants.

2.4.3. Geo-scale EC Termination Protocol. We now present the GEC *termination* protocol that allows geo-scale systems, undergoing node failures, guarantee both safety and liveness. To ensure liveness we require each node to wait on a timer and *timeout* on expiry of its timer. A system undergoing an agreement protocol is live if the nodes are able to progress and not block. A geo-scale system is referred as safe, if at no instant its nodes are in conflicting states (refer Section 2.2).

Cluster Consultation. GEC termination protocol introduces the notion of Cluster Consultation to attain twin guarantees of safety and liveness. Cluster Consultation allows the master node to communicate with the participants of a cluster. This process occurs when the coordinator of a cluster fails, which in turn causes the master to *timeout*. Hence, the master apprises the associated cluster about the failed coordinator and requests them to attain a common decision. Interestingly, if the participants detect the failed coordinator, prior to communication by the master, then they reach a common ground and initiate *Reverse Cluster Consultation* (communication with the master and/or coordinators).

Master Timeout.

(A1) **In `G-WAIT` state**: If the master timeouts in this state, then it implies that the master did not receive the *Local-commit* or *Local-abort* message from one of the coordinators. Hence, the master would add a log entry (*global-abort-decision-reached*), transmit the *Global-abort* message to all other coordinators and initiate *Cluster Consultation.*

(A2) **In `WAIT-ACK` state**: If the master timeouts in this state, then it implies that the master did not receive a *G-Ack* from one of the coordinators. Hence, the master initiates *Cluster Consultation* and then follows the global decision.

Coordinator Timeout.

(A3) **In `INITIAL` state**: If the coordinator timeouts in this state, then it implies that the coordinator did not receive the *G-Prepare* message from the coordinator. Hence, the coordinator communicates with other active coordinators to reach a common decision.

(A4) **In `L-WAIT` state**: If the coordinator timeouts in this state, then it implies that the coordinator did not receive the vote from one of the participants. Hence, the coordinator adds a log entry (*local-abort-decision-reached*) and transmits the *Local-abort* message to the master.

(A5) **In `G-WAIT` state**: If the coordinator timeouts in this state, then it implies that the coordinator did not receive the global decision from the master. Hence, the coordinator interacts with other active coordinators to reach a common decision.

(A6) **In `WAIT-ACK` state**: If the coordinator timeouts in this state, then it implies that the coordinator did not receive the *G-Ack* message from one of the coordinators. Hence, the coordinator communicates with the master node. If the master has the required *G-Ack* message, then it forwards the same, otherwise the master proceeds similarly to case (B).

Participant Timeout.

(A7) **In `INITIAL` state**: If the participant timeouts in this state, then it implies that it did not receive the *Prepare* message from the coordinator. Hence, the participant consults other active participants, for agreement.

(A8) **In `READY` state**: If the participant timeouts in this state, then it implies that it did not receive the global decision from the coordinator. Hence, the participant interacts with other active participants.

(A9) **In `WAIT-ACK` state**: If the participant timeouts in this state, then it implies that it did not receive the *A-Ack* message from the coordinator. Hence, the participant decides to consult other active participants.

Master Election. When the coordinator timeouts while waiting for a response from the master (cases C and E), it interacts with other coordinators to reach a common decision. In such cases, it is often necessary to designate one of the active coordinators as the *new* master node. The election of the new master is akin to *leader election* in the EC protocol. The new master, communicates with the cluster of the failed master and requests them to select a new representative for the cluster. Next, the new master attempts to move the system to safe state. If the new master is in `INITIAL` or `L-WAIT` states, then it decides to globally abort the transaction. If the new master is in `G-WAIT` state, then it first checks whether any other coordinator previously received a global decision. If there exists a previous global decision then the new leader follows that decision, otherwise it proceeds to abort the transaction. If the new leader is in `WAIT-ACK` state, then evidently it knows the global decision and simply ensures that every other node is also in the same state.

Coordinator Election. The participant nodes of a cluster initiate election of a new leader when they detect failure of their current coordinator. The new coordinator helps them to reach an agreement and also performs the *Reverse Cluster Consultation.* If the new coordinator is in initial state, then it transmits *Local-abort* as the decision of the cluster, to the master. If the new coordinator is in `READY` state, then it consults other participants to check if anyone received the global decision. In case none of the active participants are aware of the global decision, the new coordinator performs *Reverse Cluster Consultation.* If the new coordinator is in `WAIT-ACK` state, then it simply performs *Reverse Cluster Consultation.*

It is important to understand that either using *Cluster Consultation* or *Reverse Cluster Consultation* the new coordinator can make the system reach a stable state. If *Cluster Consultation* occur prior to election of new coordinator, then the new coordinator knows the correct state to proceed. Otherwise, the new coordinator either conveys its state to the master or acquires the information about global state.

2.4.4. GEC Correctness: Safety and Liveness. The EasyCommit extensions for the geo-scale systems are intended towards achieving a safe and scalable agreement protocol. Hence, we need to illustrate that GEC is non-blocking in scenarios considered earlier in this section.

For the sake of completeness, we revisit the scenario. We assume existence of four clusters, coordinators and a master node, akin to Figure 2.11. The master node asks all the coordinators to send a decision and they reply with their local decisions (say *Local-commit*). The master sends the global decision to all the coordinators and its participants and dies. All but one coordinator (cluster 4) receives the global decision and forward it to their cluster participants. We assume that coordinator for cluster 4 could not receive the global decision as it failed. This implies that participant nodes of cluster 4 are not aware of the global decision. Furthermore, consider that except for cluster 4, all the nodes in every other cluster have failed. The nodes of cluster 4 can still make progress. Cluster 4 nodes have a guarantee that no other node could have committed (or aborted) the transaction. It is important to understand that cluster 4 nodes did not receive the global decision from their coordinator. Hence, their coordinator could not have sent an *G-Ack* message, which in turn ensures that other nodes could not have committed the results. The cluster 4 nodes can independently select a new leader and progress to safe state.

To prove the twin guarantees of safety and liveness for GEC it is easy to generate a representation similar to Table 2.10. The modified table will replace the hidden states `TRANSMIT-A` and `TRANSMIT-C` with `WAIT-ACK`. It is important to understand that `WAIT-ACK` is the state when nodes know the global decision. Hence, it is not an `UNDECIDED` state.

THEOREM 2.4.1. *Geo-scale EasyCommit protocol is safe, that is, in the presence of only node failures, for a specific transaction, two nodes cannot be in both Aborted and Committed states, at any instant.*

Proof. Let us assume that two nodes `p` and `q`, in different clusters, are in conflicting states (say `p` voted to abort the transaction and `q` voted to commit). This implies that one of them received *Global-abort* message, while other received the *Global-commit* message. This indicates that the master sent conflicting global decisions, which is a contradiction.

Another possibility is that due to master failure, at least one of the coordinators did not receive the message from the master and transmitted conflicting decision to its cluster. Such an assumption is again a contradiction, as cases (C) and (E) require coordinators to communicate with each other.

It is possible that the coordinator of `p`'s cluster has failed and the active nodes decide to move to a conflicting state. This is in contradiction with cases (G) to (I) as active participants need to elect a new coordinator and initiate *Reverse Cluster Consultation*. If during *Reverse Cluster Consultation* they find master to have failed, then they would initiate election of new master. Furthermore, if `p`'s cluster is unaware of the global decision then rules (I) to (IV) safeguard them from transitioning to a state in conflict with `q`. Note: `p`'s failed coordinator could not have transmitted an *G-Ack*. However, if `p`'s cluster knows the global decision and is waiting for an *A-Ack* then they can simply perform *Reverse Cluster Consultation*. At this stage, it is also possible that every cluster except `p`'s cluster has also failed. Still, the nodes can safely follow the global decision. □

Theorem 2.4.2. *Geo-scale EasyCommit protocol is live that is in the presence of only node failures, it does not block.*

Proof. The proof for liveness property follows directly from the proof for Theorem 2.4.1. The termination protocol cases (A) to (I) provide sufficient guarantee that active nodes continue making progress under node failures. □

2.5. EasyCommit Implementation

We now present a discussion on our implementation of the EasyCommit protocol. We have implemented EC protocol in the ExpoDB platform. ExpoDB is an in-memory, distributed transactional platform that incorporates and extends the Deneva [68] testbed. ExpoDB also offers secure transactional capability, and presents a flexible framework to study distributed ledger–blockchain [113, 114].

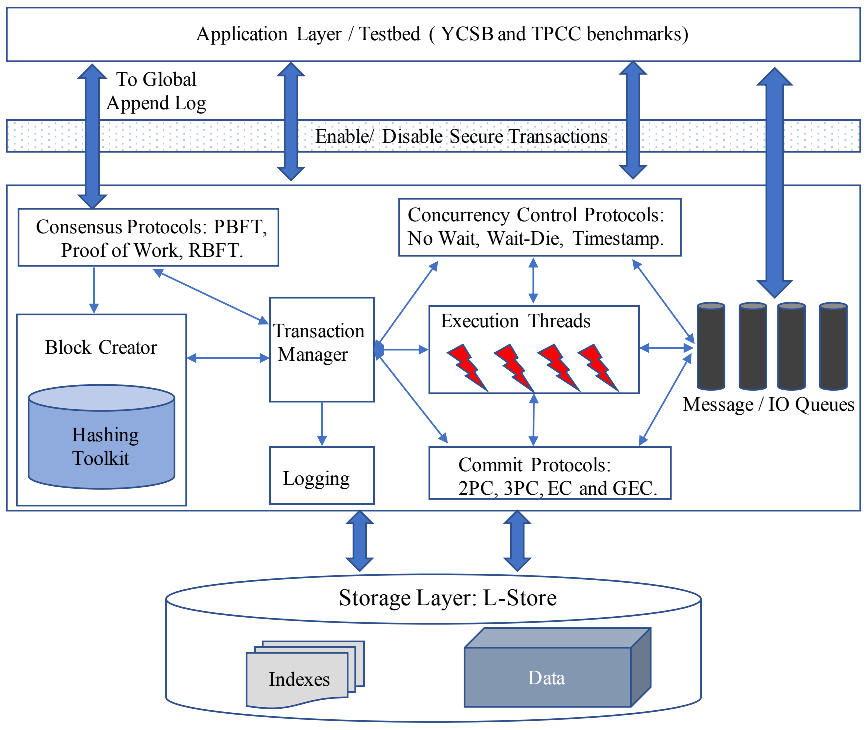

Application Layer / Testbed (YCSB and TPCC benchmarks)
To Global Append Log
Enable/ Disable Secure Transactions
Consensus Protocols: PBFT, Proof of Work, RBFT.
Concurrency Control Protocols: No Wait, Wait-Die, Timestamp.
Block Creator
Hashing Toolkit
Transaction Manager
Execution Threads
Message / IO Queues
Logging
Commit Protocols: 2PC, 3PC, EC and GEC.
Storage Layer: L-Store
Indexes
Data

FIGURE 2.16. **ExpoDB Framework** - executed at each server process, hosted on a cloud node. Each server process receives a set of messages (from clients and other servers), and uses multiple threads to interact with various distributed database components.

2.5.1. Architectural Overview. ExpoDB includes a lightweight layer for testing distributed protocols and design strategy. Figure 2.16 presents the block diagram representation of the ExpoDB framework. It supports a client-server architecture, where each client or server process is hosted on one of the cloud nodes. To maintain inherent characteristics of a distributed system, we opt for a shared nothing architecture. Each partition is mapped to one server node.

A transaction is expressed as a stored procedure that contains both program logic and database queries, which read or modify the records. The clients and server processes communicate with each other using TCP/IP sockets. In practice, the client and server processes are hosted on different cloud nodes, and we maintain an equal number of client and server cloud instances.

Each client creates one or more transactions and sends these transactions to a server process. The server process in turn executes these transaction by accessing the local data and runs the transaction until further execution requires access to remote data. The server process then communicates with other server processes that have access to remote data (remote partitions). Once, these processes return the result, the server process continues execution till completion. Next, it takes a decision to commit or abort the transaction (that is, executes the associated *commit protocol*).

In case a transaction has to be aborted then the coordinating server sends messages to the remote servers to rollback the changes. Such a transaction is resumed after an exponential back-off time. On successful completion of a transaction, the coordinating server process sends an acknowledgment to the client process and performs necessary garbage collection.

2.5.2. Design of 2PC and 3PC. 2PC: The 2PC protocol starts after the completion of the transaction execution. The read-only transactions and single partition transactions do not make use of the commit protocol. Hence, the commit protocol comes into play when the transaction is multi-partition and performs updates to the data-storage. The coordinating server sends a *Prepare* message to all the participating servers and waits for their response. The participating servers respond with the *Vote-commit* message[5]. On receiving the *Vote-commit* message the coordinating server starts the final phase and transmits the *Global-commit* message to all the participants. Each participant on receiving the *Global-commit* message commits the transaction, releases the local transactional resources, and responds with an acknowledgment for the coordinator. The coordinator waits on a counter for response from each participant and then commits the transaction, sends a response to the client node, and releases the associated transactional data-structures.

3PC: To gauge the performance of the EC protocol, we also implemented the 3PC commit protocol. The 3PC protocol implementation is a straightforward extension to the 2PC protocol. We add an extra `PRE-COMMIT` phase before the final phase. On receiving, all the *Vote-commit* messages, the coordinator sends the *Prepare-to-commit* message to each participant. The participating nodes acknowledge the reception of the *Prepare-to-commit* message from the coordinator. The coordinating server on receiving these acknowledgments, starts the finish phase.

2.5.3. EasyCommit Design. We now explain the design of EasyCommit protocol in the ExpoDB framework. The first phase (that is the `INITIAL` phase) is same for both the 2PC and the EC protocol. In the EC protocol, once the coordinator receives the *Vote-commit* message

[5]Without node failures, any transaction that reaches the prepare phase is assumed to successfully commit.

from all the nodes, it first sends the *Global-commit* message to each of the participating processes and then commits the transaction. Next, it responds to the client with the transaction completion notification. When the participating nodes receive the *Global-commit* message from the coordinator, they forward the *Global-commit* message to all the other nodes (including the coordinator), and then commit the transaction.

Although, in the EC protocol the coordinator has a faster response rate to the client, but its throughput takes a slight dip due to additional, implementation enforced wait. It can be noted that we have not performed any cleanup tasks (such as releasing the transactional resources) yet. The cleanup of the transactional resources is performed once it is ensured that neither of those resources would be ever used, nor any messages associated with the transaction would be further received. Hence, we have to force all the nodes (both the coordinator and the participants) to poll the message queue and wait till they have received the messages from each other node. Once all the messages are received, each node performs the cleanup.

To implement EC protocol we had to extend the message being transmitted with a new field which identifies all the participants of the transaction. This array contains the `Id` for each participant, and is updated by the coordinator (as only the coordinator has information about all the partitions) and transmitted as part of the *Global-commit* message.

2.5.4. Geo-scale EasyCommit Design. To extend EasyCommit design to geographically large distributed systems, we adapt Geo-scale EasyCommit algorithm into a topology-aware implementation. We ensure that during the execution of GEC protocol none of the cluster members, except the coordinator communicate outside the clusters. Our implementation allows each cluster node to statically compute the identifiers of other nodes in the cluster. This requirement is met by informing each node about the cluster size, during initial system setup.

The master node needs to apprise each coordinator about identity of other coordinators. It is important to note that two transactions may not have the same master. We dedicate the node receiving the transaction as the master node for that transaction. This node can statically compute the coordinators for its transaction (using *modulo* operation). These coordinators acknowledge their elevated *status* once they receive a *G-Prepare* message. Similarly, coordinators request remaining nodes in the cluster to act as participants.

A key consideration in our implementation is the design of the master node. GEC algorithm states requirement of a master and a set of coordinators. This approach allows us to have a separate node demarcated as the coordinator in the cluster accommodating the master. However, we opt for an efficient design scheme where only one node performs the tasks of both the master and coordinator. This change requires us integrate the master and coordinator algorithms in a manner that prevents redundancy. For instance, once the master nodes transmits the *G-Prepare* message to other coordinators, it initiates the task of transmitting *Prepare* to its cluster participants. Similarly, the master tracks the incoming votes from its participants and the local decisions from other coordinators. Once the master has received all the votes, it transmits the global decision, to its cluster and the coordinators.

A key takeaway from our discussion in Section 2.4 was existence of intermediate states. We claim that these states could easily be merged with other states and do not introduce additional load. Our GEC implementation helps us to validate this claim. We allow each coordinator to transmit the *G-Ack* message as soon as its participants receive the global decision. Moreover, a coordinator could receive the *G-Ack* message from another coordinator prior to the global decision. Hence, the coordinator can track the number of *G-Ack* messages it has received and may piggyback *A-Ack* message to its participants along with the global decision from the master.

2.6. Evauation

In this section, we present a comprehensive evaluation of our novel EasyCommit protocol against 2PC and 3PC. As discussed in Section 2.5, we use the ExpoDB framework for implementing the EC protocol. For our experimentation, we adopt the evaluation scheme of [68].

To evaluate various commit protocols, we deploy the ExpoDB framework on the Microsoft Azure cloud. For running the client and server processes, we use upto 64 `Standard_D8S_V3` instances, deployed in the US East region. Each `Standard_D8S_V3` instance consists of 8 virtual CPU cores and 32GB of memory. For our experiments, we ensure a one-to-one mapping between the server (or client) process and the hosting `Standard_D8S_V3` instance. On each server process, we allowed creation of 4 worker threads, each of which were attached to a dedicated core, and 8 I/O threads. At each server node, a load of 10000 open client connections is applied. For each experiment, we first initiated a warmup phase for 60 seconds, followed by 60 seconds of execution. The measured throughput does not include the transactions completed during warmup phase. If a transaction gets aborted then it is restarted again, only after a fixed time. To attenuate the noise in our readings, we average our results over three runs.

To evaluate the commit protocols, we use the `NO_WAIT` concurrency control algorithm. We use the `NO_WAIT` algorithm as: (i) it is the simplest algorithm, amongst all the concurrency control algorithms present in the ExpoDB framework, and (ii) has been proved to achieve high system throughput. It has to be noted that the use of underlying concurrency control algorithm is orthogonal to our approach. We present the design of a new commit protocol, and hence other concurrency control algorithms (except Calvin) available in the ExpoDB framework, can also employ EC protocol during the commit phase. We present a discussion on the different concurrency control algorithms, later in this section.

In `NO_WAIT` protocol, a transaction requesting access to a locked record is aborted. On aborting the transaction, all the locks held with this transaction are released, which allows other transactions waiting on these locks to progress. `NO_WAIT` algorithm prevents deadlock by aborting transactions in case of conflicts, and hence, has high abort rate. The simple design of `NO_WAIT` algorithm, and its ability to achieve high system throughput [68] motivated us to use it for concurrency control.

2.6.1. Benchmark Workloads. We test our experiments on two different benchmark suites: YCSB [29] and TPC-C [33]. We use YCSB benchmark to evaluate EC protocol on characteristics interesting to the OLTP database designers (Section 2.6.2 to Section 2.6.5) and use TPC-C to gauge the performance of EC protocol on a real world benchmark (Section 2.6.6 and Section 2.6.7).

YCSB – The Yahoo! Cloud Serving Benchmark consists of 11 columns (including a primary key) and 100B random characters. In our experiments we used a YCSB table of size 16 million records per partition. Hence, the size of our database was 16 GB per node. For all our experiments we ensured that each YCSB transaction accessed 10 records (we mention changes to this scheme explicitly). Each access to YCSB data followed the Zipfian distribution. Zipfian distribution tunes the access to hot records through the *skew factor* (`theta`). When `theta` is set to 0.1, the resulting distribution is uniform, while the `theta` value 0.9 corresponds to extremely skewed distribution. In

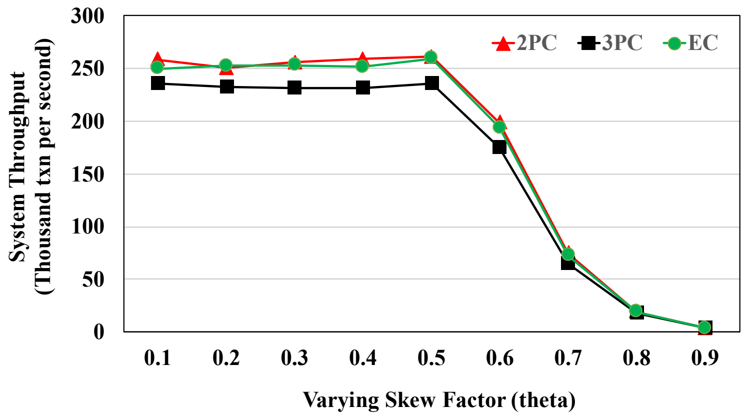


FIGURE 2.17. System throughput (transactions per second) on varying the skew factor (`theta`) for the 2PC, 3PC and EC protocols. These experiments run the YCSB benchmark. Number of server nodes are set to 16 and partitions per transaction are set to 2.

our evaluation using YCSB data, we only executed multi-partition transactions, as single partition transactions do not require use of commit algorithms.

TPC-C – The TPC-C benchmark helps to evaluate system performance by modeling an application for warehouse order processing. It consists of a read-only, item table that is replicated at each server node while rest of the tables are partitioned using the warehouse ID. ExpoDB supports *Payment* and *NewOrder* transactions, which constitute 88% of the workload. Each transaction of *Payment* type accesses at most 2 partitions. These transaction first update the payment amounts for the local warehouse and district, and then update the customer data. The probability that a customer belongs to a remote warehouse is 0.15. In case of transactions of type *NewOrder*, first the transaction reads the local warehouse and district records and then modifies the district record. Next, it modifies item entries in the stock table. Only, 10% *NewOrder* transactions are multi-partition , as only 1% of the updates require remote access.

2.6.2. Varying Skew factor (Theta). We evaluate the system throughput by tuning the skew factor (theta), available in YCSB benchmarks, from 0.1 to 0.9. Figure 2.17 presents the statistics when the number of partitions per transaction are set to 2. In this experiment, we use 16 server nodes to analyze the effects induced by the three commit protocols.

A key takeaway from this plot is that, for `theta` ≤ 0.7 the system throughputs for EC and 2PC protocols are better than the system throughput for the 3PC protocol. On increasing the theta

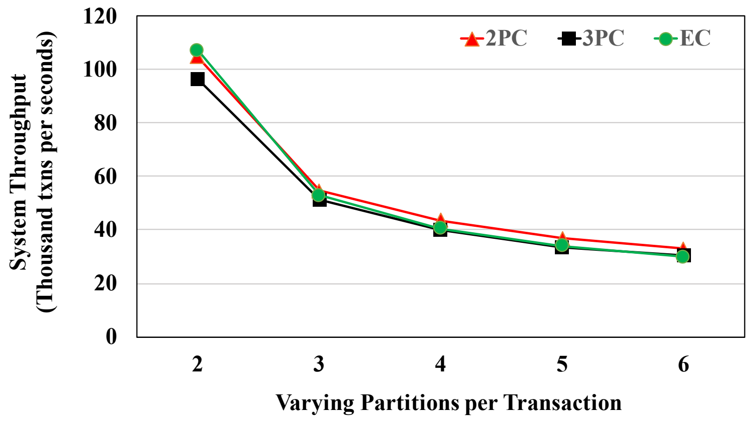


FIGURE 2.18. System throughput (transactions per second) on varying the number of partitions per transactions for the commit protocols. These experiments use YCSB benchmark. The number of server nodes are set to 16 and `theta` is set to 0.6.

further the transactional access becomes highly skewed. This results in an increased contention between the transactions as they try to access (read or write) the same record. Hence, there is a significant reduction in the system throughput across various commit protocols. Thus, it can be observed that the magnitude of difference in the system throughputs for 2PC, 3PC and EC protocol is relatively insignificant. It is important to note that on highly skewed data, the gains due to the choice of underlying commit protocols are overshadowed by other system overheads (such as cleanup, transaction management and so on).

In the YCSB benchmark, for `theta` ≤ 0.5 the data access is uniform across the nodes, which implies that the client transactions access data on various partitions – low contention. Hence, each server node achieves nearly the same throughput. It can be observed that for all the three commit protocols the throughput is nearly constant (not same). We attribute the delta difference in the throughputs of the EC and 2PC protocols to the system induced overheads, network communication latency, and resource contention between the threads (for access to CPU and cache).

2.6.3. Varying Partitions per Transaction. We now measure the system throughput achieved by the three commit protocols on varying the number of partitions per transactions from 2 to 6. Figure 2.18 presents the throughput achieved on the YCSB benchmark, when `theta` is fixed to 0.6, and number of server nodes are set to 16. The number of operations accessed by each transaction are set to 16, and the transaction read-write ratio is maintained at 1 : 1.

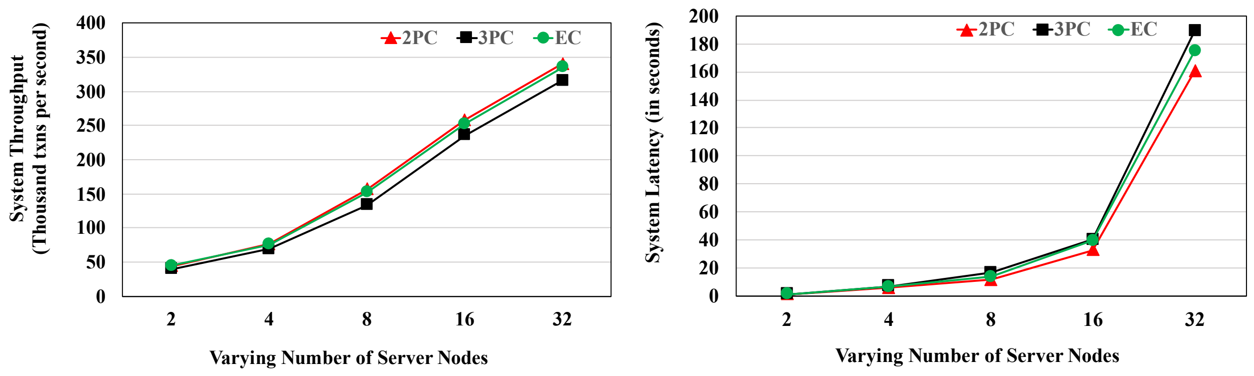
It can be observed that on increasing the number of partitions per transaction there is a dip in the system throughput, across all of the commit protocols. On moving from 2 to 4 partitions there is an approximate decrease of 55%, while the reduction is system performance is around 25% from 4 partitions to 6 partitions, for the three commit protocol. As the number of partitions per transaction increase, the number of messages being exchanged in each round increases linearly for 2PC and 3PC, and quadratically for EC. Also, an increase in partitions imply the transactional resources are held longer across multiple sites, which leads to throughput degradation for all the protocols. Note: in practice, the number of partitions per transaction are not more than four [33].

2.6.4. Varying Server Nodes. We study the effect of varying the number of server nodes (from 2 nodes to 32 nodes) on the system throughput and latency, for the 2PC, 3PC and EC protocols. In Figure 2.19 we set the number of partitions per transaction to 2 and plot graphs for the low contention (`theta` = 0.1), medium contention (`theta` = 0.6) and high contention (`theta` = 0.7). In these experiments, we increase size of YCSB table in accordance to the the increase in number of server nodes.

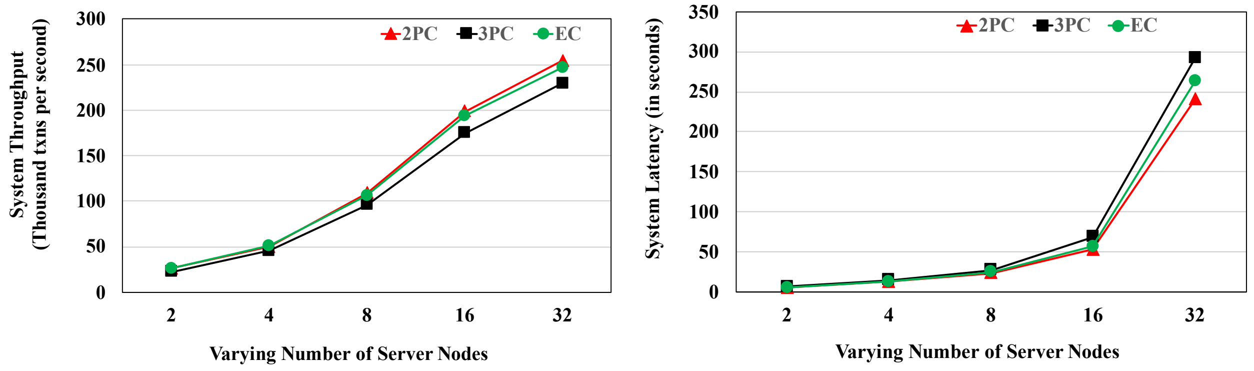
In Figure 2.19, we use the plots on the left to study the system throughput on varying the number of server nodes. It can be observed that as the contention (or skew factor) increases the system throughput decreases, and such a reduction is sharply evident on moving from `theta` = 0.6 to `theta` = 0.7. Another interesting observation is that the system throughput attained by the EC protocol is significantly greater than the throughput attained under 3PC protocol. The gains in system throughput are due to reduction of an extra phase which compensates for the extra messages communicated during the EC protocol.

In comparison to the 2PC protocol the system throughput under EC protocol is marginally lower at low contention and medium contention, and relatively same at high contention. These gains are the result of zero acknowledgment messages required by the coordinating node, in the commit phase, which helps EC protocol perform nearly as efficient as the 2PC protocol. This helps us to conclude that a database system using EC is as scalable as its counterpart employing 2PC.

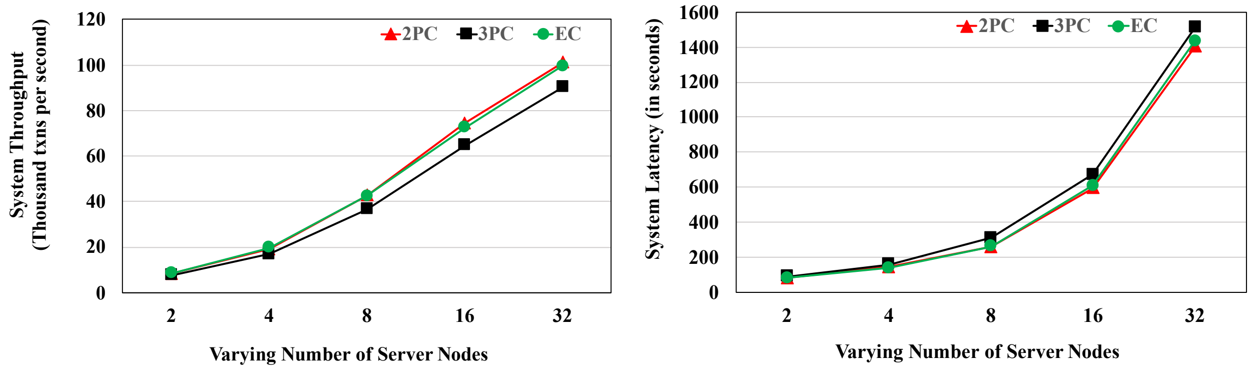
2.6.4.1. *Latency.* In Figure 2.19, we use the plots on the right, to shows the 99 percentile system latency when one of the three commit protocols are employed by the system. We again vary the number of server nodes from 2 to 32. The 99 percentile latency is measured from the first commit



(A) Low contention – (`theta` = 0.1).



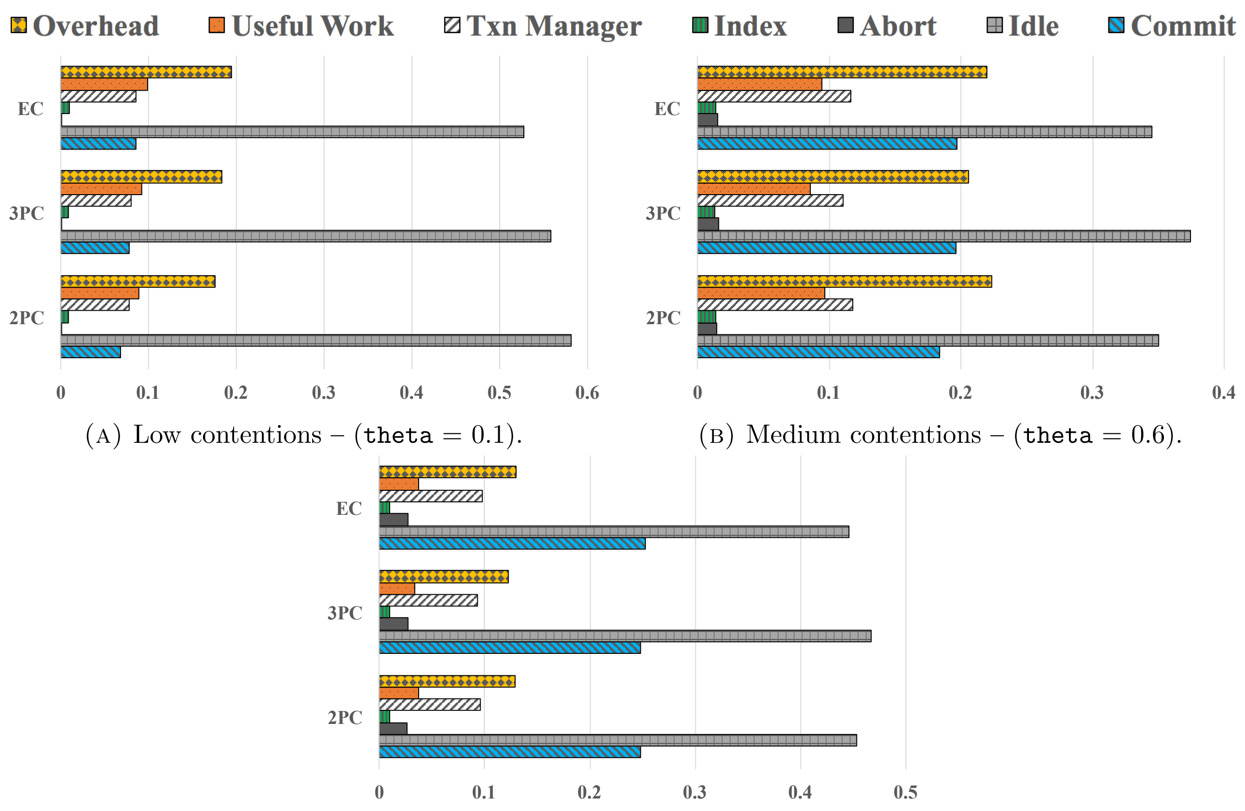
(B) Medium contention – (`theta` = 0.6).



(C) High contention – (`theta` = 0.7).

FIGURE 2.19. System Throughput (transactions per second) and System Latency (in seconds), on varying the number of server nodes for the 2PC, 3PC and EC protocols. The measured latency is the 99-percentile latency, that is, latency from the first start to final commit of a transaction. For these experiments we use the YCSB benchmarks and set the number of partitions per transaction to 2.

to the final commit of a transaction. On increasing the number of server nodes there is a steep increase in latency for each commit protocol. The high latency values for 3PC protocol can be easily cited to the extra phase of communication.



(A) Low contentions – (`theta` = 0.1). (B) Medium contentions – (`theta` = 0.6).

(C) High contentions – (`theta` = 0.7).

FIGURE 2.20. Percentage of time spent by various database components, on executing the YCSB benchmark. We set the number of server nodes to 16 and partitions per transaction to 2.

2.6.4.2. *Proportion of time consumed by various components:* Figure 2.20 presents the time spent on various components of the distributed database system. We show the time distribution for the different degree of contention (*theta*). We categorize these measures under seven different heads.

Useful Work is the time spent by worker threads doing computation for read and write operations. **Txn Manager** is the time spent in maintaining transaction associated resources. **Index** is the time spent in transaction indexing. **Abort** is the time spent in cleaning up aborted transactions. **Idle** is the time worker thread spends when not performing any task. **Commit** is the time spent in executing the commit protocol. **Overhead** represents the time to fetch transaction table, transaction cleanup and releasing transaction table.

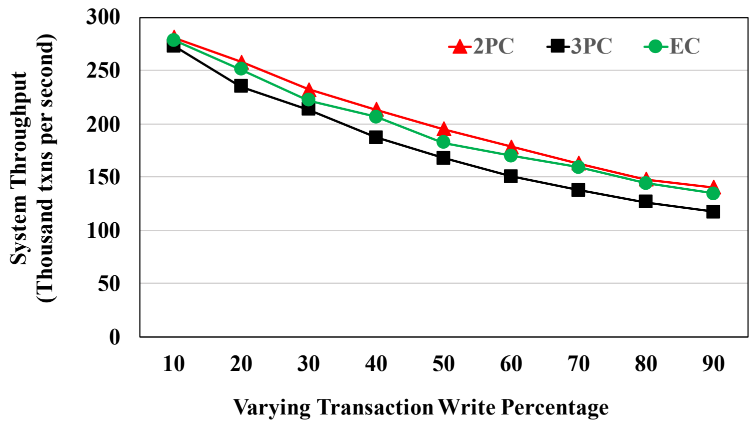
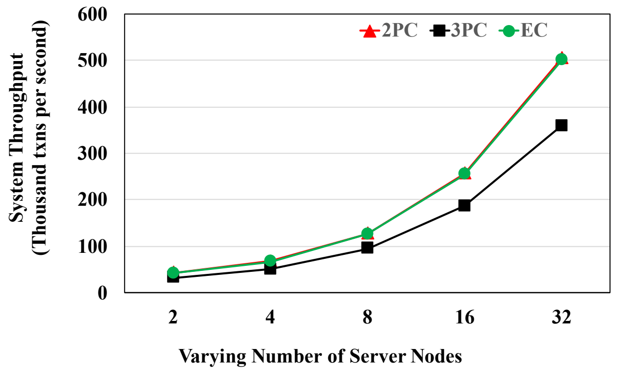


FIGURE 2.21. System throughput (transactions per second) on varying the transaction write percentage for the 2PC, 3PC and EC protocols. These experiments use YCSB benchmark, and set the number of server nodes to 16 and partitions per transactions to 2.

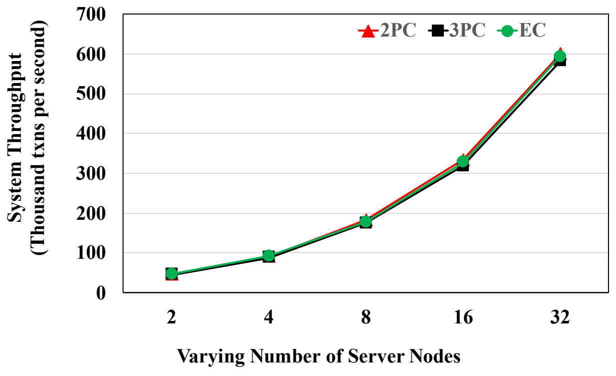
The key intuition from these plots is that as the contention (`theta`) increases there is an increase in time spent in abort. At low contention as most of the transactions are read-only, so the time spent in commit phase is least, and as contention increase, commit phase plays an important role in achieving high throughput from databases. Also, it can be observed at medium and high contention, worker threads executing 3PC protocol are idle for the maximum time and perform the least amount of useful work, which indicates a decrease in system throughput under 3PC protocol due to an extra phase of communication.

2.6.5. Varying Transaction Write Percentage. We now vary the transactional write percentage, and draw out comparisons between the system throughput achieved by the ExpoDB when employing one of the three commit protocols. These experiments (refer Figure 2.21) are based on YCSB benchmark, and vary the percentage of write operations accessed by each transaction from 10 to 90. We set the skew factor to 0.6, number of server nodes to 16 and partitions per transaction to 2.

It can be seen that when only 10% of the operations are write then all the protocols achieve nearly the same system throughput. This is because most of the requests sent by the client consists of read-only transactions, and under read only transactions, the commit protocols are not executed. However, as the write percentage increases the gap between the system throughput achieved by



(A) Payment Transaction



(B) NewOrder Transaction

FIGURE 2.22. System throughput on varying the number of server nodes, on the TPC-C benchmark. The number of warehouses per server are set to 128.

3PC protocol and the other two commit protocols increases. This indicates that 3PC protocol performs poorly when the underlying application consists of write intensive transactions.

In comparison to the 2PC protocol, EC protocol undergoes marginal reduction in throughput. As the number of write operations increase, the number of transactions undergoing the commit protocol also increase. We have already seen that under EC protocol (i) the amount of message communication is higher than the 2PC protocol, and (ii) each node needs to wait for additional *wait-time* before releasing the transactional resources. Some of these held resources include locks on data items, and it is easy to surmise that under EC protocol locks are held longer than the 2PC protocol. The increase in duration of locks being held also leads to an increased abort rate, which is another important factor for reduced system throughput.

2.6.6. Scalability of TPC-C Benchmarks. We now gauge the performance of the EC protocol with respect to a real-world application, that is using TPC-C benchmark. Figure 2.22 presents the characteristics of the 2PC, 3PC and EC protocols, under TPC-C benchmark, on varying the number of server nodes. It has to be noted that a major chunk of TPC-C transactions are single-partition, while most of the multi-partition transactions access only two partitions. Our evaluation scheme sets 128 warehouses per server, and, hence a multi-partition can access two co-located partitions (that is on a single server).

Figure 2.22a represents the scalability of the *Payment* transactions for the three commit protocols. It is evident from this plot that as the number of server nodes increase, the system throughput

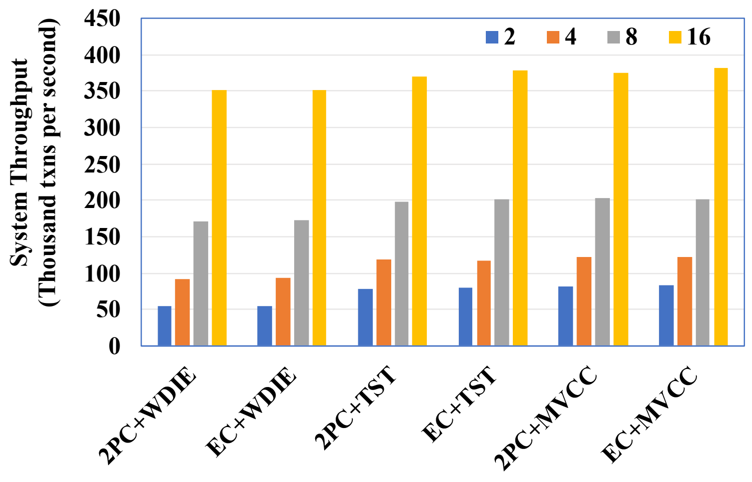


FIGURE 2.23. System throughput achieved by three different concurrency control algorithms. For experimentation, we use the TPC-C *Payment* transaction, and vary the number of server nodes to 16. The number of warehouses per server are set to 128. Here WDIE and TST refer to `WAIT-DIE` and `TIMESTAMP`, respectively.

increases for each commit protocol. However, there is a performance bottleneck in case of 3PC protocol. In case of payment transactions as updates are performed at the home warehouse, which requires exclusive access, so there is an increase in abort rate for the underlying concurrency control algorithm (in our case `NO_WAIT`). Now, as 3PC protocol requires an additional phase to commit the transaction, hence there is an increase in the abort rate. Interestingly, the throughput achieved by the EC protocol is approximately equal to the system throughput under 2PC protocol.

Figure 2.22b depicts the system throughput on executing TPC-C *NewOrder* transactions. The performance bottleneck is reduced for these transactions as there only 10 districts per warehouse, and hence, the commit protocols achieve comparatively higher throughput. Also, as there are only 10% multi-partition transactions, so all the protocols achieve nearly the same performance.

2.6.7. Concurrency Control. The presence of read/write data conflicts between transactional accesses necessitates the use of concurrency control algorithms by the database management system. The ExpoDB framework implements multiple state-of-the-art concurrency control algorithms. Although, in this work, we use `NO_WAIT` concurrency control algorithm, but EC protocol can be easily integrated to work alongside other concurrency control algorithms.

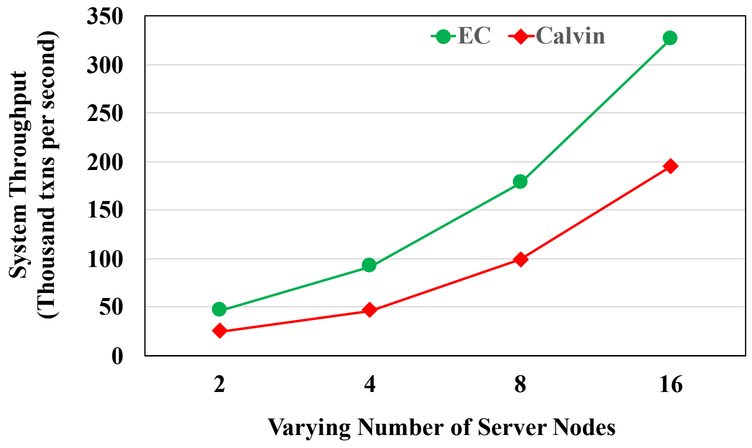


FIGURE 2.24. Comparison of throughput achieved by the system executing Calvin versus the system implementing the combination of No-Wait+EC protocol. In this experiment we use the TPC-C *Neworder* transaction, and vary the number of server nodes to 16. The number of warehouses per server are set to 128.

Figure 2.23 measures the system throughput for three different concurrency control algorithms. We use TPC-C *Payment* transactions for these experiments, and increase the number of server nodes upto 16. We also set the number of warehouses per server to 128. We compare the performance of EC protocol against the 2PC protocol, when the underlying concurrency control algorithm is `WAIT-DIE` [13], `TIMESTAMP` [13] and `MVCC` [14]. It is evident from these experiments that the EC protocol is able to achieve as high efficiency as the 2PC protocol, irrespective of the mechanism used for ensuring concurrency control.

We also analyze our commit protocol against an interesting deterministic concurrency control algorithm – *Calvin* [131]. *Calvin* is a deterministic algorithm that requires the prior knowledge of the read/write sets of the transaction before its execution. When the transaction's read/write sets are not known, at prior, then *Calvin* causes some transactions to execute twice. Interestingly, in the second pass, if some records modify then the transaction is aborted and restarted again. Hence, prior works [68] have shown *Calvin* to perform poorly in such settings. Another strong critic against *Calvin* is that in case of failures, it requires a replica node that executes the same set of operations as the node responding to client query. This implies that Calvin is not suitable under failures for use with partitioned databases. Also, the requirement for replica node, reduces the system throughput.

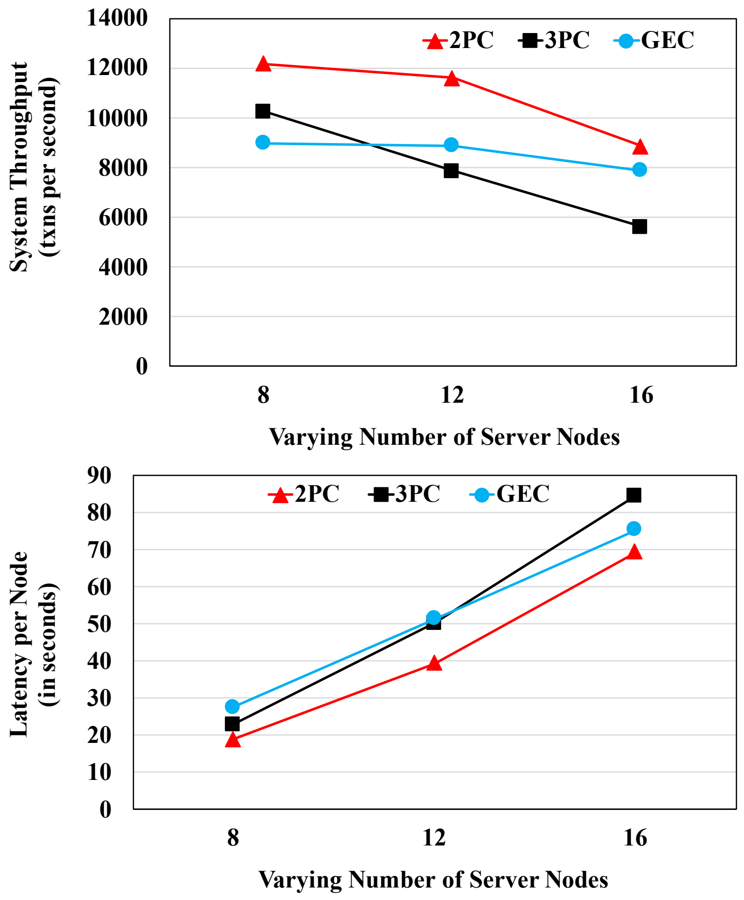


FIGURE 2.25. System throughput and latency per node on varying the number of server nodes, across four regions, where each region consists of equal number of nodes. For these experiments we use YCSB benchmark and set both partitions per transaction and requests per transaction equal to total number of nodes in each run.

Figure 2.24 presents a comparison of `NO_WAIT` algorithm (employing EC protocol) and Calvin. For this experiment we use the TPC-C *Neworder* transactions, and vary the number of server nodes from 2 to 16. These transactions are required to update the order number in their districts. Hence, the deterministic protocols such as Calvin suffer performance degradation.

2.6.8. Geo-scale EasyCommit. We now present an evaluation of our GEC algorithm against 2PC and 3PC algorithms. In Section 2.4 we present the need for a topology-aware algorithm that performs efficiently in the presence of geographically large clusters. Geo-scale systems require existence of algorithms that can reduce the latency and do not require communication between all

Nodes per region	**4**	**5**
2PC	9725.07	8984.80
3PC	8845.07	7664.05
GEC	9187.95	8282.70

FIGURE 2.26. System throughput on increasing number of nodes per region. These experiments were run across three regions and both partitions per transaction and requests per transaction were equal to total number of nodes in each run.

the nodes, across clusters. Figure 2.25 illustrates the performance achieved by GEC and validates our aforementioned claim through interesting comparisons against 3PC and 2PC protocols.

In these figures we run experiments across four geographically distant regions: US East (Ohio), US West (N. California), EU (Ireland) and Asia Pacific (Mumbai). We use Amazon AWS to run these experiments and establish Virtual Private Network to facilitate these experiments. We use `m5x2large` nodes for servers and `t2x2large` nodes for clients. In each run, we ensure there is a one-to-one mapping between the server and the client, that is we have equal number of client and server nodes. We set the number of partitions per transaction and requests per transaction equal to number of servers. We vary the number of server nodes in each region from 2 to 4.

Figure 2.25 illustrates that on increasing the number of server nodes the system throughput decreases across all the three protocols. This phenomena can be easily attributed to: (i) increase in number of partitions per transaction, and (ii) increase in number of requests per transaction. This implies that as the number of server nodes increase, each transaction needs to complete more requests and each request refers to a different partition. An increase in number of server nodes also leads to faster degradation in the performance of 3PC and 2PC protocols, in comparison to GEC protocol. This behavior arises as traditional agreement protocols are oblivious to the underlying cluster topology. This in turn causes existence of a single master that communicates with all the nodes. Interestingly, the throughput of GEC is poor when the cluster size is small, as it performs more work in comparison to 3PC protocol. On increasing the number of nodes per cluster, the throughput reduction is less for GEC (a proof that it is *topology-aware*). Moreover, GEC performs significantly better than 3PC and nearly as good as 2PC protocol.

Figure 2.25 also presents the statistics for 99 percentile latency incurred at each server node. It is easy to gauge that the latency per node increases significantly, on increasing the number of server nodes. When the number of nodes per region is small, the GEC protocol suffers from high latency, as it requires higher communication. However, on increasing the number of nodes per cluster, GEC protocol outperforms 3PC and has latency closer to 2PC protocol. It is our assertion that a further increase in the number of nodes per cluster should bridge the gap between 2PC and GEC throughput and latency values. It is important to understand that GEC is non-blocking and attains performance of the order of 2PC protocol. Further, we use Figure 2.26 to affirm our insights across *three* regions. GEC protocol attains higher throughput than 3PC protocol even on a smaller setup consisting of just three clusters. This proves that GEC protocol is useful across setups of different topologies.

2.7. Optimizations

In earlier sections, we presented a theoretical proof and an evaluation of EasyCommit protocol, which proved its relevance in the space of existing commit protocols. We now discuss some optimizations for the EC protocol.

An optimized version of the EC protocol would allow achieving further gains in comparison to both the 2PC and 3PC protocols. A simple approach is to reduce the number of messages transmitted in the second phase. In the optimized protocol, each node only forwards messages to those nodes from which it has not received a *Global-Commit* or *Global-Abort* message. Another simple optimization is to ensure early cleanup, that is reduction of implementation enforced wait (refer Section 2.5.3). To achieve this, each node would maintain a lookup table, where an entry for each transaction is added, on receiving the first *Global-Commit* or *Global-Abort* message. The remaining messages, addressed to the same transaction, would be matched in the table and deleted. We would also need to periodically, flush some of the entries of the table, to reclaim memory. Interestingly, such an optimization would allow implementing a variant of EC protocol that does not require any "implicit" acknowledgments. Note a similar limited variant for 3PC protocol can be constructed where the coordinator does not wait for acknowledgments after sending the *Prepare-to-Commit* messages, and directly transmits *Global-Commit* message to all the cohorts. Our proposed optimized version is comparable to this 3PC variant.

2.8. Concluding Remarks

In this chapter, we presented a novel commit protocol – EasyCommit. Our design of EasyCommit, leverages the best of twin worlds (2PC and 3PC), it is non-blocking (like 3PC) and requires two phases (like 2PC). EasyCommit achieves these goals by ensuring two key observations: (i) first transmit and then commit, and (ii) message redundancy. We presented the design of the EasyCommit protocol and proved that it guarantees both safety and liveness. We also presented the associated termination protocol and stated cases where EasyCommit can perform independent recovery. We learnt from our EC protocol and designed a novel agreement protocol (Geo-scale EasyCommit) that caters to the needs of a geographically large scale system. GEC protocol limits cost-expensive inter-cluster communication by facilitating cost-inexpensive within cluster communication. We performed a detailed evaluation of EC protocol on a 64 node cloud, and illustrated that it is nearly as efficient as the 2PC protocol. We also evaluated GEC protocol on a setup scaling three continents and showed that GEC is an efficient alternative to 3PC and performs nearly as good as blocking 2PC.

2.9. Bibliographic Notes

Prior to our EC protocol, several interesting research works have suggested the design of a *one phase commit* protocol [2, 52, 127]. These works are aimed towards achieving higher performance than stronger consistency. Clearly, none of these works satisfy the non-blocking requirement expected of a commit protocol.

Over the past three decades, several variants of the 2PC protocol have been proposed, each of which aim to reduce the costs associated with its design. [44, 53, 69, 78, 93, 99, 110, 119, 122]. Presumed-commit and presumed-abort [99] work by reducing a single round of message transmission between the coordinator and the participants, when the transaction is to be committed or aborted, respectively. Gray and Reuter [53] present a series of optimizations for enhancing the 2PC protocol such as lazy commit, read-only commit and balancing the load by coordinator transfer. Group commit [44, 110] helps to reduce the commit overhead by committing a batch of transactions together. Samaras et al. [122] design several interesting optimizations to improve the performance of 2PC protocol. They present heuristics to reduce the overhead of logging, network contention and resource conflicts. Compared to all of these works, we present EC protocol, which is not only efficient, but also satisfies the non-blocking property.

Levy et al. [93] present an optimistic 2PC protocol that releases the locks held by a transaction once all the nodes agree to commit. In the case a node decides to abort the transaction, to prevent violation of database atomicity, compensating transactions are issued to rollback the changes. Although their approach does not guarantee non-blocking behavior, we believe the idea of optimistic resource release can be integrated with our EC protocol to achieve further performance.

Boutros and Desai [119] present another variant to the 2PC protocol, which forces each node to send an additional message in the case of a communication failure between the coordinator and the participant. Although their protocol attempts to handle message loss, their protocol blocks under site failures. We believe an integration of their design with out EC protocol may yield some resilience during message loss.

Haritsa et al. [69] improve the performance of the 2PC protocol, in the context of *real-time* distributed systems. Their protocol permits a conflicting transaction to access the non-committed data. This can lead to cascading aborts, and is not suitable for use with the traditional distributed databases. Our technique, on the other hand, is independent of the underlying concurrency control mechanism and does not cause any special aborts.

Jiménez-Peris et al. [78] allow their system to optimistically fetch the uncommitted data, thereby improving the performance achieved by the 2PC. However, their protocol is tailored for usage alongside strict two-phase locking, and assumes existence of an additional replica of each process. Our technique is neither tailored to any specific concurrency control mechanism, nor it assumes existence of any extra process. However, we believe these heuristics can be used alongside EC protocol to yield further benefits.

Reddy and Kitsuregawa [117] modify the 3PC protocol by introducing the notion of backup sites. With the help of backup sites they are able to achieve higher throughput, but their approach blocks in case of multiple failures. In comparison, our EasyCommit protocol is non-blocking and does not require any backup sites.

CHAPTER 3

Proof-of-Execution: Reaching Consensus through Fault-Tolerant Speculation

In *federated data management* a single common database is managed by many independent stakeholders (e.g., an industry consortium). In doing so, federated data management can ease data sharing and improve data quality [39, 75, 118]. At the core of federated data management is *reaching agreement* on any updates on the common database in an efficient manner, this to enable fast query processing, data retrieval, and data modifications. One can achieve federated data management by *replicating* the common database among all participant, this by replicating the sequence of transactions that affect the database to all stakeholders. One can do so using commit protocols designed for distributed databases such as two-phase [50] and three-phase commit [125], or by using crash-resilient replication protocols such as Paxos [89] and Raft [107].

These solutions are error-prone in a federated *decentralized* environment in which each stakeholder manages its own replicas and replicas of each stakeholder can fail (e.g., due to software, hardware, or network failure) or act malicious: commit protocols and replication protocols can only deal with crashes. Consequently, recent federated designs propose the usage of Byzantine Fault-Tolerant (BFT) consensus protocols. BFT consensus aims at *ordering client requests among a set of replicas, some of which could be Byzantine, such that all non-faulty replicas reach agreement on a common order for these requests* [21, 48, 63, 87, 137]. Furthermore, BFT consensus comes with the added benefit of *democracy*, as BFT consensus gives all replicas an equal vote in all agreement decisions, while the resilience of BFT can aid in dealing with the billions of dollars losses associated with prevalent attacks on data management systems [106].

Akin to commit protocols, the majority of BFT consensus protocols use a *primary-backup model* in which one replica is designated *the primary* that coordinates agreement, while the remaining replicas act as backups and follow the protocol [109]. This primary-backup BFT consensus was first popularized by the influential PBFT consensus protocol of Castro and Liskov [21]. The design of PBFT requires at least $\mathbf{3f} + 1$ replicas to deal with up-to-$\mathbf{f}$ malicious replicas and operates in *three communication phases*, two of which necessitate quadratic communication complexity. As such, PBFT is considered costly when compared to commit or replication protocols, which has negatively impacted the usage of BFT consensus in large-scale data management systems. This has led to the development of new BFT consensus protocols that promise efficiency at the cost of flexibility (e.g., [48, 87, 137]). As a result, the majority of BFT-fueled systems [6, 41, 102] still employ the classical time-tested, flexible, and safe design of PBFT, however.

In this paper, we explore different design principles that can enable implementing a scalable and reliable agreement protocol that shields against malicious attacks. We use these design principles to introduce Proof-of-Execution (PoE), a novel BFT protocol that achieves resilient agreement in just three linear phases. To concoct PoE's scalable and resilient design, we start with PBFT and successively add *four* design elements:

(I1) **Non-Divergent Speculative Execution.** In PBFT, when the primary replica receives a client request, it forwards that request to the backups. Each backup on receiving a request from the primary agrees to support by broadcasting a PREPARE message. When a replica receives PREPARE message from the majority of other replicas, it marks itself as *prepared* and broadcasts a COMMIT message. Each replica that has prepared, and receives COMMIT messages from a majority of other replicas, executes the request.

Evidently, PBFT requires two phases of *all-to-all* communication. Our first ingredient towards faster consensus is speculative execution. In PBFT terminology, PoE replicas execute requests after they get *prepared*, that is, they do not broadcast COMMIT messages. This speculative execution is non-divergent as each replica has a partial guarantee–it has prepared–prior to execution.

(I2) **Safe Rollbacks and Robustness under Failures.** Due to speculative execution, a malicious primary in PoE can ensure that only a subset of replicas prepare and execute a request. Hence, a client may or may not receive a sufficient number of matching responses. PoE ensures that if a client receives a *full proof-of-execution*, consisting of responses from a majority of the non-faulty replicas, then such a request persists in time. Otherwise, PoE permits replicas to *rollback* their state if necessary. This proof-of-execution is the cornerstone of the correctness of PoE.

(I3) **Agnostic Signatures and Linear Communication.** BFT protocols are run among distrusting parties. To provide security, these protocols employ cryptographic primitives for signing the messages and generating message digests. Prior works have shown that the choice of cryptographic signature scheme can impact the performance of the underlying system [21, 85]. Hence, we allow replicas to either employ *message authentication codes* (MACs) or *threshold signatures* (TSs) for signing [85]. When few replicas are participating in consensus (up to 16), then a single phase of all-to-all communication is inexpensive and using MACs for such setups can make computations cheap. For larger setups, we employ TSs to achieve linear communication complexity. TSs permit us to split a phase of all-to-all communication into *two linear phases* [48, 137].

(I4) **Avoid Response Aggregation.** SBFT [48], a recently-proposed BFT protocol, suggests the use of a single replica (designated as the *executor*) to act as a response aggregator. In specific, all replicas execute each client request and send their response to the executor. It is the duty of the executor to reply to the client and *send a proof* that a majority of the replicas not only executed this request, but also outputted the same result. In PoE, we avoid this additional communication between the replicas by allowing each replica to respond directly to the client.

In specific, we make the following contributions:

(1) We introduce PoE, a novel Byzantine fault-tolerant consensus protocol that uses *speculative execution* to reach agreement among replicas.

(2) To guarantee failure recovery in the presence of speculative execution and Byzantine behavior, we introduce a novel view-change protocol that can rollback requests.

(3) PoE supports batching, out-of-order processing, and is signature-scheme agnostic and can be made to employ either MACs or threshold signatures.

(4) PoE does not rely on non-faulty replicas, clients, or trusted hardware to achieve safe and efficient consensus.

(5) To validate our vision of using PoE in resilient federated data management systems, we implement PoE and four other BFT protocols (ZYZZYVA, PBFT, SBFT, and HOTSTUFF) in our efficient RESILIENTDB fabric.

(6) We extensively evaluate PoE against these protocols on a Google Cloud deployment consisting of 91 replicas and 320 k clients under (i) no failure, (ii) backup failure, (iii) primary

failure, (iv) batching of requests, (v) zero payload, and (vi) scaling the number of replicas. Further, to prove the correctness of our results, we also stress test PoE and other protocols in a simulated environment. Our results show that PoE can achieve up to 80% more throughput than existing BFT protocols in the presence of failures.

To the best of our knowledge, PoE is the first protocol that achieves consensus in *only two phases* while being able to deal with Byzantine failures and without relying on trusted clients (e.g., Zyzzyva [87]) or on trusted hardware (e.g., MinBFT [133]). Hence, PoE can serve as a drop-in replacement of Pbft to improve scalability and performance in permissioned blockchain fabrics such as Hyperledger Fabric [8]; and in sharding protocols such as AHL [35].

3.1. System model and notations

Before providing a full description of our PoE protocol, we present the system model we use and the relevant notations.

A system is a set $\mathfrak{R}$ of *replicas* that process client requests. We assign each replica $R \in \mathfrak{R}$ a unique identifier $\mathsf{id}(R)$ with $0 \leq \mathsf{id}(R) < |\mathfrak{R}|$. We write $\mathcal{F} \subseteq \mathfrak{R}$ to denote the set of *Byzantine replicas* that can behave in arbitrary, possibly coordinated and malicious, manners. We assume that non-faulty replicas (those in $\mathfrak{R} \setminus \mathcal{F}$) behave in accordance to the protocol and are deterministic: on identical inputs, all non-faulty replicas must produce identical outputs. We do not make any assumptions on clients: all client can be malicious without affecting PoE. We write $\mathbf{n} = |\mathfrak{R}|$, $\mathbf{f} = |\mathcal{F}|$, and $\mathbf{nf} = |\mathfrak{R} \setminus \mathcal{F}|$ to denote the number of replicas, faulty replicas, and non-faulty replicas, respectively. We assume that $\mathbf{n} > 3\mathbf{f}$ ($\mathbf{nf} > 2\mathbf{f}$).

We assume *authenticated communication*: Byzantine replicas are able to impersonate each other, but replicas cannot impersonate non-faulty replicas. Authenticated communication is a minimal requirement to deal with Byzantine behavior. Depending on the type of message, we use message authentication codes (MACs) or threshold signatures (TSs) to achieve authenticated communication [85]. MACs are based on symmetric cryptography in which every pair of communicating nodes has a *secret key*. We expect non-faulty replicas to keep their *secret keys* hidden.

TSs are based on asymmetric cryptography. In specific, each replica holds a distinct *private key*, which it can use to create a signature share. Next, one can produce a valid threshold signature given at least $\mathbf{nf}$ such signature shares (from distinct replicas). We write $s\langle v\rangle_i$ to denote the signature share of the i-th replica for signing value v. Anyone that receives a set $T = \{s\langle v\rangle_j \mid j \in T'\}$ of signature shares for v from $|T'| = \mathbf{nf}$ distinct replicas, can aggregate T into a single signature $\langle v\rangle$. This digital signature can then be verified using a public key.

We also employ a *collision-resistant cryptographic hash function* $\text{digest}(\cdot)$ that can map an arbitrary value v to a constant-sized digest $\text{digest}(v)$ [85]. We assume that it is practically impossible to find another value v', $v \neq v'$, such that $\text{digest}(v) = \text{digest}(v')$. We use notation $v||w$ to denotes the *concatenation* of two values v and w.

Next, we define the consensus provided by PoE.

Definition 3.1.1. *A single run of any* consensus protocol *should satisfy the following:*

Termination: *Each non-faulty replica executes a transaction.*

Non-divergence: *All non-faulty replicas execute the same transaction.*

Termination is typically referred to as liveness*, whereas non-divergence is typically referred to as* safety. *In* PoE*, execution is speculative: replicas can execute and rollback transactions. To provide safety,* PoE *provides speculative non-divergence instead of non-divergence:*

Speculative non-divergence: *If* $\mathbf{nf} - \mathbf{f} \geq \mathbf{f} + 1$ *non-faulty replicas accept and execute the same transaction T, then all non-faulty replicas will eventually accept and execute T (after rolling back any other executed transactions).*

3.2. Primer on BFT Consensus

Prior to exploring the design of our PoE protocol, we first analyze in brief the age-old problem of achieving byzantine fault-tolerant (BFT) consensus. In this direction, next, we present the designs of some of the state-of-the-art BFT protocols.

3.2.1. Practical Byzantine Fault-Tolerance. A majority of modern-day BFT protocols improve the BFT consensus algorithm described by the *Practical Byzantine Fault-Tolerance* (PBFT) protocol. Hence, PBFT is a good fit to lead this discussion. PBFT expects that in a system of $\mathbf{n} = 3\mathbf{f} + 1$ replicas at most $\mathbf{f}$ replicas are byzantine. PBFT follows the *primary-backup* model where one replica is designated as the primary and leads the consensus, while all the other replicas act

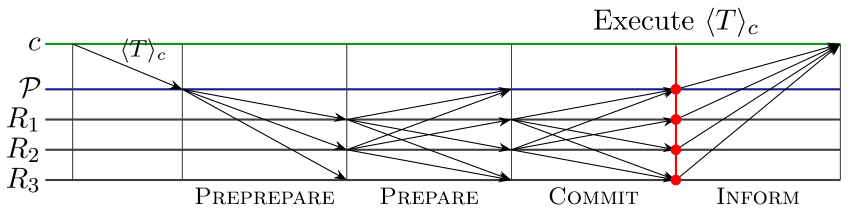


FIGURE 3.1. A schematic representation of the *preprepare-prepare-commit protocol* of PBFT. First, a client c requests transaction T and the primary P proposes T to all replicas via a PREPREPARE message. Next, replicas commit to T via a two-phase message exchange (PREPARE and COMMIT messages). Finally, replicas execute the proposal and inform the client.

as backups and follow the protocol. Next, we summarize the *normal case* algorithm of the PBFT protocol. We use Figure 3.1 to illustrate these steps.

(1) **Client Request.** The PBFT protocol gets into action when a client c wants a transaction T to be processed. To fulfill this task, c creates a message $\langle T\rangle_c$ and sends this message to the primary replica $\mathcal{P}$.

(2) **Pre-prepare.** When the primary $\mathcal{P}$ receives an incoming client request $m := \langle T\rangle_c$, it checks if m is well-formed. If this is the case, then $\mathcal{P}$ assigns m a sequence number k and sends it as a PREPREPARE message to all the backups. This message also hashes the message m to include a digest(m), which helps in future communication.

(3) **Prepare.** When a replica $R \in \mathfrak{R}$ receives a PREPREPARE message from $\mathcal{P}$, it performs three checks: (i) the message is well-formed, (ii) digest(m) is the hash of m, and (iii) this is the first PREPREPARE from $\mathcal{P}$ with sequence number k. If the checks are met, then R agrees to support the order k for this client request and broadcasts a PREPARE message.

(4) **Commit.** When a replica R receives identical PREPARE messages from $\mathbf{nf} = 2\mathbf{f}+1$ replicas, it marks the request m as *prepared* and broadcasts a COMMIT message. Next, R waits for arrival of identical COMMIT messages from $\mathbf{nf} = 2\mathbf{f} + 1$ replicas If this is the case, then R proceeds to mark m as *committed*.

(5) **Execute.** Once a replica has received $\mathbf{nf}$ COMMIT messages, and it has already executed transaction at sequence $(k - 1)$, it executes T and sends the result of execution to the client c. The client c waits for identical responses from $\mathbf{f} + 1$ replicas and marks the request as complete.

3.2.2. Other Consensus Protocols. Since the introduction of PBFT, several new BFT protocols have been proposed with the aim of improving the consensus offered by the PBFT protocol. In this section, we look at the design of some of these protocols.

Zyzzyva. It has a optimal-case path due to which the performance of ZYZZYVA provides an upper-bound for any primary-backup protocol (when no failures occur). Unfortunately, the failure-handling of ZYZZYVA is costly, making ZYZZYVA unable to deal with any failures efficiently.

In the case of ZYZZYVA, each replica executes the client request as soon as it receives a PREPREPARE message from the primary. Hence, replicas do not wait to confirm: (i) if other replicas also got the same request from the primary, and (ii) if the order for client request is same for all the replicas. This forces the client to wait on a timer for identical responses from all the **n** replicas. In case the client does not receive **n** responses, it *timeouts* and informs all the replicas. Each replica, on receiving a message from the client replies with an ACK message. Once the client receives ACK messages from **nf** replicas, it marks the request complete.

SBFT. Like ZYZZYVA, SBFT [48] also adheres to a twin-path design: a linear *fast-path* when there are no byzantine failures and an expensive *slow-path* if some replicas act byzantine. SBFT builds on top of the PBFT protocol and uses threshold signatures to linearize the communication. Essentially, SBFT splits the quadratic phases of PBFT into two.

HotStuff. In any primary-backup BFT protocol, if the primary acts byzantine, then the protocol employs the accompanying view-change algorithm to detect and replace the primary. This view-change algorithm leads to a momentary disruption in system throughput until the resumption of service. HOTSTUFF [137] removes the dependence of the protocol from one primary by replacing primary at the end of every consensus.

As SBFT, HOTSTUFF uses threshold signatures to minimize communication. The state-exchange of HOTSTUFF has an extra phase compared to PBFT. This additional phase simplifies replacing primaries in HOTSTUFF, and enables HOTSTUFF to regularly switch primaries (which limits the influence of any faulty replicas). Due to this design, HOTSTUFF does not support out-of-order processing. As a consequence, HOTSTUFF is more affected by message delays than by bandwidth.

Protocol	Phases	Messages	Resilience	Requirements
Zyzzyva	1	$\mathcal{O}(\mathbf{n})$	0	Reliable clients and unsafe
PoE	3	$\mathcal{O}(3\mathbf{n})$	$\mathbf{f}$	Sign. agnostic
Pbft	3	$\mathcal{O}(\mathbf{n}+2\mathbf{n}^2)$	$\mathbf{f}$	
HotStuff	8	$\mathcal{O}(8\mathbf{n})$	$\mathbf{f}$	Sequential Consensus
Sbft	5	$\mathcal{O}(5\mathbf{n})$	0	Optimistic path

Figure 3.2. Comparison of bft consensus protocols in a system with $\mathbf{n}$ replicas of which $\mathbf{f}$ are faulty. The costs given are for the normal-case behavior.

3.3. Analysis of Design Principles

To arrive at an optimal design for our PoE protocol, we studied practices followed by state-of-the-art distributed data management systems and applied their principles to the design of PoE where possible. In Figure 3.2, we present a comparison of PoE against *four* well-known resilient consensus protocols.

To illustrate the merits of PoE's design, we first briefly look at Pbft. The last phase of Pbft ensures that non-faulty replicas only execute requests and inform clients when there is a guarantee that such a transaction will be recovered after any failures. Hence, clients need to wait for only $\mathbf{f}+1$ identical responses, of which at-least one is from a non-faulty replica, to ensure *guaranteed execution*. By eliminating this last phase, replicas speculatively execute requests before obtaining recovery guarantees. This impacts Pbft-style consensus in two ways:

(1) First, clients need a way to determine *proof-of-execution* after which they have a guarantee that their requests are executed and maintained by the system. We shall show that such a proof-of-execution can be obtained using $\mathbf{nf} \geq 2\mathbf{f}+1$ identical responses (instead of $\mathbf{f}+1$ responses).
(2) Second, as requests are executed before they are guaranteed, replicas need to be able to rollback requests that are dropped during periods of recovery.

PoE's speculative execution guarantees that requests with a proof-of-execution will never rollback and that only a single request can obtain a proof-of-execution per round. Hence, speculative execution provides the same strong consistency (safety) of Pbft in all cases, this at much lower cost under normal operations. Furthermore, we show that speculative execution is fully compatible with other scalable design principles applied to PBFT, e.g., batching and out-of-order processing to maximize throughput, even with high message delays.

Out-of-order execution. Typical BFT systems follow the *order-execute* model: first replicas agree on a unique order of the client request, and only then they execute the requests in order [21, 48, 87, 137]. Unfortunately, this prevents these systems from providing any support for concurrent execution. A few BFT systems suggest executing prior to ordering, but even such systems need to re-verify their results prior to committing changes [8, 84]. Our PoE protocol lies between these two extremes: the replicas speculatively execute using only partial ordering guarantees. By doing so, PoE can eliminate communication costs and minimize latencies of typical BFT systems, this without needing to re-verify results in the normal case.

Out-of-order processing. Although BFT consensus protocols typically execute requests in-order, this does not imply that they need to process proposals to order requests sequentially. To maximize throughput, PBFT and other primary-backup protocols support *out-of-order processing* in which all available bandwidth of the primary is used to continuously propose requests (even when previous proposals are still being processed by the system). By doing so, out-of-order processing can eliminate the impact of high message delays. To provide out-of-order processing, all replicas will process any request proposed as the k-th request whenever k is within some *active window* bounded by a *low-watermark* and *high-watermark* [21]. These watermarks are increased as the system progresses. The size of this active window is—in practice—only limited by the memory resources available to replicas. As out-of-order processing is an essential technique to deliver high throughputs in environments with high message delays, we have included out-of-order processing in the design of PoE.

Twin-path consensus. Speculative execution employed by PoE is different that the *twin-path model* utilized by ZYZZYVA [87] and SBFT [48]. These twin-path protocols have an optimistic *fast* path that works only if none of the replicas are *faulty* and require aid to determine whether these optimistic condition hold.

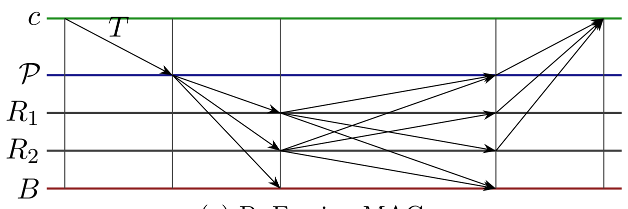
In the fast path of ZYZZYVA, primaries propose requests, and backups directly execute such proposals and inform the client (without further coordination). The client waits for responses from all $\mathbf{n}$ replicas before marking the request executed. When the client does not receive $\mathbf{n}$ responses, it *timeouts* and sends a message to all replicas, after which the replicas perform an expensive client-dependent *slow-path* recovery process (which is prone to errors when communication is unreliable [3]).

The fast path of SBFT can deal with up to $\mathbf{c}$ crash-failures using $3\mathbf{f} + 2\mathbf{c} + 1$ replicas and uses threshold signatures to make communication linear. The fast path of SBFT requires a reliable collector and executor to aggregate messages and to send only *a single* (instead of at-least-$\mathbf{f} + 1$) response to the client. Due to aggregating execution, the fast path of SBFT still performs four rounds of communication before the client gets a response, whereas PoE only uses two rounds of communication (or three when PoE uses threshold signatures). If the fast path *timeouts* (e.g., the collector or executor fails), then SBFT falls back to a threshold-version of PBFT that takes an additional round before the client gets a response. Twin-path consensus is in sharp contrast with the design of PoE, which does not need outside aid (reliable clients, collectors, or executors), and can operate optimally even while dealing with replica failures.

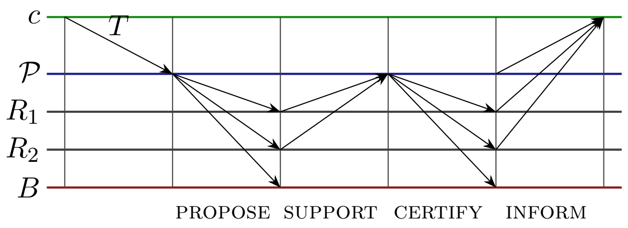
Primary rotation. To minimize the influence of any single replica on BFT consensus, HOTSTUFF opts to replace the primary after every consensus decision. To efficiently do so, HOTSTUFF uses an extra communication phase (as compared to PBFT), which minimizes the cost of primary replacement. Furthermore, HOTSTUFF uses threshold signatures to make its communication linear (resulting in eight communication phases before a client gets responses). The event-based version of HOTSTUFF can overlap phases of consecutive rounds, thereby assuring that consensus of a client request starts in every one-to-all-to-one communication phase. Unfortunately, the primary replacements require that all consensus rounds are performed in a strictly *sequential* manner, eliminating any possibility of *out-of-order processing*.

3.4. Proof-of-Execution

In our *Proof-of-Execution consensus protocol* (PoE), the primary replica is responsible for proposing transactions requested by clients to all backup replicas. Each backup replica *speculatively* executes these transactions with the belief that the primary is behaving correctly. Speculative execution expedites processing of transactions in all cases. Finally, when malicious behavior



(a) PoE using MACs



(b) PoE using TSs.

FIGURE 3.3. Normal-case algorithm of PoE: Client c sends its request containing transaction T to the primary $\mathcal{P}$, which proposes this request to all replicas. Although replica B is Byzantine, it fails to affect PoE.

is detected, replicas can recover by *rolling back transactions*, which ensures correctness without depending on any twin-path model.

3.4.1. The Normal-Case Algorithm of PoE. PoE operates in *views* $v = 0, 1, \ldots$. In view v, replica R with $\mathsf{id}(R) = v \bmod \mathbf{n}$ is elected as the primary. The design of PoE relies on authenticated communication, which can be provided using MACs or TSs. In Figure 3.3, we sketch the normal-case working of PoE for both cases. For the sake of brevity, we will describe PoE built on top of TSs, which results in a protocol with low—*linear*—message complexity in the normal case. The full pseudo-code for this algorithm can be found in Figure 3.4. In Section 3.4.5, we detail the minimal changes to PoE necessary when switching to MACs.

Consider a view v with primary $\mathcal{P}$. To request execution of transaction T, a client c signs transaction T and sends the signed transaction $\langle T \rangle_c$ to $\mathcal{P}$. The usage of signatures assures that malicious primaries cannot forge transactions. To initiate replication and execution of T as the k-th transaction, the primary proposes T to all replicas via a PROPOSE message.

After the i-th replica R receives a PROPOSE message m from $\mathcal{P}$, it checks whether at least $\mathbf{nf}$ other replicas received the same proposal m from primary $\mathcal{P}$. This check assures R that at least $\mathbf{nf} - \mathbf{f}$ non-faulty replicas received the same proposal, which will play a central role in achieving speculative non-divergence. To perform this check, each replica *supports* the first proposal m it receives from the primary by computing a *signature share* $s\langle m\rangle_i$ and sending a SUPPORT message containing this share to the primary.

The primary $\mathcal{P}$ waits for SUPPORT messages with valid signature shares from $\mathbf{nf}$ distinct replicas, which can then be aggregated into a single signature $\langle m\rangle$. After generating such a signature, the primary broadcasts this signature to all replicas via a CERTIFY message.

After a replica R receives a valid CERTIFY message, it *view-commits* to T as the k-th transaction in view v. The replica logs this view-commit decision as $\texttt{VCommit}_R(\langle T\rangle_c, v, k)$. After R view-commits to T, R schedules T for speculative execution as the k-th transaction of view v. Consequently, T will be executed by R after all preceding transactions are executed. We write $\texttt{Execute}_R(\langle T\rangle_c, v, k)$ to log this execution.

After execution, R informs the client of the order of execution and of execution result r (if any) via a message INFORM. In turn, client c will wait for a *proof-of-execution* for the transaction T it requested, which consists of identical INFORM messages from $\mathbf{nf}$ distinct replicas. This proof-of-execution guarantees that at least $\mathbf{nf} - \mathbf{f} \geq \mathbf{f} + 1$ non-faulty replicas executed T as the k-th transaction and in Section 3.4.2, we will see that such transactions are always preserved by PoE when recovering from failures.

If client c does not know the current primary or does not get any timely response for its requests, then it can broadcast its request $\langle T\rangle_c$ to all replicas. The non-faulty replicas will then forward this request to the current primary (if T is not yet executed) and ensure that the primary initiates successful proposal of this request in a timely manner.

To prove correctness of PoE in all cases, we will need the following technical safety-related property of view-commits.

PROPOSITION 3.4.1. *Let R_i, $i \in \{1, 2\}$, be two non-faulty replicas that view-committed to $\langle T_i\rangle_{c_i}$ as the k-th transaction of view v ($\texttt{VCommit}_R(\langle T\rangle_c, v, k)$). If $\mathbf{n} > 3\mathbf{f}$, then $\langle T_1\rangle_{c_1} = \langle T_2\rangle_{c_2}$.*

PROOF. Replica R_i only view-committed to $\langle T_i\rangle_{c_i}$ after R_i received CERTIFY$(\langle h\rangle, v, k)$ from the primary $\mathcal{P}$ (Line 14 of Figure 3.4). This message includes a threshold signature $\langle h\rangle$, whose construction requires signature shares from a set S_i of $\mathbf{nf}$ distinct replicas. Let $X_i = S_i \setminus \mathcal{F}$ be the

```
   Client-role (used by client $c$ to request transaction $T$) :
1: Send $\langle T\rangle_c$ to the primary $\mathcal{P}$.
2: Await receipt of messages $\textsc{inform}(\langle T\rangle_c, v, k, r)$ from $\mathbf{nf}$ replicas.
3: Considers $T$ executed, with result $r$, as the $k$-th transaction.

   Primary-role (running at the primary $\mathcal{P}$ of view $v$, $\mathsf{id}(\mathcal{P}) = v \bmod \mathbf{n}$) :
4: Let view $v$ start after execution of the $k$-th transaction.
5: event $\mathcal{P}$ awaits receipt of message $\langle T\rangle_c$ from client $c$ do
6:    Broadcast $\textsc{propose}(\langle T\rangle_c, v, k)$ to all replicas.
7:    $k := k + 1$.
8: event $\mathcal{P}$ receives $\mathbf{nf}$ message $\textsc{support}(s\langle h\rangle_i, v, k)$ such that:
          (1) each message was sent by a distinct replica, $i \in \{1, \dots, n\}$; and
          (2) All $s\langle h\rangle_i$ in this set can be combined to generate signature $\langle h\rangle$.
   do
9:    Broadcast $\textsc{certify}(\langle h\rangle, v, k)$ to all replicas.

   Backup-role (running at every $i$-th replica $R$.) :
10: event $R$ receives message $m := \textsc{propose}(\langle T\rangle_c, v, k)$ such that:
          (1) $v$ is the current view;
          (2) $m$ is sent by the primary of $v$; and
          (3) $R$ did not accept a $k$-th proposal in $v$
    do
11:    Compute $h := \text{digest}(\langle T\rangle_c || v || k)$.
12:    Compute signature share $s\langle h\rangle_i$.
13:    Transmit $\textsc{support}(s\langle h\rangle_i, v, k)$ to $\mathcal{P}$.
14: event $R$ receives messages $\textsc{certify}(\langle h\rangle, v, k)$ from $\mathcal{P}$ such that:
          (1) $R$ transmitted $\textsc{support}(s\langle h\rangle_i, v, k)$ to $\mathcal{P}$; and
          (2) $\langle h\rangle$ is a valid threshold signature
    do
15:    View-commit $T$, the $k$-th transaction of $v$ ($\texttt{VCommit}_R(\langle T\rangle_c, v, k)$).
16: event $R$ logged $\texttt{VCommit}_R(\langle T\rangle_c, v, k)$ and
             has logged $\texttt{Execute}_R(t', v', k')$ for all $0 \leq k' < k$ do
17:    Execute $T$ as the $k$-th transaction of $v$ ($\texttt{Execute}_R(\langle T\rangle_c, v, k)$).
18:    Let $r$ be the result of execution of $T$ (if there is any result).
19:    Send $\textsc{inform}(\text{digest}(\langle T\rangle_c), v, k, r)$ to $c$.
```

FIGURE 3.4. The normal-case algorithm of PoE.

non-faulty replicas in S_i. As $|S_i| = \mathbf{nf}$ and $|\mathcal{F}| = \mathbf{f}$, we have $|X_i| \geq \mathbf{nf} - \mathbf{f}$. The non-faulty replicas in X_i will only send a single SUPPORT message for the k-th transaction in view v (Line 10 of Figure 3.4). Hence, if $\langle T_1\rangle_{c_1} \neq \langle T_2\rangle_{c_2}$, then X_1 and X_2 must not overlap and $\mathbf{nf} \geq |X_1 \cup X_2| \geq 2(\mathbf{nf} - \mathbf{f})$ must hold. As $\mathbf{n} = \mathbf{nf} + \mathbf{f}$, this simplifies to $3\mathbf{f} \geq \mathbf{n}$, which contradicts $\mathbf{n} > 3\mathbf{f}$. Hence, we conclude $\langle T_1\rangle_{c_1} = \langle T_2\rangle_{c_2}$. □

We will later use Proposition 3.4.1 to show that PoE provides speculative non-divergence. Next, we look at typical cases in which the normal-case of PoE is interrupted:

Example 3.4.1. *A malicious primary can try to affect* PoE *by not conforming to the normal-case algorithm in the following ways:*

(1) By sending proposals for different transactions to different non-faulty replicas. In this case, Proposition 3.4.1 guarantees that at most a single such proposed transaction will get view-committed by any non-faulty replica.

(2) By keeping some non-faulty replicas in the dark by not sending proposals to them. In this case, the remaining non-faulty replicas can still end up view-committing the transactions as long as at least $\mathbf{nf} - \mathbf{f}$ *non-faulty replicas receive proposals: the faulty replicas in* $\mathcal{F}$ *can take over the role of up to* $\mathbf{f}$ *non-faulty replicas left in the dark (giving the false illusion that the non-faulty replicas in the dark are malicious).*

(3) By preventing execution by not proposing a k-th transaction, even though transactions following the k-th transaction are being proposed.

When the network is unreliable and messages do not get delivered (or not on time), then the behavior of a non-faulty primary can match that of the malicious primary in the above example. Indeed, failure of the normal-case of PoE has only two possible causes: primary failure and unreliable communication. If communication is unreliable, then there is no way to guarantee continuous service [42]. Hence, replicas simply assume failure of the current primary if the normal-case behavior of PoE is interrupted, while the design of PoE guarantees that unreliable communication does not affect the correctness of PoE.

To deal with primary failure, each replica maintains a timer for each request. If this timer expires (*timeout*) and it has not been able to execute the request, it assumes that the primary is malicious. To deal with such a failure, replicas will replace the primary. Next, we present the *view-change algorithm* that performs primary replacement.

3.4.2. The View-Change Algorithm. If PoE observes failure of the primary $\mathcal{P}$ of view v, then PoE will elect a new primary and move to the next view, view $v + 1$, via the *view-change algorithm.* The goals of the view-change are:

(1) to assure that each request that *is considered executed* by any client is preserved under all circumstances; and

(2) to assure that the replicas are able to agree on a new view whenever communication is reliable.

As described in the previous section, a client will consider its request executed if it receives a *proof-of-execution* consisting of identical INFORM responses from at-least **nf** distinct replicas. Of these **nf** responses, at-most **f** can come from faulty replicas. Hence, a client can only consider its request executed whenever the requested transaction was executed (and view-committed) by at-least $\mathbf{nf} - \mathbf{f} \geq \mathbf{f} + 1$ non-faulty replicas in the system. We note the similarity with the view-change algorithm of PBFT, which will preserve any request that is *prepared* by at-least $\mathbf{nf} - \mathbf{f} \geq \mathbf{f} + 1$ non-faulty replicas.

The view-change algorithm of PoE consists of three steps. First, failure of the current primary $\mathcal{P}$ needs to be detected by all non-faulty replicas. Second, all replicas exchange information to establish which transactions were included in view v and which were not. Third, the new primary $\mathcal{P}\prime$ proposes a new view. This new view proposal contains a list of the transactions executed in the previous views (based on the information exchanged earlier). Finally, if the new view proposal is valid, then replicas switch to this view; otherwise, replicas detect failure of $\mathcal{P}\prime$ and initiate a view-change for the next view $(v+2)$. The communication of the view-change algorithm of PoE is sketched in Figure 3.5 and the full pseudo-code of the algorithm can be found in Figure 3.6. Next, we discuss each step in detail.

3.4.2.1. *Failure Detection and View-Change Requests.* If a replica R detects failure of the primary of view v, then it halts the normal-case algorithm of PoE for view v and informs all other replicas of this failure by requesting a view-change. The replica R does so by broadcasting a message VC-REQUEST(v, E), in which E is a summary of all transactions executed by R (Figure 3.6, Line 1). Each replica R can detect the failure of primary in two ways:

(1) R *timeouts* while expecting normal-case operations toward executing a client request. E.g., when R forwards a client request to the current primary, and the current primary fails to propose this request on time.
(2) R receives VC-REQUEST messages, indicating that the primary of view v failed, from $\mathbf{f} + 1$ distinct replicas. As at most **f** of these messages can come from faulty replicas, at least

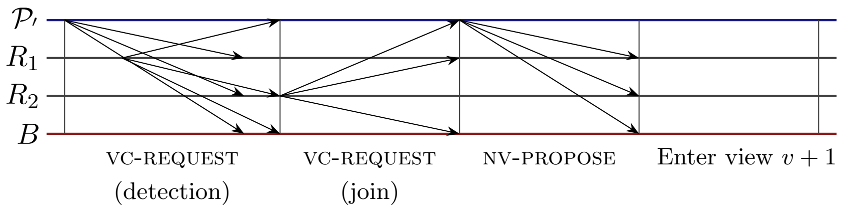


FIGURE 3.5. The current primary B of view v is faulty and needs to be replaced. The next primary, $\mathcal{P}\prime$, and the replica R_1 detected this failure first and request view-change via VC-REQUEST messages. The replica R_2 joins these requests.

```
    vc-request (used by replica $R$ to request view-change) :
 1: event $R$ detects failure of the primary do
 2:     $R$ halts the normal-case algorithm of Figure 3.4 for view $v$.
 3:     $E := \{(\text{CERTIFY}(\langle h\rangle, w, k), \langle T\rangle_c) \mid$
                $w \leq v$ and $\texttt{Execute}_R(\langle T\rangle_c, w, k)$ and $h = \text{digest}(\langle T\rangle_c \| w \| k)\}$.
 4:     Broadcast VC-REQUEST$(v, E)$ to all replicas.
 5: event $R$ receives $\mathbf{f}+1$ messages VC-REQUEST$(v_i, E_i)$ such that
            (1) each message was sent by a distinct replica; and
            (2) $v_i$, $1 \leq i \leq \mathbf{f}+1$, is the current view
    do
 6:     $R$ detects failure of the primary (join).

    On receiving nv-propose (used by replica $R$) :
 7: event $R$ receives $m = \text{NV-PROPOSE}(v+1, m_1, m_2, ..., m_{\mathbf{nf}})$ do
 8:     if $m$ is a valid new-view proposal (similar to creating NV-PROPOSE) then
 9:         Derive the transactions $N$ for the new-view from $m_1, m_2, \ldots, m_{\mathbf{nf}}$.
10:         Rollback any executed transactions not included in $N$.
11:         Execute the transactions in $N$ not yet executed.
12:         Move into view $v+1$ (see Section 3.4.2.3 for details).

    nv-propose (used by replica $\mathcal{P}\prime$ that will act as the new primary) :
13: event $\mathcal{P}\prime$ receives $\mathbf{nf}$ messages $m_i = \text{VC-REQUEST}(v_i, E_i)$ such that
            (1) these messages are sent by a set $S$, $|S| = \mathbf{nf}$, of distinct replicas;
            (2) for each $m_i$, $1 \leq i \leq \mathbf{nf}$, sent by replica $Q_i \in S$, $E_i$ consists of a consecutive sequence of entries
                $(\text{CERTIFY}(\langle h\rangle, v, k), \langle T\rangle_c)$;
            (3) $v_i$, $1 \leq i \leq \mathbf{nf}$, is the current view $v$; and
            (4) $\mathcal{P}\prime$ is the next primary ($\mathsf{id}(\mathcal{P}\prime) = (v+1) \bmod \mathbf{n}$)
    do
14:     Broadcast NV-PROPOSE$(v+1, m_1, m_2, ..., m_{\mathbf{nf}})$ to all replicas.
```

FIGURE 3.6. The view-change algorithm of PoE.

one non-faulty replica must have detected a failure. In this case, R joins the view-change (Figure 3.6, Line 5).

3.4.2.2. *Proposing the New View.* To start view $v+1$, the new primary $\mathcal{P}\prime$ (with $\mathsf{id}(\mathcal{P}\prime) = (v+1) \bmod \mathbf{n}$) needs to propose a new view by determining a valid list of requests that need to be preserved. To do so, $\mathcal{P}\prime$ waits until it receives sufficient information. In specific, $\mathcal{P}\prime$ waits until it received *valid* VC-REQUEST messages from a set $S \subseteq \mathfrak{R}$ of $|S| = \mathbf{nf}$ distinct replicas.

An i-th view-change request m_i is considered valid if it includes a *consecutive sequence* of pairs $(c, \langle T \rangle_c)$, where c is a valid CERTIFY message for request $\langle T \rangle_c$. Such a set S is guaranteed to exist when communication is reliable, as all non-faulty replicas will participate in the view-change algorithm. The new primary collects the set S of $|S| = \mathbf{nf}$ valid VC-REQUEST and proposes them in a new view message NV-PROPOSE to all replicas.

3.4.2.3. *Move to the New View.* After a replica R receives a NV-PROPOSE message containing a new-view proposal from the new primary $\mathcal{P}\prime$, R validates the content of this message. From the set of VC-REQUEST messages in the new-view proposal, R chooses, for each k, the pair $(\text{CERTIFY}(\langle h \rangle, w, k), \langle T \rangle_c)$ proposed in the most-recent view w. Furthermore, R determines the total number of such requests $k_{\max}$. Then, R view-commits and executes all $k_{\max}$ chosen requests that happened before view $v + 1$. Notice that replica R can skip execution of any transaction it already executed. If R executed transactions not included in the new-view proposal, then R needs to *rollback* these transactions before it can proceed executing requests in view $v + 1$. After these steps, R can switch to the new view $v + 1$. In the new view, the new primary $\mathcal{P}\prime$ starts by proposing the $k_{\max} + 1$-th transaction.

When moving into the new view, we see the cost of speculative execution: some replicas can be forced to *rollback execution* of transactions:

EXAMPLE 3.4.2. *Consider a system with non-faulty replica R. When deciding the k-th request, communication became unreliable, due to which only R received a* CERTIFY *message for request $\langle T \rangle_c$. Consequently, R speculatively executes T and informs the client c. During the view-change, all other replicas—none of which have a* CERTIFY *message for $\langle T \rangle_c$—provide their local state to the new primary, which proposes a new view that does not include any k-th request. Hence, the new primary will start its view by proposing client request $\langle T' \rangle_{c'}$ as the k-th request, which gets accepted. Consequently, R needs to rollback execution of T. Luckily, this is not an issue: the client c only got at-most $\mathbf{f} + 1 < \mathbf{nf}$ responses for request, does not yet have a proof-of-execution, and, consequently, does not consider T executed.*

In practice, rollbacks can be supported by, e.g., undoing the operations of transaction in reverse order, or by reverting to an old state. For the correct working of PoE, the exact working of rollbacks is not important as long as the execution layer provides support for rollbacks.

3.4.3. Correctness of PoE. First, we show that the normal-case algorithm of PoE provides non-divergent speculative consensus when the primary is non-faulty and communication is reliable.

THEOREM 3.4.3. *Consider a system in view v, in which the first $k-1$ transactions have been executed by all non-faulty replicas, in which the primary is non-faulty, and communication is reliable. If the primary received $\langle T\rangle_c$, then it can use the algorithm in Figure 3.4 to ensure that:*

(1) there is non-divergent execution of T;

(2) c considers T executed as the k-th transaction; and

(3) c learns the result of executing T (if any),

this independent of any malicious behavior by faulty replicas.

PROOF. Each non-faulty primary would follow the algorithm of PoE described in Figure 3.4 and send PROPOSE$(\langle T\rangle_c, v, k)$ to all replicas (Line 6). In response, all **nf** non-faulty replicas will compute a signature share and send a SUPPORT message to the primary (Line 13). Consequently, the primary will receive signature shares from **nf** replicas and will combine them to generate a threshold signature $\langle h\rangle$. The primary will include this signature $\langle h\rangle$ in a CERTIFY message and broadcast it to all replicas. Each replica will successfully verify $\langle h\rangle$ and will view-commit to T (Line 14). As the first $k-1$ transactions have already been executed, every non-faulty replica will execute T. As all non-faulty replicas behave deterministically, execution will yield the same result r (if any) across all non-faulty replicas. Hence, when the non-faulty replicas inform c, they do so by all sending identical messages INFORM(digest$(\langle T\rangle_c), v, k, r)$ to c (Line 16–Line 19). As all **nf** non-faulty replicas executed T, we have non-divergent execution. Finally, as there are at most **f** faulty replicas, the faulty replicas can only forge up to **f** invalid INFORM messages. Consequently, the client c will only receive the message INFORM(digest$(\langle T\rangle_c), v, k, r)$ from at least **nf** distinct replicas, and will conclude that T is executed yielding result r (Line 3). □

At the core of the correctness of PoE, under all conditions, is that no replica will rollback requests $\langle T\rangle_c$ for which client c already received a proof-of-execution. We prove this next:

PROPOSITION 3.4.2. *Let $\langle T \rangle_c$ be a request for which client c already received a proof-of-execution showing that T was executed as the k-th transaction of view v. If $\mathbf{n} > 3\mathbf{f}$, then every non-faulty replica that switches to a view $v' > v$ will preserve T as the k-th transaction of view v.*

PROOF. Client c considers $\langle T \rangle_c$ executed as the k-th transaction of view v when it received identical INFORM-messages for T from a set A of $|A| = \mathbf{nf}$ distinct replicas (Figure 3.4, Line 3). Let $B = A \setminus \mathcal{F}$ be the set of non-faulty replicas in A.

Now consider a non-faulty replica R that switches to view $v' > v$. Before doing so, R must have received a valid proposal $m = \text{NV-PROPOSE}(v', m_1, ..., m_{\mathbf{nf}})$ from the primary of view v'. Let C be the set of $\mathbf{nf}$ distinct replicas that provided messages $m_1, \dots, m_{\mathbf{nf}}$ and let $D = C \setminus \mathcal{F}$ be the set of non-faulty replicas in C. We have $|B| \geq \mathbf{nf} - \mathbf{f}$ and $|D| \geq \mathbf{nf} - \mathbf{f}$. Hence, using a contradiction argument similar to the one in the proof of Proposition 3.4.1, we conclude that there must exists a non-faulty replica $Q \in (B \cap D)$ that executed $\langle T \rangle_c$, informed c, and requested a view-change.

To complete the proof, we need to show that $\langle T \rangle_c$ was proposed and executed in the last view that proposed and view-committed a k-th transaction and, hence, that Q will include $\langle T \rangle_c$ in its VC-REQUEST message for view v'. We do so by induction on the difference $v' - v$. As the base case, we have $v' - v = 1$, in which case no view after v exists yet and, hence, $\langle T \rangle_c$ must be the newest k-th transaction available to Q. As the induction hypothesis, we assume that all non-faulty replicas will preserve T when entering a new view w, $v < w \leq w'$. Hence, non-faulty replicas participating in view w will not support any k-th transactions proposed in view w. Consequently, no CERTIFY messages can be constructed for any k-th transaction in view w. Hence, the new-view proposal for $w' + 1$ will include $\langle T \rangle_c$, completing the proof. □

As a direct consequence of the above, we have

COROLLARY 3.4.1 (Safety of POE). POE *provides speculative non-divergence if* $\mathbf{n} > 3\mathbf{f}$.

We notice that the view-change algorithm does not deal with minor malicious behavior (e.g., a single replica left in the dark). Furthermore, the presented view-change algorithm will recover all transactions since the start of the system, which will result in unreasonable large messages when many transactions have already been proposed. In practice, both these issues can be resolved by regularly making *checkpoints* (e.g., after every 100 requests) and only including requests since the last checkpoint in each VC-REQUEST message. To do so, PoE uses a standard fully-decentralized PBFT-style checkpoint algorithm that enables the independent checkpointing and recovery of any request that is executed by at least $\mathbf{f}+1$ non-faulty replicas whenever communication is reliable [21]. Finally, utilizing the view-change algorithm and checkpoints, we prove

THEOREM 3.4.4 (Liveness of PoE). PoE *provides termination in periods of reliable bounded-delay communication if* $\mathbf{n} > 3\mathbf{f}$.

PROOF. When the primary is non-faulty, Theorem 3.4.3 guarantees termination as replicas continuously accept and execute requests. If the primary is Byzantine and fails to guarantee termination for at most $\mathbf{f}$ non-faulty replicas, then the checkpoint algorithm will assure termination of these non-faulty replicas. Finally, if the primary is Byzantine and fails to guarantee termination for at least $\mathbf{f} + 1$ non-faulty replicas, then it will be replaced using the view-change algorithm. For the view-change process, each replica will start with a timeout δ after it receives $\mathbf{nf}$ matching VC-REQUESTS and double this timeout after each view-change (exponential backoff). When communication becomes reliable, this mechanism guarantees that all replicas will eventually view-change to the same view at the same time. After this point, a non-faulty replica will become primary in at most $\mathbf{f}$ view-changes, after which Theorem 3.4.3 guarantees termination. □

3.4.4. Fine-Tuning and Optimizations. To keep presentation simple, we did not include the following optimizations in the protocol description:

(1) To reach $\mathbf{nf}$ signature shares, the primary can generate one itself. Hence, it only needs $\mathbf{nf} - 1$ shares of other replicas.
(2) The PROPOSE, SUPPORT, INFORM, and NV-PROPOSE messages are not forwarded and only need MACs to provide message authentication. The CERTIFY messages need not be signed, as tampering them would invalidate the threshold signature. The VC-REQUEST messages need to be signed, as they need to be forwarded without tampering.

Finally, the design of PoE is fully compatible with *out-of-order processing* as a replica only supports proposals for a k-th transaction if it had not previously supported another k-th proposal (Figure 3.4, Line 10) and only executes a k-th transaction if it has already executed all the preceding transactions (Figure 3.4, Line 16). As the size of the active out-of-order processing window determines how many client requests are being processed at the same time (without receiving a proof-of-execution), the size of the active window determines the number of transactions that can be rolled back during view-changes.

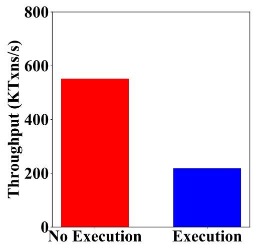
3.4.5. Designing PoE using MACs. The design of PoE can be adapted to only use message authentication codes (MACs) to authenticate communication. This will sharply reduce the computational complexity of PoE and eliminate one round of communication, this at the cost of higher *quadratic* overall communication costs (see Figure 3.3).

The usage of only MACs makes it impossible to obtain threshold signatures or reliably forward messages (as forwarding replicas can tamper with the content of unsigned messages). Hence, using MACs requires changes to how client requests are included in proposals (as client requests are forwarded), to the normal-case algorithm of PoE (which uses threshold signatures), and to the view-change algorithm of PoE (which forwards VC-REQUEST messages). The changes to the proposal of client requests and to the view-change algorithm can be derived from the strategies used by PBFT to support MACs [21]. Hence, next we only review the changes to the normal-case algorithm of PoE.

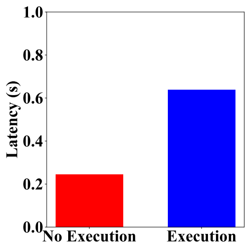
Consider a replica R that receives a PROPOSE message from the primary $\mathcal{P}$. Next, R needs to determine whether at least **nf** other replicas received the same proposal, which is required to achieve speculative non-divergence (see Proposition 3.4.1). When using MACs, R can do so by replacing the all-to-one support and one-to-all certify phases by a single all-to-all *support phase*. In the support phase, each replica agrees to *support* the first proposal $\text{PROPOSE}(\langle T \rangle_c, v, k)$ it receives from the primary by broadcasting a message $\text{SUPPORT}(\text{digest}(\langle T \rangle_c), v, k)$ to all replicas. After this broadcast, each replica waits until it receives SUPPORT messages, identical to the message it sent, from **nf** distinct replicas. If R receives these messages, it *view-commits* to T as the k-th transaction in view v and schedules T for execution. We have sketched this algorithm in Figure 3.3.

3.5. Evaluation

We now analyze our design principles in practice. To do so, we evaluate our PoE protocol against four state-of-the-art BFT protocols in our high throughput yielding permissioned blockchain fabric, RESILIENTDB (refer to Chapter 5 for design details). There are many BFT protocols we



(A) System throughput.



(B) Latency.

FIGURE 3.7. Upper bound on performance when primary only replies to clients (*No exec.*) and when primary executes a request and replies to clients (*Exec.*).

could compare with. Hence, we pick a representative sample: (1) ZYZZYVA—as it has the absolute minimal cost in the fault-free case, (2) PBFT—as it is a common baseline, (3) SBFT—as it is a safer variation of ZYZZYVA, and (3) HOTSTUFF—as it is a linear-communication protocol that adopts the notion of rotating leaders.

For fairness, we reimplemented all the protocols in our RESILIENTDB fabric. Further, we associated our out-of-ordering optimizations with all the protocols except HOTSTUFF, which by design disallows out-of-ordering. Moreover, our PBFT implementation learns from the architecture proposed by BFTSmart [17] and adds pipelining, request batching, and out-of-ordering to its PBFT design. Through our experiments, we want to answer the following questions:

(Q1) How does POE fare in comparison with the other protocols under failures?

(Q2) Does POE benefits from batching client requests?

(Q3) How does POE perform under zero payload?

(Q4) How scalable is POE on increasing the number of replicas participating in the consensus, in the normal-case?

Setup. We run our experiments on the Google Cloud, and deploy each replicas on a *c*2 machine having a 16-core Intel Xeon Cascade Lake CPU running at 3.8 GHz with 32 GB memory. We deploy up to 320 k clients on 16 machines. To collect results after reaching a steady-state, we run each experiment for 180 s: the first 60 s are warmup, and measurement results are collected over the next 120 s. We average our results over three runs.

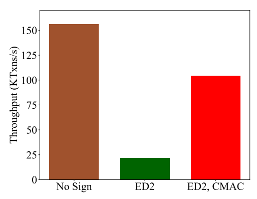
Configuration and Benchmarking. For evaluating the protocols, we employed YCSB [29] from Blockbench's macro benchmarks [38]. Each client request queries a YCSB table that holds half a million active records. We require 90% of the requests to be write queries as the majority of typical blockchain transactions are updates to existing records. Prior to the experiments, each replica is initialized with an identical copy of the YCSB table. The client requests generated by YCSB follow a Zipfian distribution and are heavily skewed (skew factor 0.9).

Unless *explicitly* stated, we use the following configuration for all experiments. We perform scaling experiments by varying replicas from 4 to 91. We divide our experiments in two dimensions: (1) *Zero Payload* or *Standard Payload*, and (2) *Failures* or *Non-Failures*. We employ batching with a batch size of 100 as the percentage increase in throughput on larger batch sizes is small.

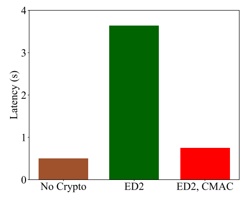
Under Zero Payload conditions, all replicas execute 100 dummy instructions per batch, while the primary sends an empty proposal (and not a batch of 100 requests). Under Standard Payload, with a batch size of 100, the size of PROPOSE message is 5400 B, of RESPONSE message is 1748 B, and of other messages is around 250 B. For experiments with failures, we force one backup replica to crash. Additionally, we present an experiment that illustrates the effect of primary failure. We measure *throughput* as transactions executed per second. We measure *latency* as the time from when the client sends a request to the time when the client receives a response.

3.5.1. System Characterization. We first determine the upper bounds on the performance of RESILIENTDB. In Figures 3.7a and 3.7b, we present the maximum throughput and latency of RESILIENTDB when there is *no communication* among the replicas. We use the term *No Execution* to refer to the case where all clients send their request to the primary replica and primary simply responds back to the client. We count every query responded back in the system throughput. We use the term *Execution* to refer to the case where the primary replica executes each query before responding back to the client.

The architecture of RESILIENTDB (see Chapter 5) states the use of one worker thread. In these experiments, we maximize system performance by allowing up to two threads to work independently at the primary replica without ordering any queries. Our results indicate that the system can attain high throughputs (up to 500 ktxn/s) and can reach low latencies (up to 0.25 s). Notice that if we employ additional worker-threads, our RESILIENTDB fabric can easily attain higher throughput.



(A) System throughput.



(B) Latency.

FIGURE 3.8. System performance using three different signature schemes. In all cases, $\mathbf{n} = 16$ replicas participate in consensus.

3.5.2. Effect of Cryptographic Signatures. RESILIENTDB enables a flexible design where replicas and clients can employ both digital signatures (threshold signatures) and message authentication codes. This helps us to implement POE and other consensus protocols in RESILIENTDB.

To achieve authenticated communication using symmetric cryptography, we employ a combination of CMAC and AES [85]. Further, we employ ED25519-based digital signatures to enable asymmetric cryptographic signing. For generating efficient threshold signature scheme, we use Boneh–Lynn–Shacham (BLS) signatures [85]. To create message digests and for hashing purposes, we use the SHA256 algorithm.

Next, we determine the cost of different cryptographic signing schemes. For this purpose, we run three different experiments in which (i) no signature scheme is used (*None*); (ii) everyone uses digital signatures based on ED25519 (*ED*); and (iii) all replicas use CMAC+AES for signing, while clients sign their message using ED25519 (*MAC*). In these three experiments, we run PBFT consensus among 16 replicas. In Figures 3.8a and 3.8b, we illustrate the throughput attained and latency incurred by RESILIENTDB for the experiments. Clearly, the system attains its highest throughput when no signatures are employed. However, such a system cannot handle malicious attacks. Further, using just digital signatures for signing messages can prove to be expensive. An optimal configuration can require clients to sign their messages using digital signatures, while replicas can communicate using MACs. As can be seen from the results, the costs associated with digital signatures are huge, as their usage reduces performance by 86%, whereas message authentication codes only reduce performance by 33%.

3.5.3. Scaling Replicas under Standard Payload. In this section, we evaluate scalability of POE both under backup failure and no failures.

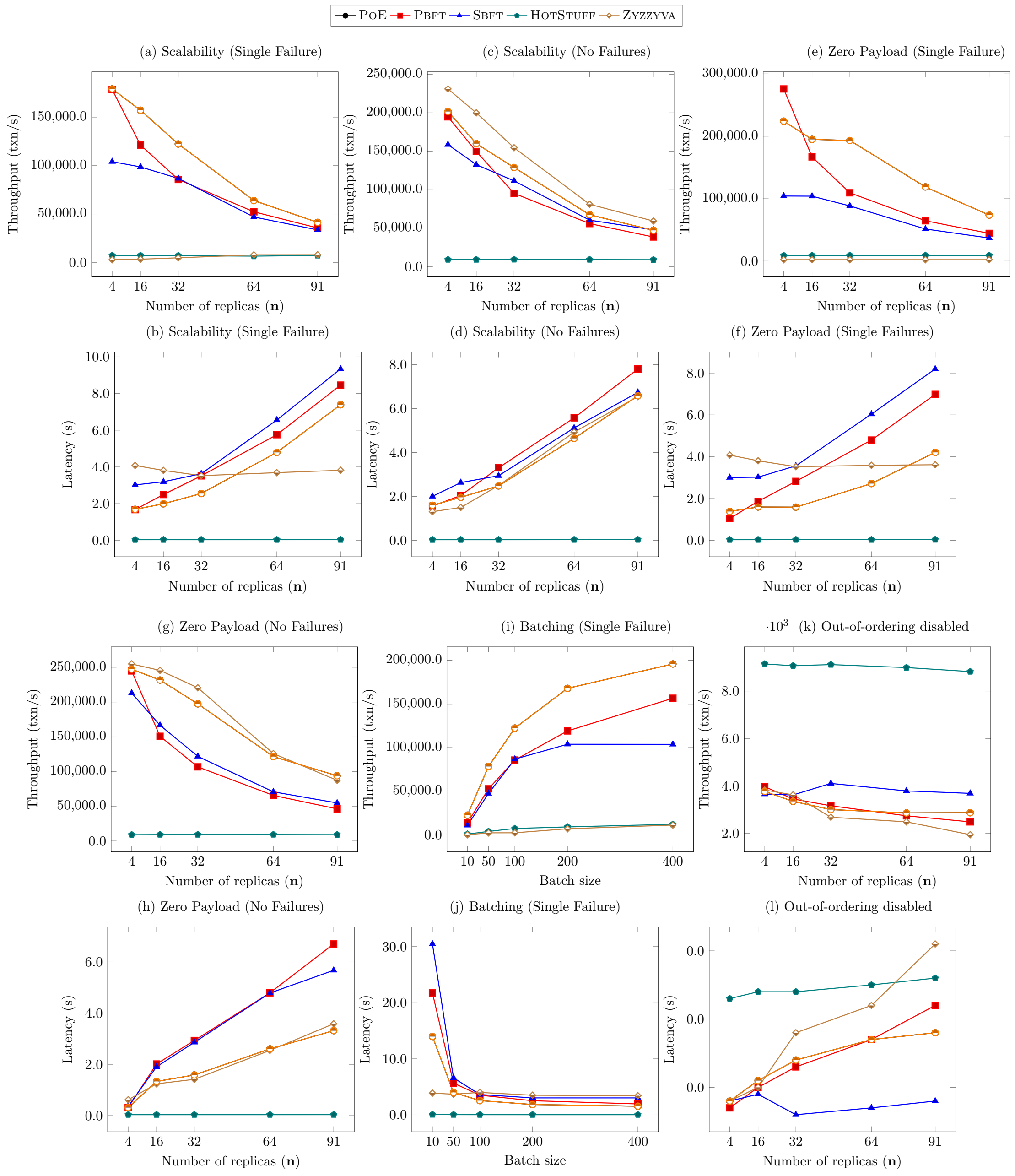


FIGURE 3.9. Evaluating system throughput and average latency incurred by PoE and other BFT protocols.

(1) ***Single Backup Failure.*** We use Figures 3.9(a) and 3.9(b) to illustrate the throughput and latency attained by the system on running different consensus protocols under a backup failure. These graphs affirm our claim that PoE attains higher throughput and incurs lower latency than all other protocols.

In case of PBFT, each replica participates in two phases of quadratic communication, which limits its throughput. For the twin-path protocols such as ZYZZYVA and SBFT, a single failure is sufficient to cause massive reductions in their system throughputs. Notice that the collector in SBFT and the clients in ZYZZYVA have to wait for messages from all **n** replicas, respectively. As predicting an optimal value for timeouts is hard [25, 26], we chose a very small value for the timeout (3 s) for replicas and clients. We justify these values, as the experiments we show later in this section show that the average latency can be as large as 6 s. We note that high timeouts affect ZYZZYVA more than SBFT. In ZYZZYVA, clients are waiting for timeouts during which they stop sending requests, which empties the pipeline at the primary, starving it from new request to propose. To alleviate such issues in real-world deployments of ZYZZYVA, clients need to be able to precisely predict the latency to minimize the time the clients needs to wait between requests. Unfortunately, this is hard and runs the risk of ending up in the expensive slow path of ZYZZYVA whenever the predicted latency is slightly off. In SBFT, the collector may timeout waiting for threshold shares for the k-th round while the primary can continues propose requests for future round l, $l > k$. Hence, in SBFT replicas have more opportunity to occupy themselves with useful work.

HOTSTUFF attains significantly low throughput due to its sequential primary-rotation model in which each of its primaries has to wait for the previous primary before proposing the next request, which leads to a huge reduction in its throughput. Interestingly, HOTSTUFF incurs the least average latency among all protocols. This is a result of intensive load on the system when running other protocols. As these protocols process several requests concurrently (see the multi-threaded architecture in Chapter 5), these requests spend on average more time in the queue before being processed by a replica. Notice that all out-of-order consensus protocols employ this trade off: a small sacrifice on latency yields higher gains on system throughput.

In case of POE, its high throughputs under failures is a result of its three-phase linear protocol that does not rely on any twin-path model. To summarize, POE attains up to 43%, 72%, 24× and 62× more throughputs than PBFT, SBFT, HOTSTUFF and ZYZZYVA.

(2) ***No Replica Failure.*** We use Figures 3.9(c) and 3.9(d) to illustrate the throughput and latency attained by the system on running different consensus protocols in fault-free conditions.

These plots help us to bound the maximum throughput that can be attained by different consensus protocols in our system.

First, as expected, in comparison to the Figures 3.9(a) and 3.9(b), the throughputs for PoE and Pbft are slightly higher. Second, PoE continues to outperform both Pbft and HotStuff, for the reasons described earlier. Third, both Zyzzyva and Sbft are now attaining higher throughputs as their clients and collector no longer timeout, respectively. The key reason Sbft's gains are limited is because Sbft requires five phases and becomes computation bounded. Although Pbft is quadratic, it employs MAC, which are cheaper to sign and verify.

Notice that the differences in throughputs of PoE and Zyzzyva are small. PoE has 20% (on 91 replicas) to 13% (on 4 replicas) less throughputs than Zyzzyva. An interesting observation is that on 91 replicas, Zyzzyva incurs almost the same latency as PoE, even though it has higher throughput. This happens as clients in PoE have to wait for only the fastest $\mathbf{nf} = 61$ replies, whereas a client for Zyzzyva has to wait for replies from all replicas (even the slowest ones). To conclude, PoE attains up to 35%, 27% and 21× more throughput than Pbft, Sbft and HotStuff, respectively.

3.5.4. Scaling Replicas under Zero Payload. We now measure the performance of different protocols under zero payload. In any bft protocol, the primary starts consensus by sending a Propose message that includes all transactions. As a result, this message has the largest size and is responsible for consuming the majority of the bandwidth. A zero payload experiment ensures that each replica executes dummy instructions. Hence, the primary is no longer a bottleneck.

We again run these experiments for both ***Single Failure*** and ***Failure-Free*** cases, and use Figures 3.9(e) to 3.9(h) to illustrate our observations. It is evident from these figures that zero payload experiments have helped in increasing PoE's gains. PoE attains up to 85%, 62% and 27× more throughputs than Pbft, Sbft and HotStuff, respectively. In fact, under failure-free conditions, the throughput attained by PoE is comparable to Zyzzyva. This is easily explained. First, both PoE and Zyzzyva are linear protocols. Second, although in failure-free cases Zyzzyva attains consensus in one phase, its clients need to wait for response from all $\mathbf{n}$ replicas, which gives PoE an opportunity to cover the gap. However, Sbft being a linear protocol does not perform as good as its other linear counterparts. Its throughput is impacted by the delay of five phases.

3.5.5. Impact of Batching under Failures. Next, we study the effect of batching client requests on BFT protocols [21, 137]. To answer this question, we measure performance as function of the number of requests in a batch (*the batch-size*), which we vary between 10 and 400. For this experiment, we use a system with 32 available replicas, of which one replica has failed.

We use Figures 3.9(i) and 3.9(j) to illustrate, for each consensus protocol, the throughput and average latency attained by the system. For each protocol, increasing the batch-size also increases throughput, while decreasing the latency. This happens as larger batch-sizes require fewer consensus rounds to complete the exact same set of requests, reducing the cost of ordering and executing the transactions. This not only improves throughput, but also reduces client latencies as clients receive faster responses for their requests. Although increasing the batch-size reduces the number of consensus rounds, the large message size causes a proportional decrease in throughput (or increase in latency). This is evident from the experiments at higher batch-sizes: increasing the batch-size beyond 100 gradually curves the throughput plots towards a limit for PoE, PBFT and SBFT. For example, on increasing the batch size from 100 to 400, PoE and PBFT see an increase in throughput by 60% and 80%, respectively, while the gap in throughput reduces from 43% to 25%. As in the previous experiments, ZYZZYVA yields a significantly lower throughput as it cannot handle failures. In case of HOTSTUFF, an increase in batch size does increases its throughput but due to high scaling of the graph this change seems insignificant.

3.5.6. Disabling Out-of-Ordering. Until now, we allowed protocols like PBFT, PoE, SBFT and ZYZZYVA to process requests *out-of-order*. As a result, these protocols achieve much higher throughputs than HOTSTUFF, which is restricted by its sequential primary-rotation model. In Figures 3.9(k) and 3.9(l), we evaluate the performance of the protocols when there are no opportunities for out-of-ordering.

In this setting, we require each client to only send its request when it has accepted a response for its previous query. As HOTSTUFF pipelines its phases of consensus into a *four*-phase pipeline, so we allow it to access four client requests (each on a distinct subsequent replica) at any time. As expected, HOTSTUFF performs better than all other protocols at the expense of a higher latency as it rotates primaries at the end of each consensus, which allows it to pipeline four requests. However, notice that once out-of-ordering is disabled, throughput drops from 200 ktransactions/s

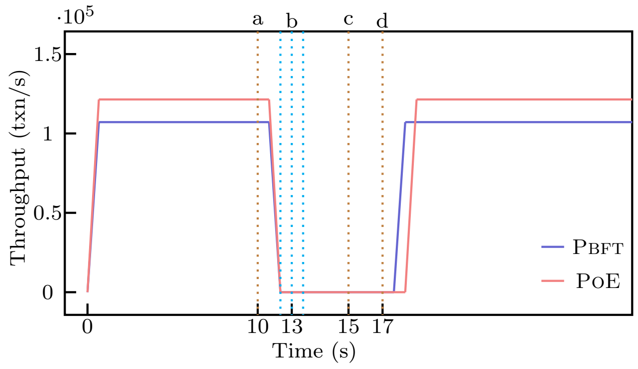


FIGURE 3.10. System throughput under instance failures ($\mathbf{n} = 32$). (a) replicas detect failure of primary and broadcast VC-REQUEST; (b) replicas receives VC-REQUEST from others; (c) replicas receives NV-PROPOSE from new primary; (d) state recovery;

to just under a few *thousand* transactions/s. Hence, from a practical standpoint, out-of-ordering is simply crucial. Further, the difference in latency of different protocols is quite small, and the visible variation is a result of graph scaling while the actual numbers are in the range of 20 ms–40 ms.

3.5.7. Primary Failure–View Change. In Figure 3.10, we study the impact of of a benign primary failure on PoE and Pbft. To recover from a primary failure, backup replicas run the view-change protocol. We skip illustrating view-change plots for Zyzzyva and Sbft as they already face severe reduction in throughput for a single backup failure. Further, Zyzzyva has an *unsafe* view-change algorithm and Sbft's view-change algorithm is no less expensive than Pbft. For HotStuff, we do not show results as it changes primary at the end of every consensus. Although single primary protocols face a momentary loss in throughput during view-change, these protocols easily cover this gap through their ability to process messages out-of-order.

For our experiments, we let the primary replica complete consensus for 10 s (or around a million transactions) and then fail. This causes clients to timeout while waiting for responses for their pending transactions. Hence, these clients forward their requests to backup replicas.

When a backup replica receives a client request, it forwards that request to the primary and waits on a timer. Once a replicas timeouts, it detects a primary failure and broadcasts a VC-REQUEST message to all other replicas—initiate view-change protocol (a). Next, each replica waits for a new view message from the next primary. In the meantime, a replica may receive

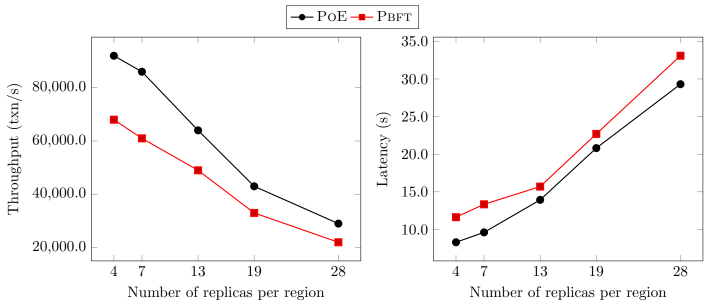


FIGURE 3.11. System throughput and average latency incurred by PoE and PBFT in a WAN deployment of five regions under a single failure. In the largest deployment, we have 140 replicas spread equally over these regions.

VC-REQUEST messages from other replicas (b). Once a replica receives NV-PROPOSE message from the new primary (c), it moves to the next view.

3.5.8. WAN Scalability. In this section, we use Figure 3.11 to illustrate the throughputs and latencies for different PoE and PBFT deployments on a wide-area network in the presence of a single failure. In specific, we deploy clients and replicas across *five* locations across the globe: Oregon, Iowa, Montreal, the Netherlands, and Taiwan. Next, we vary the number of replicas from 20 to 140 by equally distributing these replicas across each region.

These plots affirm our existing observations that PoE outperforms existing state-of-the-art protocols and scales well in wide-area deployments. In specific, PoE achieves up to 1.41× higher throughput and incurs 28.67% less latency than PBFT. We skip presenting plots for SBFT, HOTSTUFF and ZYZZYVA due to their low throughputs under failures.

3.6. Concluding Remarks

We present Proof-of-Execution (PoE), a novel Byzantine fault-tolerant consensus protocol that guarantees safety and liveness and does so in only three linear phases. PoE decouples ordering from execution by allowing replicas to process messages out-of-order and execute client-transactions speculatively. Despite these properties, PoE ensures that all the replicas reach a single unique order for all the transactions. Further, PoE guarantees that if a client observes identical results of execution from a majority of the replicas, then it can reliably mark its transaction committed. Due to speculative execution, PoE may require replicas to revert executed transactions, however. To evaluate PoE's design, we implement it in our RESILIENTDB fabric. Our evaluation shows that PoE achieves up-to-80% higher throughputs than four state-of-the-art BFT protocols in the presence of failures.

3.7. Bibliographic Notes

Consensus is an age-old problem that received much theoretical and practical attention (see, e.g., [80, 89, 107]). Further, the use of rollbacks is common in distributed systems. E.g., the crash-resilient replication protocol Raft [107] allows primaries to re-write the log of any replica. In a Byzantine environment, such an approach would delegate too much power to the primary, as they can maliciously overwrite transactions that need to be preserved.

The interest in practical BFT consensus protocols took off with the introduction of PBFT [21]. Apart from the protocols that we already discussed, there are some interesting protocols that achieve efficient consensus by requiring $5\mathbf{f} + 1$ replicas [1, 34]. However, these protocols have been shown to work only in the cases where transactions are non-conflicting [87]. Some other BFT protocols [24, 133] suggest the use of *trusted components* to reduce the cost of BFT consensus. These works require only $2\mathbf{f} + 1$ replicas as the trusted component helps to guarantee a correct ordering. The safety of these protocols relies on the security of trusted component. In comparison, PoE does (i) not require extra replicas, (ii) not depend on clients, (iii) not require trusted components, and (iv) not need the two phases of quadratic communication required by PBFT.

As a promising future direction, Castro [21] also suggested exploring speculative optimizations for PBFT, which he referred to as tentative execution. However, this lacked: (i) formal description, (ii) non-divergence safety property, (iii) specification of rollback under attacks, (iv) re-examination of the view change protocol, and (v) any actual evaluation.

Consensus for Blockchains. Since the introduction of Bitcoin [101], the well-known cryptocurrency that led to the coining of the term blockchain, several new BFT consensus protocols that cater to cryptocurrencies have been designed [77, 86]. Bitcoin [101] employs the *Proof-of-Work* [77] consensus protocol (PoW), which is computationally intensive, achieves low throughput, and can cause forks (divergence) in the blockchain: separate chains can exist on non-faulty replicas, which in turn can cause *double-spending attacks* [57]. Due to these limitations, several other similar algorithms have been proposed. E.g., *Proof-of-Stake* (PoS) [86], which is design such that any replica owning $n\%$ of the total resources gets the opportunity to create $n\%$ of the new blocks. As PoS is resource driven, it can face attacks where replicas are incentivized to work simultaneously on several forks of the blokchain, without ever trying to eliminate these forks.

There are also a set of interesting alternative designs such as ConFlux [95], Caper [6] and MeshCash [12] that suggest the use of directed acyclic graphs (DAGs) to store a blockchain to improve the performance of Bitcoin. However, these protocols either rely on PoW or Pbft for consensus. PoE does not face the limitations faced by PoW [77] and PoS [86]. The use of DAGs [6, 12, 95], and sharding [35, 138] is orthogonal to the design of PoE. Hence, their use with PoE can reap further benefits. Further, PoE can be employed by meta-protocols and does not restrict consensus to any subset of replicas.

CHAPTER 4

RCC: Resilient Concurrent Consensus for High-Throughput Secure Transaction Processing

At the core of consensus-based systems are *consensus protocols* that enable independent participants to manage a single common database by reliably and continuously replicating a unique sequence of transactions among all participants. Most practical systems use consensus protocols that follow the classical primary-backup design of PBFT [21] in which a single replica, the *primary*, proposes transactions by broadcasting them to all other replicas, after which all replicas *exchange state* to determine whether the primary correctly proposes the same transaction to all replicas and to deal with failure of the primary. Well-known examples of such protocols are PBFT [21], ZYZZYVA [88], SBFT [48], HOTSTUFF [137], and POE [65], and fully-optimized implementations of these protocols are able to process up-to tens-of-thousands transactions per second [64].

4.1. The Limitations of Traditional Consensus

Unfortunately, a close look at the design of primary-backup consensus protocols reveals that their design *underutilized available network resources*, which prevents the *maximization of transaction throughput*: the throughput of these protocols is determined mainly by the outgoing bandwidth of the primary. To illustrate this, we consider the maximum throughput by which primaries can replicate transactions. Consider a system with $\mathbf{n}$ replicas of which $\mathbf{f}$ are faulty and the remaining $\mathbf{nf} = \mathbf{n} - \mathbf{f}$ are non-faulty. The maximum throughput $T_{\max}$ of any such protocol is determined by the outgoing bandwidth B of the primary, the number of replicas $\mathbf{n}$, and the size of transactions st: $T_{\max} = B/((\mathbf{n}-1)st)$. No practical consensus protocol will be able to achieve this throughput, as dealing with crashes and malicious behavior requires substantial state exchange. Protocols such as ZYZZYVA [88] can come close, however, by optimizing for the case in which no faults occur, this at the cost of their ability to deal with faults efficiently.

For PBFT, the minimum amount of state exchange consists of two rounds in which PREPARE and COMMIT messages are exchanged between all replicas (a quadratic amount, see Example 4.4.1 in Section 4.4). Assuming that these messages have size *sm*, the maximum throughput of PBFT is $T_{\text{PBFT}} = B/((\mathbf{n}-1)(st+3sm))$. To minimize overhead, typical implementations of PBFT group hundreds of transactions together, assuring that $st \gg sm$ and, hence, $T_{\max} \approx T_{\text{PBFT}}$.

The above not only shows a maximum on throughput, but also that primary-backup consensus protocols such as PBFT and ZYZZYVA severely *underutilize resources* of non-primary replicas: when $st \gg sm$, the primary sends and receives roughly $(\mathbf{n}-1)st$ bytes, whereas all other replicas only send and receive roughly *st* bytes. The obvious solution would be to use several primaries. Unfortunately, recent protocols such as HOTSTUFF [137], SPINNING [132], and PRIME [5] that regularly *switch primaries* all require that a switch from a primary happens after all proposals of that primary are processed. Hence, such primary switching does load balance overall resource usage among the replicas, but does not address the underutilization of resources we observe.

4.2. Our Solution: Towards Resilient Concurrent Consensus

The only way to push throughput of consensus-based databases and data processing systems beyond the limit $T_{\max}$, is by better utilizing available resources. In this work, we propose to do so via *concurrent consensus*, in which we use many primaries that *concurrently* propose transactions. We also propose RCC, a paradigm for the realization of concurrent consensus. Our contributions are as follows:

(1) First, in Section 4.3, we propose *concurrent consensus* and show that concurrent consensus can achieve much higher throughput than primary-backup consensus by effectively utilizing all available system resources.
(2) Then, in Section 4.4, we propose RCC, a paradigm for turning any primary-backup consensus protocol into a concurrent consensus protocol and that is designed for maximizing throughput in all cases, even during malicious activity.
(3) Then, in Section 4.5, we show that RCC can be utilized to make systems *more resilient*, as it can mitigate the effects of order-based attacks and throttling attacks (which are not prevented by traditional consensus protocols), and can provide better load balancing.

(4) Finally, in Section 4.6, we put the design RCC to the test by implementing it in ResilientDB, our high-performance resilient blockchain fabric, and compare RCC with state-of-the-art primary-backup consensus protocols. Our comparison shows that RCC answers the promises of *concurrent consensus*: it achieves up to $2.75\times$ higher throughput than other consensus protocols, has a peak throughput of $365\,\text{ktxn/batch}$ and can be easily scaled to 91 replicas.

4.3. The Promise of Concurrent Consensus

To deal with the underutilization of resources and the low throughput of primary-backup consensus, we propose *concurrent consensus*. In specific, we design for a system that is optimized for high-throughput scenarios in which a plentitude of transactions are available, and we make every replica a *concurrent primary* that is responsible for proposing and replicating some of these transactions. As we have $\mathbf{nf}$ non-faulty replicas, we can expect to always *concurrently* propose at least $\mathbf{nf}$ transactions if sufficient transactions are available. Such concurrent processing has the potential to drastically improve throughput: in each round, each primary will send out one proposal to all other replicas, and receive $\mathbf{nf}-1$ proposals from other primaries. Hence, the maximum concurrent throughput is $T_{\text{cmax}} = \mathbf{nf}B/((\mathbf{n}-1)st + (\mathbf{nf}-1)st)$.

In practice, of course, the primaries also need to participate in state exchange to determine the correct operations of all concurrent primaries. If we use Pbft-style state exchange, we end up with a concurrent throughput of $T_{\text{cP\scriptsize BFT}} = \mathbf{nf}B/((\mathbf{n}-1)(st+3sm) + (\mathbf{nf}-1)(st+4(\mathbf{n}-1)sm))$. In Figure 4.1, we have sketched the maximum throughputs T_{max}, $T_{\text{P\scriptsize BFT}}$, T_{cmax}, and $T_{\text{cP\scriptsize BFT}}$. As one can see, concurrent consensus not only promises greatly improved throughput, but also sharply reduces the costs associated with scaling consensus. We remark, however, that these figures provide best-case *upper-bounds*, as they only focus on bandwidth usage. In practice, replicas are also limited by computational power and available memory buffers that puts limits on the number of transactions they can process in parallel and can execute (see Section 4.6.2).

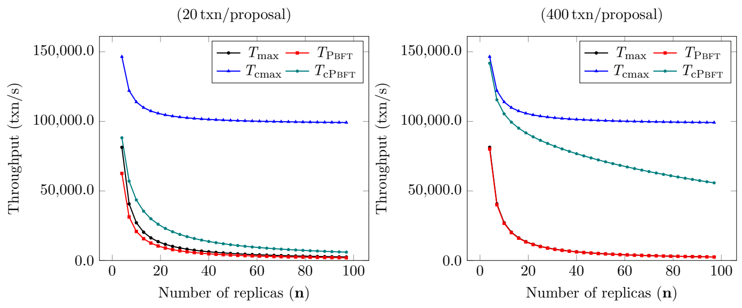


FIGURE 4.1. Maximum throughput of replication in a system with $B = 1$ Gbit/s, $\mathbf{n} = 3\mathbf{f} + 1$, $\mathbf{nf} = 2\mathbf{f} + 1$, $sm = 1$ KiB, and individual transactions are 512 B. On the *left*, each proposal groups 20 transactions ($st = 10$ KiB) and on the *right*, each proposal groups 400 transactions ($st = 2$ MiB).

4.4. RCC: Resilient Concurrent Consensus

The idea behind concurrent consensus, as outlined in the previous section, is straightforward: improve overall throughput by using all available resources via concurrency. Designing and implementing a concurrent consensus system that operates correctly, even during crashes and malicious behavior of some replicas, is challenging, however. In this section, we describe how to design correct *consensus protocols* that deliver on the promises of concurrent consensus. We do so by introducing RCC, a paradigm that can turn any primary-backup consensus protocol into a concurrent consensus protocol. At its basis, RCC makes every replica a primary of a consensus-instance that replicates transactions among all replicas. Furthermore, RCC provides the necessary coordination between these consensus-instances to coordinate execution and deal with faulty primaries. To assure resilience and maximize throughput, we put the following design goals in RCC:

(1) RCC provides *consensus* among replicas on the client transactions that are to be executed and the order in which they are executed.
(2) Clients can interact with RCC to force execution of their transactions and learn the outcome of execution.
(3) RCC is a design paradigm that can be applied to any primary-backup consensus protocol, turning it into a concurrent consensus protocol.

(4) In RCC, consensus-instances with non-faulty primaries are *always* able to propose transactions at maximum throughput (with respect to the resources available to any replica), this independent of faulty behavior by any other replica.
(5) In RCC, dealing with faulty primaries does not interfere with the operations of other consensus-instances.

Combined, design goals D4 and D5 imply that instances with non-faulty primaries can propose transactions *wait-free*: transactions are proposed concurrent to any other activities and does not require any coordination with other instances.

4.4.1. Background on Primary-Backup Consensus and PBFT. Before we present RCC, we provide the necessary background and notation for primary-backup consensus. Typical primary-backup consensus protocols operate in *views*. Within each view, a primary can propose client transactions, which will then be executed by all non-faulty replicas. To assure that all non-faulty replicas maintain the same state, transactions are required to be *deterministic*: on identical inputs, execution of a transaction must always produce identical outcomes. To deal with faulty behavior by the primary or by any other replicas during a view, three complimentary mechanisms are used:

Byzantine commit. The primary uses a *Byzantine commit algorithm* BCA to propose a client transaction T to all replicas. Next, BCA will perform state exchange to determine whether the primary successfully proposed a transaction. If the primary is non-faulty, then all replicas will receive T and determine success. If the primary is faulty and more than $\mathbf{f}$ non-faulty replicas do *not* receive a proposal or receive different proposals than the other replicas, then the state exchange step of BCA will detect this failure of the primary.

Primary replacement. The replicas use a *view-change algorithm* to replace the primary of the current view v when this primary is detected to be faulty by non-faulty replicas. This view-change algorithm will collect the state of sufficient replicas in view v to determine a correct starting state for the next view $v + 1$ and assign new primary that will propose client transactions in view $v + 1$.

Recovery. A faulty primary can keep up to $\mathbf{f}$ non-faulty replicas *in the dark* without being detected, as $\mathbf{f}$ faulty replicas can cover for this malicious behavior. Such behavior is not detected and, consequently, does not trigger a view-change. Via a *checkpoint algorithm* the at-most-$\mathbf{f}$ non-faulty replicas that are in the dark will learn the proposed client transactions that are successfully proposed to the remaining at-least-$\mathbf{nf} - \mathbf{f} > \mathbf{f}$ non-fault replicas (that are not in the dark).

EXAMPLE 4.4.1. *Next, we illustrate these mechanisms in* PBFT. *At the core of* PBFT *is the* preprepare-prepare-commit *Byzantine commit algorithm. This algorithm operates in three phases, which are sketched in Figure 3.1.*

First, the current primary chooses a client request of the form $\langle c \rangle_T$, a transaction T signed by client c, and proposes this request as the ρ-th transaction by broadcasting it to all replicas via a PREPREPARE *message m. Next, each non-faulty replica R* prepares *the first proposed ρ-th transaction it receives by broadcasting a* PREPARE *message for m. If a replica R receives* $\mathbf{nf}$ PREPARE *messages for m from* $\mathbf{nf}$ *distinct replicas, then it has the guarantee that any group of* $\mathbf{nf}$ *replicas will contain a* non-faulty replica *that has received m. Hence, R has the guarantee that m can be recovered from any group of* $\mathbf{nf}$ *replicas, independent of the behavior of the current primary. With this guarantee, R* commits *to m by broadcasting a* COMMIT *message for m. Finally, if a replica R receives* $\mathbf{nf}$ COMMIT *messages for m from* $\mathbf{nf}$ *distinct replicas, then it* accepts *m. In* PBFT, *accepted proposals are then* executed *and the client is informed of the outcome.*

Each replica R participating in preprepare-prepare-commit uses an internal timeout value *to detect failure: whenever the primary fails to coordinate a round of preprepare-prepare-commit—which should result in R accepting some proposal—R will detect* failure *of the primary and halt participation in preprepare-prepare-commit. If* $\mathbf{f} + 1$ *non-faulty replicas detect such a failure and communication is reliable, then they can cooperate to assure that all non-faulty replicas detect the failure. We call this a* confirmed failure *of preprepare-prepare-commit. In* PBFT, *confirmed failures trigger a* view-change. *Finally,* PBFT *employs a majority-vote* checkpoint protocol *that allows replicas that are kept in the dark to learn* accepted *proposals without help of the primary.*

4.4.2. The Design of RCC**.** We now present RCC in detail. Consider a primary-backup consensus protocol P that utilizes Byzantine commit algorithm BCA (e.g., PBFT with preprepare-prepare-commit). At the core of applying our RCC paradigm to P is running $\mathbf{m}$, $1 \leq \mathbf{m} \leq \mathbf{n}$, instances of BCA *concurrently*, while providing sufficient coordination between the instances to deal with any malicious behavior. To do so, RCC makes BCA *concurrent* and uses a checkpoint protocol for per-instance recovery of in-the-dark replicas (see Section 4.4.4). Instead of view-changes, RCC uses a novel wait-free mechanism, that does not involve replacing primaries, to deal with detectable primary failures (see Section 4.4.3). RCC requires the following guarantees on BCA:

ASSUMPTION. *Consider an instance of* BCA *running in a system with* $\mathbf{n}$ *replicas,* $\mathbf{n} > 3\mathbf{f}$.

(1) *If no failures are detected in round* ρ *of* BCA *(the round is* successful*), then at least* $\mathbf{nf} - \mathbf{f}$ *non-faulty replicas have* accepted *a proposed transaction in round* ρ.

(2) *If a non-faulty replica* accepts *a proposed transaction* T *in round* ρ *of* BCA*, then all other non-faulty replicas that accepted a proposed transaction, accepted* T.

(3) *If a non-faulty replica* accepts *a transaction* T*, then* T *can be recovered from the state of any subset of* $\mathbf{nf} - \mathbf{f}$ *non-faulty replicas.*

(4) *If the primary is non-faulty and communication is reliable, then all non-faulty replicas will accept a proposal in round* ρ *of* BCA.

With minor fine-tuning, these assumptions are met by PBFT, ZYZZYVA, SBFT, HOTSTUFF, and many other primary-backup consensus protocols, meeting design goal D3.

RCC operates in rounds. In each round, RCC replicates $\mathbf{m}$ client transactions (or, as discussed in Section 4.1, $\mathbf{m}$ sets of client transactions), one for each instance. We write $\mathcal{I}_i$ to denote the i-th instance of BCA. To enforce that each instance is coordinated by a distinct primary, the i-th replica $\mathcal{P}_i$ is assigned as the primary coordinating $\mathcal{I}_i$. Initially, RCC operates with $\mathbf{m} = \mathbf{n}$ instances. In RCC, instances can fail and be *stopped*, e.g., when coordinated by malicious primaries or during periods of unreliable communication. Each round ρ of RCC operates in three steps:

(1) *Concurrent* BCA. First, each replica participates in $\mathbf{m}$ instances of BCA, in which each instance is proposing a transaction requested by a client among all replicas.

(2) *Ordering.* Then, each replica collects all successfully replicated client transactions and puts them in the same—deterministically determined—*order.*

(3) *Execution.* Finally, each replica *executes* the transactions of round ρ in order and informs the clients of the outcome of their requested transactions.

Figure 4.2 sketches a high-level overview of running $\mathbf{m}$ *concurrent instances of* BCA.

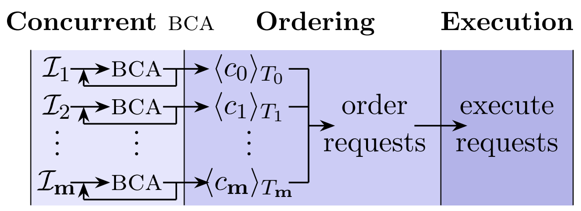

Concurrent BCA
Ordering
Execution
$\mathcal{I}_1$
BCA
$\langle c_0 \rangle_{T_0}$
$\mathcal{I}_2$
BCA
$\langle c_1 \rangle_{T_1}$
$\mathcal{I}_{\mathbf{m}}$
BCA
$\langle c_{\mathbf{m}} \rangle_{T_{\mathbf{m}}}$
order requests
execute requests

FIGURE 4.2. A high-level overview of RCC running at replica R. Replica R participates in $\mathbf{m}$ concurrent instances of BCA (that run independently and continuously output transactions). The instances yield $\mathbf{m}$ transactions, which are executed in a deterministic order.

To maximize performance, we want every instance to propose distinct transactions, such that every round results in $\mathbf{m}$ distinct transactions. In Section 4.4.5, we delve into the details by which primaries can choose transactions to propose.

To meet design goal D4 and D5, individual BCA instances in RCC can continuously propose and replicate transactions: ordering and execution of the transactions replicated in a round by the $\mathbf{m}$ instances is done *in parallel* to the proposal and replication of transactions for future rounds. Consequently, non-faulty primaries can utilize their entire outgoing network bandwidth for proposing transactions, even if other replicas or primaries are acting malicious.

Let $\langle c_i \rangle_{T_i}$ be the transaction T_i requested by c_i and proposed by $\mathcal{P}_i$ in round ρ. After all $\mathbf{m}$ instances complete round ρ, each replica can collect the set of transactions $S = \{\langle c_i \rangle_{T_i} \mid 1 \leq i \leq \mathbf{m}\}$. By Assumption A2, all non-faulty replicas will obtain the same set S. Next, all replicas choose an order on S and execute all transactions in that order. For now, we assume that the transaction $\langle c_i \rangle_{T_i}$ is executed as the i-th transaction of round ρ. In Section 4.5, we show that a more advanced ordering-scheme can further improve the resilience of consensus against malicious behavior. As a direct consequence of Assumption A4, we have the following:

PROPOSITION 4.4.1. *Consider* RCC *running in a system with* $\mathbf{n}$ *replicas,* $\mathbf{n} > 3\mathbf{f}$. *If all* $\mathbf{m}$ *instances have non-faulty primaries and communication is reliable, then, in each round, all non-faulty replicas will accept the same set of* $\mathbf{m}$ *transactions and execute them in the same order.*

As all non-faulty replicas will execute each transaction in $\langle c_i \rangle_{T_i} \in S$, there are $\mathbf{nf}$ distinct non-faulty replicas that can inform the client of the outcome of execution. As all non-faulty replicas operate deterministically and execute the transactions in the same order, client c_i will receive identical outcomes of $\mathbf{nf} > \mathbf{f}$ replicas, guaranteeing that this outcome is correct.

In the above, we described the normal-case operations of RCC. As in normal primary-backup protocols, individual instances in RCC can be subject to both *detectable* and *undetectable* failures. Next, we deal with these two types of failures.

4.4.3. Dealing with Detectable Failures. Consensus-based systems typically operate in an environment with asynchronous communication: messages can get lost, arrive with arbitrary delays, and in arbitrary order. Consequently, it is impossible to distinguish between, on the one hand, a primary that is malicious and does not send out proposals and, on the other hand, a primary that does send out proposals that get lost in the network. As such, asynchronous consensus protocols can only provide *progress* in periods of *reliable bounded-delay communication* during which all messages sent by non-faulty replicas will arrive at their destination within some maximum delay [42, 47].

To be able to deal with failures, RCC assumes that *any failure* of non-faulty replicas to receive proposals from a primary $\mathcal{P}_i$, $1 \leq i \leq \mathbf{m}$, is due to *failure* of $\mathcal{P}_i$, and we design the recovery process such that it can also recover from failures due to unreliable communication. Furthermore, in accordance with the wait-free design goals D4 and D5, the recovery process will be designed so that it does not interfere with other BCA instances or other recovery processes. Now assume that primary $\mathcal{P}_i$ of $\mathcal{I}_i$, $1 \leq i \leq \mathbf{m}$, fails in round ρ. The recovery process consists of three steps:

(1) All non-faulty replicas need to detect failure of the $\mathcal{P}_i$.
(2) All non-faulty replicas need to reach agreement on the state of $\mathcal{I}_i$: which transactions have been proposed by $\mathcal{P}_i$ and have been accepted in the rounds up-to-ρ.
(3) To deal with unreliable communication, all non-faulty replicas need to determine the round in which $\mathcal{P}_i$ is allowed to resume its operations.

To reach agreement on the state of $\mathcal{I}_i$, we rely on a separate instance of the consensus protocol P that is only used to coordinate agreement on the state of $\mathcal{I}_i$ during failure. This coordinating consensus protocol P replicates $\texttt{stop}(i; E)$ operations, in which E is a set of $\mathbf{nf}$ FAILURE messages sent by $\mathbf{nf}$ distinct replicas from which all accepted proposals in instance $\mathcal{I}_i$ can be derived. We notice that P is—itself—an instance of a primary-backup protocol that is coordinated by some primary $\mathcal{L}_i$ (based on the current view in which the instance of P operates), and we use the standard

```
    Recovery request role (used by replica $R$) :
1:  event $R$ detects failure of the primary $\mathcal{P}_i$, $1 \leq i \leq \mathbf{m}$, in round $\rho$ do
2:      $R$ halts $\mathcal{I}_i$.
3:      Let $P$ be the state of $R$ in accordance to Assumption A3.
4:      Broadcast $\text{FAILURE}(i, \rho, P)$ to all replicas.
5:  event $R$ receives $\mathbf{f}+1$ messages $m_j = \text{FAILURE}(i, \rho_j, P_j)$ such that:
            (1) these messages are sent by a set $S$ of $|S| = \mathbf{f}+1$ distinct replicas;
            (2) all $\mathbf{f}+1$ messages are well-formed; and
            (3) $\rho_j$, $1 \leq j \leq \mathbf{f}+1$, comes after the round in which $\mathcal{I}_i$ started last
    do
6:      $R$ detects failure of $\mathcal{P}_i$ (if not yet done so).

    Recovery leader role (used by leader $\mathcal{L}_i$ of P) :
7:  event $\mathcal{L}_i$ receives $\mathbf{nf}$ messages $m_j = \text{FAILURE}(i, \rho_j, P_j)$ such that
            (1) these messages are sent by a set $S$ of $|S| = \mathbf{f}+1$ distinct replicas;
            (2) all $\mathbf{nf}$ messages are well-formed; and
            (3) $\rho_j$, $1 \leq j \leq \mathbf{f}+1$, comes after the round in which $\mathcal{I}_i$ started last
    do
8:      Propose $\texttt{stop}(i; \{m_1, \dots, m_{\mathbf{nf}}\})$ via P.

    State recovery role (used by replica $R$) :
9:  event $R$ accepts $\texttt{stop}(i; E)$ from $\mathcal{L}_i$ via P do
10:     Recover the state of $\mathcal{I}_i$ using $E$ in accordance to Assumption A3.
11:     Determine the last round $\rho$ for which $\mathcal{I}_i$ accepted a proposal.
12:     Set $\rho + 2^f$, with $f$ the number of accepted $\texttt{stop}(i; E')$ operations, as the next valid round number for instance
        $\mathcal{I}_i$.
```

FIGURE 4.3. The *recovery algorithm* of RCC.

machinery of P to deal with failures of that leader (see Section 4.4.1). Next, we shall describe how the recovery process is initiated. The details of this protocol can be found in Figure 4.3.

When a replica R detects failure of instance $\mathcal{I}_i$, $0 \leq i < \mathbf{m}$, in round ρ, it broadcasts a message $\text{FAILURE}(i, \rho, P)$, in which P is the state of R in accordance to Assumption A3 (Line 1 of Figure 4.3). To deal with unreliable communication, R will continuously broadcast this FAILURE message with an exponentially-growing delay until it learns on how to proceed with $\mathcal{I}_i$. To reduce communication in the normal-case operations of P, one can send the full message $\text{FAILURE}(i, \rho, P)$ to only $\mathcal{L}_i$, while sending $\text{FAILURE}(i, \rho)$ to all other replicas.

If a replica receives $\mathbf{f}+1$ FAILURE messages from distinct replicas for a certain instance $\mathcal{I}_i$, then it received at least one such message from a non-faulty replica. Hence, it can detect failure of $\mathcal{I}_i$ (Line 5 of Figure 4.3). Finally, if a replica R receives $\mathbf{nf}$ FAILURE messages from distinct replicas for a certain instance $\mathcal{I}_i$, then we say there is a *confirmed failure*, as R has the guarantee that eventually—within at most two message delays—also the primary $\mathcal{L}_i$ of P will receive $\mathbf{nf}$ FAILURE messages (if communication is reliable). Hence, at this point, R sets a timer based on some internal timeout value (that estimates the message delay) and waits on the leader $\mathcal{L}_i$ to propose a valid `stop`-operation or for the timer to run out. In the latter case, replica R detects failure of the leader $\mathcal{L}_i$ and follows the steps of a view-change in P to (try to) replace $\mathcal{L}_i$. When the leader $\mathcal{L}_i$ receives $\mathbf{nf}$ FAILURE messages, it can and must construct a valid `stop`-operation and reach consensus on this operation (Line 7 of Figure 4.3). After reaching consensus, each replica can recover to a common state of $\mathcal{I}_i$:

THEOREM 4.4.2. *Consider* RCC *running in a system with* $\mathbf{n}$ *replicas. If* $\mathbf{n} > 3\mathbf{f}$*, an instance* $\mathcal{I}_i$*,* $0 \leq i < \mathbf{m}$*, has a confirmed failure, and the last proposal of* $\mathcal{P}_i$ *accepted by a non-faulty replica was in round* ρ*, then—whenever communication becomes reliable—the recovery protocol of Figure 4.3 will assure that all non-faulty replicas will recover the same state, which will include all proposals accepted by non-faulty replicas before-or-at round* ρ*.*

PROOF. If communication is reliable and instance $\mathcal{I}_i$ has a confirmed failure, then all non-faulty replicas will detect this failure and send FAILURE messages (Line 1 of Figure 4.3). Hence, all replicas are guaranteed to receive at least $\mathbf{nf}$ FAILURE messages, and any replica will be able to construct a well-formed operation $\texttt{stop}(i; E)$. Hence, P will eventually be forced to reach consensus on $\texttt{stop}(i; E)$. Consequently, all non-faulty replicas will conclude on the same state for instance $\mathcal{I}_i$. Now consider a transaction T accepted by non-faulty replica Q in instance $\mathcal{I}_i$. Due to Assumption A3, Q will only accept T if T can be recovered from the state of any set of $\mathbf{nf} - \mathbf{f}$ non-faulty replicas. As $|E| = \mathbf{nf}$ (Line 7 of Figure 4.3), the set E contains the state of $\mathbf{nf} - \mathbf{f}$ non-faulty replicas. Hence, T must be recoverable from E. □

We notice that the recovery algorithm of RCC, as outlined in Figure 4.3, only affects the capabilities of the BCA instance that is stopped. All other BCA instances can concurrently propose transactions for current and for future rounds. Hence, the recovery algorithm adheres to the wait-free design goals D4 and D5. Furthermore, we reiterate that we have separate instance of the coordinating consensus protocol for each instance $\mathcal{I}_i$, $1 \leq i \leq \mathbf{m}$. Hence, recovery of several instances can happen concurrently, which minimizes the time it takes to recover from several simultaneous primary failures and, consequently, minimizes the delay before a round can be executed during primary failures.

Confirmed failures not only happen due to malicious behavior. Instances can also fail due to periods of unreliable communication. To deal with this, we eventually restart any stopped instances. To prevent instances coordinated by malicious replicas to continuously cause recovery of their instances, every failure will incur an exponentially growing restart penalty (Line 12 of Figure 4.3). The exact round in which an instance can resume operations can be determined deterministically from the accepted history of `stop`-requests. When all instances have round failures due to unreliable communication (which can be detected from the history of `stop`-requests), any instance is allowed to resume operations in the earliest available round (after which all other instances are also required to resume operations).

4.4.4. Dealing with Undetectable Failures. As stated in Assumption A1, a malicious primary $\mathcal{P}_i$ of a BCA instance $\mathcal{I}_i$ is able to keep up to $\mathbf{f}$ non-faulty replicas in the dark without being detected. In normal primary-backup protocols, this is not a huge issue: at least $\mathbf{nf} - \mathbf{f} > \mathbf{f}$ non-faulty replicas still accept transactions, and these replicas can execute and reliably inform the client of the outcome of execution. This is not the case in RCC, however:

EXAMPLE 4.4.3. *Consider a system with* $\mathbf{n} = 3\mathbf{f} + 1 = 7$ *replicas. Assume that primaries* $\mathcal{P}_1$ *and* $\mathcal{P}_2$ *are malicious, while all other primaries are non-faulty. We partition the non-faulty replicas into three sets* A_1*,* A_2*, and* B *with* $|A_1| = |A_2| = \mathbf{f}$ *and* $|B| = 1$*. In round* ρ*, the malicious primary* $\mathcal{P}_i$*,* $i \in \{1, 2\}$*, proposes transaction* $\langle c_i \rangle_{T_i}$ *to only the non-faulty replicas in* $A_i \cup B$*. This situation is sketched in Figure 4.4. After all concurrent instances of* BCA *finish round* ρ*, we see that the replicas in* A_1 *have accepted* $\langle c_1 \rangle_{T_1}$*, the replicas in* A_2 *have accepted* $\langle c_2 \rangle_{T_2}$*, and only the replica in* B *has accepted both* $\langle c_1 \rangle_{T_1}$ *and* $\langle c_2 \rangle_{T_2}$*. Hence, only the single replica in* B *can proceed with execution of round* ρ*. Notice that, due to Assumption A1, we consider all instances as finished successfully. If* $\mathbf{n} \geq 10$ *and* $\mathbf{f} \geq 3$*, this example attack can be generalized such that also the replica in* B *is missing at least a single client transaction.*

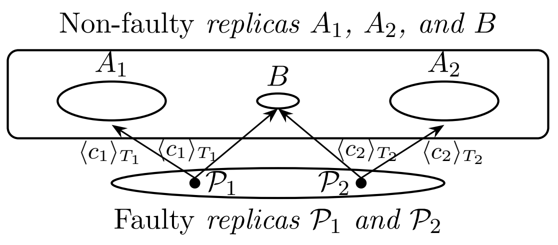

Non-faulty *replicas* A_1, A_2, *and* B
A_1
B
A_2
$\langle c_1 \rangle_{T_1}$
$\langle c_1 \rangle_{T_1}$
$\langle c_2 \rangle_{T_2}$
$\langle c_2 \rangle_{T_2}$
$\mathcal{P}_1$
$\mathcal{P}_2$
Faulty *replicas* $\mathcal{P}_1$ *and* $\mathcal{P}_2$

FIGURE 4.4. An attack possible when parallelizing BCA: malicious primaries can prevent non-faulty replicas from learning all client requests in a round, thereby preventing timely round execution. The faulty primary $\mathcal{P}_i$, $i \in \{1, 2\}$, does so by only letting non-faulty replicas $A_i \cup B$ participate in instance $\mathcal{I}_i$.

To deal with *in-the-dark* attacks of Example 4.4.3, we can run a standard checkpoint algorithm for each BCA instance: if the system does not reach confirmed failure of $\mathcal{P}_i$ in round ρ, $1 \leq i \leq \mathbf{m}$, then, by Assumption A1 and A2, at-least-$\mathbf{nf} - \mathbf{f}$ non-faulty replicas have accepted the same transaction T in round ρ of $\mathcal{I}_i$. Hence, by Assumption A3, a standard checkpoint algorithm (e.g., the one of PBFT or one based on delayed replication [70]) that exchanges the state of these at-least-$\mathbf{nf} - \mathbf{f}$ non-faulty replicas among all other replicas is sufficient to assure that all non-faulty replicas eventually accept T. We notice that these checkpoint algorithms can be run concurrently with the operations of BCA instances, thereby adhering to our wait-free design goals D4 and D5.

To reduce the cost of checkpoints, typical consensus systems only perform checkpoints after every x-th round for some system-defined constant x. Due to in-the-dark attacks, applying such a strategy to RCC means choosing between execution latency and throughput. Consequently, in RCC we do checkpoints on a dynamic *per-need basis*: when replica R receives $\mathbf{nf} - \mathbf{f}$ claims of failure of primaries (via the FAILURE messages of the recovery protocol) in round ρ and R itself finished round ρ for all its instances, then it will participate in any attempt for a checkpoint for round ρ. Hence, if an in-the-dark attack affects more than $\mathbf{f}$ distinct non-faulty replicas in round ρ, then a successful checkpoint will be made and all non-faulty replicas recover from the attack, accept all transactions in round ρ, and execute all these transactions.

Using Theorem 4.4.2 to deal with detectable failures and using checkpoint protocols to deal with replicas in-the-dark, we conclude that RCC adheres to design goal D1:

THEOREM 4.4.4. *Consider* RCC *running in a system with* $\mathbf{n}$ *replicas. If* $\mathbf{n} > 3\mathbf{f}$*, then* RCC *provides consensus in periods in which communication is reliable.*

4.4.5. Client Interactions with RCC**.** To maximize performance, it is important that every instance proposes distinct client transactions, as proposing the same client transaction several times would reduce throughput. We have designed RCC with faulty clients in mind, hence, we do not expect cooperation of clients to assure that they send their transactions to only a single primary.

To be able to do so, the design of RCC is optimized for the case in which there are always many more concurrent clients than replicas in the system. In this setting, we assign every client c to a single primary $\mathcal{P}_i$, $1 \leq i \leq \mathbf{m} = \mathbf{n}$, such that only instance $\mathcal{I}_i$ can propose client requests of c. For this design to work in *all cases*, we need to solve two issues, however: we need to deal with situations in which primaries do not receive client requests (e.g., during downtime periods in which only few transactions are requested), and we need to deal with faulty primaries that refuse to propose requests of some clients.

First, if there are less concurrent clients than replicas in the system, e.g., when demand for services is low, then RCC still needs to process client transactions correctly, but it can do so without optimally utilizing resources available, as this would not impact throughput in this case due to the low demands. If a primary $\mathcal{P}_i$, $1 \leq i \leq \mathbf{m}$, does not have transactions to propose in any round ρ and $\mathcal{P}_i$ detects that other BCA instances are proposing for round ρ (e.g., as it receives proposals), then $\mathcal{P}_i$ proposes a small no-op-request instead.

Second, to deal with a primary $\mathcal{P}_i$, $1 \leq i \leq \mathbf{m}$, that refuses to propose requests of some clients, we take a two-step approach. First, we incentivize malicious primaries to *not refuse* services, as otherwise they will be detected faulty and loose the ability to propose transactions altogether. To detect failure of $\mathcal{P}_i$, RCC uses standard techniques to enable a client c to *force* execution of a transaction T. First, c broadcasts $\langle c \rangle_T$ to all replicas. Each non-faulty replica R will then forward $\langle c \rangle_T$ to the appropriate primary $\mathcal{P}_i$, $1 \leq i \leq \mathbf{m}$. Next, if the primary $\mathcal{P}_i$ does not propose any transaction requested by c within a reasonable amount of time, then R detects failure of $\mathcal{P}_i$. Hence, refusal of $\mathcal{P}_i$ to propose $\langle c \rangle_T$ will lead to primary failure, incentivizing malicious primaries to provide service.

Finally, we need to deal with primaries that are unwilling or incapable of proposing requests of c, e.g., when the primary crashes. To do so, c can request to be reassigned to another instance $\mathcal{I}_j$, $1 \leq j \leq \mathbf{m}$, by broadcasting a request $m := \textsc{SwitchInstance}(c, j)$ to all replicas. Reassignment is handled by the coordinating consensus protocol P for $\mathcal{I}_i$, that will reach consensus on m. Malicious clients can try to use reassignment to propose transactions in several instances at the same time. To deal with this, we assume that no instance is more than σ rounds behind any other instance (see Section 4.5). Now, consider the moment at which replica R accepts m and let $\rho(m, R)$ be the maximum round in which any request has been proposed by any instance in which R participates. The primary $\mathcal{P}_i$ will stop proposing transactions of c immediately. Any non-faulty replica R will stop accepting transactions of c by $\mathcal{I}_i$ after round $\rho(m, R) + \sigma$ and will start accepting transactions of c by $\mathcal{I}_j$ after round $\rho(m, R) + 2\sigma$. Finally, $\mathcal{P}_j$ will start proposing transactions of c in round $\rho(m, \mathcal{P}_j) + 3\sigma$.

4.5. RCC: Improving Resilience of Consensus

Traditional primary-backup consensus protocols rely heavily on the operations of their primary. Although these protocols are designed to deal with primaries that *completely fail* proposing client transactions, they are not designed to deal with many other types of malicious behavior.

EXAMPLE 4.5.1. *Consider a financial service running on a traditional* PBFT *consensus-based system. In this setting, a malicious primary can affect operations in two malicious ways:*

(1) Ordering attack. *The primary sets the order in which transactions are processed and, hence, can choose an ordering that best fits its own interests. To illustrate this, we consider client transactions of the form:*

$$\text{transfer}(A, B, n, m) := \textbf{\textit{if}}\ \ \text{amount}(A) > n\ \ \textbf{\textit{then}}\ \ \text{withdraw}(A, m);\ \ \text{deposit}(B, m).$$

Let $T_1 = \text{transfer}(\mathit{Alice}, \mathit{Bob}, 500, 200)$ *and* $T_2 = \text{transfer}(\mathit{Bob}, \mathit{Eve}, 400, 300)$. *Before processing these transaction, the balance for Alice is* 800, *for Bob* 300, *and for Eve* 100. *In Figure 4.5, we summarize the results of either first executing* T_1 *or first executing* T_2. *As is clear from the figure, execution of* T_1 *influences the outcome of execution of* T_2. *As primaries choose the ordering of transactions, a malicious primary can chose an ordering whose outcome benefits its own interests, e.g., formulate targeted attacks to affect the execution of the transaction of some clients.*

	Original	*First T_1, then T_2*		*First T_2, then T_1*	
	Balance	T_1	T_2	T_2	T_1
Alice	*800*	*600*	*600*	*800*	*600*
Bob	*300*	*500*	*200*	*300*	*500*
Eve	*100*	*100*	*400*	*100*	*100*

FIGURE 4.5. Illustration of the influence of execution order on the outcome: switching around requests affects the transfer of T_2.

(2) Throttling attack. *The primary sets the pace at which the system processes transactions. We recall that individual replicas rely on* time-outs *to detect malicious behavior of the primary. This approach will fail to detect or deal with primaries that* throttle throughput *by proposing transactions as slow as possible, while preventing failure detection due to time-outs.*

Besides malicious primaries, also other malicious entities can take advantage of a primary-backup consensus protocol:

(3) Targeted attack. *As the throughput of a primary-backup system is entirely determined by the primary, attackers can send arbitrary messages to the primary. Even if the primary recognizes that these messages are irrelevant for its operations, it has spend resources (network bandwidth, computational power, and memory) to do so, thereby reducing throughput. Notice that—in the worst case—this can even lead to failure of a non-faulty primary to propose transactions in a timely manner.*

Where traditional consensus-based systems fail to deal with these attacks, the concurrent design of RCC can be used to mitigate these attacks.

First, we look at *ordering attacks*. To mitigate this type of attack, we propose a method to deterministically select a different permutation of the order of execution in every round in such a way that this ordering is practically impossible to predict or influence by faulty replicas. Note that for any sequence S of $k = |S|$ values, there exist $k!$ distinct permutations. We write $P(S)$ to denote these permutations of S. To deterministically select one of these permutations, we construct a function that maps an integer $h \in \{0, \dots, k! - 1\}$ to a unique permutation in $P(S)$. Then we discuss how replicas will uniformly pick h. As $|P(S)| = k!$, we can construct the following bijection

$f_S : \{0, \dots, k! - 1\} \to P(S)$

$$f_S(i) = \begin{cases} S & \text{if } |S| = 1; \\ f_{S \setminus S[q]}(r) \oplus S[q] & \text{if } |S| > 1, \end{cases}$$

in which $q = i \operatorname{div} (|S| - 1)!$ is the quotient and $r = i \bmod (|S| - 1)!$ is the remainder of integer division by $(|S| - 1)!$. Using induction on the size of S, we can prove:

LEMMA 4.5.2. *f_S is a bijection from $\{0, \dots, |S|! - 1\}$ to all possible permutations of S.*

Let S be the sequence of all transactions accepted in round ρ, ordered on increasing instance. The replicas uniformly pick $h = \text{digest}(S) \bmod (k! - 1)$, in which $\text{digest}(S)$ is a *strong cryptographic hash function* that maps an arbitrary value v to a numeric digest value in a bounded range such that it is practically impossible to find another value S', $S \neq S'$, with $\text{digest}(S) = \text{digest}(S')$. When at least one primary is non-malicious ($\mathbf{m} > \mathbf{f}$), the final value h is only known after completion of round ρ and it is practically impossible to predictably influence this value. After selecting h, all replicas execute the transactions in S in the order given by $f_S(h)$.

To deal with primaries that throttle their instances, non-faulty replicas will detect failure of those instances that lag behind other instances. In specific, if an instance $\mathcal{I}_i$, $1 \leq i \leq \mathbf{m}$, is σ rounds behind any other instances (for some system-dependent constant σ), then R detects failure of $\mathcal{P}_i$.

Finally, we notice that concurrent consensus and RCC—by design—provides *load balancing* with respect to the tasks of the primary, this by spreading the total workload of the system over many primaries. As such, RCC not only improves performance when bounded by the primary bandwidth, but also when performance is bounded by computational power (e.g., due to costly cryptographic primitives), or by message delays. Furthermore, this *load balancing* reduces the load on any single primary to propose and process a given amount of transactions, dampening the effects of any targeted attacks against the resources of a single primary.

4.6. Evaluation of the Performance of RCC

In the previous sections, we proposed concurrent consensus and presented the design of RCC, our concurrent consensus paradigm. To show that concurrent consensus not only provides benefits in theory, we study the performance of RCC and the effects of concurrent consensus in a practical setting. To do so, we measure the performance of RCC in ResilientDB—our high-performance resilient blockchain fabric—and compare RCC with the well-known primary-backup consensus protocols Pbft, Zyzzyva, Sbft, and HotStuff. Specifically, we aim to answer the following:

(1) What is the performance of RCC: does RCC deliver on the promises of concurrent consensus and provide more throughput than any primary-backup consensus protocol can provide?
(2) What is the scalability of RCC: does RCC deliver on the promises of concurrent consensus and provide better scalability than primary-backup consensus protocols?
(3) Does RCC provide sufficient load balancing of primary tasks to improve performance of consensus by offsetting any high costs incurred by the primary?
(4) How does RCC fare under failures?
(5) What is the impact of batching client transactions on the performance of RCC?

First, in Section 4.6.1, we describe the experimental setup. Then, in Section 4.6.2, we provide a high-level overview of ResilientDB and of its general performance characteristics. Next, in Section 3.2.2, we provide details on the consensus protocols we use in this evaluation. Then, in Section 4.6.3, we present the experiments we performed and the measurements obtained. Finally, in Section 4.6.4, we interpret these measurements and answer the above research questions.

4.6.1. Experimental Setup. To be able to study the practical performance of RCC and other consensus protocols, we choose to study these protocols in a full resilient database system. To do so, we implemented RCC in ResilientDB. To generate a workload for the protocols, we used the *Yahoo Cloud Serving Benchmark* [29] provided by the Blockbench macro benchmarks [38]. In the generated workload, each client transaction queries a YCSB table with half a million active records and 90% of the transactions write and modify records. Prior to the experiments, each replica is initialized with an identical copy of the YCSB table. We perform all experiments in the Google Cloud. In specific, each replica is deployed on a `c2`-machine with a 16-core Intel Xeon Cascade Lake CPU running at 3.8 GHz and with 32 GB memory. We use up to 320 k clients, deployed on 16 machines.

4.6.2. The ResilientDB **Blochchain Fabric.** The ResilientDB fabric incorporates secure permissioned blockchain technologies to provide resilient data processing. A detailed description of how ResilientDB achieves high-throughput consensus in a practical settings can be found in Gupta et al. [59, 61, 62, 64, 115]. The architecture of ResilientDB is optimized for maximizing throughput via *multi-threading* and *pipelining*. To further maximize throughput and minimize the overhead of any consensus protocol, ResilientDB has built-in support for *batching* of client transactions.

We typically group 100 txn/batch. In this case, the size of a proposal is 5400 B and of a client reply (for 100 transactions) is 1748 B. The other messages exchanged between replicas during the Byzantine commit algorithm have a size of 250 B. ResilientDB supports *out-of-order processing* of transactions in which primaries can propose future transactions before current transactions are executed. This allows ResilientDB to maximize throughput of any primary-backup protocol that supports out-of-order processing (e.g., Pbft, Zyzzyva, and Sbft) by maximizing bandwidth utilization at the primary.

In ResilientDB, each replica maintains a *blockchain ledger* (a journal) that holds an ordered copy of all executed transactions. The ledger not only stores all transactions, but also proofs of their acceptance by a consensus protocols. As these proofs are built using strong cryptographic primitives, the ledger is *immutable* and, hence, can be used to provide *strong data provenance*.

In our experiments replicas not only perform consensus, but also communicate with clients and execute transactions. In this practical setting, performance is not fully determined by bandwidth usage due to consensus (as outlined in Section 4.1), but also by the cost of *communicating with clients*, of sequential *execution* of all transactions, of *cryptography*, and of other steps involved in processing messages and transactions, and by the available memory limitations.

4.6.3. The Experiments. To be able to answer Question Q1–Q5, we perform four experiments in which we measure the performance of RCC. In each experiment, we measure the *throughput* as the number of transactions that are executed per second, and we measure the *latency* as the time from when a client sends a transaction to the time where that client receives a response. We run each experiment for 180 s: the first 60 s are warm-up, and measurement results are collected

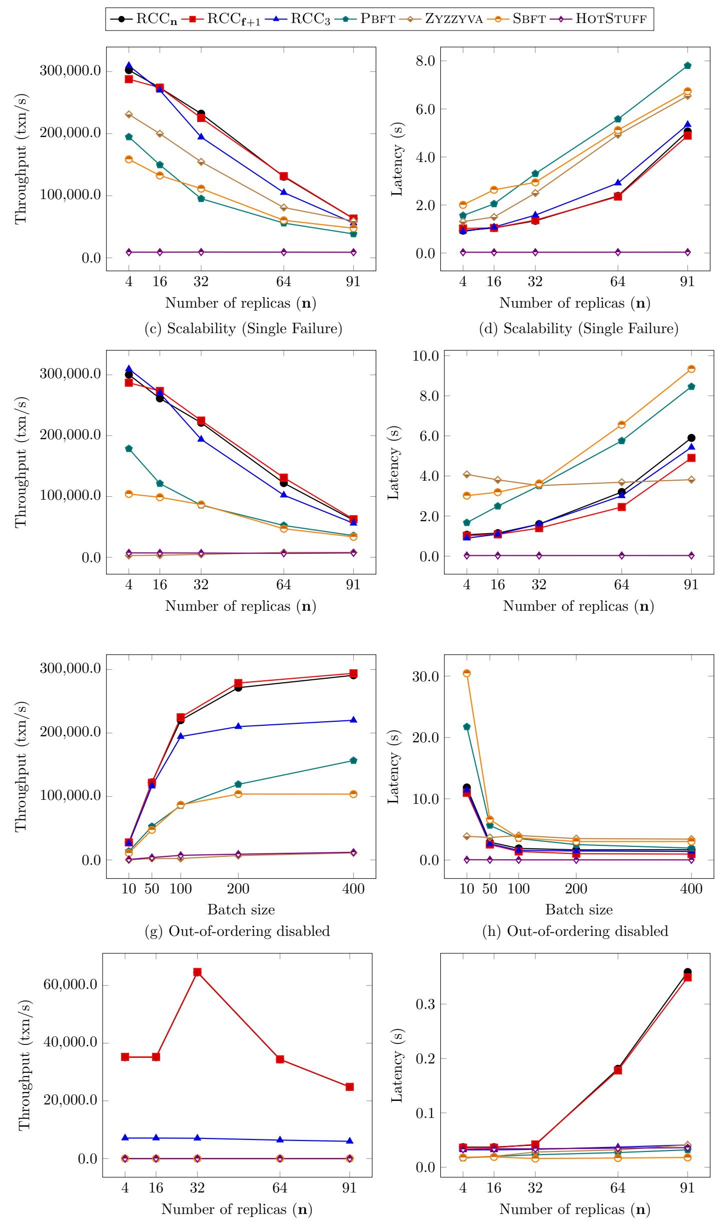


FIGURE 4.6. Evaluating system throughput and average latency incurred by RCC and other consensus protocols.

over the next 120 s. We average our results over three runs. The results of all four experiments can be found in Figure 4.6.

In the first experiment, we measure the *best-case performance* of the consensus protocols as a function of the number of replicas when all replicas are non-faulty. We vary the number of replicas between $\mathbf{n} = 4$ and $\mathbf{n} = 91$ and we use a batch size of 100 txn/batch. The results can be found in Figure 4.6, (a) and (b).

In the second experiment, we measure the performance of the consensus protocols as a function of the number of replicas during failure of a single replica. Again, we vary the number of replicas between $\mathbf{n} = 4$ and $\mathbf{n} = 91$ and we use a batch size of 100 txn/batch. The results can be found in Figure 4.6, (c) and (d).

In the third experiment, we measure the performance of the consensus protocols as a function of the number of replicas during failure of a single replica while varying the batch size between 10 txn/batch and 400 txn/batch. We use $\mathbf{n} = 32$ replicas. The results can be found in Figure 4.6, (e) and (f).

In the fourth and final experiment, we measure the performance of the consensus protocols when outgoing primary bandwidth is not the limiting factor. We do so by disabling *out-of-order processing* in all protocols that support out-of-order processing. This makes the performance of these protocols inherently bounded by the message delay and not by network bandwidth. We study this case by varying the number of replicas between $\mathbf{n} = 4$ and $\mathbf{n} = 91$ and we use a batch size of 100 txn/batch. The results can be found in Figure 4.6, (g) and (h).

4.6.4. Discussion. From the experiments, a few obvious patterns emerge. First, we see that increasing the batch size ((e) and (f)) increases performance of all consensus protocols (Q5). This is in line with what one can expect (See Section 4.1 and Section 4.3). As the gains beyond 100 txn/batch are small, we have chosen to use 100 txn/batch in all other experiments.

Second, we see that the three versions of RCC outperform all other protocols, and the performance of RCC with or without failures is comparable ((a)–(d)). Furthermore, we see that adding concurrency by adding more instances improves performance, as RCC_3 is outperformed by the other RCC versions. On small deployments with $\mathbf{n} = 4, \dots, 16$ replicas, the strength of RCC is most evident, as our RCC implementations approach the maximum rate at which ResilientDB can execute transactions (see Section 4.6.2).

Third, we see that RCC easily outperforms Zyzzyva, even in the best-case scenario of no failures ((a) and (b)). We also see that Zyzzyva is—indeed—the fastest primary-backup consensus protocol when no failures happen. This underlines the ability of RCC, and of concurrent consensus in general, to reach throughputs no primary-backup consensus protocol can reach. We also notice that Zyzzyva fails to deal with failures ((c) and (d)), in which case its performance plummets, a case that the other protocols have no issues dealing with.

Finally, due to the lack of out-of-order processing capabilities in HotStuff, HotStuff is uncompetitive to out-of-order protocols. When we disable out-of-order processing for all other protocols ((g) and (h)), the strength of the simple design of HotStuff shows: its event-based single-phase design outperforms all other primary-backup consensus protocols. Due to the concurrent design of RCC, a non-out-of-order-RCC is still able to greatly outperform HotStuff, however, as the non-out-of-order variants of RCC balance the entire workload over many primaries. Furthermore, as the throughput is not bound by any replica resources in this case (and only by network delays), the non-out-of-order variants $\text{RCC}_{\mathbf{f}+1}$ and $\text{RCC}_{\mathbf{n}}$ benefit from increasing the number of replicas, as this also increases the amount of concurrent processing (due to increasing the number of instances).

Summary. RCC implementations achieve up to 2.77×, 1.53×, 38×, and 82× higher throughput than Sbft, Pbft, HotStuff, and Zyzzyva in single failure experiments. RCC implementations achieve up to 2×, 1.83×, 33×, and 1.45× higher throughput than Sbft, Pbft, HotStuff, and Zyzzyva in no failure experiments, respectively.

Based on these observations, we conclude that RCC delivers on the promises of concurrent consensus. RCC provides more throughput than any primary-backup consensus protocol can provide (Q1). Moreover, RCC provides great scalability if throughput is only bounded by the primaries: as the non-out-of-order results show, the load-balancing capabilities of RCC can even offset inefficiencies in other parts of the consensus protocol (Q2, Q3). Finally, we conclude that RCC can efficiently deal with failures (Q4). Hence, RCC meets the design goals D1–D5 that we set out in Section 4.4.

4.6.5. Analyzing RCC as a Paradigm. Finally, we experimentally illustrate the ability of RCC to act as a paradigm. To do so, we apply RCC to not only Pbft, but also to Zyzzyva and Sbft. In Figure 4.7, we plot the performance of these three variants of RCC: RCC-P (RCC+Pbft), RCC-Z (RCC+Zyzzyva), and RCC-S (RCC+Sbft). To evaluate the scalability of these protocols, we perform experiments in the optimistic setting with no failures and $\mathbf{m} = \mathbf{n}$ concurrent instances.

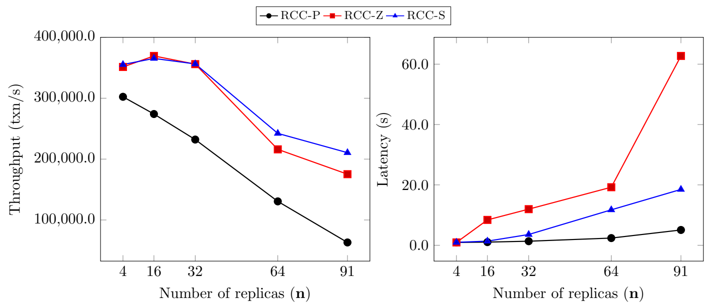


FIGURE 4.7. Evaluating system throughput and latency attained by three RCC variants: RCC-P, RCC-Z and RCC-S when there are no failures.

It is evident from these plots that all RCC variants achieve extremely high throughput. As SBFT and ZYZZYVA only require linear communication in the optimistic case, RCC-S and RCC-Z are able achieve up to 3.33× and 2.78× higher throughputs than RCC-P, respectively.

Notice that RCC-S consistently attains equal or higher throughput than RCC-Z, even though ZYZZYVA scales better than SBFT. This phenomena is caused by the way RCC-Z interacts with clients. In specific, like ZYZZYVA, RCC-Z requires its clients to wait for responses of all **n** replicas. Hence, clients have to wait longer to place new transactions, and consequently RCC-Z requires more clients than RCC-S to attain maximum performance. Even if we ran RCC-Z with 5 million clients, the largest amount at our disposal, we would not see maximum performance. Due to the low single-primary performance of ZYZZYVA, this phenomena does not prevent ZYZZYVA to already reach its maximum performance.

4.7. Concluding Remarks

In this chapter, we proposed *concurrent consensus* as a major step toward enabling high-throughput and more scalable consensus-based database systems. We have shown that concurrent consensus is in theory able to achieve throughputs that primary-backup consensus systems are unable to achieve. To put the idea of concurrent consensus in practice, we proposed the RCC paradigm that can be used to make normal primary-backup consensus protocols *concurrent*. Furthermore, we showed that RCC is capable of making consensus-based systems more resilient to failures by sharply reducing the impact of faulty replicas on the throughput and operations of the system. We have also put the design of the RCC paradigm to the test by implementing it in

ResilientDB, our high-performance resilient blockchain fabric, and comparing it with state-of-the-art primary-backup consensus protocols. Our experiments show that RCC is able to fulfill the promises of concurrent consensus, as it significantly outperforms other consensus protocols and provides better scalability. As such, we believe that RCC opens the door to the development of new high-throughput resilient database and federated transaction processing systems.

4.8. Bibliographic Notes

Parallelizing consensus. Several recent consensus designs propose to run several primaries concurrently, e.g., [10, 40, 94, 128]. None of these proposals satisfy all design goals of RCC, however. In specific, these proposals all fall short with respect to maximizing potential throughput in all cases, as none of these proposals satisfy the *wait-free* design goals D4 and D5 of RCC.

Example 4.8.1. *The* MirBFT *protocol proposes to run concurrent instances of* Pbft*, this in a similar fashion as* RCC*. The key difference is how* MirBFT *deals with failures:* MirBFT *operates in global* epochs *in which a* super-primary *decides which instances are enabled. During any failure,* MirBFT *will switch to a new epoch via a view-change protocol that temporarily shuts-down all instances and subsequently reduces throughput to zero. This is in sharp contrast to the wait-free design of* RCC*, in which failures are handled on a per-instance level. In Figure 4.8, we illustrated these differences in the failure recovery of* RCC *and* MirBFT*.*

As is clear from the figure, the fully-coordinated approach of MirBFT *results in substantial performance degradation during failure recovery. Hence,* MirBFT *does not meet design goals D4 and D5, which is sharply limits the throughput of* MirBFT *when compared to* RCC*.*

Reducing malicious behavior. Several works have observed that traditional consensus protocols only address a narrow set of malicious behavior, namely behavior that prevents any progress [5, 10, 27, 132]. Hence, several designs have been proposed to also address behavior that impedes performance without completely preventing progress. One such design is Rbft, which uses concurrent primaries *not* to improve performance—as we propose—but only to mitigate throttling attacks in a way similar to what we described in Section 4.5. In practice, the design of Rbft results in poor performance at high costs.

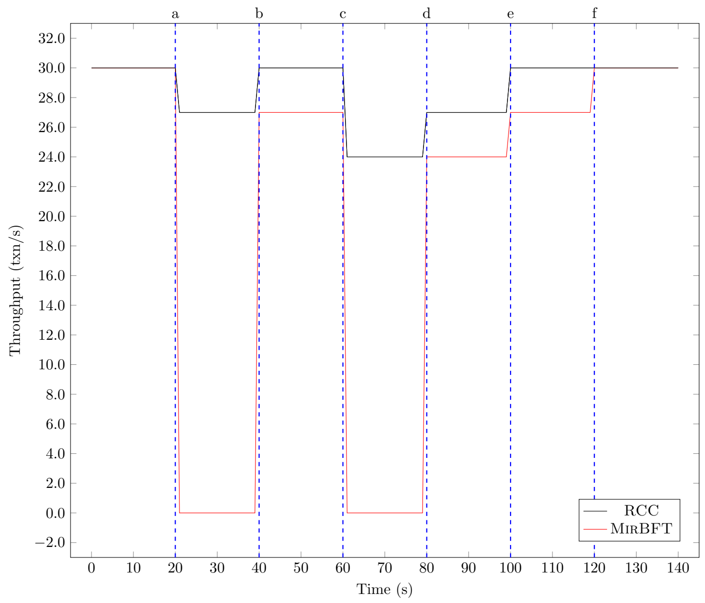


Figure 4.8. Throughput of RCC versus MirBFT during instance failures with $\mathbf{m} = 11$ instances. At (a), primary $\mathcal{P}_1$ fails. In RCC, all other instances are unaffected, whereas in MirBFT all replicas need to coordinate recovery. At (b), recovery is finished. In RCC, all instances can resume work, whereas MirBFT halts an instance due to recovery. At (c) primaries $\mathcal{P}_1$ and $\mathcal{P}_2$ fail. In RCC, $\mathcal{P}_2$ will be recovered at (d) and $\mathcal{P}_1$ at (e) (as $\mathcal{P}_1$ failed twice, its recovery in RCC takes twice as long). In MirBFT, recovery is finished at (d), after which MirBFT operates with only $\mathbf{m} = 9$ instances. At (e) and (f), MirBFT decides that the system is sufficiently reliable, and MirBFT enables the remaining instances one at a time.

HotStuff [137], Spinning [132], and Prime [5] all proposes to minimize the influence of malicious primaries by replacing the primary every round. This would not incur the costs of Rbft, while still reducing—but not eliminating—the impact of faulty replicas to severely reduce throughput. Unfortunately, these protocols follow the design of primary-backup consensus protocols and, as discussed in Section 4.3, these designs are unable to achieve throughputs close to those reached by a concurrent consensus such as RCC.

Concurrent consensus via sharding. Several recent works have proposed to speed up consensus-based systems by incorporating sharding, this either at the data level (e.g., [6, 7, 35, 41, 71, 116]) or at the consensus level (e.g., [63]). In these approaches only a small subset of all replicas, those in a single shard, participate in the consensus on any given transaction, thereby reducing the costs to replicate this transaction and enabling concurrent transaction processing in independent shards.

As such, sharded designs can promise huge scalability benefits for easily-sharded workloads. To do so, sharded designs utilize a weaker failure model than the fully-replicated model RCC uses, however. Consider, e.g., a sharded system with z shards of $\mathbf{n} = 3\mathbf{f} + 1$ replicas each. In this setting, the system can only tolerate failure of up to $\mathbf{f}$ replicas in a single shard, whereas a fully-replicated system using z replicas could tolerate the failure of any choice of $\lfloor (z\mathbf{n} - 1)/3 \rfloor$ replicas. Furthermore, sharded designs typically operate consensus protocols such as Pbft in each shard to order local transactions, which opens the opportunity of concurrent consensus and RCC to achieve even higher performance in these designs.

CHAPTER 5

ResilientDB: High Throughput Yielding Permissioned Blockchain Fabric

The *Practical Byzantine Fault-Tolerance* (PBFT) protocol has been around for more than two decades. Since its inception, several new BFT protocols, such as ZYZZYVA [87], SBFT [48], and HOTSTUFF [137], have been proposed, which aim to optimize the consensus presented by PBFT. However, PBFT remains as the first choice for system designers due to its simple and robust design. Despite this, the major use-case of blockchain technology and byzantine fault-tolerant applications remains as a crypto-currency. This leads us to a key observation: *Why have blockchain (or* BFT*) applications seen such a slow adoption?*

The low throughput and high latency are the key reasons why BFT algorithms are often ignored. In Chapter 2, we saw that the traditional distributed systems are capable of achieving throughputs of the order 100K transactions per second. However, the throughputs of current permissioned blockchain applications are still of the order 10K transactions per second [6, 8, 35]. Several prior works blame the low throughput and scalability of a permissioned blockchain system on to its underlying BFT consensus algorithm [35, 38, 87, 137]. Although these claims are not false, we believe they only represent a one-sided story.

We claim that the low throughput of a blockchain system is due to missed opportunities during its design and implementation. Hence, we want to raise a question: *can a well-crafted system-centric architecture based on a classical* BFT *protocol outperform a protocol-centric architecture?* Essentially, we wish to show that even a slow-perceived classical BFT protocol, such as PBFT [21], if implemented on skillfully-optimized blockchain fabric, can outperform a fast niche-case and optimized for fault-free consensus, BFT protocol, such as ZYZZYVA [88]. We use Figure 5.1 to illustrate such a possibility. In this figure, we measure the throughput of an optimally designed permissioned blockchain system (RESILIENTDB) and intentionally make it employ the slow PBFT protocol. Next, we compare the throughput of ResilientDB against a *protocol-centric* permissioned blockchain system that adopts practices suggested in BFTSmart [17] and employs the fast Zyzzyva protocol. We observe that the *system-centric* design of ResilientDB, even after employing the three-phase Pbft protocol (two of the three phases require quadratic communication among the replicas) outperforms the system having a single-phase linear protocol Zyzzyva. Further, ResilientDB achieves a throughput of 175K transactions per second, scales up to 32 replicas, and attains up to 79% more throughput.

This chapter aims to illustrating that the design and architecture of the underlying system are as important as optimizing bft consensus. Further, we employ decades of academic research and industry experience to design of a high-throughput yielding permissioned blockchain fabric, ResilientDB. In specific, we *dissect* existing permissioned blockchain systems, identify different performance bottlenecks, and illustrate mechanisms to eliminate these bottlenecks from the design. For example, we show that even for a blockchain system, ordering of transactions can be easily relaxed without affecting the security. Further, most of the tasks associated with transaction ordering can be extensively parallelized and pipelined. A highlight of our other observations:

- Optimal batching of transactions can help a system gain up to 66× throughput.
- Clever use of cryptographic signature schemes can increase throughput by 103×.
- Employing in-memory storage with blockchains can yield up to 18× throughput gains.
- Decoupling execution from the ordering of client transactions can increase throughput gains by 10%.
- Out-of-order processing of client transactions can help gain 60% more throughput.
- bft protocols optimized for fault-free executions can result in a loss of 39× throughput under failures.

These observations allow us to perceive ResilientDB as a reliable test-bed to implement and evaluate enterprise-grade blockchain applications. We now enlist our contributions:

- We dissect existing permissioned blockchain systems and enlist different factors that affect their performance.
- We carefully measure the impact of these factors and present ways to mitigate the effects of these factors.

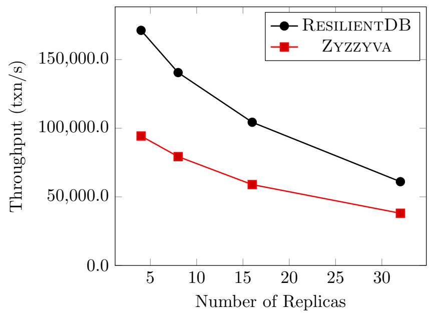


FIGURE 5.1. Two permissioned blockchains employing distinct BFT consensus protocols ($80K$ clients used for each experiment).

- We design a permissioned blockchain system, ResilientDB that yields high throughput, incurs low latency, and scales even a slow protocol like Pbft. ResilientDB includes an extensively parallelized and deeply pipelined architecture that efficiently balances the load at a replica.
- We raise *eleven* questions and rigorously evaluate our ResilientDB platform in light of these questions.

5.1. Dissecting Existing Permissioned Blockchains

As discussed earlier, most of the existing works focus on: (i) optimizing the underlying BFT consensus algorithm, and/or (ii) restructuring the way a blockchain is maintained. We believe there is much more to render in the design of a permissioned blockchain system beyond these strategies. Hence, we identify several other key *factors* that reduce the throughput and increase the latency of a permisisoned blockchain system or database.

Single-threaded Monolithic Design. There are ample opportunities available in the design of a permissioned blockchain application to extract parallelism. Several existing permissioned systems provide minimal to no discussion on how they can benefit from the underlying hardware or cores [6, 35, 138]. Due to the sustained reduction in hardware cost (as a consequence of Moore's Law [100]), it is easy for each replica to have at least *eight* cores. Hence, by parallelizing the tasks across different threads and pipelining several transactions, a blockchain application can highly benefit from the available computational power.

Successive Phases of Consensus. Several works advocate the benefits of performing consensus on one request at a time [6, 76], while others promote aggregating client requests into large batches [8, 101]. We believe there is a communication and computation trade-off that needs to be analyzed before reaching such a decision. Hence, an optimal batching limit needs to discovered.

Decoupling Ordering and Execution. On receiving a client request, each replica of a permissioned blockchain application has to order and execute that request. Although these tasks share a dependency, it is a useful design practice to separate them at the physical or logical level. At the physical level, distinct replicas can be used for execution. However, such an approach would incur additional communication costs. At the logical level, distinct threads can be asked to process requests in parallel, but additional hardware cores would be needed to facilitate such parallelism. In specific, a single entity performing both ordering and execution loses an opportunity to gain from inherent parallelism.

Strict Ordering. Permissioned blockchain applications rely on BFT protocols, which necessitate ordering of client requests in accordance with linearizability [21, 74]. Although linearizability helps in guaranteeing a safe state across all the replicas, it is an expensive property to achieve. Hence, we need an approach that can provide linearizability but is inexpensive. We observe that permissioned blockchain applications can benefit from delaying the ordering of client requests until execution. This delay ensures that although several client requests are processed in parallel, the result of their execution is in order.

Off-Memory Chain Management. Blockchain applications work on a large set of records or data. Hence, they require access to databases to store these records. There is a clear trade-off when applications store data in-memory or on an off-the-shelf database. Off-memory storage requires several CPU cycles to fetch data [72]. Hence, employing in-memory storage can ensure faster access, which in turn can lead to high system throughput.

Expensive Cryptographic Practices. Blockchain applications expect the exchange of several messages among the participating replicas and the clients, of which some may be byzantine. Hence, each blockchain application requires strong cryptographic constructs that allow a client or a replica to validate any message. These cryptographic constructs find a variety of uses in a

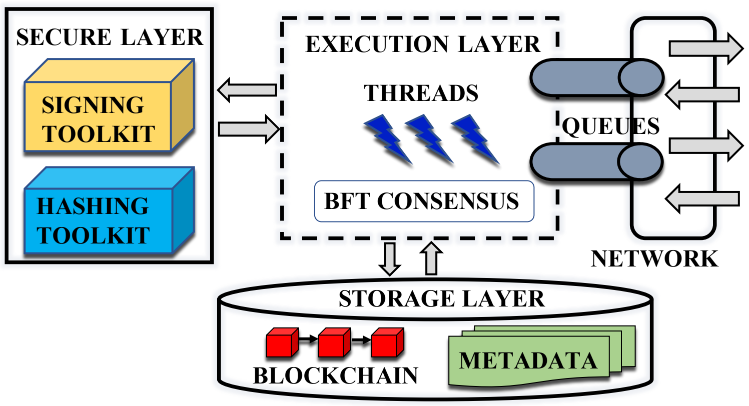


FIGURE 5.2. RESILIENTDB Architecture.

blockchain application: (i) To sign a message. (ii) To verify an incoming message. (iii) To generate the digest of a client request. (iv) To hash a record or data.

To sign and verify a message, a blockchain application can employ either symmetric-key cryptography or asymmetric-key cryptography [85]. Although symmetric-key signatures, such as Message Authentication Code (MAC), are faster to generate than asymmetric-key signatures, such as Digital Signature (DS), DSs offer the key property of non-repudiation, which is not guaranteed by MACs [85]. Hence, several works suggest using DSs [6, 8, 35, 138]. However, a cleverly designed permissioned blockchain system can skip using DSs for a majority of its communication, which in turn will help increase its throughput. For generating digests or hash, a blockchain application needs to employ standard Hash functions, such as SHA256 or SHA3, which are secure.

5.2. ResilientDB Permissioned Blockchain Fabric

We now present our RESILIENTDB fabric, which incorporates our insights and fulfills the promise of an efficient permissioned blockchain system. In Figure 5.2, we illustrate the overall architecture of RESILIENTDB, which lays down an efficient client-server architecture. At the *application layer*, we allow multiple clients to co-exist, each of which creates its own requests. For this purpose, they can either employ an existing benchmark suite or design a *Smart Contract* suiting to the active application. Next, clients and replicas use the *transport layer* to exchange messages across the network. RESILIENTDB also provides a *storage layer* where all the metadata corresponding to a request and the blockchain is stored. At each replica, there is an *execution layer* where the underlying consensus protocol is run on the client request, and the request is ordered and executed. During ordering, the *secure layer* provides cryptographic support.

Since our aim is to present the design of a high-throughput permissioned blockchain system, for the rest of the discussion we assume that ResilientDB employs the Pbft protocol for reaching consensus among the replicas. It is important to note that the succeeding insights are also relevant in the context of other bft protocols.

5.2.1. Multi-Threaded Deep Pipeline. For implementing Pbft, we require ResilientDB to follow the primary-backup model. On receiving a client request, the primary replica must initiate Pbft consensus among all the backup replicas and ensure all the replicas execute this client request in the same order. Note that depending on the choice of bft protocol, ResilientDB can be molded to adopt a different model (e.g. leaderless architecture).

In Figure 5.3, we illustrate the threaded-pipelined architecture of ResilientDB replicas. We permit increasing (or decreasing) the number of threads of each type. In fact one of the key goals of this paper is to study the effect of varying these threads on a permissioned blockchain. With each replica, we associate multiple *input* and *output* threads. In specific, we balance the tasks assigned to the input-threads, by requiring one input-thread to solely receive client requests, while two other input-threads to collect messages sent by other replicas. ResilientDB also balances the task of transmitting messages between the two output-threads by assigning equal clients and replicas to each output-thread. To facilitate this division, we need to associate a distinct *queue* with each output-thread.

5.2.2. Transaction Batching. ResilientDB allows both clients and replicas to batch their transactions. Using an optimal batching policy can help mask communication and consensus costs. A client can send a burst of transactions as a single request to the primary. Examples of applications where a client may batch multiple transactions are stock-trading, monetary-exchanges, and service level-agreements. The primary replica can also aggregate client requests together to significantly reduce the number of times a consensus protocol needs to be run among the replicas.

5.2.3. Modeling a Primary Replica. To facilitate efficient batching of requests, we require ResilientDB to associate multiple *batch-threads* with the primary replica. When the primary

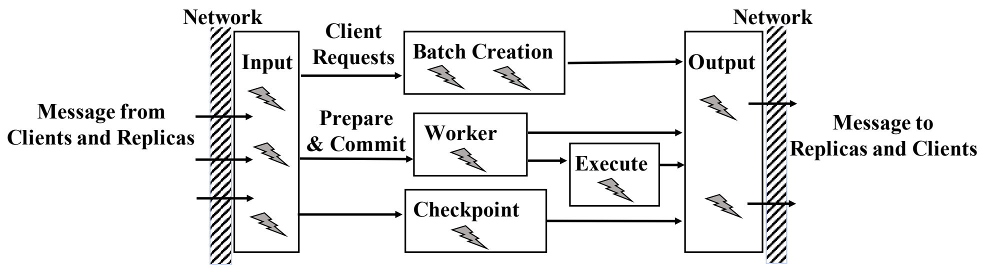


FIGURE 5.3. Schematic representation of the multi-threaded deep-pipelines at each RESILIENTDB replica. The number of threads of each type can be varied depending on the requirements of the underlying consensus protocol.

replica receives a batch of requests from the client, it treats it as a single request. The input-thread at the primary assigns a monotonically increasing sequence number to each incoming client request and enqueues it into the common queue for the batch-threads. To prevent contention among the batch-threads, we design the common queue as *lock-free*. But *why have a common queue?* This allows us to ensure that any enqueued request is consumed as soon as any batch-thread is available.

Each batch-thread also performs the task of verifying the signature of the client request. If the verification is successful, then it creates a batch and names it as the PRE-PREPARE message. PBFT also requires the primary to generate the digest of the client request and send this digest as part of the PRE-PREPARE message. This digest helps in identifying the client request in future communication. Hence, each batch-thread also hashes a batch and marks this hash as a digest. Finally, the batch-thread signs and enqueues the corresponding PRE-PREPARE message into the queue for an output-thread.

Apart from the client requests, the primary replica receives PREPARE and COMMIT messages from backup replicas. As the system is partially asynchronous, so the primary may receive both the PREPARE and COMMIT messages from a backup replica X before the PREPARE message from a backup Y. *How is this possible?* The replica X could have received sufficient number of PREPARE messages (that is $2f$) before the primary receives PREPARE from replica Y (total number of replicas are $n = 3f + 1$). In such a case, X would proceed to the next phase and broadcast its COMMIT message. Hence, to prevent any resource contention, we designate only one *worker-thread* the task of processing all these messages.

When the input-thread receives a PREPARE message, it enqueues that message in the *work-queue*. The worker-thread dequeues a message and verifies the signature on this message. If the verification is successful, then it records this message and continues collecting PREPARE messages corresponding to a PRE-PREPARE message until its count reaches $2f$. Once it reaches this count, then it creates a COMMIT message, signs and broadcasts this message. The worker-thread follows similar steps for a COMMIT message, except that it needs a total of $2f + 1$ messages, and once it reaches this count, it informs the *execute-thread* to execute the client requests.

5.2.4. Modeling a Backup Replica. As a backup replica does not create batches of client requests, RESILIENTDB assigns it fewer threads. When the input-thread at a backup replica receives a PRE-PREPARE message from the primary, then it enqueues it in the work-queue. The worker-thread at a backup dequeues a PRE-PREPARE message and checks if the message has a valid signature of the primary. If this is the case, then the worker-thread creates a PREPARE message, signs this message, and enqueues it in the queue for output-thread. Note that this PREPARE message includes the digest from the PRE-PREPARE message and the sequence number suggested by the primary. The output-thread broadcasts this PREPARE message on the network. Similar to the primary, each backup replica also collects $2f$ PREPARE messages, creates and broadcasts a COMMIT message, collects $2f + 1$ COMMIT messages, and informs the execute-thread.

5.2.5. Out-of-Order Message Processing. The key to the fast ordering of client requests is to allow ordering of multiple client requests to happen in parallel. RESILIENTDB supports parallel ordering of client requests, while ensuring a single common order across all the replicas.

EXAMPLE 5.2.1. *Say a client C sends the primary replica P first request m_1 and then request m_2. The input-thread at the primary P would assign a sequence number k to request m_1 and $k + 1$ to request m_2. However, as the batch-threads can work at varying speeds, so it is possible that the consensuses for requests m_1 and m_2 may either overlap, or some replica R may receive $2f + 1$* COMMIT *messages for m_2 before m_1.*

In principle, Example 5.2.1 seems like a challenge for a blockchain application, as a replica may receive requests at sequence number $k+1, k+2, ...$ before it commits request at number k. However, the property of out-of-order message processing is inherent in the design of most BFT protocols and is often overlooked.

Existing BFT protocols expect all the non-faulty replicas to act deterministic, that is, on identical inputs present identical outputs [21, 87, 137]. Further, they only accept a request after they have a guarantee that a majority of other replicas have also accepted the same request. For example, in the PBFT protocol, say a backup replica R receives a PRE-PREPARE message for client request m_1 with sequence number k. This replica R will not send a COMMIT message in support of the request m_1 until it receives $2f$ identical PREPARE messages from distinct replicas in support of m_1. Further, the replica R will only execute request m_1 when it receives $2f+1$ COMMIT messages from distinct replicas.

In the case of out-of-order message processing, if a replica gets $2f+1$ COMMIT messages for a request with sequence number $k+1$ before the request with number k, it will *not execute* $(k+1)$-th request before k-th request. Hence, the execution of all the succeeding requests has to be kept on hold. This ensures that the order of execution is identical across all the non-faulty replicas.

Of course, the primary $\mathcal{P}$ could act malicious and could send all but the k-th request. To tackle such a scenario, BFT protocols already provide a primary-replacement (or *view-change*) algorithm [21, 87]. The aim of the view-change algorithm is to deterministically replace the malicious primary $\mathcal{P}$ with a new primary $\mathcal{P}'$. It is the duty of this new primary $\mathcal{P}'$ to ensure all the replicas reach the common state otherwise it will also be replaced. As RESILIENTDB uses existing BFT protocols, we skip presenting the details of existing view-change algorithm.

5.2.6. Efficient Ordered Execution. Although we parallelize consensus, we ensure execution happens in order. For instance, the requests m_1 and m_2 from Example 5.2.1 are executed in sequence order, that is, m_1 is executed before m_2, irrespective of the order their consensuses completed. At each replica, we dedicate a separate *execution-thread* to execute the requests. But, the key question remains: *how can we reduce the execution-thread's overhead of ordering.*

It is evident that the execution-thread has to wait for a notification from the worker-thread. In specific, we require the worker-thread to create an EXECUTE message and place this message in the *appropriate* queue for the execution-thread. This EXECUTE message contains the identifier for the starting and ending transactions of a batch, which need to be executed. Note that we associate a large set of queues with the execution-thread. To determine the number of required queues for the execution-thread, we use the parameter QC.

$$QC = 2 \times Num\_Clients \times Num\_Req$$

Here, $Num\_Clients$ represent the total number of clients in the system, while Num_Req represents the maximum number of requests a client can send without waiting for any response. We assume both of these parameters to be finite. Although QC can be very large, the queues are logical. So, the space complexity remains almost the same as for a single queue. But why is this practice advantageous?

Using this design the execute-thread can deterministically select the queue to dequeue. If k was the sequence number for last executed request, the execute-thread calculates $r = (k+1) \mod QC$ and waits for an EXECUTE message to be enqueued in its r-th queue. This design is more efficient than having a single queue, as a single queue would have forced several dequeues and enqueues until finding the next request in order to execute. Alternatively, we could have employed *hash-maps* but collision resistant hash functions are expensive to compute and verify [85].

Once the execution is complete, the execution-thread creates a RESPONSE message and enqueues it in the queue for output-threads to send to the client. Notice that by executing client requests in order we achieve the guarantee that a single common order is established across all the non-faulty replicas.

Block Generation. A blockchain is an immutable ledger that consists of a set of *blocks*. Each block contains necessary information regarding the executed transaction and the *previous* block in its chain. The data about the previous block helps any blockchain achieve immutability. The i-th block in the chain can be represented as: $B_i := \{k, d, v, H(B_{i-1})\}$

This block B_i contains the sequence number (k) of the client request, the digest (d) of the request, the identifier of the primary v who initiated the consensus, and the hash of the previous block, $H(B_{i-1})$. In each blockchain application, every replica independently maintains its copy of the blockchain. Prior to the start of consensus, the blockchain of each replica has no element. Hence, it is initialized with a *genesis* block [101]. The genesis block is marked as the first block

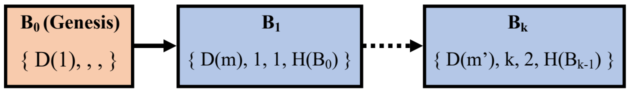


FIGURE 5.4. A formal representation of the blockchain.

in the chain and contains dummy data. For instance, a genesis block can contain the hash of the identifier of the first primary, $H(\mathbb{P})$.

Post executing the client requests, we require the execution thread to create a block representing this batch of requests. As the execute-thread has access to the previous block in the chain, so it can easily hash this previous block and store this hash in the new block. Note that this step provides another opportunity for parallelism where the execute-thread can delegate the task of creating a new block to another thread.

5.2.7. Checkpointing. We also require replicas to periodically generate and exchange *checkpoints*. These checkpoints serve *two* purposes: (1) Help a failed replica to update itself to the current state. (2) Facilitate cleaning of old requests, messages and blocks. However, as checkpointing requires exchange of large messages, so we ensure it does not impact the throughput of the system. RESILIENTDB deploys a separate *checkpoint-thread* at each replica to collect and process incoming CHECKPOINT messages. These checkpoint messages simply include all the blocks generated since the last checkpoint. In specific, a CHECKPOINT message is sent only after a replica has executed Δ requests. Once execute-thread completes executing a batch, it checks if the sequence number of the batch is a *multiple of* Δ. If such is the case, it sends a CHECKPOINT message to all the replicas. When a replica receives $2f+1$ identical CHECKPOINT messages from distinct replicas, then it marks the checkpoint and clears all the data before the previous checkpoint [21, 87].

5.2.8. Buffer Pool Management. Until now, our description revolved around how a replica uses messages and transactions. In RESILIENTDB, we designed a *base class* that represents all the messages. To create a new message type, one has to simply inherit this base class and add required properties. Although on delivery to the network, each message is simply a buffer of characters, this typed representation helps us to easily manipulate the required properties. Similarly, we have designed a *base class* to represent all client transactions. An object of this transaction class includes: transaction identifier, client identifier, and transaction data, among many other properties.

When a message arrives in the system, a replica needs to allocate (`malloc`) space for that messages. Similarly, when a replica receives a client request, it needs to allocate corresponding transaction objects. When the lifetime of a message ends (or a new checkpoint is established), then the memory occupied by that message (or transactions object) needs to be released (`free`). To avoid such frequent allocations and de-allocations, we adopt the standard practice of maintaining a set of *buffer pools*. At the system initialization stage, we create a large number of empty objects representing the messages and transactions. So instead of doing a `malloc`, these objects are extracted from their respective pools and are placed back in the pool during the `free` operation.

5.3. Experimental Analysis

We now analyze how various parameters affect the throughput and latency of a Permissioned Blockchain (henceforth abbreviated as PBC) system. For the purpose of this study we use our RESILIENTDB fabric. Although RESILIENTDB can employ any BFT consensus protocol, we use the PBFT protocol to ensure that the system design remains as our key focus. To ensure a holistic evaluation, we attempt to answer the following questions:

(Q1) Can a well-crafted system based on a classical BFT protocol outperform a modern protocol?

(Q2) How much gains in throughput can a PBC achieve from pipelining and threading?

(Q3) Can pipelining help a PBC become more scalable?

(Q4) What impact does batching of requests has on a PBC?

(Q5) Do multi-operation requests impact the throughput and latency of a PBC?

(Q6) How increasing the message size impacts a PBC?

(Q7) What effect do different types of cryptographic signature schemes have on the throughput of a PBC?

(Q8) How does a PBC fare with in-memory storage versus a storage provided by a standard database?

(Q9) Can an increased number of clients impact the latency of a PBC, while its throughput remains unaffected?

(Q10) Can a PBC sustain high throughput on a setup having fewer number of cores?

(Q11) How impactful are replica failures for a PBC?

5.3.1. Evaluation Setup. We employ Google Cloud infrastructure at Iowa region to deploy our ResilientDB. For replicas, we use `c2` machines with an 8-core Intel Xeon Cascade Lake CPU running at 3.8GHz and having 16GB memory, while for clients we use `c2` 4-core machines. We run each experiment for 180 seconds, and collect results over *three* runs to average out any noise.

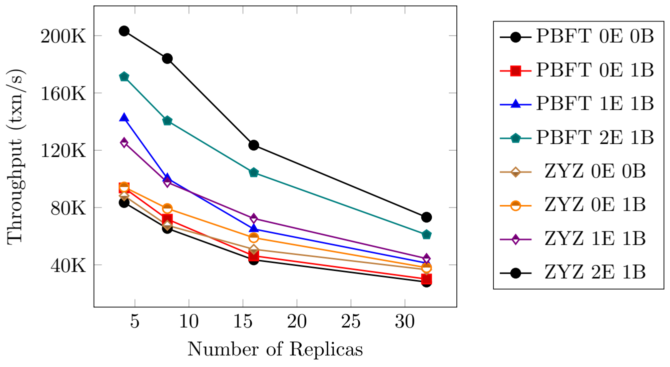
We use YCSB [29, 38] for generating workload for client requests. For creating a request, each client indexes a YCSB table with an active set of 600K records. In our evaluation, we require client requests to contain only write accesses, as a majority of blockchain requests are updates to the existing data. During the initialization phase, we ensure each replica has an identical copy of the table. Each client YCSB request is generated from a uniform Zipfian distribution.

Unless *explicitly* stated otherwise, we use the following setup: We invoke up to 80K clients on 4 machines and run consensus among 16 replicas. We employ batching to create batches of 100 requests. For communication among replicas and clients we employ digital signatures based on ED25519, and for communication among replicas we use a combination of CMAC and AES [85]. At each replica, we permit one worker-thread, one execute-thread and two batch-threads

5.3.2. Effect of Threading and Pipelining. In this section, we analyze and answer questions Q1 to Q3. For this study, we vary the system parameters in two dimensions: (i) We increase the number of replicas participating in the consensus from 4 to 32. (ii) We expand the pipeline and gradually balance the load among parallel threads.

In this experiment, we take two consensus protocols: Pbft and Zyzzyva, and we ensure that at least $3f + 1$ replicas are participating in the consensus. We gradually move our system towards the architecture of Figure 5.3. In Figure 5.5, we show the effects of this gradual increase. We denote the number of execution-threads with symbol E, and batch-threads with symbol B. For all these experiments, we used only *one* worker-thread. The key intuition behind these plots is to continue expanding the stages of pipeline and the number of threads, until system can no longer increase its throughput. In this manner, it would be easy to observe design choices that could make even Pbft outperform Zyzzyva, that is, benefits of a *well-crafted implementation.*

On close observation of Figure 5.5, we can trivially highlight the benefits of a good implementation. Further, these plots help to confirm our intuition that a multi-threaded pipelined



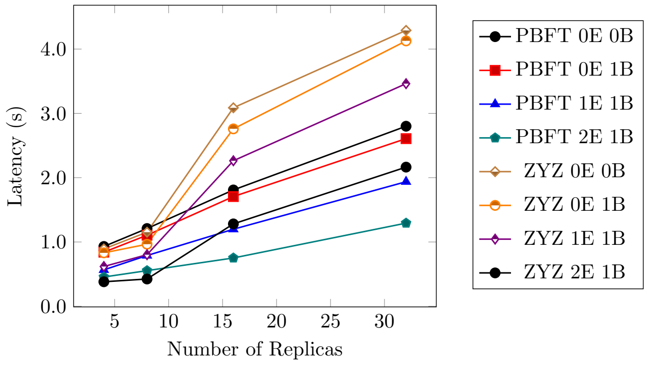


FIGURE 5.5. System throughput and latency, on varying the number of replicas participating in the consensus. Here, E denotes number of execution-threads, while B denotes batch-threads.

architecture for a PBC outperforms a single-threaded design. This is the key reason why our design of RESILIENTDB employs *one* execution-thread and *two* batch-threads apart from a single worker-thread.

Next, we explain our methodology for gradual changes. We first modified RESILIENTDB to ensure there are no additional threads for execution and batching, that is, all tasks are done by one worker-thread (0E 0B). On scaling this system we realized that this worker-thread was getting fully utilized. Hence, we partially divide the load by having an execute-thread (1E 0B). However, we again observed that the worker-thread at the primary was getting completely utilized. So we had an opportunity to introduce a separate thread to create batches (1E 1B). Although worker-thread was no longer saturating, the batch-thread was overloaded with the task of creating batches. Hence, we further divided the task of batching among multiple batch-threads (1E 2B) and ensured none of the batch-threads were fully utilized. Figures 5.6 and 5.7 show the utilization level for different threads

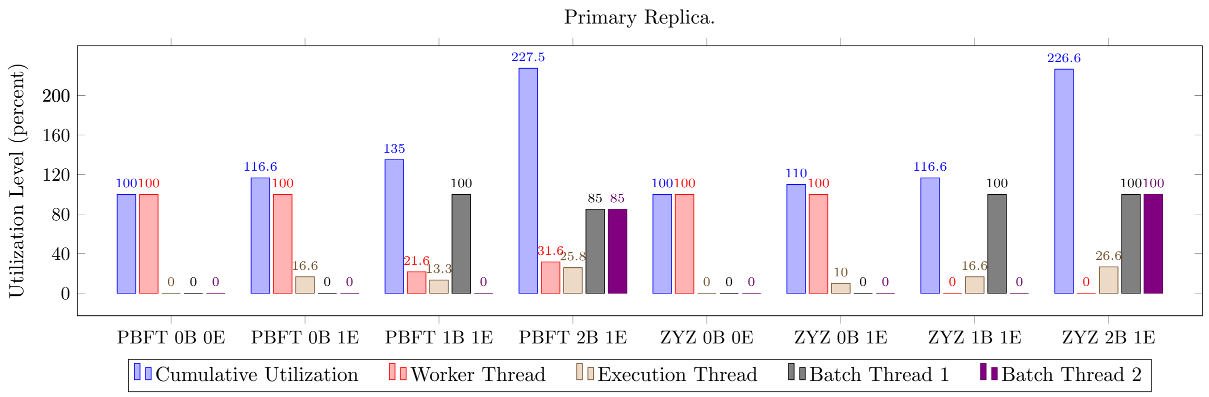


FIGURE 5.6. Utilization level of different threads at a Primary. The mean is at 100%, which implies the thread is completely utilized.

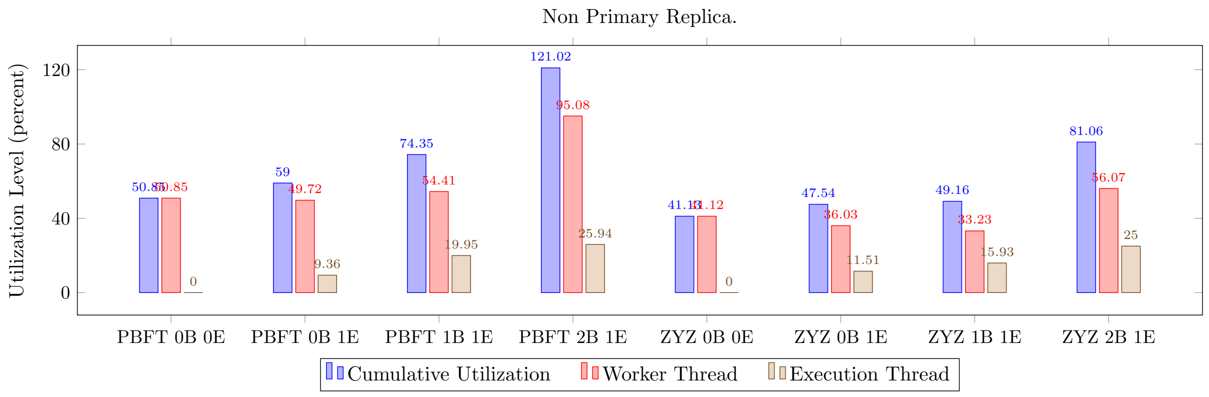
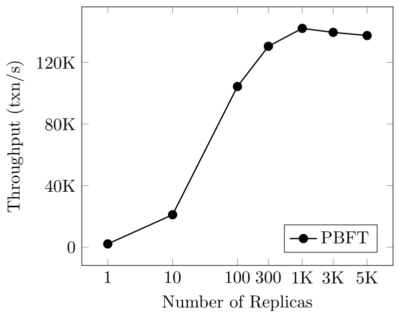


FIGURE 5.7. Utilization level of different threads at a replica. The mean is at 100%, which implies the thread is completely utilized.

at a replica. In this figure, we mark 100% as the maximum utilization for any thread. Using the bar for *cumulative utilization*, we show a summation of the utilization for all the threads, for any experiment. Note that for PBFT 1E 2B, the worker-thread at the backup replicas have started to saturate. But, as the architecture at the non-primary is following our design, so we split no further.

It can be observed that if PBFT is given benefit of RESILIENTDB's standard pipeline (1E 2B), then it can attain higher throughput than all but one ZYZZYVA implementations. The only ZYZZYVA implementation (1E 2B) that outperforms PBFT is the one that employs RESILIENTDB's standard threaded-pipeline. Further, even the simpler implementation for PBFT (1E 1B) attains higher throughput than ZYZZYVA's 0E 0B and 1E 0B implementations.



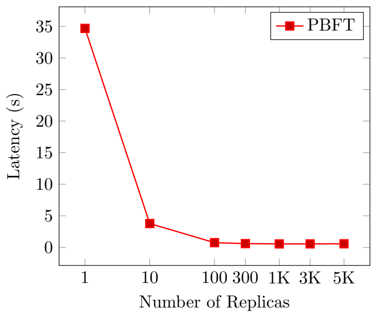


FIGURE 5.8. System throughput and latency on varying the number of transactions per batch. In this experiment, 16 replicas participate in consensus.

As stated earlier, the design of ResilientDB is independent of the underlying consensus protocol. This can be observed from the fact that when Zyzzyva is given ResilientDB's standard pipeline, then it can achieve throughput of 200K txns/s. Note that in majority of the settings Pbft incurs less latency than Zyzzyva. This is an effect of Zyzzyva's algorithm, which requires the client to wait for replies from all the n replicas, where for Pbft the client only needs $f + 1$ responses. To ***summarize***: (i) Pbft's throughput (latency) increases (reduces) by 1.39× (58.4%) on moving from 0E 0B setup to 1E 2B. (ii) Zyzzyva's throughput (latency) increases (reduces) by 1.72× (63.19%) on moving from 0E 0B setup to 1E 2B. (iii) Throughput gains up to 1.07× are possible on running Pbft on an efficient setup, in comparison to basic setups for Zyzzyva.

5.3.3. Effect of Transaction Batching. We now try to answer question Q4 by studying how batching the client transactions impacts the throughput and latency of a Pbc. For this study, we increase the size of a batch from 1 to 5000.

In Figure 5.8, we observe that as the number of transactions in a batch increases, the throughput increases until a limit (at 1000) and then starts decreasing (at 3000). At smaller batches, more consensuses are taking place, and hence communication impacts the system throughput. Hence, larger batches help reduce the consensuses. However, when the transactions in a batch are increased further, then the size of the resulting message and the time taken to create a batch by a batch-thread, reduces the system throughput. Hence, any Pbc needs to find an optimal number of client transactions that it can batch. To ***summarize:*** batching can increase throughput by up to 66× and reduce latency by up to 98.4%.

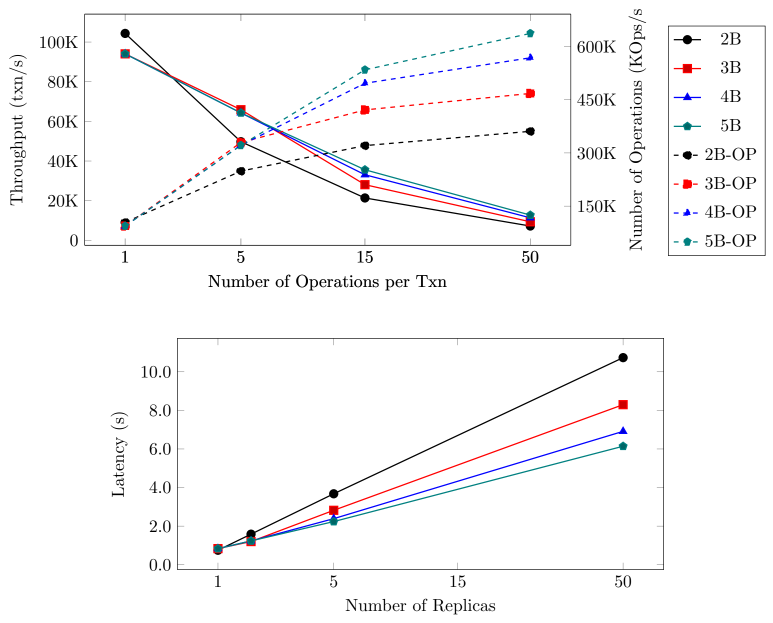
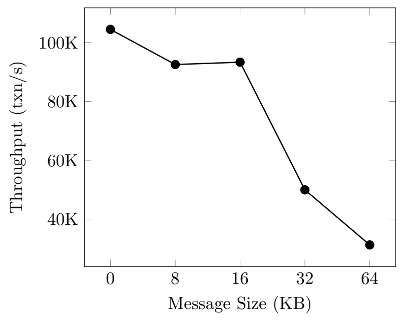


FIGURE 5.9. System throughput and latency on varying the number of operations per transaction. Here, B denotes the number of batch-threads used in the experiment. Legends with suffix OP refer to plotlines that illustrate total number of operations executed.

5.3.4. Effect of Multi-Operation Transactions. We now answer question Q5, that is, understand how multi-operation transactions affect the throughput of a system? In Figure 5.9, we increase the number of operations per transaction from 1 to 50. Further, we increase the number of batch-threads from 2 to 5, while having one worker-thread and one execute-thread. Although multi-operation transactions are common, prior works do not provide any discussion on such transactions. Notice that these experiments are orthogonal counterparts of the experiments in the previous section.

It is evident from these figures that on increasing the number of operations per transaction, the system throughput decreases. This decrease is a consequence of batch-threads getting saturated as they perform task of batching and allocating resources for transaction. Hence, we ran several experiments with distinct counts for batch-threads. An increase in the number of batch-threads helps the system to increase its throughput, but the gap reduces significantly after the transaction becomes too large (at 50 operations). Similarly, more batch-threads help to decrease the latency incurred by the system.



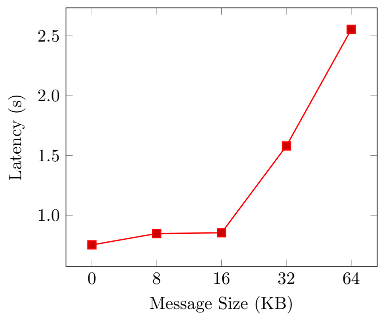


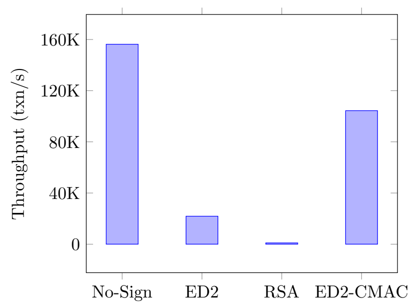
FIGURE 5.10. System throughput and latency on varying the message size. Here, 16 replicas participate in consensus.

Alternatively, we also measure the total number of operations completed in each experiment. Notice that if we base the throughput on the number of operations executed per second, then the trend has completely reversed. Indeed, this makes sense as in fewer rounds of consensus, more operations have been executed. To ***summarize:*** multi-operation transactions can cause a decrease of 93% in throughput and an increase of 13.29× in latency, on the two batch-threads setup. An increase in batch-threads from two to five increases throughputs up to 66% and reduces latencies up to 39%.

5.3.5. Effect of Message Size. We now try to answer question Q6 by increasing the size of the PRE-PREPARE message in each consensus. The key intuition behind this experiment is to gauge how well a PBC system performs when the requests sent by a client are large. Although each batch includes only 100 client transactions, individually, these requests can be large. Hence, these experiments are aimed at exploiting a different system parameter than the plots of Figure 5.8.

In Figure 5.10, we study the variation in throughput and latency by increasing the size of a PRE-PREPARE message. We do this by adding a payload to each message, which includes a set of integers (8byte each). The cardinality of this set is kept equal to the desired message size.

It is evident from these plots that as the message size increases, there is a decrease in the system throughput and an increase in the latency incurred by the client. This is a result of network bandwidth becoming a limitation, due to which it takes extra time to push more data onto the network. Hence, in this experiment, the system reaches a network bound before any thread can hit its computational bound. This leads to all the threads being idle. To ***summarize:*** On moving from 8KB to 64KB messages, there is a 52% decrease in throughput and 1.09× increase in latency.



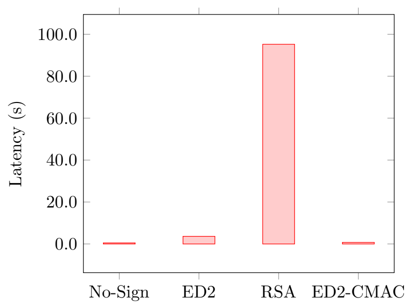


FIGURE 5.11. System throughput and latency with different signature schemes. Here, 16 replicas participate in consensus.

5.3.6. Effect of Cryptographic Signatures. In this section, we answer question Q7 by studying the impact of different cryptographic signature schemes. The key intuition behind these experiments is to determine which signing scheme helps a PBC achieve the highest throughput while preventing byzantine attacks. For this purpose, we run four different experiments to measure the system throughput and latency when: (i) no signature scheme is used, (ii) everyone uses digital signatures based on ED25519, (iii) everyone uses digital signatures based on RSA, and (iv) all replicas use CMAC+AES for signing, while clients sign their message using ED25519.

Figure 5.11 helps us to illustrate the throughput attained and latency incurred by RESILIENTDB for different configurations. It is evident that RESILIENTDB attains maximum throughput when no signatures are employed. However, such a system does not fulfill the minimal requirements of a permissioned blockchain system. Further, using just digital signatures for signing messages is not exactly the best practice. An optimal configuration can require clients to sign their messages using digital signatures, while replicas can communicate using MACs. To ***summarize:*** (i) use of cryptography reduces throughput by at least 49% and increases latency by 33%. (ii) choosing RSA over CMAC, ED25519 combination would increase latency by 125×.

5.3.7. Effect of Memory Storage. We now try to answer question Q8 by studying the trade-off of having in-memory storage versus off-memory storage in a PBC. For testing off-memory storage, we integrate SQLite [36] with our RESILIENTDB architecture. We use SQLite to store and access the transactional records. As SQLite is external to our RESILIENTDB fabric, so we developed API calls to read and write its tables. Note that until now, for all the experiments, we

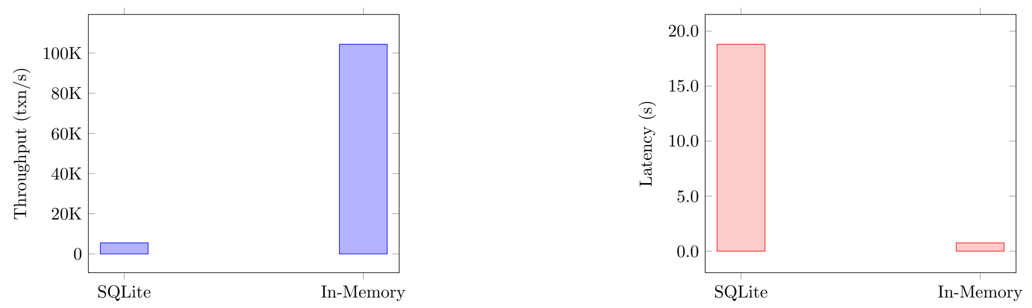


FIGURE 5.12. System throughput and latency for in-memory storage vs. off-memory storage. Here, 16 replicas used for consensus.

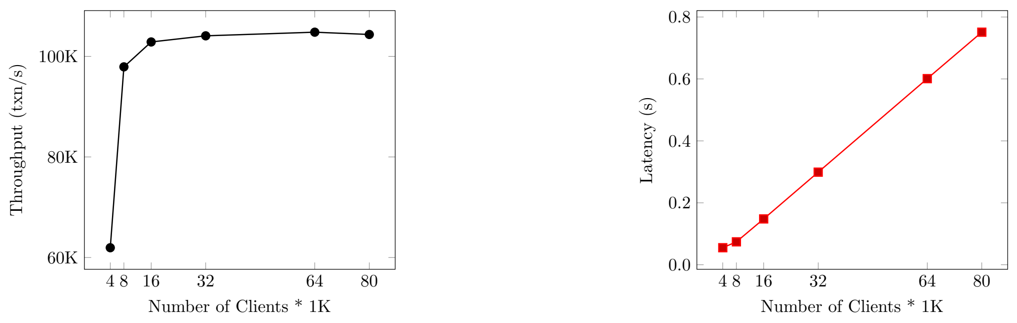


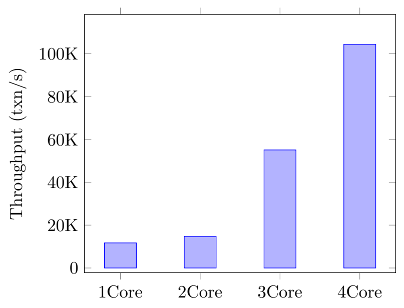
FIGURE 5.13. System throughput and latency on varying the number of clients. Here, 16 replicas participate in consensus.

assumed in-memory storage, that is, records are written and accessed in an in-memory key-value data-structure.

In Figure 5.12, we illustrate the impact on system throughput and latency in the two cases. For the in-memory storage, we require the execute-thread to read/write the key-value data-structure. For SQLite, execute-thread initiates an API call and waits for the results. It is evident from these plots that access to off-memory storage (SQLite) is quite expensive. Further, as execute-thread is busy-waiting for a reply, it performs no useful task. To ***summarize:***, choosing SQLite over in-memory storage reduces throughput by 94% and increase latency by 24×.

5.3.8. Effect of Clients. We now study the impact of clients on a PBC system, and as a result, work towards answering question Q9. We observe the changes in throughput and latency on increasing the number of clients sending requests to a PBC from 4K to 80K.

Through Figure 5.13 we conclude that on increasing the number of clients, the throughput for the system increases to some extent (up to 32K), and then it becomes constant. This is a result of



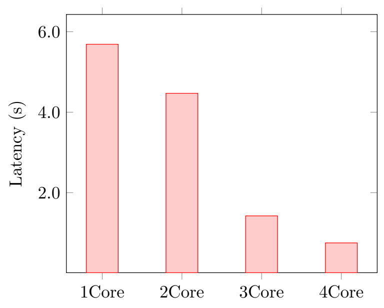
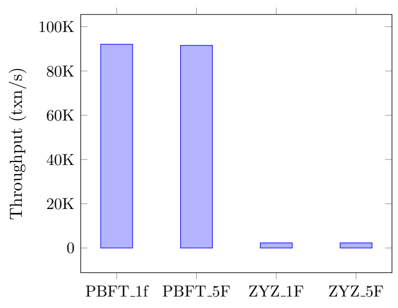


FIGURE 5.14. System throughput and latency on varying the number of hardware cores. Here, 16 replicas participate in consensus.

all the threads processing at their maximum capacities, that is, the system is unable to handle any more client requests. As the number of clients increases, an increased set of requests have to wait in the queue before they can be processed. This wait can even cause a slight dip in throughput (on moving from 64K to 80K clients). This delay in processing causes a linear increase in the latency incurred by the clients (as shown in Figure 5.13). To ***summarize:*** we observe that an increase in the number of clients from 16K to 80K helps the system to gain an additional 1.44% throughput but incurs 5× more latency.

5.3.9. Effect of Hardware Cores. We now answer question Q10 by analyzing the effects of a deployed hardware on a PBC application. In specific, we want to deploy our replicas on different Google Cloud machines having 1, 2, 4 and 8 cores. We use Figure 5.14 to illustrate the throughput and latency attained by our RESILIENTDB system on different machines. For all these experiments, we require 16 replicas to participate in the consensus. These figures affirm our claim that if replicas run on a machine with fewer cores, then the overall system throughput will be reduced (and higher latency will be incurred). As our architecture (refer to Figure 5.3) requires several threads, so on a machine with fewer cores our threads face resource contention. Hence, RESILIENTDB attains maximum throughput on the 8-core machines. To ***summarize:*** deploying RESILIENTDB replicas on an 8-core machine, in comparison to the 1-core machines, leads to an 8.92× increase in throughput.



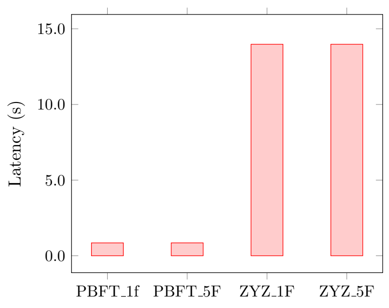


FIGURE 5.15. System throughput and latency on failing non-primary replicas. Here, 16 replicas participate in consensus.

5.3.10. Effect of Replica Failures. We now try to answer question Q11 by analyzing whether a fast BFT consensus protocol can withstand replica failures. This experiment also illustrates the impact of failures on a PBC. In specific, we perform a head-on comparison of ZYZZYVA against PBFT, while allowing some backup replicas to fail.

In Figure 5.15, we illustrate the impact of failure of *one* replica and *five* replicas on the two protocols. For this experiment we require at most 16 replicas to participate in consensus. Note that for $n = 16$, the maximum number of failures a BFT system can handle are $f = 5$. Hence, we evaluate both the protocols under minimum and maximum simultaneous failures.

On increasing the number of failures from one to five, there is a small dip in the throughput for both the protocols. This dip is not visible due to the high scaling of the graph. For PBFT, in comparison to the failure-free case, there is not a significant decrease in throughput as none of its phases require more than $2f + 1$ messages.

In case of ZYZZYVA, the system faces a pronounced reduction in its throughput with just one failure. The key issue with ZYZZYVA is that its clients need responses from all the replicas. So even one failure makes a client *wait* until it *timeouts*. This wait causes a significant reduction in its throughput. Note that finding an optimal amount of time a client should wait is a hard problem [25, 26]. Hence, we approximate this by requiring clients to wait for only a small time.

Protocols like ZYZZYVA advocate for a twin path model [48, 87]. In these protocols, each replica achieves consensus by following a fast path until the system faces a failure. Once a failure happens, these protocols decide to switch to a slower path. Such a design heavily relies on the value of timeout. If the timeout is large, then these protocols face a large reduction in throughput.

For example, in ZYZZYVA, a larger timeout implies clients have to wait for a larger amount of time before initiating the next phase. If the network is dynamic, then the value of timeout can continuously change. Thus, finding the optimal value of timeout is hard. Another way to boost the throughput of these protocols is to assume there are a sufficient number of clients that can help offset the effects of timeout.

5.4. Observation

Based on the results presented in the previous section, we make two high-level conclusions:

- A slow classical BFT protocol running on a well-crafted implementation (like RESILIENTDB), can outperform a fast BFT protocol implemented on a protocol-centric design.
- No single parameter can alone substantially improve the throughput (or reduce latency) of the underlying PBC. The key reason our RESILIENTDB framework can attain high throughputs and incurs low latency is that it attempts at optimally utilizing several parameters.

Threading and Pipelining. Earlier in this chapter, we discussed several works that either present new protocols to improve the performance of a PBC or illustrate novel use-cases for blockchain. These works rarely focus on the implementation of a replica itself and can significantly gain throughput by adopting an architecture similar to our RESILIENTDB. Further, caution needs to be taken while introducing parallelism as unnecessary threads can cause resource contention or deadlocks (e.g., multiple execution-threads can cause data-conflicts).

Batching and Multiple Operations. Several works suggest batching client requests, while others have vetoed against such a choice. Our results show that the optimal use of batching can help to reduce the cost of consensus by merging multiple consensuses into one. However, over-batching does introduce a communication trade-off. Hence, each PBC application should determine the optimal set of client requests to batch. Clients can also employ multi-operation transactions. In practice, such a transaction includes at most ten operations. Hence, employing operations per second as a metric to measure throughput may be a good idea.

Message Size and Payload. Depending on the application targeted by a PBC, the clients can send requests that have a large size. For example, a client can require the execution of a specific code. If multiple large requests are batched together, then the network may consume resources in splitting a message into packets, transmitting these packets, and aggregating these packets at the destination. Hence, depending on the application, batching just ten large requests may allow the system to return high throughput.

Cryptographic Signatures. Although the use of cryptographic signatures bottlenecks the system throughput, their use is essential for safety. We observe that a combination of MACs and DSs can help guarantee both safety and high throughput. For instance, digital signatures are only necessary for messages that need to be *forwarded*. Hence, in a PBC, only clients need to digitally sign their requests. For communication among the replicas, MACs suffice, as in most of the BFT protocols, no replica forwards messages of any other replica. Hence, the property of non-repudiation is implicitly satisfied.

Chain Storage. PBC applications need to store client records and other metadata. We observed that the use of in-memory data-structures is better than off-memory storage, such as SQLite. The key reason a PBC system can avoid frequent access to off-memory storage is that at all times, at most f replicas can fail. Hence, if persistent storage is required, then it can be performed asynchronously or delayed until periods of low contention.

Replica Failures. We know that failures are common. Either a replica may fail, or messages may get lost. A PBC system needs to be ready to face these situations. Hence, the system design must not rely on a BFT protocol that works well in non-failure cases but attains low throughput under simple failures. We observed that designs employing protocols like ZYZZYVA can have negligible throughput with just one failure. Further, some protocols suggest the use of two modes, fast path and slow path [48]. Although such protocols attain high throughputs in the fast path, they switch to the slow path on failures. Note that this switch happens when some replica or client timeouts. Determining the optimal value for timeouts is hard [25, 26]. Thus, twin path protocols may not be suitable if the network is dynamic.

5.5. Concluding Remarks

In this chapter, we present a high-throughput yielding permissioned blockchain framework, RESILIENTDB. By dissecting RESILIENTDB, we analyze several factors that affect the performance of a permissioned blockchain system. This allows us to raise a simple question: *can a well-crafted system based on a classical* BFT *protocol outperform a modern protocol?* We show that the extensively parallel and pipelined design of our ResilientDB fabric does allow even Pbft to gain high throughputs (up to 175K) and outperform common implementations of Zyzzyva. Further, we perform a rigorous evaluation of ResilientDB and illustrate the impact of different factors such as cryptography, chain management, monolithic design, and so on. We envision the practices adopted in ResilientDB to be included in designing and testing newer BFT protocols and permissioned blockchain applications.

CHAPTER 6

Future Directions: Scalable Permissioned Blockchain Applications

Until now, we aimed at designing efficient BFT consensus protocols (PoE and RCC) and scalable permissioned blockchain fabric. We now discuss ways in which our designs can be applied to practical systems. We vision our designs to help accelerate the development of efficient and scalable permissioned blockchain applications. In this chapter, we achieve this goal as follows.

- First, we try to reliably reduce the impact of byzantine replicas. To achieve this goal, we move towards the design of BFT protocols that can prevent byzantine replicas from equivocating their decisions. Several prior works have presented interesting solutions in this direction by making use of trusted components [24, 92, 133]. However, these protocols make several assumptions in their design, which prevent their application to real-world setting. We illustrate how these protocols provide limited safety and liveness guarantees, and envision solutions that can eliminate these limitations.
- Second, we envision the scaling of time-critical edge applications. These edge applications need low-latency response, and as edge devices have limited processing power, application developers have to process requests on in-house clouds. As in-house clouds are hard to scale or maintain, we envision the design of serverless-edge infrastructure. However, neither the edge nodes nor the serverless cloud can be trusted. Hence, we present the design of our ServerlessBFT protocol that facilitates the efficient processing of transactions in the serverless-edge infrastructure.
- Finally, it is common for replicated databases to have their replicas spread across a wide-area network [30]. The key issue for such geographically spread database deployments is that the communication between replicas becomes visibly expensive. In these systems, two or more replicas are often connected by networks that offer low bandwidth and high ping costs. To resolve this challenge, we envision the design of a BFT consensus protocol for geo-replicated databases.

6.1. Reliable Trusted Consensus

Until now, we have illustrated that a BFT protocol can guarantee a *safe* and *live* consensus if at most $\mathbf{f}$ of the $\mathbf{n}$ replicas participating in the consensus are byzantine [21, 87]. Specifically, $\mathbf{n} \geq 3\mathbf{f} + 1$. This implies that less than one-third replicas can act byzantine. As each replica stores the same data, BFT protocols require $\mathbf{f}$ more copies of data in comparison to their crash fault-tolerant counterparts (CFT) [89, 107]. CFT consensus protocols help $\mathbf{n} = 2\mathbf{f} + 1$ replicas reach consensus if at most $\mathbf{f}$ replicas crash, but not act byzantine. Hence, if we can rein in the byzantine acts of replica, then we can reduce the amount of replication.

Prior works have employed this insight and presented the design of BFT protocols where replicas host some *trusted components* [24, 92, 133, 135]. These trusted components prevent byzantine replicas from *equivocating* their decisions. Byzantine replicas equivocate by committing to multiple conflicting orders for client requests. Such behaviors can degrade system throughput, affect system progress, and cause safety violations [24, 28].

The common approach to prevent equivocation is to ask each replica to get each message it sends attested by a trusted component. This is under the assumption that trusted components are honest and cannot be compromised. For the sake of discussion, we refer to these BFT protocols that employ trusted components as TRUST-BFT protocols. In existing TRUST-BFT protocols, each replica hosts a trusted component that participates in the consensus protocol. These trusted components come in several flavors, such as Intel SGX [31], Sanctum [32], Keystone [91], and so on. Irrespective of the choice of trusted components, these trusted components provide each host replica access to append-only logs and/or monotonically increasing counters.

The interest in these TRUST-BFT protocols is so profound that in recent years several new consensus protocols, such as PBFT-EA [24], TRINC [92], MINBFT [133], MINZZ [133], CHEAP-BFT [83] and HOTSTUFF-M [135] have been proposed. Although all of these protocols yield interesting designs, we believe *too much trust is made on trusted components.*

In this chapter, we analyze the design of existing TRUST-BFT protocols and unearth the overlooked design limitations of these TRUST-BFT protocols. We want to argue against the prevalent belief that existing TRUST-BFT protocols yield higher throughputs than their BFT counterparts. Our study illustrates that existing TRUST-BFT protocols provide limited safety, liveness and concurrency in comparison to BFT protocols. The goal of our study is not to demerit the opportunities provide by existing TRUST-BFT protocol, but to illustrate their design limitations and provide straightforward, yet intuitive solutions. Next, we enlist our observations and solutions:

Observations: Our analysis of existing TRUST-BFT protocols has illustrated several challenges posed by their designs, which we illustrate next.

(A1) Weak Quorums limit Liveness. As TRUST-BFT protocols have access to only $\mathbf{f}+1$ non-faulty replicas, their consensuses depend on weaker quorums. Unlike BFT protocols where at least two-thirds of the replicas (at least $\mathbf{f}+1$ of these would be non-faulty) need to agree on the primary proposed order for a request, TRUST-BFT protocols just need a majority of replicas to agree. As a result, in TRUST-BFT protocols, $\mathbf{f}$ byzantine replicas need just one non-faulty replica to get any request committed.

If the primary is byzantine, and the network is unreliable, then the byzantine replicas can ensure that neither the system makes progress, nor the primary is replaced. Such an attack only occurs in existing TRUST-BFT protocols, a consequence of their support for weak quorums.

(A2) Dependence on Byzantine Host. TRUST-BFT protocols assume that none of the trusted components are compromised, this despite some of the host replicas being byzantine. However, trusted components can crash fail, lose power, or restarted by the host. Under such conditions, a trusted component may have its logs wiped-out or counters reset.

Prior works present no solutions on how to recover such a crashed trusted component without violating safety. Further, our observations reveal that in the worst case, a recovering trusted component may need to wait for states from all the replicas.

(A3) Sequential Transaction Processing. Existing TRUST-BFT protocols vision higher throughputs than their BFT counterparts. However, they offer designs which are sequential and can only process one client request at a time. This severely impacts their performance as BFT protocols have been shown to employ pipelining, multi-threading, and out-of-order message processing. As a result, despite having $\mathbf{f}$ less replication, existing TRUST-BFT protocols yield lower throughputs than their BFT counterparts.

Solutions: Following our observations that illustrate implicit challenges in the architecture of existing TRUST-BFT protocols, we set out to resolve these challenges. Next, we enlist our solutions.

(S1) Crash Recovery Protocol. To allow a crashed trusted component recover post failure, we present a succinct recovery protocol. Our recovery protocol ensures that the crashed trusted component successfully recovers its state once it receives states from $\mathbf{f} + 1$ other replicas.

(S2) Concurrent Trusted Consensus. To facilitate concurrent request processing by existing TRUST-BFT protocols, we present simple modifications in the way a host replica accesses its trusted components. Further, we state simple steps which can be employed to convert an existing TRUST-BFT protocol into its concurrent variant (we refer to these variants as CTRUST-BFT).

(S3) Lightweight Live Consensus. Existing TRUST-BFT protocols require each replica to host a trusted component. Further, prior to sending any message, the host replica needs to get it attested by its trusted component. This process is neither efficient, nor it prevents liveness anomalies (Observation A1). Hence, we present straightforward extensions of existing BFT protocols, which we refer to as LFT-BFT protocols to resolve these challenges. Our LFT-BFT protocols require only the primary replica to access trusted components and offer efficient consensus than both BFT and TRUST-BFT protocols.

6.1.1. Trusted Byzantine Fault Tolerance.

Although PBFT facilitates a BFT consensus among the replicas of a system, it is clearly expensive. PBFT requires multiple phases of quadratic communication complexity among $3\mathbf{f} + 1$ replicas. Several recent works try to reduce the costs associated with PBFT by reducing the number of phases or linearising the communication [48, 87, 137]. However, all these optimized BFT protocols still face two challenges:

(C1) **Excessive Replicas.** These BFT consensus protocols require at least $\mathbf{f}$ more replicas than their crash fault-tolerant counterparts [89, 107].

(C2) **Equivocation.** Replicas have the power to lie, that is, they can communicate different messages to different replicas.

Chun et al. [24] introduced the concept of trusted components in BFT consensus to resolve challenges C1 and C2.

Expanded Notations. We assume our replicated service $\mathfrak{S}$ to include a set $\mathfrak{I}$ of trusted components apart from sets $\mathfrak{R}$ and $\mathfrak{C}$.

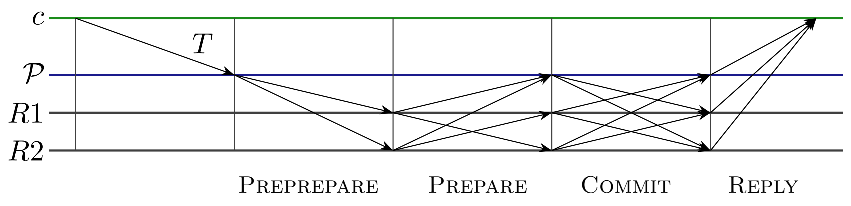


Figure 6.1. Schematic representation of the Pbft-EA consensus protocol.

Definition 6.1.1. *A trusted component* $\text{T} \in \mathfrak{I}$ *is a cryptographically secure entity, which has a negligible probability of being compromised by byzantine adversaries.* T *provides access to a shielded execution environment. Any replica* R *can communicate with* T *only through a series of predefined functions. Updates to* T*'s state are additive in nature and cannot be rollbacked.*

As our service $\mathfrak{S}$ is distributed, it is assumed that each $R \in \mathfrak{R}$ has access to an independent co-located trusted component T_R. This allows T_R to guarantee *integrity* of any function computation. Further, T_R can be requested to *attest* its computation using cryptographic signature schemes. To achieve all of these properties, our service $\mathfrak{S}$ makes the following two assumptions:

(1) Each T_R is honest or non-faulty.

(2) if an adversary compromises a replica R, it cannot compromise its T_R.

The assumptions made by our service are not new and our so prevalent in literature that in the past two decades several novel BFT protocols have been proposed that employ trusted components to achieve *arguably* efficient consensuses. In the rest of this chapter, we refer to these protocols as TRUST-BFT protocols.

Optimal Trusted Computations: It is clear by now that these trusted components perform some computation on behalf of the replicas, which allow TRUST-BFT protocols to reduce replication and provide non-equivocation. Prior works have shown that trusted components either provide access to a trusted log [24] or trusted counter [92]. A2M [24] requires each trusted component to provide access to several *append-only* trusted logs. Each replica could access a specific trusted log through simple functions, such as append, advance, truncate, lookup, and end. TrInc [92] proposed equipping each trusted component with monotonically increasing counters, and with slight modifications, aforementioned functions can be used to access these counters. Following the use of trusted counters by TrInc, almost all the subsequent TRUST-BFT protocols have adhered to this design [28, 83, 133]. As a result, in the rest of this chapter, we primarily assume the use of trusted counters and will explain the necessary differences in the case of trusted logs.

Description of Functionality: Irrespective of whether a trusted component provides access to logs or counters, the underlying TRUST-BFT protocol needs to provide APIs or functions that can be used by replicas to access the corresponding trusted component. Further, each trusted component provides access to *multiple* trusted-counters or trusted-logs. Generally, these counters or logs can be accessed through the following two functions:

(1) $\text{Append}(id, k, x)$ – Assume the id-th counter at some T_R has value ct.

- If $k = \perp$ (sequence number is unspecified), T_R binds x to $ct + 1$.
- If $k > ct$, T_R updates $ct = k$ and binds x to ct.

In the case of a trusted log, x is appended to the position ct in the log. Finally, Append function returns the following attestation $\langle \text{Attest}(id, ct, x)\rangle_{\mathrm{T}_R}$ as a proof of this binding.

(2) $\text{Lookup}(id, ct)$ – In the case of trusted logs, returns an attestation $\langle \text{Attest}(id, ct, x)\rangle_{\mathrm{T}_R}$ for the value bound at position ct in log id.

Fault-Tolerance Requirements: Existing TRUST-BFT protocols expect that in a system of $\mathbf{n} = 2\mathbf{f} + 1$ replicas at most $\mathbf{f}$ replicas are byzantine.

This implies that these TRUST-BFT protocols are able to provide higher fault-tolerance with a smaller set of replicas (reduced replication). As these protocols employ trusted components, their replicas are unable to equivocate, which in turn restricts the possible range of byzantine attacks [28].

To illustrate the general working of a trusted protocol, we analyze the design of the PBFT-EA protocol, which is inspired from PBFT but employs trusted components and requires only $2\mathbf{f} + 1$ replicas [24]. We use Figure 6.1 to illustrate the steps. Both PBFT and PBFT-EA require same number of phases to achieve consensus on a client transaction. However, there are some noteworthy differences, which we state next:

(O1) **Less Communication.** Owing to a smaller set of replicas participating in each phase of the consensus, PBFT-EA communicates less messages than PBFT.

(O2) **Smaller Quorums.** TRUST-BFT protocols permit replicas to wait on smaller quorums. For instance, in the PBFT-EA protocol, each replica R only waits for $\mathbf{nf} = \mathbf{f} + 1$ PREPARE messages before it can broadcast COMMIT messages. Similarly, when R receives $\mathbf{nf} = \mathbf{f}+1$ COMMIT messages, it marks the client transaction as committed and executes the same.

(O3) **Sequential Trusted Computations.** Existing TRUST-BFT protocols like PBFT-EA require each message sent by a replica to be ordered and signed by the local trusted component. To achieve this task, the trusted component T_R at a replica R maintains a distinct set of counters for each *type of messages* communicated during consensus. In the case of PBFT-EA protocol, T_R stores *five* distinct counters, corresponding to PREPREPARE, PREPARE, COMMIT, CHECKPOINT, and VIEWCHANGE messages.

The primary $\mathcal{P}$ calls the $\text{APPEND}(id, m, k)$ on a client request m to assign it a sequence number k. If k is valid, then $\text{T}_\mathcal{P}$ increments the id-th *preprepare counter* accordingly and returns a attestation $\langle \text{ATTEST}(id, k, m) \rangle_{\text{T}_\mathcal{P}}$, which primary forwards to all replicas along with the PREPREPARE message.

Each replica R on receiving the $m' :=$ PREPARE message can check if the accompanying attestation is valid. If it is the case, then R calls the $\text{APPEND}(id, m', k)$ to assign it a sequence number k. For a valid k, T_R updates its id-th *prepare* counter and returns the attestation $\langle \text{ATTEST}(id, k, m') \rangle_{\text{T}_R}$. On receiving $\mathbf{f}+1$ identical PREPARE messages with attestations, R sends out $m'' :=$ COMMIT messages with attestation $\langle \text{ATTEST}(id, k, m'') \rangle_{\text{T}_R}$ where this time k is the value of id-th *commit* counter. Similarly, other counters are accessed as required by the underlying consensus protocol. For trusted logs, the messages are appended at the k-th position in their respective logs (with identifier id).

This design of PBFT-EA protocol is inherited by every other TRUST-BFT protocol and sharply contrasts the design followed by existing BFT protocols. In existing TRUST-BFT protocols, every replica has to return an attestation for each message it sends to other replicas. This attestation serves as a *proof* that the message has been assigned a monotonically increasing counter value or log position. This forces these protocols to process each message one-by-one [24, 28, 92, 133, 135]

(O4) **Checkpoints.** Like BFT protocols, TRUST-BFT protocols also share checkpoints, periodically. These checkpoints reflect the state of a replica R's trusted component T_R and help to *truncate* logs. If T_R employs trusted logs, then it attests a snapshot of its logs and attaches it to the outgoing CHECKPOINT messages.

In the case $\textsc{t}_R$ employs trusted counters, a snapshot of its counters provides very little state information. Hence, protocols employing trusted counters also include a *minuscule log* that can hold a constant number of recent entries [92]. As a result, these protocols also a snapshot of this minuscule log.

Unlike existing BFT protocols, in a TRUST-BFT protocol, each replica R marks its checkpoint as stable if it receives CHECKPOINT messages from $\mathbf{f}+1$ other replicas (can include its own message).

(O5) **Message Retransmission.** Unlike BFT protocols that allow messages to be lost, TRUST-BFT protocols expect that each message is eventually delivered. To achieve this goal, they expect existence of some mechanism or technology that supports message retransmissions [24, 28, 133, 135].

6.1.2. Guarantees and Challenges.

Past two decades have given rise to several TRUST-BFT protocols, such as PBFT-EA [24], MINBFT [133], MINZYZZYVA [133], CHEAPBFT [83], and HOTSTUFF-M that stick by observations O1 to O5. We now look at the safety and liveness guarantees offered by these protocols.

- ***Safety.*** TRUST-BFT protocols provide same safety guarantees as their BFT counterparts under the assumption that no trusted component is faulty.
- ***Liveness.*** TRUST-BFT protocols provide same liveness guarantees as their BFT counterparts under the assumption that all messages are eventually delivered.

Akin to existing BFT protocols, the primary replica in TRUST-BFT protocols can also act byzantine. Although the primary cannot equivocate, it can avoid sending messages to some replicas. Further, the primary $\mathcal{P}$ can decide to ignore one or more client requests. If such is the case, then TRUST-BFT protocols also provide access to a ***view-change*** protocol [24, 83, 133]. If at least $\mathbf{nf} = \mathbf{f}+1$ replicas trigger the view-change protocol (by broadcasting VIEWCHANGE message), then $\mathcal{P}$ is replaced and another replica $R \in \mathfrak{R} \setminus \mathcal{P}$ is designated as the new primary. Notice that the quorums to replace the primary in TRUST-BFT protocols are also smaller than their BFT counterparts.

Despite their exciting designs, these TRUST-BFT protocols possess several *design limitations.* In the rest of this chapter, we attempt to illustrate these overlooked design limitations. First, we illustrate how these TRUST-BFT protocols offer *restricted liveness* in comparison to their BFT counterparts. Next, we illustrate how these TRUST-BFT protocols are *unsafe* if trusted components at byzantine replicas crash. Finally, we argue that these TRUST-BFT protocols lack opportunities to run consensuses on multiple requests in parallel.

6.1.3. Lack of Liveness under Message Loss.

Existing TRUST-BFT protocols claim to provide same liveness guarantees as their BFT counterparts. However, these TRUST-BFT protocols expect eventual message delivery through message re-transmissions [24, 83, 133]. This is a *stronger* requirement than what is expected by traditional BFT protocols. Prior to arguing the practicality of message re-transmission assumption, we first prove our claim.

CLAIM 1. *In a replicated service $\mathfrak{S}$ of replicas $\mathfrak{R}$, clients $\mathfrak{C}$, and trusted components $\mathfrak{I}$, where* $|\mathfrak{R}| = \mathbf{n} = 2\mathbf{f} + 1$*, loss of messages to at least one replica can affect system liveness.*

PROOF. Assume a run of the PBFT-EA protocol. We know that $\mathcal{F} \subset \mathfrak{R}$ replicas are faulty and $|\mathcal{F}| = \mathbf{f}$. Let us distribute the $\mathbf{nf} = |\mathfrak{R} \setminus \mathcal{F}|$ non-faulty replicas into sets D and G, such that $1 \leq |D| \leq \mathbf{f}$ and $|G| = \mathbf{nf} - |D| \geq 1$. Assume that the primary $\mathcal{P}$ is byzantine ($\mathcal{P}_{\in}\mathcal{F}$) and all the replicas in $\mathcal{F}$ want to prevent replicas in set D from participating in consensus for a client transaction T. As a result, the replicas in $\mathcal{F}$ do not send any messages to replicas in D. Further, assume that the PREPARE and COMMIT messages from the replicas in G to those in D are lost.

Say, the replicas in $\mathcal{F}$ work together to ensure that the replicas in G successfully commit the transaction T, but choose not to reply to the client. As a result, the client receives less than $\mathbf{f} + 1$ responses and eventually, complains to all the replicas. Replicas in G will act on the complain by replying back to the client as they have already executed T, while replicas in D will trigger the view-change protocol. We know that for a view-change to take place at least $\mathbf{f} + 1$ replicas should agree. Hence, neither the view-change will take place, nor the client will receive sufficient responses for transaction T. As a result client can no longer make progress, which violates the termination guarantee as defined in Section 2.1. □

Our Claim 1 illustrates that existing TRUST-BFT protocols cannot make progress even when one non-faulty replica suffers message loss. For these protocols to continuously make progress, despite message loss, there needs to be a mechanism that can guarantee timely message re-transmission.

6.1.4. Weak Quorums.

Although Claim 1 proves that existing TRUST-BFT protocols are not live, it is unclear how the traditional BFT protocols thwart such an attack. To explain this, we wish to highlight the existence of *weak quorums* in TRUST-BFT protocols.

As described earlier, TRUST-BFT protocols have access to smaller quorums in comparison to traditional BFT protocols. For example, in the PBFT-EA protocol, $\mathbf{f}$ faulty replicas can get any transaction committed if they receive support of one non-faulty replica. This implies that these TRUST-BFT protocols are limited in design by their weaker quorums–a little over 50% support is needed to commit a transaction.

In comparison, BFT protocols such as PBFT require at least $2\mathbf{f}+1$ COMMIT messages to mark a request committed (around 67% support). Notice that in BFT protocols, there are $\mathbf{n} \geq 3\mathbf{f}+1$ replicas. The larger set of replicas in a BFT protocol ensures that each committed transaction receives support of at least a majority of non-faulty replicas ($\mathbf{f}+1$). As a result, the replicas in $\mathcal{F}$ can prevent at most $1 \leq |D| \leq \mathbf{f}$ replicas from participating in the consensus. Hence, the faulty replicas are never in majority while determining the fate of any transaction.

This yields a stronger guarantee as either the client will receive $\mathbf{f}+1$ matching responses, or at least $\mathbf{f}+1$ non-faulty replicas will be eager to replace the current primary. As a result, the system continues making progress.

6.1.5. Message Retransmissions.

A key reason for existing TRUST-BFT protocols to ignore the liveness attack stated in Claim 1 is that they assume support for message re-transmissions. If every non-faulty replica in a TRUST-BFT protocol is guaranteed to participate in each consensus, then the system will continue making progress. We wish to claim that this is a strong assumption, which is hard for any system to meet.

Prior to proving that message re-transmissions are hard to achieve, we need to design a simple algorithm that dictates message re-transmissions under failures. Note: although all the existing TRUST-BFT protocols require message retransmissions, they *do not* present any algorithm or implementation detail regarding the same. We use Figure 6.2 to show how a replica $R1$ can try transmission of message m to another replica $R2$.

```
Initialization:
// $R1$ sends a message $m$ to $R2$.
// $\Delta :=$ digest($m$), digest of message $m$
retransmit (used by $R1$ to resend message to $R2$) :
1: event $R1$ timeouts waiting for $\langle \text{Ack}(\Delta) \rangle_{R2}$ message from $R2$ do
2:     Resends the message $m$ to $R2$.
3: event $R1$ receives an $\langle \text{Ack}(\Delta) \rangle_{R2}$ message from $R2$ do
4:     Marks the message $m$ as sent.

ack-transmit (used by $R2$ to resend Ack message to $R1$) :
5: event $R2$ receives message $m$ from $R1$ do
6:     Sends $\langle \text{Ack}(\Delta) \rangle_{R2}$ to $R1$.
```

Figure 6.2. A simple message retransmission protocol.

Although message retransmission protocol stated in Figure 6.2 requires just *six* steps, it does not guarantee termination. For each message m transmitted by $R1$ to $R2$, there needs to be a subsequent acknowledgement message $\langle \text{Ack}(\Delta) \rangle_{R2}$ from $R2$ to $R1$. This $\langle \text{Ack}(\Delta) \rangle_{R2}$ includes a digest $\Delta :=$ digest(m) of m and is signed by $R2$. The reception of an $\langle \text{Ack}(\Delta) \rangle_{R2}$ message by $R1$ proves to $R1$ that $R2$ did receive m. Further, as m can get lost, $R1$ may never receive an Ack from $R2$. As a result, $R1$ needs to set a timer for each message m it transmits to $R2$. Once, the timer expires, $R1$ retransmits the message m to $R2$.

Moreover, any message retransmission protocol should detect non-responsiveness of the byzantine replicas. For example, if $R2$ is indeed byzantine, then it may never send an Ack message to $R1$. Under such a case, the protocol of Figure 6.2 would never terminate. Next, we prove that impossibility of message retransmissions.

Claim 2. *Given a sender $R1$ and a receiver $R2$, such that $R1$ wishes to transmit a message m to $R2$, it is impossible for $R1$ to conclude that $R2$ received m if the network is unreliable or $R2$ is byzantine.*

Proof. Let us assume that it is always possible for the sender $R1$ to conclude if the receiver $R2$ received m or not. Hence, we need to consider two cases: (i) when the network is unreliable, and (ii) when $R2$ is byzantine.

Case 1. Say $R1$ can conclude that $R2$ received m despite existence of an unreliable network. This implies that $R1$ will be able to terminate the protocol in Figure 6.2. If this is the case, then $R1$ will always receive an Ack for m. As the network is unreliable, it is safe of assume that message

m always gets lost. This implies that $R2$ may never receive m and as a result, never sends an ACK. As a consequence, $R1$ will periodically timeout and resend m. This is a *contradiction* to our assumption that $R1$ is able to terminate the protocol in Figure 6.2.

Case 2. Say $R1$ can conclude that $R2$ received m even if $R2$ is byzantine. This implies that $R1$ will be able to terminate the protocol in Figure 6.2. If this is the case, then $R1$ will somehow receive an ACK for m. As $R2$ is byzantine, despite receiving m, it may never send an ACK to $R1$. As a consequence, $R1$ will periodically timeout and resend m. Notice that $R1$ cannot mark $R2$ as byzantine as it is impossible for it to distinguish whether $R2$ is byzantine or the network is unreliable. This *contradicts* our initial assumption. □

6.1.6. Lack of Safety under Trusted Component Failure.

With the advancement of technology, it is safe to assume that compromising these trusted components is hard. However, any trusted component can simply crash or lose access to its log. As a consequence, the trusted component will lose all its data or counter values. In this section, we claim that a simple crash failure of a trusted component can make the system unsafe.

Crash Failures vs. Message Loss. We visualize *crash failures* different from *message loss* or *indefinite delays.* A replica suffering message loss is similar to one that has been *partitioned* from rest of the network (unable to communicate with other replicas). A key cause for partitioned replicas or message loss is an unreliable network. Despite partitioning, a replica is aware of its state and will eventually timeout and blame it on the primary by requesting a *view-change*. Whether the view-change is successful or not depends on how quickly is the partitioning is resolved and how many other replicas are interested to replace the primary. Under a crash failure, a replica may lose its data. Once a replica is up again after a crash, it tries to recover its state by communicating with other replicas. During a crash, a trusted component at a byzantine replica may loose value of its logs or counters.

To prove our claim, we need to crash the trusted component of only one byzantine replicas. Notice that we are not expecting any more failures than $\mathbf{f}$ as we only require the trusted component at a byzantine replica to fail. Further, imagining such an attack is not hard as although a byzantine replica cannot compromise its trusted components, it can always restart its hardware, destroy logs, and perform other external attacks. Prior to arguing the nature of our presented attack, we first prove our claim.

CLAIM 3. *In a service $\mathfrak{S}$, crash-failure of the trusted component $\text{T}_R \in \mathfrak{I}$ at a byzantine replica $R \in \mathcal{F}$ violates safety.*

PROOF. Assume a run of the PBFT-EA protocol. We know that $\mathcal{F} \subset \mathfrak{R}$ replicas are faulty and $|\mathcal{F}| = \mathbf{f}$. Let us distribute the $\mathbf{nf} = |\mathfrak{R} \setminus \mathcal{F}|$ non-faulty replicas into sets D and G, such that $|D| = \mathbf{f}$ and $|G| = \mathbf{nf} - \mathbf{f} = 1$. Assume that the primary $\mathcal{P}$ is byzantine ($\mathcal{P}_{\in}\mathcal{F}$) and all the replicas in $\mathcal{F}$ want to prevent replicas in set D from participating in consensus for a client transaction T. Further, assume that the PREPARE and COMMIT messages from the replica in G to those in D are indefinitely delayed. However, replicas in $\mathcal{F}$ and G are successful in committing T and the client receives $\mathbf{f} + 1$ identical responses and marks the transaction complete.

Now, assume that the trusted component $\text{T}_R \in \mathfrak{I}$ for a replica $R \in \mathcal{F}$ crashes, loses the value of its counters and restarts. On recovery, T_R needs to wait for states from $\mathbf{f}$ other trusted components, total $\mathbf{f} + 1$ by including its own state. Assume T_R only receives states from trusted components in D. As all of these states have no knowledge of transaction T, so T_R concludes no transaction has been ordered. This results in a safety violation as we now have a majority of replicas in $\mathfrak{R}$ that does not marks T as committed even though the client assumes T as completed. □

6.1.7. Dependence on Byzantine Host.

To understand the safety violation presented in Claim 3, we need to analyze the steps to recovery. First, failures of trusted component is an unexplored topic in the TRUST-BFT literatures. Existing TRUST-BFT claim to guarantee consensus safety with small quorums of $\mathbf{f}+1$ replicas until no trusted component is byzantine. In Claim 3, we fail a trusted component.

The key reason these TRUST-BFT protocols face the safety violation of Claim 3 is because they have a *dependence* on trusted components at byzantine replicas. In comparison, in BFT protocols, a byzantine replica is free to *assume* any state after a crash as the underlying protocol is not dependent on the safe recovery of the byzantine replica. This implies that in a BFT protocol, if a byzantine replica does not want to recover its state, it can start participating in the consensus as soon as it restarts after the crash. This behavior does not affect the fate of already committed transactions as for each such transaction, there are always $\mathbf{f}+1$ non-faulty replicas that have marked the same.

For TRUST-BFT protocols to prevent the attack of Claim 3, every crashed trusted component, irrespective of whether it is co-located with a faulty or non-faulty replica, needs to wait for signed states from $\mathbf{f}+1$ other trusted components before marking its recovery as complete. This implies a failed trusted component cannot include its own state to reach this $\mathbf{f}+1$ count, something which it could do during the PREPARE, COMMIT, and CHECKPOINT phases. We summarize this in the following definition.

DEFINITION 6.1.2. *A fail-crashed trusted component* $\mathrm{T}_R \in \mathfrak{I}$ *at a replica* R *needs to wait for attested states from* $\mathbf{f}+1$ *other trusted components to recover its state. Further,* T_R *cannot participate in any phase of the consensus during its recovery.*

6.1.8. Recovery Protocol.

Definition 6.1.2 illustrates that any crashed trusted component that wishes to recover its state needs to wait for states from $\mathbf{f}+1$ other distinct trusted components. As at most $\mathbf{f}+1$ replicas out of $\mathbf{n}$ are non-faulty so a failed T_R may need to wait for messages from all the non-faulty replicas. *Why?* Because the remaining $\mathbf{f}-1$ replicas may act byzantine and may not want to help T_R recover its state. As a consequence, they may not send their states to T_R.

To facilitate efficient recovery of T_R, we present a recovery protocol in Figure 6.3. Using this protocol, a recovering T_R attempts to learn committed client requests (if any) corresponding to each sequence number since the last checkpoint. If the primary $\mathcal{P}$ is byzantine, then it may ensure that only one non-faulty replica commits each client request and all the $\mathbf{f}$ byzantine replicas participate in each consensus. As a result, T_R may end up waiting for messages from all the non-faulty replicas.

Furthermore, $\mathcal{P}$ may ask its $\mathrm{T}_{\mathcal{P}}$ to assign client requests non-consecutive but monotonically increasing sequence numbers. This implies that no message may be assigned to some sequence numbers. Such empty sequence numbers are considered *skipped* and delay the recovery protocol as unless T_R receives states from all the non-faulty replicas, it cannot generate the complete state.

```
recover (used by $R$ to facilitate its recovery) :
event Trusted component $\mathrm{T}_R$ finds its logs empty or counters reset do
    $\mathrm{T}_R$ creates $m := \langle \textsc{Recover} \rangle_{\mathrm{T}_R}$ and forwards $m$ to $R$.
    $R$ forwards $m$ to all the replicas.

event $R$ receives $m := \langle \textsc{Snapshot}(l, \mathcal{S}, \mathcal{C}) \rangle_{\mathrm{T}_{R'}}$ from $R'$ do
    Forwards $m$ to $\mathrm{T}_{Replica}$, which performs following steps.
event $\mathrm{T}_R$ waits for $\langle \textsc{Snapshot}(l, \mathcal{S}, \mathcal{C}) \rangle_{\mathrm{T}_{R'}}$ from $\mathbf{f}+1$ replicas do
    $mc$ := Identifier of most recent checkpoint among all $\mathcal{C}$.
    $ml$ := Largest identifier among all $l$.
    $\mathbb{S}$ := Set of $\mathbf{f}+1$ $\mathcal{S}$.
    for $i$ in $mc$ to $ml$ do
        if Exists a request with sequence number $i$ in $\mathbb{S}$ then
            Update the state and add it to the log.

snapshot (used by $R'$ to help a crashed replica recover) :
event $R'$ receives message $m := \langle \textsc{Recover} \rangle_{\mathrm{T}_R}$ do
    Forwards $m$ to its trusted component $\mathrm{T}_{R'}$.
event $\mathrm{T}_{R'}$ receives $\langle \textsc{Recover} \rangle_{\mathrm{T}_R}$ do
    $\mathcal{C}$ := Last checkpoint with proofs (Checkpoint messages from $\mathbf{f}+1$ replicas).
    $\mathcal{S}$ := All the committed requests with proofs since last checkpoint.
    $l$ := Highest sequencer number in $\mathcal{S}$.
    $\mathrm{T}_{R'}$ creates $m := \langle \textsc{Snapshot}(l, \mathcal{S}, \mathcal{C}) \rangle_{\mathrm{T}_{R'}}$ and forwards to $R'$, which forwards $m$ to $R$.
```

FIGURE 6.3. Recovery protocol to help a fail-crashed trusted component at a replica recover its state.

Once T_R has determined all the client requests, it updates its associated counters or logs and is ready to participate in the consensus protocol. Further, during the recovery process, T_R ignores any other messages it receives.

Protocol Description. Our recover protocol requires *two* phases of communication and guarantees termination if the network is reliable. The protocol expects trusted components to log their requests as counter-based designs cannot provide commit-proofs unless they also store their exchanged messages. In the *first* phase, the recovering replica R's trusted component T_R discovers that its logs are empty or counters are reset. So, T_R constructs a RECOVER message and requests R to forward this message to all the replicas.

When a replica $R' \neq R$ receives a valid RECOVER message, then it forwards it to its $\mathrm{T}_{R'}$. This allows the $\mathrm{T}_{R'}$ to create a SNAPSHOT message. This SNAPSHOT message includes the highest agreed checkpoint $\mathcal{C}$ and all the committed requests with proofs $\mathcal{S}$ since the last checkpoint. When the recovering replica R receives SNAPSHOT messages from $\mathbf{f}+1$ distinct replicas, it initiates the process of updating its state. For this, T_R tries to find a request corresponding to each sequence number between the highest checkpoint and the largest committed identifier. Next, we argue termination of our recovery protocol.

THEOREM 6.1.3. *If the network is reliable and a replica R delivers all the incoming* SNAPSHOT *messages to its crashed trusted component* T_R*, then the* T_R *can recover its state successfully if it employs the algorithm of Figure 6.3.*

PROOF. Assume that the algorithm of Figure 6.3 does not terminate. This implies that a crashed T_R can never successfully recover. If such is the case, it implies that T_R never got SNAPSHOT messages from $\mathbf{f}+1$ distinct replicas. We know that the network is reliable and the replica R delivers any SNAPSHOT messages it receives to T_R. Further, at most $\mathbf{f}-1$ replicas could act faulty. As a result, at least $\mathbf{f}+1$ non-faulty replicas will send a SNAPSHOT message to R. This contradicts our assumption. □

6.1.9. Lack of Concurrency Opportunities.

In this section, we illustrate how existing TRUST-BFT protocols inhibit opportunities of concurrently processing multiple requests. Specifically, we want to argue that existing TRUST-BFT protocols are slower than their BFT counterparts.

As illustrated in Observation O3, TRUST-BFT protocols require their trusted components to attest each message prior to broadcasting that message on the network. This attestation acts as a proof that the trusted component logged the message at the required position or assigned the message corresponding value of the counter. We use Claim 4 to prove that this design inhibits concurrent processing of client requests.

CLAIM 4. *In a service* $\mathfrak{S}$ *employing* PBFT-EA *protocol, if a non-faulty primary* $\mathcal{P}$ *assigns two transactions* Ti *and* Tj *sequence numbers* i, j*, such that* $i < j$*, then* $\mathcal{P}$ *cannot initiate consensus on* Tj *before completing consensus on* Ti*.*

PROOF. Assume that $\mathcal{P}$ allows consensuses of Ti and Tj to run concurrently. This implies that a replica R may receive PREPREPARE for Tj before Ti. Further, we know that when R receives any transaction, it asks its T_R to append it to the log or assign it a counter value and expects an attestation in return. As a result, R would have an attestation $\langle \text{ATTEST}(id, j, Tj)\rangle_{\mathrm{T}_R}$, which it will forward with the PREPARE message to all the replicas.

Now, when R receives PREPREPARE message for Ti, its T_R will ignore this message as its log or counter are already set at j $(i < j)$ and the APPEND() function prevents processing a lower sequence number request. As a result, R cannot forward a PREPARE message for Ti. This implies that the consensus of Ti may never complete, which contradicts our assumption as a non-faulty primary always tries to successfully reach consensus on each request. □

Our Claim 4 applies to all the existing TRUST-BFT protocols, irrespective of whether they employ trusted logs or counters. Hence, these protocols are unable to concurrently run consensus on multiple client requests, which is employed by existing BFT protocols to increase their throughput [21]. The key reason TRUST-BFT protocols are unable to process requests concurrently is because the APPEND operation only allows appending to monotonically increasing positions in the log. This inhibits the primary replica from initiating multiple concurrent consensuses. We wish to claim that TRUST-BFT designs employing trusted-logs can be allowed to process client requests concurrently. Notice that our claim do not extend to implementations employing trusted-counters.

6.1.10. Facilitating Concurrent Consensuses in Trusted Logs.

TRUST-BFT protocols employing trusted components with trusted logs require a small set of changes to facilitate concurrent request processing. As these protocols log all the requests since last checkpoint, it is easy for non-faulty replicas to delay ordering client requests until execution. This implies that to guarantee safety, each trusted component only needs to log each request and present a proof when required. This allows us to present *updated* APIs for the log accessing functions.

(1) APPENDC(id, x) – Assume the id-th log of a T_R is at position ct. This function appends x to the position ct, moves ct to $ct + 1$, and returns the attestation $\langle \text{ATTEST}(id, ct, x)\rangle_{\text{T}_R}$ as a proof of this binding.

(2) LOOKUPC(id, m) – Returns an attestation $\langle \text{ATTEST}(id, S)\rangle_{\text{T}_R}$ where S is a snapshot of the log holding m.

In the APPENDC function, a T_R appends a message m to the next available position in the log. Notice that the replica R no longer proposes any sequence number for m. As a result, the sequence number for each request corresponds to its position in the log. Similarly, in the LOOKUPC function, a T_R returns as attestation a snapshot of the log that includes the message m.

6.1.10.1. **Protocol for Concurrent Consensus.** With the help of our AppendC and LookupC functions, it is easy to redesign a majority of existing primary-backup TRUST-BFT protocols such that they provide support for concurrent request processing. Next, we state the general flow for these redesigned protocols.

- When the primary $\mathcal{P}$ receives a client request m, it uses AppendC function to log m at its trusted component $\mathrm{T}_{\mathcal{P}}$. In return, $\mathcal{P}$ receives an attestation $\langle \textsc{Attest}(id, ct, m) \rangle_{\mathrm{T}_{\mathcal{P}}}$, which suggests that $\mathrm{T}_{\mathcal{P}}$ added m to the id-th *preprepare log* and ct acts as the sequence number for m.
- Prior to sending any message m, each replica R calls the $\textsc{AppendC}(id, m)$ function to access its T_R. As a result, T_R appends m to the log with identifier id.
- Once T_R has appended m to the log, it takes a snapshot S of its log id and generates an attestation $\langle \textsc{Attest}(id, S) \rangle_{\mathrm{T}_R}$. The snapshot could be as simple as the last appended message m.
- Although we allow concurrent processing, we require access to trusted logs be either single-threaded or in a mutex.
- Each replica R executes each request in the order suggested by $\mathrm{T}_{\mathcal{P}}$. As a result, once a replica R has completed consensus on a client request Tj, it delays executing Tj until it has executed every transaction Ti, such that $i < j$.

Using the steps stated above, we can transform an existing primary-backup TRUST-BFT protocol into a variation that allows replicas to process requests concurrently. In the rest of the chapter, we refer to such variations as Concurrent TRUST-BFT (CTRUST-BFT). Our CTRUST-BFT protocols employ the existing view-change protocols to replace the primary in case of failures. Further, the recovery protocol we introduced in Section 6.1.8 can be applied to these protocols without any changes. Such is the case because these CTRUST-BFT protocols require their trusted components to log all the transactions, albeit not in order. Moreover, all the replicas execute requests in the order proposed by the primary[1]. Next, we prove that our CTRUST-BFT protocols are safe.

THEOREM 6.1.4. *A service* $\mathfrak{S} = \{\mathfrak{R}, \mathfrak{C}, \mathrm{T}\}$ *employing a* CTRUST-BFT *protocol where each trusted component has access to trusted logs yields a safe consensus.*

PROOF. We prove this as follows:

[1]Our CTRUST-BFT protocols can also employ optimizations that allow parallel execution of independent transactions without affecting their safety [84].

If the primary $\mathcal{P}$ is non-faulty, then all the non-faulty replicas in $\mathfrak{R}$ participate in consensus of each request proposed by $\mathcal{P}$ and execute them in the order they are logged by $\textsc{t}_{\mathcal{P}}$.

If the primary $\mathcal{P}$ is byzantine, then it can prevent at most D, $1 \leq D \leq f$, non-faulty replicas in executing any request. Such a primary may force the replicas in D to participate in consensus of several requests but cannot force them to execute any request out-of-order. Notice that the AppendC function disallows skipping any log position unlike the Append function. Further, $\mathcal{P}$ needs support of at least $g = D - \mathbf{f}$ replicas to make client believe that its request is complete as a client needs $\mathbf{f} + 1$ identical responses. Moreover, $\mathcal{P}$ cannot equivocate as each request is assigned a sequence number by its $\textsc{t}_{\mathcal{P}}$.

If $\mathcal{P}$ prevents a non-faulty replicas in executing any request, then it may get replaced. Post committing a request, each replica R starts a timer which runs until the request is executed. If the timer timeouts, then R sends a ViewChange message to all the replicas. When $\mathbf{f} + 1$ replicas support view-change, then the new primary helps all the replicas reach a common state and starts the new view.

If the network is unreliable, then the messages sent by primary or replicas may get indefinitely delayed or dropped. This will either prevent some replicas in executing the requests, or may cause a primary replacement. However, it cannot make the system unsafe. □

6.1.11. Concurrency Dilemma of Trusted Counters.

In the previous section, we illustrated that it is possible to parallelize existing trust-bft protocols that employ trusted logs. However, we cannot extend similar insights to the trusted counter based implementations. In specific, if our Ctrust-bft protocols employ trusted counters instead of trusted logs, then such implementations would be unsafe. Any Ctrust-bft protocol that wants its trusted components to employ a trusted counters need to also provide a support for trusted logs. We prove this next.

Claim 5. *A service $\mathfrak{S} = \{\mathfrak{R}, \mathfrak{C}, \textsc{t}\}$ employing a* Ctrust-bft *protocol where each trusted component has access to only trusted counters cannot guarantee a safe consensus.*

PROOF. Assume that a CTRUST-BFT protocol employing trusted counters yields a safe consensus. This implies that each request executed by a replica R will persist across view-changes. In specific, it is safe for a client to mark its request as complete if it receives $\mathbf{f}+1$ identical responses.

Let us divide $\mathbf{n}$ replicas into three sets: $\mathcal{F} = \mathbf{f}$ be the faulty replicas, $D = \mathbf{f}$ be the non-faulty replicas, and $g = \mathcal{F} - D$ be the remaining non-faulty replica. Assume that the primary $\mathcal{P}$ is byzantine, $\mathcal{P} \in \mathcal{F}$. Let m_i denote the message m with sequence number i assigned by $\mathrm{T}_{\mathcal{P}}$ (value of the counter). We assume that $\mathcal{P}$ receives three client requests m_i, $i \in [1, 3]$.

Now, byzantine $\mathcal{P}$ sends PREPREPARE(m_1) to g, PREPREPARE(m_2) to D, and the replicas in $\mathcal{F}$ receive PREPREPARE for all messages from m_1 to m_3. Further, assume that messages from g to D and vice versa are lost and replicas in $\mathcal{F}$ only send required number of messages, to help g and D to commit their respective messages.

As a result, replicas in g and D will successfully commit messages m_1 and m_2, respectively. Further, g can execute m_1 and reply to the client. Notice that replicas in D have to wait as they do not have access to m_1. However, replicas in $\mathcal{F}$ can successfully execute both m_1 and m_2 and can reply to the clients.

The client for request m_1 will receive $\mathbf{f}+1$ responses and will mark the request as complete. Notice that all the trusted counters for replicas in g and D are at one. Assume that the replicas in $\mathcal{F} - \mathcal{P}$ avoid processing request m_3. In such a case, the counters of replicas in $\mathcal{F} - \mathcal{P}$ are at two.

Assume a view change takes place and g is partitioned (unable to communicate with any replica). Let the new primary $\mathcal{P}\prime$ be from D. $\mathcal{P}\prime$ initiates new view once it receives VIEWCHANGE messages from $\mathbf{f}+1$ replicas. In such a case, $\mathcal{P}\prime$ may receive messages from replicas in D and one replica in $\mathcal{F}$. As replicas in D only committed m_2, they have no knowledge of m_1. The one faulty replica from $\mathcal{F}$ will claim no knowledge of m_1 by stating that it only received m_2 and m_3 from the primary. To prove this it will show the value of its counters set at two. Hence, $\mathcal{P}\prime$ will conclude that no request got executed in previous view, and only m_2 was committed. This decision makes the system unsafe. □

6.1.12. Lightweight Trusted Consensus.

CTRUST-BFT protocols aim to bridge the gap between the performance achieved by existing BFT and TRUST-BFT protocols. Although CTRUST-BFT protocols scale better than their BFT counterparts, their throughputs are limited by excessive logging. We now envision the design of byzantine fault-tolerant protocols employing trusted components that can meet the following two goals:

(G1) **No Trusted Logging.** A CTRUST-BFT protocol that does not require messages to be logged in trusted logs.

(G2) **Minimal Active Trusted Component.** A CTRUST-BFT protocol that requires only few active trusted components.

Assume that we are able to derive the design of a CTRUST-BFT protocol that satisfies our Goals G1 and G2. Such a protocol would be *lightweight*, *fast* and *trustable*. Hence, we would refer to these protocols as LFT-BFT protocols. These LFT-BFT protocols satisfy Goal G1 by requiring each trusted component to only use trusted counters. To fulfill Goal G2, our LFT-BFT protocols will require only a subset of trusted components to participate in each consensus. These participating trusted components are termed as *active* while rest are designated as *passive*.

Our LFT-BFT protocols require only the primary replica to make use of its trusted component for assign sequence numbers and attest the incoming client requests. Hence, this trusted component at the primary is termed as active. At every other replica, the trusted components remain idle (passive) and do not participate in the consensus.

$3\mathbf{f}+1$ Replicas: To reduce the amount of trusted logging and to facilitate concurrent request processing with trusted counters, we need to expand our set of replicas. In specific, we fall back to the number of replicas needed by existing BFT protocols. Our LFT-BFT protocols expect a system of $\mathbf{n} = 3\mathbf{f}+1$ replicas. This larger set of replicas provides us with stronger quorums, which helps to reduce the number of active trusted components and facilitates use of trusted counters.

Why use lft-bft protocols: Before we illustrate the design of our LFT-BFT protocols, we need to argue the merit behind these designs. One of the prime reasons for opting TRUST-BFT protocols is that they help in reducing replication. This implies that by switching back to consensus protocols that require $3\mathbf{f}+1$ replicas, we will be forgoing the gains. Although such an observation is partly true, it overlooks the costs introduced by existing TRUST-BFT protocols to a replicated system.

We have already illustrated that existing TRUST-BFT protocols cannot run concurrent consensuses on client requests. To resolve this challenge, we presented the design of our CTRUST-BFT protocols. However, CTRUST-BFT protocols do not meet our Goals G1 and G2. Managing trusted logs is expensive. In each phase of consensus, replicas need to exchange attested states among each other. Our LFT-BFT protocols not only avoid these costs but also require less communication than their BFT counterparts.

Further, our LFT-BFT protocols neither face the liveness anomaly of our Claim 1, nor do they need message re-transmissions to avoid simple message losses. This allows them to provide same liveness guarantees as their BFT counterparts. Finally, as the trusted component at each replica is passive, our LFT-BFT protocols do not require a special recovery mechanism to recover their state.

6.1.13. Designing a lft-bft protocol.

We now present steps to convert a primary-backup TRUST-BFT protocol into a LFT-BFT protocol. We expect each trusted component to be equipped with trusted counters. Further, we need to make slight modifications in the APIs used by a replica R to access its trusted component T_R.

(1) APPENDL(id, x) – Assume the id-th counter of T_R has value ct. This function associates ct with message x and returns an attestation $\langle \text{ATTEST}(id, ct, x)\rangle_{\text{T}_R}$ as a proof of this binding. Post binding, T_R increments ct to $ct + 1$.

(2) LOOKUP(id, ct) – Performs no action and returns nothing.

Our LFT-BFT protocols do not require a LOOKUP function as non-primary replicas do not access their trusted components during the consensus phases. This implies that only the primary replica makes use of the APPEND function to access its trusted component. Next, we lay down the design of a LFT-BFT protocol.

• A service $\mathfrak{S} = \{\mathfrak{R}, \mathfrak{C}, \text{T}\}$ employing a LFT-BFT protocol to achieve consensus among its replicas assumes $|\mathfrak{R}| = \mathbf{n} \geq 3\mathbf{f} + 1$.

• On receiving a client request $m := T$, the primary $\mathcal{P}$ calls the APPEND(id, m) to access its trusted component $\text{T}_{\mathcal{P}}$. As a result, $\text{T}_{\mathcal{P}}$ binds the current value of its id-th counter ct to m and returns an attestation $\langle \text{ATTEST}(id, ct, m)\rangle_{\text{T}_{\mathcal{P}}}$. Next, $\mathcal{P}$ initiates consensus by sending the PREPREPARE message and its attestation for m to all the replicas in $\mathfrak{R}$.

• When a replica R receives a PREPREPARE message from $\mathcal{P}$, it verifies the attestation. If the attestation is valid, it agrees to order m at sequence ct and follows the remainder steps of the consensus protocol. However, R never accesses its passive T_R.

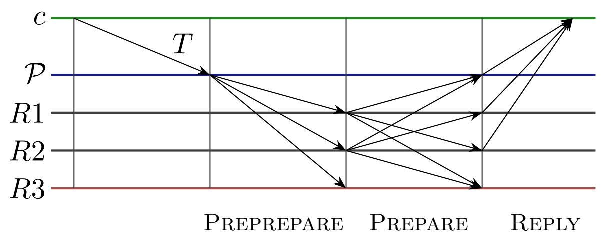


FIGURE 6.4. Schematic representation of the MIN-LFT protocol where only primary requires access to trusted components.

- Each replica R executes each request in the order suggested by $\mathrm{T}_{\mathcal{P}}$. As a result, once a replica R has completed consensus on a client request Tj, it delays executing Tj until it has executed every transaction Ti, such that $i < j$.

6.1.14. Case Study: Min-LFT.

In Section 2.1, we introduced the design of PBFT protocol. Although PBFT yields a safe consensus, it requires three phases, of which two necessitate quadratic communication complexity. The design of PBFT led to the PBFT-EA protocol, which reduces the amount of replication by $\mathbf{f}$. Following this, Veronese et al. [133] introduced the MINBFT protocol, which reduces the cost of consensus by eliminating one phase of quadratic communication of PBFT-EA. However, MINBFT makes use of trusted counters under the reduced replication setting ($n = 2\mathbf{f} + 1$). As a result, MINBFT cannot be converted into a CTRUST-BFT protocol.

Using steps stated in Section 6.1.13, we can design a LFT-BFT-variant of PBFT. We refer to this protocol as MIN-LFT and use Figure 6.4 to represent its normal-case consensus flow. Although the consensus steps for this protocol are self-explanatory, we highlight its key properties next.

(1) MIN-LFT achieves consensus in two phases, and only one of those phases requires quadratic communication complexity.

(2) MIN-LFT requires active trusted components only at the primary replica.

(3) The primary replica require no trusted logs and uses only trusted counters.

(4) MIN-LFT supports concurrent request processing.

(5) Each replica marks a request as *prepared* only it receives PREPARE messages from $2\mathbf{f} + 1$ replicas. Hence, MIN-LFT makes use of stronger quorums.

In case of primary failure, MIN-LFT employs PBFT's view-change protocol. Like PBFT, the new primary can only initiate a view-change if it receives VIEWCHANGE message from $2\mathbf{f}+1$ replicas. Next, We prove the safety of MIN-LFT's consensus.

THEOREM 6.1.5. *In a service* $\mathfrak{S} = \{\mathfrak{R}, \mathfrak{C}, \textsc{t}\}$ *where* $|\mathfrak{R}| = \mathbf{n} = 3\mathbf{f}+1$*,* MIN-LFT *protocol guarantees a safe consensus.*

PROOF. If the primary $\mathcal{P} \in \mathfrak{R}$ is non-faulty, then $\mathcal{P}$ will require its trusted component $\textsc{t}_{\mathcal{P}}$ to attest each client request m. This attestation will associate a unique counter value ct with m and all the non-faulty replicas will successfully execute m as the ct-th request in order.

In the case $\mathcal{P}$ is byzantine, it can prevent $1 \leq D \leq f$ non-faulty replicas from participating in consensus for m. However, a non-faulty replica R can mark the ordering of m complete when it receives PREPARE messages from $2\mathbf{f}+1$ distinct replicas. For this to happen, $\mathcal{P}$ needs to send the PREPREPARE for m (along with its attestation) to a majority of non-faulty replicas (at least $\mathbf{f}+1$). As a result, if a view-change takes place in future, there will be at least one non-faulty replica that received m from $\mathcal{P}$. Hence, m will persist across views.

In the case, $\mathcal{P}$ attempts to prevent replicas in $\mathfrak{R}$ from executing one or more requests by not sending PREPREPARE messages for consecutively sequenced requests, then $\mathcal{P}$ will be replaced through a view-change. Post view-change, replicas will converge to a common state and will be able to execute each request. □

6.2. Byzantine Fault-Tolerant Serverless-Edge Architecture

With the rise of interactive and time-critical applications, the expected performance requirements are becoming more stringent. Emerging edge applications, specifically the Internet of Things (IoT) applications, such as the ones in Industry 4.0, smart spaces, and transport technologies, expect time-critical responses. Further, emerging edge applications also include mobile applications that employ immersive technologies such as Virtual and Augmented Reality (V/AR). These applications require fast compute in the order of *ten* milliseconds. This stringent latency requirement makes relying on cloud data centers infeasible as the round-trip latency is in the range of 100 milliseconds to seconds.

The *edge-compute* architecture can mitigate high latencies due to cloud-based wide-area compute as the data is processed at the edge of the network close to clients. However, edge-compute architectures often struggle due to limited available storage. As a result, in this work, we envision an *edge-cloud model*, where the storage and compute are distributed across the edge and cloud nodes. The goal of this edge-cloud model is to reap the benefits of both worlds: edge nodes (by placing time-critical compute and storage functions on edge nodes) and cloud nodes (by placing extensive compute and storage functions on cloud nodes).

The edge-cloud model faces two daunting challenges: (1) edge nodes cannot be trusted, and (2) the model needs to be lightweight. The edge nodes are untrusted as they may be operated by multiple third-party providers that are not in the trust domain of the application. This creates opportunities for byzantine or malicious attacks. As a result, the edge-cloud model needs to be *Byzantine Fault-Tolerant* (BFT). However, we may leverage the trust on the cloud—which is typically in the trust domain of the application such as a private cloud operated by the application owner or operated by a public cloud partner. Hence, in our edge-cloud model, we trust the cloud, but expect edge nodes to act byzantine.

We also know that edge nodes have access to limited compute resources in comparison to cloud nodes. However, time-critical edge applications require access to a varying degree of computational resources. Depending on the number of application users, the required number of resources throttles between the high and low. This indicates that an edge application needs to reserve a large amount

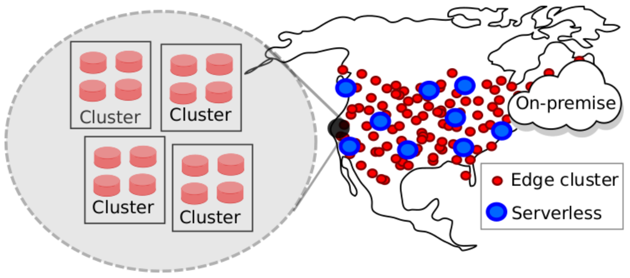


FIGURE 6.5. The serverless-edge system model.

of compute nodes ahead of time. However, reserving hundreds of edge nodes ahead of time is neither feasible nor beneficial.

In our edge-cloud model, we resolve this challenge by adopting the *serverless* cloud infrastructure, which enables lightweight and fine-grained reservation of compute resources. Relying on the serverless architecture ensures that an application only reserves what it needs and when it needs, which helps it to monetize its costs. However, a key challenge remains: how can the untrusted edge nodes interact with the untrusted serverless cloud infrastructure to fulfill a client transaction.

In this work, we propose our SERVERLESSBFT protocol that tackles the challenge of reliably connecting untrusted edge nodes and serverless cloud. To ensure that the combination of these seemingly incompatible technologies does not require extensive coordination, our SERVERLESSBFT protocol presents a modular pipeline that tackles each challenge in a specific stage of the pipeline: BFT coordination of edge devices, BFT compute processing on serverless cloud, and data storage on a trusted private database.

6.2.1. The Case for Serverless-Edge Model. Serverless technology eases the use of cloud resources by allowing users to simply upload their functions (code) and leave the tasks of function execution, server provisioning, and administration to the cloud. The success of serverless technology can be gauged from the fact that all the major cloud providers also provide access to some serverless infrastructure [79, 97]. When a user sends its function to a serverless cloud, the service provider processes the request by spawning some workers or *executor*. As a result, the user's focus is only on *how to design* its application, while the serverless cloud focusses on *how to execute* the application.

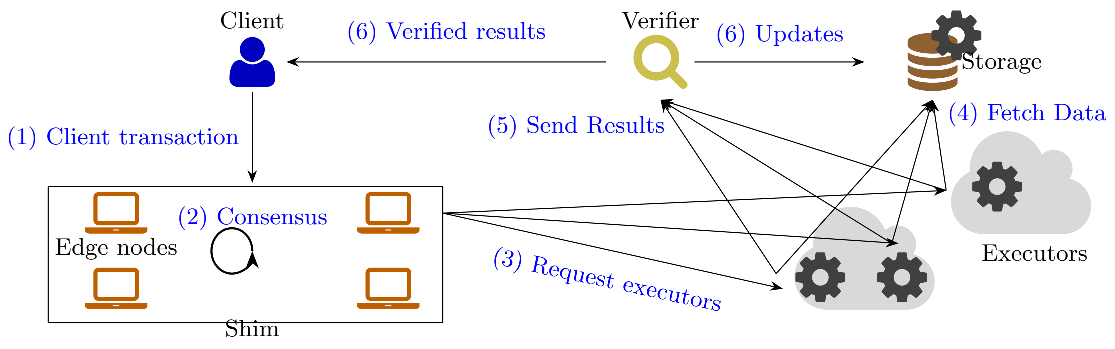


FIGURE 6.6. Byzantine-Fault Tolerant Serverless Architecture.

Traditional edge applications assume two types of nodes: (i) untrusted *edge devices* that are deployed at the edge (for instance on cars for a smart traffic application), and (ii) trusted on-premise nodes maintained privately by the application organization. Edge devices perform some in-situ operations, such as data compression, before sending it to the on-premise nodes for processing and storage.

There are two key challenges with the existing edge model: (i) edge nodes cannot be trusted, and (ii) application organization may lack resources for a scalable on-premise cloud. To resolve these challenges, we envision a new *serverless-edge* model (refer to Figure 6.5). Our serverless-edge model consists of *three* major components: untrusted edge nodes, untrusted serverless cloud, and trusted private storage. To ensure a reliable and lightweight coordination among these components, we design our SERVERLESSBFT protocol. Our SERVERLESSBFT protocol views the set of edge nodes as a BFTsystem and employs appropriate consensus protocols to help edge nodes reach a reliable agreement.

6.2.2. Serverless-Edge Infrastructure.

Our serverless-edge architecture comprises of three key components: (1) untrusted edge nodes, (2) untrusted serverless cloud *executors*, and (3) trusted *storage*. We vision clients or end-users to interact with our serverless-edge infrastructure to process their requests. As the three components in our serverless-edge architecture may be operated by different organizations, which may not trust each other, we need a resilient protocol that can sustain any adversarial actions. We achieve this task through our SERVERLESSBFT protocol that is resilient against byzantine attacks.

We use Figure 6.6 to illustrate the vision of our serverless-edge architecture. In this figure, we introduce the notion of a lightweight *shim* and a trusted *verifier* apart from our key components. Next, we explain the tasks of each component.

• ***Client.*** Any user that accesses an edge application employing our serverless-edge architecture becomes a client in our system. We perceive each client interaction as a request or transaction.

• ***Shim.*** As the edge nodes perform in situ processing of client request, we cluster the neighboring edge nodes into a shim. Shim nodes work together to validate and order client requests, prior to forwarding them to the serverless cloud for processing. To achieve this task, shim nodes participate in a BFT consensus protocol. Our shim can employ any existing BFT consensus protocol to achieve agreement among the edge nodes.

• ***Serverless Exxecutors.*** Once the shim orders a client request, it forwards the request and its ordering information to the serverless cloud provider. As we cannot trust the cloud, we spawn multiple executors at the cloud to process the request. Further, cloud does not trust the shim (edge nodes). Hence, prior to executing the client request, each executor validates the order for this request. While executing a client request, executors may require access to the data. As a result, we allow executors to communicate directly with the trusted storage.

• ***Verifier.*** As we trust the storage, we cannot permit executors (some of which could be byzantine) to issue updates to the database post-execution of a client transaction. Hence, we place a lightweight trusted verifier at the storage, which controls what updates can be applied to the storage. The verifier collects results from the executors and once it has a quorum of matching results, it replies to the client and updates the database.

• ***Storage.*** The trusted storage replies to the read requests from the executors and updates the database on requests from the verifier.

6.2.3. Preliminaries.

To explain how our ServerlessBFT protocol manages resilient communication across our serverless-edge infrastructure, we need to lay down some of the notations and assumption.

We represent our serverless-edge architecture $\mathcal{A}$ through a quintuple, $\mathcal{A} = \{\mathfrak{C}, \mathfrak{R}, \mathcal{E}, \mathcal{S}, \mathcal{V}\}$. In this architecture, we use $\mathfrak{C}$ to denote the set of clients, $\mathfrak{R}$ to denote the shim of edge nodes, $\mathcal{E}$ to denote the serverless cloud executors, $\mathcal{V}$ and $\mathcal{S}$ to denote the verifier and storage, respectively. Next, we denote the byzantine fault-tolerance requirement expected by our ServerlessBFT protocol.

Fault-Tolerance Requirement at Shim. We use the notation $\mathcal{F}$ to denote the set of faulty nodes in $\mathfrak{R}$, and notation $\mathcal{NF} = \mathfrak{R} \setminus \mathcal{F}$ to denote the set of non-faulty nodes. Faulty nodes can crash-fail or act byzantine. We write $\mathbf{n}_{\mathfrak{R}} = |\mathfrak{R}|$, $\mathbf{f}_{\mathfrak{R}} = |\mathcal{F}|$, and $\mathbf{nf}_{\mathfrak{R}} = |\mathcal{NF}|$. As shim nodes work together to order client requests, they need to participate in a BFT consensus protocol. Prior works have illustrated that for a system of nodes to be BFT, it should have at least $\mathbf{n}_{\mathfrak{R}}$ nodes, where $\mathbf{n}_{\mathfrak{R}} \geq 3\mathbf{f}_{\mathfrak{R}} + 1$ ($\mathbf{nf}_{\mathfrak{R}} > 2\mathbf{f}_{\mathfrak{R}} + 1$) [21, 48, 87]. We expect our shim to meet this requirement.

Fault-Tolerance Requirement at Cloud. To handle attacks at the serverless cloud, we require the service provider to spawn $\mathbf{n}_{\mathcal{E}} = |\mathcal{E}|$ executors. Like shim, we denote the number of faulty and non-faulty executors as $\mathbf{f}_{\mathcal{E}}$ and $\mathbf{nf}_{\mathcal{E}}$, respectively. However, to handle $\mathbf{f}_{\mathcal{E}}$ faulty executors, we only require $\mathbf{n}_{\mathcal{E}} \geq 2\mathbf{f}_{\mathcal{E}} + 1$. This leads us to following two important observations:

(1) The values for $\mathbf{f}_{\mathcal{E}}$ and $\mathbf{f}_{\mathfrak{R}}$ may or may not be same and depends on the application developer.

(2) If $\mathbf{f}_{\mathcal{E}} = \mathbf{f}_{\mathfrak{R}} = \mathbf{f}$, then our ServerlessBFT protocol expects at least $\mathbf{f}$ less executors than the edge nodes in the shim. Our reduced set of executors is based on a noteworthy insight by Yin et al. [136].

We use a function $\mathsf{id}()$ to assign an identifier to each replica $R \in \mathfrak{R}$ and each executor $\text{E} \in \mathcal{E}$. We assume that non-faulty replicas and executors follow the protocol: on deterministic inputs produce deterministic outputs. We *do not* make any assumptions on the behavior of the clients and *permit* faulty replicas or executors to behave arbitrarily and perform coordinated attacks.

6.2.4. Architecture.

We now explain in detail how our ServerlessBFT protocol ensures resilient transaction processing in the serverless-edge architecture. ServerlessBFT protocol aims to make the serverless-edge architecture byzantine fault-tolerant by shielding clients and their trusted storage from attacks while guaranteeing continuous transaction processing. We use Figure 6.7 to present the transactional flow through our serverless-edge infrastructure.

In this figure, the shim consists of $\mathbf{n}_{\mathfrak{R}} = 4$ nodes, which participate in a BFT consensus protocol to order an incoming client transaction. As stated earlier, shim can employ any existing BFT protocol, such as Pbft [21], PoE [65], Zyzzyva [87], Sbft [48], and so on. For the sake of

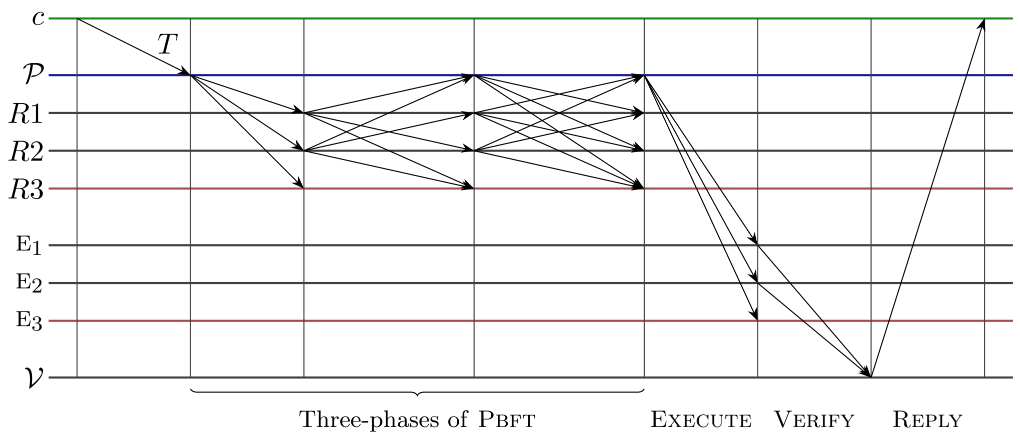


FIGURE 6.7. Schematic representation of the transactional flow in SERVERLESS-BFT protocol. Given a client transaction T, the nodes of the shim work together to order this transaction, following which the primary $\mathcal{P}$ invokes the executors at the service provider to execute T. Post execution, the executors send their results to the verifier, which replies to the client.

explanation, we assume the nodes of the shim follow a primary-backup protocol, such as PBFT, for ordering client transactions. Post consensus, SERVERLESSBFT requires the primary node $\mathcal{P}$ to invoke $\mathbf{n}_{\mathcal{E}} = 3$ executors at the serverless cloud. These executors send the results of executing client transactions to the verifier $\mathcal{V}$ for validation and updating the database.

Client Request and Response. The SERVERLESSBFT framework gets into action when a client c wants a transaction T to be processed. To fulfill this task, c creates a message $\langle T \rangle_c$ and sends this message to the primary node $\mathcal{P}$ of the shim.[2] Notice that c employs DS to sign this message. The client c marks $\langle T \rangle_c$ as processed when it receives a RESPONSE message from the verifier $\mathcal{V}$. As c knows that $\mathcal{V}$ is a trusted entity in our infrastructure, it readily accepts the response.

Shim Ordering. To reliably order an incoming client request, SERVERLESSBFT requires the shim to run a BFT protocol. For the sake of explanation, we assume that the shim orders each client request using the PBFT protocol. As a result, one of the edge nodes of the shim is designated as the *primary* node. On receiving a client request $\langle T \rangle_c$, the primary $\mathcal{P}$ checks if $\langle T \rangle_c$ is well-formed. If this is the case, then primary initiates the consensus dictated by the PBFT protocol. Next, for the sake of completeness, we re-iterate the PBFT protocol.

[2] Some BFT protocols require a client request to be sent to all the nodes.

Pre-prepare. The primary $\mathcal{P}$ assigns a sequence number k to the well-formed client message $m := \langle T \rangle_c$ and sends it as a Preprepare to all the nodes of the shim. This Preprepare message also includes a digest $\Delta = \text{digest}(m)$, which is used in future communication. Notice that primary signs this message using MAC, which provide sufficient guarantees for this phase.

Prepare. When a node $R \in \mathfrak{R}$ receives a Preprepare message from the primary $\mathcal{P}$, it run it through a series of checks. If the checks are successful, then R agrees to support the order k for this client request and broadcasts a Prepare message.

Commit. When a node R receives identical Prepare messages from $\mathbf{nf}_{\mathfrak{R}}$ nodes, it marks the request m as *prepared* and broadcasts a Commit messages. Notice that we require each node R to employ DS to sign the Commit messages. Next, R waits for arrival of identical Commit messages from $\mathbf{nf}_{\mathfrak{R}}$ nodes If this is the case, then R proceeds to mark m as *committed.*

Serverless Cloud. When the primary $\mathcal{P}$ marks a request committed, it initiates the communication between the shim and the serverless cloud. Specifically, $\mathcal{P}$ invokes $\mathbf{n}_{\mathcal{E}}$ executors at the cloud and sends each of them the Execute message for processing.

The Execute message includes the client request $\langle T \rangle_c$ and a *certificate.* This certificate $\mathfrak{X}$ includes signatures of $\mathbf{nf}_{\mathfrak{R}}$ distinct nodes and proves that a majority of shim nodes agreed to order this request. Prior to executing the transaction T, each executor $\text{E} \in \mathcal{E}$ checks if the certificate $\mathfrak{X}$ is valid. If this is the case, then E attempts to execute T. During execution, if E requires access to some data, it connects with the storage $\mathcal{S}$ and fetches the required data.

Although $\mathcal{P}$ invokes $\mathbf{n}_{\mathcal{E}}$ executors at the cloud, at no point during the execution, we require these executors to interact with each other. Hence, each executor works independently of other executors. Post execution, each executor E sends a Verify message to the verifier $\mathcal{V}$, which includes the result r of executing the transaction T.

Verifier and Storage. When the verifier $\mathcal{V}$ receives well-formed and identical Verify messages from $\mathbf{nf}_{\mathcal{E}}$ executors, then it has a guarantee that a majority of executors reached on the result r. Hence, it sends a Response message to the client c. Next, $\mathcal{V}$ communicates with the storage $\mathcal{S}$ and forwards the updates corresponding to the result r.

```
    Client-role (used by client $c$ to request transaction $T$) :
 1: Sends $\langle T \rangle_c$ to the primary $\mathcal{P}$.
 2: Awaits receipt of message $\textsc{Response}(\langle T \rangle_c, k, r)$ from $\mathcal{V}$.
 3: Considers $T$ executed, with result $r$, as the $k$-th transaction.

    Primary-role (running at the primary node $\mathcal{P}$) :
 4: event $\mathcal{P}$ receives $\langle T \rangle_c$ do
 5:    Calculate digest $\Delta := \text{digest}(\langle T \rangle_c)$.
 6:    Broadcast $\textsc{Preprepare}(\langle T \rangle_c, \Delta, k)$ to all nodes (order at sequence $k$).

 7: event $\mathcal{P}$ receives $m := \langle \textsc{Commit}(\Delta, k) \rangle_R$ messages from $\mathbf{nf}_{\mathfrak{R}}$ nodes such that:
            (1) each message $m$ is well-formed and is sent by a distinct node $R \in \mathfrak{R}$.
    do
 8:    $\mathfrak{X}$ := set of DS of these $\mathbf{nf}_{\mathfrak{R}}$ messages.
 9:    Send $\langle \textsc{Execute}(\langle T \rangle_c, \mathfrak{X}, m, \Delta) \rangle_{\mathcal{P}}$ to all executors $\textsc{e} \in \mathcal{E}$,

    Non-Primary role (running at a node $R \in \mathfrak{R}$) :
10: event $R$ receives $\textsc{Preprepare}(\langle T \rangle_c, \Delta, k)$ from $\mathcal{P}$ such that:
            (1) message is well-formed, and $R$ did not accept a $k$-th proposal from $\mathcal{P}$.
    do
11:    Broadcast $\textsc{Prepare}(\Delta, k)$ to all nodes in $\mathfrak{R}$.

    All nodes role (running at the node $R$) :
12: event $R$ receives $\textsc{Prepare}(\Delta, k)$ messages from $\mathbf{nf}_{\mathfrak{R}}$ nodes such that:
            (1) each message is well-formed and is sent by a distinct node, $R* \in \mathfrak{R}$.
    do
13:    Broadcast $\langle \textsc{Commit}(\Delta, k) \rangle_R$ to all nodes in $\mathfrak{R}$.

    Executor-role (running at the executor $\textsc{e} \in \mathcal{E}$) :
14: event $\textsc{e}$ receives $\langle \textsc{Execute}(\langle T \rangle_c, \mathfrak{X}, m, \Delta) \rangle_{\mathcal{P}}$ from $\mathcal{P}$ such that:
            (1) message is well-formed,
            (2) $m := \textsc{Commit}(\Delta, k)$, and
            (3) Certificate $\mathfrak{X}$ includes $\mathbf{nf}_R$ distinct DS on $m$.
    do
15:    while $T$ not executed do
16:       if Need to read some data-items then
17:          Fetch required data-items from storage $\mathcal{S}$
18:    $r$ := Result of executing $T$
19:    Send $\textsc{Verify}(\langle T \rangle_c, \mathfrak{X}, m, \Delta, r)$ to verifier $\mathcal{V}$.

    Verifier-role (running at the verifier $\mathcal{V}$) :
20: event $\mathcal{V}$ receives $m' := \textsc{Verify}(\langle T \rangle_c, A, m, \Delta, r)$ messages from $\mathbf{nf}_{\mathcal{E}}$ executors such that:
            (1) each message $m'$ is well-formed and is sent by a distinct executor $\textsc{e} \in \mathcal{E}$.
            (2) all have matching result of execution $r$.
    do
21:    Send $\langle \textsc{Response}(\Delta, r) \rangle_{\mathcal{V}}$ to the client $c$.
22:    Send writes to the storage $\mathcal{S}$.
```

FIGURE 6.8. Byzantine Fault-Tolerant transaction processing by SERVERLESSBFT protocol in the serverless-edge architecture.

6.2.5. System Guarantees. Having described the flow of our ServerlessBFT protocol, we now state the guarantees offered by our infrastructure. In specific, serverless-edge infrastructure satisfies the following properties:

Shim Consistency.: If a non-faulty node commits a transaction T, then all non-faulty nodes commit T.

Shim Non-Divergence.: If two non-faulty nodes order a transaction T at sequence number k and k', then $k = k'$.

Shim Termination.: If a non-faulty client sends a transaction T, then a non-faulty node will commit T.

Executor Consistency.: If a non-faulty executor executes a transaction T, then all non-faulty executors execute T.

Executor Termination.: If a non-faulty primary sends a transaction T, then a non-faulty executor will execute T.

Verifier Non-Divergence.: If the shim commits a transaction T at sequence k, then the verifier will update the corresponding result at the storage at order k.

Shim consistency, shim non-divergence, executor consistency, and verifier non-divergence together guarantee *safety*, while shim termination and executor termination guarantee *liveness.*

ServerlessBFT framework guarantees safety in an asynchronous environment where the messages can get lost, delayed, or duplicated, and byzantine replicas can collude or act arbitrarily. Further, our ServerlessBFT framework guarantees liveness only in the periods of synchrony.

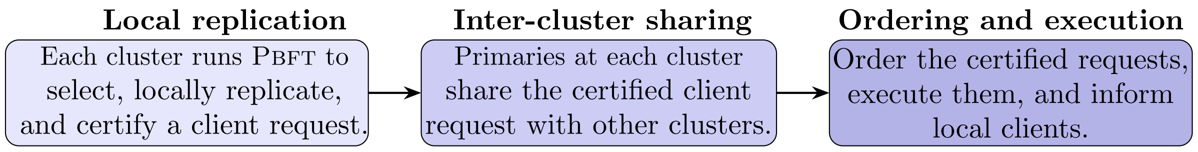

Local replication
Each cluster runs PBFT to select, locally replicate, and certify a client request.
Inter-cluster sharing
Primaries at each cluster share the certified client request with other clusters.
Ordering and execution
Order the certified requests, execute them, and inform local clients.

FIGURE 6.9. Steps in a round of the GEOBFT protocol.

6.3. Geo-Scale Consensus

To enable geo-scale deployment of a permissioned blockchain system, we believe that the underlying consensus protocol must distinguish between *local* and *global* communication. To resolve this challenge, we vision the design of a *Geo-Scale Byzantine Fault-Tolerant* consensus protocol (GEOBFT) that uses topological information to group all replicas in a single region into a single cluster. Likewise, GEOBFT assigns each client to a single cluster. This clustering helps in attaining high throughput and scalability in geo-scale deployments. GEOBFT operates in rounds, and in each round, every cluster will be able to propose a single client request for execution. Each round consists of the three steps sketched in Figure 6.9: *local replication*, *global sharing*, and *ordering and execution*, which we further detail next.

At the start of each round, each cluster chooses a single transaction of a local client. Next, each cluster *locally replicates* its chosen transaction in a Byzantine fault-tolerant manner using PBFT. At the end of successful local replication, PBFT guarantees that each non-faulty replica can prove successful local replication via a *commit certificate*.

Next, each cluster shares the locally-replicated transaction along with its commit certificate with all other clusters. To minimize inter-cluster communication, we use a novel *optimistic global sharing protocol*. Our optimistic global sharing protocol has a global phase in which clusters exchange locally-replicated transactions, followed by a local phase in which clusters distribute any received transactions locally among all local replicas. Finally, after receiving all transactions that are locally-replicated in other clusters, each replica in each cluster can deterministically *order* all these transactions and proceed with their *execution*. After execution, the replicas in each cluster inform only local clients of the outcome of the execution of their transactions (e.g., confirm execution or return any execution results).

CHAPTER 7

Conclusions

In this work, we presented protocols and design methodologies that can help scale large-scale distributed database systems. These large-scale distributed applications often face crash-failures and byzantine attacks. As a result, there has been a growing interest in designing resilient distributed applications, which is also the principle philosophy behind blockchain technology. The key focus of this work is to scale permissioned blockchain fabrics and to ensure they remain consistent.

The first work of this thesis focusses on scaling commitment protocols, which assume a distributed database is split across partitions. An incoming client request may require access to data from one or more partitions. As a result, the partitions need to agree on whether to commit or abort the transaction. Traditionally, distributed systems have employed the two-phase commit (2PC) protocol to reach such an agreement. However, prior works have illustrated that the 2PC protocol is blocking. This led to the design of the three-phase commit (3PC) protocol, which requires an additional communication phase to reach the agreement. This additional phase makes the 3PC protocol expensive and unsuitable for industrial needs. In this thesis, we resolved this problem by designing the EasyCommit protocol which leverages the best of both worlds (2PC and 3PC), that is, EC is non-blocking (like 3PC) and requires two phases (like 2PC). EasyCommit achieves these goals by ensuring two key observations: (i) first transmit and then commit, and (ii) message redundancy. Our evaluation of EC on our ResilientDB framework illustrate that EC outperforms the 3PC protocol and scales nearly as good as the 2PC protocol. On further analysis, we realized that the protocols like EC and 2PC cannot cater to the needs of geographically large scale distributed systems as the partitions could be spread across geographically distant locations. As a result, we also designed a topology-aware agreement protocol Geo-scale EasyCommit (GEC), which is non-blocking, safe, live, and outperforms all the above protocol.

Although commit protocols like EC and GEC provide good scalability, they can only handle node crash-failures. However, distributed applications can face byzantine attacks and network could be unreliable. Hence, several traditional systems employ replication to guarantee security against these attacks. Prior works have employed byzantine fault-tolerant (BFT) consensus protocols to achieve consensus among the replicas. Our *second* work aims at designing a *byzantine fault-tolerant consensus* protocol that is both efficient and secure. Existing BFT algorithms face following challenges: (i) they are communication expensive (require three phases of quadratic complexity), (ii) require a large number of replicas, (iii) depend on clients, and (iv) need trusted components.

To resolve this challenge, we presented the design of our Proof-of-Execution (PoE) consensus protocol. At the core of PoE are *out-of-order* processing and *speculative execution*, which allow PoE to execute transactions before consensus is reached among the replicas. With these techniques, PoE manages to reduce the costs of BFT in normal cases, while guaranteeing reliable consensus for clients in all cases. We envision the use of PoE in high-throughput multi-party data-management and blockchain systems.

BFT protocols like PoE and PBFT follow the primary-backup model where one replica is designated as the primary and others act as backups. Primary manages all the incoming client requests and is responsible for initiating consensus on each request. Evidently, this primary also limits the scalability of the system. To resolve this challenge, we present our RCC paradigm that takes as input a BFT protocol and parallelizes its consensus. RCC does so by requiring each replica to concurrently run multiple instances of the input BFT protocol. Further, RCC ensures that the failure of one instance does not affect the functioning of other instances. Our evaluation of RCC illustrates that it outperforms all the existing primary-backup protocols and is more resilient to byzantine attacks than the other concurrent consensus protocols

Finally, to evaluate our scalable BFT protocols, we designed ResilientDB, a high-throughput yielding permissioned blockchain fabric. ResilientDB builds on top of a key intuition, *can a well-crafted system based on a classical* BFT *protocol outperform a modern protocol?* Our ResilientDB fabric proves that designing such a well-crafted system is possible and even if such a system employs a three-phase protocol, it can outperform another systems utilizing a single-phase protocol. This endeavor requires us to dissect existing permissioned blockchain systems and highlight different factors affecting their performance. ResilientDB fabric is based on these insights, employs multi-threaded deep pipelines to balance tasks at replicas, and provides guidelines for future designs.

References

[1] M. Abd-El-Malek, G. R. Ganger, G. R. Goodson, M. K. Reiter, and J. J. Wylie. Fault-scalable byzantine fault-tolerant services. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, pages 59–74. ACM, 2005. doi: 10.1145/1095810.1095817.

[2] M. Abdallah, R. Guerraoui, and P. Pucheral. One-Phase Commit: Does it make Sense? ICPADS, 1998. ISBN 0-8186-8603-0.

[3] I. Abraham, G. Gueta, D. Malkhi, L. Alvisi, R. Kotla, and J.-P. Martin. Revisiting fast practical byzantine fault tolerance, 2017. URL `https://arxiv.org/abs/1712.01367`.

[4] D. Agrawal, A. El Abbadi, H. A. Mahmoud, F. Nawab, and K. Salem. Managing geo-replicated data in multi-datacenters. In *Proceedings of the 2013 Databases in Networked Information Systems - 8th International Workshop*, DNIS'13, pages 23–43, 2013.

[5] Y. Amir, B. Coan, J. Kirsch, and J. Lane. Prime: Byzantine replication under attack. *IEEE Trans. Depend. Secure Comput.*, 8(4):564–577, 2011. doi: 10.1109/TDSC.2010.70.

[6] M. J. Amiri, D. Agrawal, and A. E. Abbadi. CAPER: A cross-application permissioned blockchain. *Proc. VLDB Endow.*, 12(11):1385–1398, 2019. ISSN 2150-8097. doi: 10.14778/3342263.3342275.

[7] M. J. Amiri, D. Agrawal, and A. El Abbadi. SharPer: Sharding permissioned blockchains over network clusters, 2019. URL `https://arxiv.org/abs/1910.00765v1`.

[8] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, S. Muralidharan, C. Murthy, B. Nguyen, M. Sethi, G. Singh, K. Smith, A. Sorniotti, C. Stathakopoulou, M. Vukolić, S. W. Cocco, and J. Yellick. Hyperledger Fabric: A distributed operating system for permissioned blockchains. In *Proceedings of the Thirteenth EuroSys Conference*, pages 30:1–30:15. ACM, 2018. doi: 10.1145/3190508.3190538.

[9] G. Association. Blockchain for development: Emerging opportunities for mobile, identity and aid, 2017. URL `https://www.gsma.com/mobilefordevelopment/wp-content/uploads/2017/12/Blockchain-for-Development.pdf`.

[10] P.-L. Aublin, S. B. Mokhtar, and V. Quéma. RBFT: Redundant byzantine fault tolerance. In *Proceedings of the 2013 IEEE 33rd International Conference on Distributed Computing Systems*, pages 297–306. IEEE, 2013. doi: 10.1109/ICDCS.2013.53.

[11] J. Baker, C. Bond, J. C. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing Scalable, Highly Available Storage for Interactive Services. In *Proceedings of the Conference on Innovative Data system Research (CIDR)*, pages 223–234, 2011.

[12] I. Bentov, P. Hubáček, T. Moran, and A. Nadler. Tortoise and hares consensus: the meshcash framework for incentive-compatible, scalable cryptocurrencies, 2017. URL `https://eprint.iacr.org/2017/300`.

[13] P. A. Bernstein and N. Goodman. Concurrency Control in Distributed Database Systems. *ACM Comput. Surv.*, 13(2):185–221, 1981. ISSN 0360-0300.

[14] P. A. Bernstein and N. Goodman. Multiversion Concurrency Control - Theory and Algorithms. *ACM TODS*, 8(4):465–483, 1983. ISSN 0362-5915.

[15] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley Longman Publishing Co., 1987. ISBN 0-201-10715-5.

[16] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1987. ISBN 0-201-10715-5.

[17] A. Bessani, J. Sousa, and E. E. Alchieri. State machine replication for the masses with BFT-SMART. In *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 355–362. IEEE, 2014. doi: 10.1109/DSN.2014.43.

[18] B. Blechschmidt. Blockchain in Europe: Closing the strategy gap. Technical report, Cognizant Consulting, 2018. URL `https://www.cognizant.com/whitepapers/blockchain-in-europe-closing-the-strategy-gap-codex3320.pdf`.

[19] E. Buchman, J. Kwon, and Z. Milosevic. The latest gossip on BFT consensus. *CoRR*, abs/1807.04938, 2018.

[20] M. Casey, J. Crane, G. Gensler, S. Johnson, and N. Narula. The impact of blockchain technology on finance: A catalyst for change. Technical report, International Center for Monetary and Banking Studies, 2018. URL `https://www.cimb.ch/uploads/1/1/5/4/115414161/geneva21_1.pdf`.

[21] M. Castro and B. Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst.*, 20(4):398–461, 2002. doi: 10.1145/571637.571640.

[22] K. Chen, Y. Zhou, and Y. Cao. Online Data Partitioning in Distributed Database Systems. In *Proceedings of the 18th International Conference on Extending Database Technology*, pages 1–12. OpenProceeding.org, 2015. ISBN 978-3-89318-067-7.

[23] Christie's. Major collection of the fall auction season to be recorded with blockchain technology, 2018. URL `https://www.christies.com/presscenter/pdf/9160/RELEASE_ChristiesxArtoryxEbsworth_9160_1.pdf`.

[24] B.-G. Chun, P. Maniatis, S. Shenker, and J. Kubiatowicz. Attested append-only memory: Making adversaries stick to their word. *SIGOPS Oper. Syst. Rev.*, 41(6):189–204, 2007. doi: 10.1145/1323293.1294280.

[25] A. Clement, M. Kapritsos, S. Lee, Y. Wang, L. Alvisi, M. Dahlin, and T. Riche. Upright cluster services. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, pages 277–290. ACM, 2009. doi: 10.1145/1629575.1629602.

[26] A. Clement, E. Wong, L. Alvisi, M. Dahlin, and M. Marchetti. Making byzantine fault tolerant systems tolerate byzantine faults. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, pages 153–168. USENIX, 2009.

[27] A. Clement, E. Wong, L. Alvisi, M. Dahlin, and M. Marchetti. Making byzantine fault tolerant systems tolerate byzantine faults. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, pages 153–168. USENIX Association, 2009.

[28] A. Clement, F. Junqueira, A. Kate, and R. Rodrigues. On the (Limited) Power of Non-Equivocation. In *Proceedings of the 2012 ACM Symposium on Principles of Distributed Computing*, page 301308. Association for Computing Machinery, 2012. ISBN 9781450314503.

doi: 10.1145/2332432.2332490.

[29] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, pages 143–154. ACM, 2010. doi: 10.1145/1807128.1807152.

[30] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's Globally-Distributed Database. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 261–264. USENIX Association, 2012. ISBN 978-1-931971-96-6.

[31] V. Costan and S. Devadas. Intel sgx explained. Cryptology ePrint Archive, Report 2016/086, 2016. `https://ia.cr/2016/086`.

[32] V. Costan, I. Lebedev, and S. Devadas. Sanctum: Minimal Hardware Extensions for Strong Software Isolation. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 857–874, Austin, TX, Aug. 2016. USENIX Association.

[33] T. P. P. Council. Tpc benchmark c (revision 5.11). 2010.

[34] J. Cowling, D. Myers, B. Liskov, R. Rodrigues, and L. Shrira. HQ replication: A hybrid quorum protocol for byzantine fault tolerance. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, pages 177–190. USENIX, 2006.

[35] H. Dang, T. T. A. Dinh, D. Loghin, E.-C. Chang, Q. Lin, and B. C. Ooi. Towards scaling blockchain systems via sharding. In *Proceedings of the 2019 International Conference on Management of Data*, pages 123–140. ACM, 2019. doi: 10.1145/3299869.3319889.

[36] S. Developers. Sqlite home page, 2019. URL `https://sqlite.org/`.

[37] C. Diaconu, C. Freedman, E. Ismert, P.-A. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling. Hekaton: SQL Server's Memory-optimized OLTP Engine. pages 1243–1254. ACM, 2013. ISBN 978-1-4503-2037-5.

[38] T. T. A. Dinh, J. Wang, G. Chen, R. Liu, B. C. Ooi, and K.-L. Tan. BLOCKBENCH: A framework for analyzing private blockchains. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1085–1100. ACM, 2017. doi: 10.1145/3035918.

3064033.

[39] W. W. Eckerson. Data quality and the bottom line: Achieving business success through a commitment to high quality data. Technical report, The Data Warehousing Institute, 101communications LLC., 2002.

[40] M. Eischer and T. Distler. Scalable byzantine fault-tolerant state-machine replication on heterogeneous servers. *Computing*, 101:97–118, 2019. doi: 10.1007/s00607-018-0652-3.

[41] M. El-Hindi, C. Binnig, A. Arasu, D. Kossmann, and R. Ramamurthy. BlockchainDB: A shared database on blockchains. *Proc. VLDB Endow.*, 12(11):1597–1609, 2019. doi: 10.14778/3342263.3342636.

[42] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985. doi: 10.1145/3149.214121.

[43] B. Fung. The embarrassing reason behind Amazons huge cloud computing outage this week. GA, USA, 2017. The Washington Post.

[44] D. Gawlick and D. Kinkade. Varieties of concurrency control in ims/vs fast path. 8:3–10, 01 1985.

[45] L. Ge, C. Brewster, J. Spek, A. Smeenk, and J. Top. Blockchain for agriculture and food: Findings from the pilot study. Technical report, Wageningen University, 2017. URL `https://www.wur.nl/nl/Publicatie-details.htm?publicationId=publication-way-353330323634`.

[46] GideonGreenspan. MultiChain private blockchain–white paper, 2015. URL `https://www.multichain.com/download/MultiChain-White-Paper.pdf`.

[47] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, 2002. doi: 10.1145/564585.564601.

[48] G. Golan Gueta, I. Abraham, S. Grossman, D. Malkhi, B. Pinkas, M. Reiter, D.-A. Seredinschi, O. Tamir, and A. Tomescu. SBFT: A scalable and decentralized trust infrastructure. In *49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 568–580. IEEE, 2019. doi: 10.1109/DSN.2019.00063.

[49] W. J. Gordon and C. Catalini. Blockchain technology for healthcare: Facilitating the transition to patient-driven interoperability. *Computational and Structural Biotechnology Journal*, 16:224–230, 2018. doi: 10.1016/j.csbj.2018.06.003.

[50] J. Gray. Notes on data base operating systems. In *Operating Systems, An Advanced Course*, pages 393–481. Springer-Verlag, 1978. doi: 10.1007/3-540-08755-9_9.

[51] J. Gray. The Transaction Concept: Virtues and Limitations (Invited Paper). VLDB, pages 144–154, 1981.

[52] J. Gray. *A comparison of the Byzantine Agreement problem and the Transaction Commit Problem*, pages 10–17. Springer New York, 1990. ISBN 978-0-387-34812-4.

[53] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., 1st edition, 1992. ISBN 1558601902.

[54] R. Guerraoui. *Revisiting the relationship between non-blocking atomic commitment and consensus*, pages 87–100. Springer Berlin Heidelberg, 1995. ISBN 978-3-540-44783-2.

[55] S. Gupta. Resilient and scalable architecture for permissioned blockchain fabrics. In Z. Abedjan and K. Hose, editors, *Proceedings of the VLDB 2020 PhD Workshop co-located with the 46th International Conference on Very Large Databases), August 31 - September 4, 2020*, volume 2652 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2020.

[56] S. Gupta and M. Sadoghi. EasyCommit: A non-blocking two-phase commit protocol. In *Proceedings of the 21st International Conference on Extending Database Technology*, pages 157–168. Open Proceedings, 2018. doi: 10.5441/002/edbt.2018.15.

[57] S. Gupta and M. Sadoghi. *Blockchain Transaction Processing*, pages 1–11. Springer International Publishing, 2018. doi: 10.1007/978-3-319-63962-8_333-1.

[58] S. Gupta and M. Sadoghi. Efficient and non-blocking agreement protocols. *Distributed Parallel Databases*, 38(2):287–333, 2020. doi: 10.1007/s10619-019-07267-w.

[59] S. Gupta, J. Hellings, S. Rahnama, and M. Sadoghi. An in-depth look of BFT consensus in blockchain: Challenges and opportunities. In *Proceedings of the 20th International Middleware Conference Tutorials, Middleware*, pages 6–10. ACM, 2019. doi: 10.1145/3366625.3369437.

[60] S. Gupta, J. Hellings, and M. Sadoghi. Brief announcement: Revisiting consensus protocols through wait-free parallelization. In *33rd International Symposium on Distributed Computing (DISC 2019)*, volume 146, pages 44:1–44:3. Schloss Dagstuhl, 2019. doi: 10.4230/LIPIcs.DISC.2019.44.

[61] S. Gupta, J. Hellings, S. Rahnama, and M. Sadoghi. Blockchain consensus unraveled: virtues and limitations. In *Proceedings of the 14th ACM International Conference on Distributed and Event-based Systems*, pages 218–221. ACM, 2020. doi: 10.1145/3401025.3404099.

[62] S. Gupta, J. Hellings, S. Rahnama, and M. Sadoghi. Building high throughput permissioned blockchain fabrics: Challenges and opportunities. *Proc. VLDB Endow.*, 13(12):3441–3444, 2020.

[63] S. Gupta, S. Rahnama, J. Hellings, and M. Sadoghi. ResilientDB: Global scale resilient blockchain fabric. *Proc. VLDB Endow.*, 13(6):868–883, 2020. doi: 10.14778/3380750.3380757.

[64] S. Gupta, S. Rahnama, and M. Sadoghi. Permissioned blockchain through the looking glass: Architectural and implementation lessons learned. In *40th International Conference on Distributed Computing Systems*. IEEE, 2020.

[65] S. Gupta, J. Hellings, S. Rahnama, and M. Sadoghi. Proof-of-Execution: Reaching Consensus through Fault-Tolerant Speculation. In *Proceedings of the 24th International Conference on Extending Database Technology*, pages 301–312. OpenProceedings.org, 2021. doi: 10.5441/002/edbt.2021.27.

[66] S. Gupta, J. Hellings, and M. Sadoghi. *Fault-Tolerant Distributed Transactions on Blockchain.* Synthesis Lectures on Data Management. Morgan & Claypool, 2021. doi: 10.2200/S01068ED1V01Y202012DTM065.

[67] S. Gupta, J. Hellings, and M. Sadoghi. RCC: Resilient Concurrent Consensus for High-Throughput Secure Transaction Processing. In *37th IEEE International Conference on Data Engineering*, pages 1392–1403. IEEE, 2021. doi: 10.1109/ICDE51399.2021.00124.

[68] R. Harding, D. Van Aken, A. Pavlo, and M. Stonebraker. An Evaluation of Distributed Concurrency Control. *Proc. VLDB Endow.*, 10(5):553–564, 2017. ISSN 2150-8097.

[69] J. R. Haritsa, K. Ramamritham, and R. Gupta. The PROMPT Real-Time Commit Protocol. *IEEE TPDS*, 11(2):160–181, 2000. ISSN 1045-9219.

[70] J. Hellings and M. Sadoghi. Coordination-free byzantine replication with minimal communication costs. In *23rd International Conference on Database Theory (ICDT 2020)*, volume 155, pages 17:1–17:20. Schloss Dagstuhl, 2020. doi: 10.4230/LIPIcs.ICDT.2020.17.

[71] J. Hellings, D. P. Hughes, J. Primero, and M. Sadoghi. Cerberus: Minimalistic multi-shard byzantine-resilient transaction processing, 2020. URL `https://arxiv.org/abs/2008.04450`.

[72] J. L. Hennessy and D. A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach.* Morgan Kaufmann Publishers Inc., 5th edition, 2011. ISBN 012383872X, 9780123838728.

[73] M. Herlihy. Blockchains from a distributed computing perspective. *Commun. ACM*, 62(2): 78–85, 2019. doi: 10.1145/3209623.

[74] M. P. Herlihy and J. M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM TOPLAS*, 12(3):463–492, 1990. ISSN 0164-0925. doi: 10.1145/78969.78972.

[75] T. N. Herzog, F. J. Scheuren, and W. E. Winkler. *Data Quality and Record Linkage Techniques.* Springer, 2007. doi: 10.1007/0-387-69505-2.

[76] Z. István, A. Sorniotti, and M. Vukolić. Streamchain: Do blockchains need blocks? SERIAL'18, pages 1–6. ACM, 2018.

[77] M. Jakobsson and A. Juels. Proofs of work and bread pudding protocols. In *Secure Information Networks: Communications and Multimedia Security IFIP TC6/TC11 Joint Working Conference on Communications and Multimedia Security (CMS'99)*, pages 258–272. Springer, 1999. doi: 10.1007/978-0-387-35568-9_18.

[78] R. Jiménez-Peris, M. Patiño Martínez, G. Alonso, and S. Arévalo. A Low-Latency Nonblocking Commit Service. DISC'01. Springer-Verlag, 2001. ISBN 3-540-42605-1.

[79] E. Jonas, J. Schleier-Smith, V. Sreekanti, C.-C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Carreira, K. Krauth, N. Yadwadkar, J. E. Gonzalez, R. A. Popa, I. Stoica, and D. A. Patterson. Cloud Programming Simplified: A Berkeley View on Serverless Computing, 2019.

[80] F. P. Junqueira, B. C. Reed, and M. Serafini. Zab: High-performance broadcast for primary-backup systems. In *Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems&Networks*, pages 245–256. IEEE, 2011. doi: 10.1109/DSN.2011.5958223.

[81] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. B. Zdonik, E. P. C. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. J. Abadi. H-store: A High-Performance, Distributed Main Memory Transaction Processing System. *PVLDB*, 1:1496–1499, 2008.

[82] M. N. Kamel Boulos, J. T. Wilson, and K. A. Clauson. Geospatial blockchain: promises, challenges, and scenarios in health and healthcare. *International Journal of Health Geographics*, 17(1):1211–1220, 2018. doi: 10.1186/s12942-018-0144-x.

[83] R. Kapitza, J. Behl, C. Cachin, T. Distler, S. Kuhnle, S. V. Mohammadi, W. Schröder-Preikschat, and K. Stengel. CheapBFT: Resource-Efficient Byzantine Fault Tolerance. In *Proceedings of the 7th ACM European Conference on Computer Systems*, page 295308. Association for Computing Machinery, 2012. doi: 10.1145/2168836.2168866.

[84] M. Kapritsos, Y. Wang, V. Quema, A. Clement, L. Alvisi, and M. Dahlin. All about eve: Execute-verify replication for multi-core servers. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, pages 237–250. USENIX, 2012.

[85] J. Katz and Y. Lindell. *Introduction to Modern Cryptography.* Chapman and Hall/CRC, 2nd edition, 2014.

[86] S. King and S. Nadal. PPCoin: Peer-to-peer crypto-currency with Proof-of-Stake, 2012. URL `https://www.peercoin.net/whitepapers/peercoin-paper.pdf`.

[87] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: Speculative byzantine fault tolerance. *ACM Trans. Comput. Syst.*, 27(4):7:1–7:39, 2009. doi: 10.1145/1658357.1658358.

[88] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: Speculative byzantine fault tolerance. *ACM Trans. Comput. Syst.*, 27(4):7:1–7:39, 2009. doi: 10.1145/1658357.1658358.

[89] L. Lamport. Paxos made simple. *ACM SIGACT News*, 32(4):51–58, 2001. doi: 10.1145/568425.568433. Distributed Computing Column 5.

[90] L. Lao, Z. Li, S. Hou, B. Xiao, S. Guo, and Y. Yang. A survey of IoT applications in blockchain systems: Architecture, consensus, and traffic modeling. *ACM Comput. Surv.*, 53 (1), 2020. doi: 10.1145/3372136.

[91] D. Lee, D. Kohlbrenner, S. Shinde, K. Asanović, and D. Song. Keystone: An Open Framework for Architecting Trusted Execution Environments. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery. doi: 10.1145/3342195.3387532.

[92] D. Levin, J. R. Douceur, J. R. Lorch, and T. Moscibroda. TrInc: Small Trusted Hardware for Large Distributed Systems. In *6th USENIX Symposium on Networked Systems Design and Implementation*. USENIX Association, Apr. 2009.

[93] E. Levy, H. F. Korth, and A. Silberschatz. An Optimistic Commit Protocol for Distributed Transaction Management. ACM SIGMOD, pages 88–97. ACM, 1991. ISBN 0-89791-425-2.

[94] B. Li, W. Xu, M. Z. Abid, T. Distler, and R. Kapitza. SAREK: Optimistic parallel ordering in byzantine fault tolerance. In *2016 12th European Dependable Computing Conference (EDCC)*, pages 77–88. IEEE, 2016. doi: 10.1109/EDCC.2016.36.

[95] C. Li, P. Li, D. Zhou, W. Xu, F. Long, and A. Yao. Scaling nakamoto consensus to thousands of transactions per second, 2018. URL `https://arxiv.org/abs/1805.03870`.

[96] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Stronger Semantics for Low-latency Geo-replicated Storage. NSDI, pages 313–328. USENIX Association, 2013.

[97] G. McGrath and P. R. Brenner. Serverless Computing: Design, Implementation, and Performance. In *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*, pages 405–410, 2017. doi: 10.1109/ICDCSW.2017.36.

[98] MemSQL. http://www.memsql.com. 2013.

[99] C. Mohan, B. Lindsay, and R. Obermarck. Transaction Management in the R* Distributed Database Management System. *ACM TODS*, 11(4), 1986.

[100] G. E. Moore. Cramming more components onto integrated circuits, Reprinted from Electronics, volume 38, number 8, April 19, 1965, pp.114 ff. *IEEE Solid-State Circuits Society Newsletter*, 11(3):33–35, Sep. 2006. doi: 10.1109/N-SSC.2006.4785860.

[101] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2009. URL `https://bitcoin.org/bitcoin.pdf`.

[102] F. Nawab and M. Sadoghi. Blockplane: A global-scale byzantizing middleware. In *35th International Conference on Data Engineering (ICDE)*, pages 124–135. IEEE, 2019. doi:

10.1109/ICDE.2019.00020.

[103] F. Nawab, V. Arora, D. Agrawal, and A. El Abbadi. Minimizing Commit Latency of Transactions in Geo-Replicated Data Stores. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1279–1294. ACM, 2015. ISBN 978-1-4503-2758-9.

[104] NuoDB. http:://www.nuodb.com. 2010.

[105] S. A. O'Brien. Facebook, Instagram experience outages Saturday. GA, USA, 2017. CNN.

[106] T. C. of Economic Advisers. The cost of malicious cyber activity to the U.S. economy. Technical report, Executive Office of the President of the United States, 2018. URL `https://www.whitehouse.gov/wp-content/uploads/2018/03/The-Cost-of-Malicious-Cyber-Activity-to-the-U.S.-Economy.pdf`.

[107] D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, pages 305–320. USENIX, 2014.

[108] C. Oracle. Oracle 9i Real Application Clusters concepts Release 2 (9.2), Part Number A96597-01. 04 2002.

[109] M. T. Özsu and P. Valduriez. *Principles of Distributed Database Systems*. Springer, 2020. doi: 10.1007/978-3-030-26253-2.

[110] T. Park and H. Y. Yeom. A distributed group commit protocol for distributed database systems. ICPADS, 1991. ISBN 1880843293.

[111] M. Pisa and M. Juden. Blockchain and economic development: Hype vs. reality. Technical report, Center for Global Development, 2017. URL `https://www.cgdev.org/publication/blockchain-and-economic-development-hype-vs-reality`.

[112] PwC. Blockchain – an opportunity for energy producers and consumers?, 2016. URL `https://www.pwc.com/gx/en/industries/energy-utilities-resources/publications/opportunity-for-energy-producers.html`.

[113] T. Qadah, S. Gupta, and M. Sadoghi. Q-Store: Distributed, Multi-partition Transactions via Queue-oriented Execution and Communication. In *Proceedings of the 23rd International Conference on Extending Database Technology*, pages 73–84. OpenProceedings.org, 2020. doi:

10.5441/002/edbt.2020.08.

[114] T. M. Qadah and M. Sadoghi. QueCC: A queue-oriented, control-free concurrency architecture. Middleware, 2018.

[115] S. Rahnama, S. Gupta, T. Qadah, J. Hellings, and M. Sadoghi. Scalable, resilient and configurable permissioned blockchain fabric. *Proc. VLDB Endow.*, 13(12):2893–2896, 2020.

[116] S. Rahnama, S. Gupta, R. Sogani, D. Krishnan, and M. Sadoghi. RingBFT: Resilient Consensus over Sharded Ring Topology. *CoRR*, abs/2107.13047, 2021.

[117] P. K. Reddy and M. Kitsuregawa. Reducing the Blocking in Two-Phase Commit Protocol Employing Backup Sites. COOPIS'98, pages 406–416. IEEE, 1998. ISBN 0-8186-8380-5.

[118] T. C. Redman. The impact of poor data quality on the typical enterprise. *Commun. ACM*, 41(2):79–82, 1998. doi: 10.1145/269012.269025.

[119] B. S. Boutros and B. C. Desai. A two-phase commit protocol and its performance. DEXA, pages 100–105. IEEE, 1996. ISBN 0-8186-7662-0.

[120] M. Sadoghi and S. Blanas. *Transaction Processing on Modern Hardware.* Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2019. doi: 10.2200/S00896ED1V01Y201901DTM058.

[121] M. Sadoghi, S. Bhattacherjee, B. Bhattacharjee, and M. Canim. L-Store: A Real-time OLTP and OLAP System. EDBT. OpenProceeding.org, 2018.

[122] G. Samaras, K. Britton, A. Citron, and C. Mohan. Two-phase commit optimizations in a commercial distributed environment. *Distributed and Parallel Databases*, 3(4):325–360, 1995. ISSN 1573-7578.

[123] J. Shute, R. Vingralek, B. Samwel, B. Handy, C. Whipkey, E. Rollins, M. Oancea, K. Littleeld, D. Menestrina, S. Ellner, J. T. Stancescu, and H. Apte. F1: A distributed sql database that scales. In *VLDB*, 2013.

[124] D. Skeen. Nonblocking Commit Protocols. SIGMOD, pages 133–142. ACM, 1981. ISBN 0-89791-040-0.

[125] D. Skeen. A quorum-based commit protocol. Technical report, Cornell University, 1982.

[126] D. Skeen and M. Stonebraker. A Formal Model of Crash Recovery in a Distributed System. *IEEE Trans. Softw. Eng.*, 9(3):219–228, 1983. ISSN 0098-5589.

[127] J. Stamos and F. Cristian. A Low-Cost Atomic Commit Protocol. In *Proceedings of the 9th Symposium on Reliable Distributed Systems*, pages 10–17. IEEE, 1990. ISBN 0-8186-2081-1.

[128] C. Stathakopoulou, T. David, and M. Vukolic. Mir-BFT: High-throughput BFT for blockchains, 2019. URL `http://arxiv.org/abs/1906.05552`.

[129] M. Stonebraker. The Case for Shared Nothing. *Database Engineering*, 9:4–9, 1986.

[130] A. Sulleyman. Twitter Down: Social Media App And Website Not Working. UK, 2017. The Independent.

[131] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi. Calvin: Fast Distributed Transactions for Partitioned Database Systems. SIGMOD, 2012. ISBN 978-1-4503-1247-9.

[132] G. S. Veronese, M. Correia, A. N. Bessani, and L. C. Lung. Spin one's wheels? byzantine fault tolerance with a spinning primary. In *2009 28th IEEE International Symposium on Reliable Distributed Systems*, pages 135–144. IEEE, 2009. doi: 10.1109/SRDS.2009.36.

[133] G. S. Veronese, M. Correia, A. N. Bessani, L. C. Lung, and P. Verissimo. Efficient byzantine fault-tolerance. *IEEE Trans. Comput.*, 62(1):16–30, 2013. doi: 10.1109/TC.2011.221.

[134] VoltDB. https://www.voltdb.com/. 2010.

[135] S. Yandamuri, I. Abraham, K. Nayak, and M. Reiter. Brief Announcement: Communication-Efficient BFT Using Small Trusted Hardware to Tolerate Minority Corruption. In *35th International Symposium on Distributed Computing (DISC 2021)*, volume 209 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 62:1–62:4, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi: 10.4230/LIPIcs.DISC.2021.62.

[136] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin. Separating Agreement from Execution for Byzantine Fault Tolerant Services. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, page 253267, 2003. doi: 10.1145/945445.945470.

[137] M. Yin, D. Malkhi, M. K. Reiter, G. G. Gueta, and I. Abraham. HotStuff: BFT consensus with linearity and responsiveness. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, pages 347–356. ACM, 2019. doi: 10.1145/3293611.3331591.

[138] M. Zamani, M. Movahedi, and M. Raykova. RapidChain: Scaling blockchain via full sharding. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 931–948. ACM, 2018. doi: 10.1145/3243734.3243853.