Scaling Permissioned Blockchain Systems

By

SAJJAD RAHNAMA

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:

M. Sadoghi

Mohammad Sadoghi Hamedani, Chair

[illegible]

Jason Lowe-Power

Aditya Thakur

Aditya Thakur

Committee in Charge

2022

ABSTRACT

With the advent of Bitcoin, the first real-world and practical blockchain system, researchers and practitioners have turned their eyes and started developing new cryptocurrencies and blockchains. There are two types of blockchains: *Permissioned* and *Permissionless*. Public or permissionless blockchains are highly decentralized so that replicas in the system can join and leave anytime, and they are computation-intensive. Permissioned or private blockchains require a set of known replicas, and they are communication-intensive. Blockchain applications have use cases in various fields such as food supply, medical records, healthcare, supply chain, trade finance, manufacturing [14, 116, 163, 85, 192].

Every blockchain system maintains an immutable ledger of transactions and works in a trustless environment to process and replicate transactions among a set of replicas. In permissioned blockchains, replicas and identities are known, and yet they could act Byzantine (behave maliciously). At the core of each blockchain system, there is a consensus protocol, which is the main focus of our work. Due to the nature of these consensus protocols (being communication-intensive), they are poorly scalable, especially when participants of this distributed system are far apart geographically.

In the first part of our work, we introduce GEOBFT [97], a global-scale resilient permissioned blockchain system. GEOBFT scales Byzantine Fault-Tolerant (BFT) protocols in the WAN environment when the system's nodes are in different continents in the world and far apart. It uses the notation of clustering by grouping replicas in different geographical locations and requires all nodes to maintain the full ledger.

Limitations of a fully replicated ledger in a WAN environment led us to step into the Sharding world. So many researchers and works have tried to use sharding to scale BFT protocols [202, 58, 187, 8]. In the second part of this work, we present RINGBFT a high throughput resilient sharding protocol. RINGBFT introduces a topology of a *Ring* among clusters of replicas to reduce the communication and prevent deadlocks.

While designing a performant BFT protocol is crucial, implementing and building a real system that uses the protocol is even more important. A poorly build fabric can mask all the advantages of a creatively designed BFT protocol. As the third part of our work, we present a permissioned blockchain fabric called ResilientDB [160]. We have fully designed and implemented a well-crafted fabric to show the inherent ability and limitations of all BFT protocols.

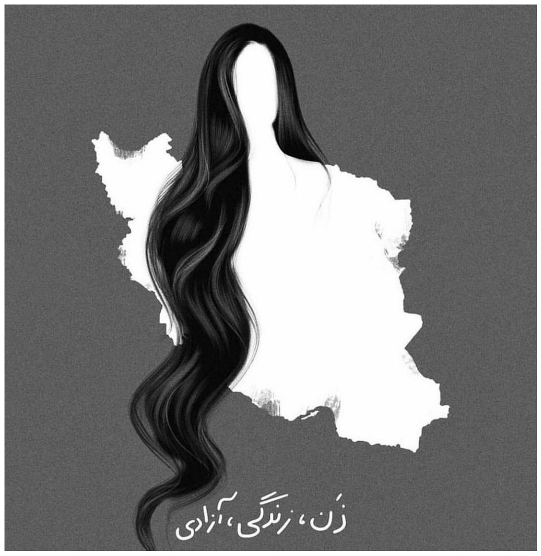
As we explored deeply in the permissioned environment, we project to step into the permissionless world as our next step, and we propose our idea for *Hybrid-Blockchains* and BFT with Trusted Components in the last chapter.

Keywords: Permissioned Blockchain, Consensus, BFT, Byzantine Fault-Tolerant, Resilient, Distributed Transaction Processing

To All Women of My Country,

To Mahsa, Nika, Sarina,

To Life and Freedom



Contents

List of Figures

List of Tables

Acknowledgments

At the end of these incredible, unimaginable four years of my life, I would like to take some time to thank those who were there for me throughout this journey. I was honored to work with several faculties, talented students, and research scholars.

I could never have imagined an incredible, supportive, kind, and understanding advisor. I would like to express my sincere gratitude to ***Dr. Mohammad Sadoghi*** for his invaluable supervision, support, and assistance at every stage of this journey during the course of my Ph.D.program. We share a lot of great memories and a few rocky roads. I always mentioned that having a good advisor makes you feel good about all the uncertainties that come with academia and should be the highest priority for people who are looking for Ph.D.programs.

Next, I would like to thank ***Dr. Jason Lowe-Power***, ***Dr. Aditya Thakur***, ***Dr. Faisal Nawab***, and ***Dr. Zubair Shafiq*** for reviewing my dissertation and their worthwhile suggestions on the candidacy exam. I am also thankful to our kind and super helpful graduate advisors at UC Davis Computer Science Jessica Stoller and Alyssa Bates for all their help and efforts.

I cannot begin to express my thanks to my dearest and oldest friend ***Rasta***, who was there for me from the first day that I landed in the US without having anyone. Thank you for your patience that cannot be underestimated! I also want to extend my gratitude to my great roommate ***Kourosh Vali***, and all my friends here, ***Parsoa***, ***Rosha***, and ***Tara***. In the end my heartful love and thanks to my family. My wonderful father, caring mother, and supportive brother.

I cannot leave UC Davis without acknowledging Exploratory Systems Lab (ExpoLab) members ***Dr. Suyash Gupta*** and ***Dr. Jelle Hellings***. In almost all the research that I did throughout my Ph.D., Suyash was involved too as a teammate, friend, and colleague. We had success and failure moments together that we passed and I am honored to work with you and learn from you. Also my sincere thanks to Jelle who was a magnificent mentor, researcher, and Cook! I would like to extend my appreciation to ***Rohan Sogani*** and ***Shubham Pandey*** for all their help in our projects.

1 Introduction

In recent years, there has been considerable development and progress in design and scaling permissioned blockchain systems [15, 30, 39, 44, 52, 93, 101, 115, 139, 142, 155, 190]. To digest these permissioned blockchains, we need to understand their core elements and properties, such as consensus, agreement, open/closed-membership, threat model, and attacks. We first investigate the differences between permissionless and permissioned blockchains; then basic components of them and how they work. Next, we provide a summary of consensus and agreement. Finally, we will explain *Practical Byzantine Fault Tolerance* (PBFT), which is the basis of our work.

1.1 Permissioned vs. Permissionless Blockchains

Bitcoin is the first real-world blockchain that is widely used. Satoshi Nakamoto presented the bitcoin as an *immutable tamper-proof ledger distributed among anonymous nodes in the public network* [139]. The ledger consists of blocks containing a list of transactions for coin exchange among anonymous users in the system. Everyone can join the network using a private and public key, and the ledger is also publicly available. The identities are not known since everything is based on public keys. The first block is called *Genesis Block*.

When you want to submit a transaction, you simply create and sign it and broadcast it to the system. Everyone can trace whether your transaction is valid or not based on your transaction history in the ledger. For your transaction to get accepted, it needs to be included in a block and added to the ledger by a consensus mechanism called **Proof of Work (PoW)**[139].

Everyone can gather a set of transactions that have been broadcasted into the network and create a block. By solving a mathematical puzzle, which is highly expensive, they can add their block to the ledger. The new block will be chained to the hash of the previous block to provide immutability. This process is called **Mining** that has rewards (coin creation) as an incentive for the system to keep working. Mining and PoW is a way of achieving consensus among nodes in the system to agree on a sequence of blocks. And since the puzzle is difficult to solve and easy to validate, divergence rarely happens, and they will reach a consensus by accepting the new block. Forks can happen, but they are short-lived and will be removed based on the longest chain rule [139].

Permissionless Blockchains: They mostly offer crypto and financial applications. They are slower than permissioned ones in terms of throughput and latency. A summary of their characteristics is as follows:

- Everyone can join and leave the network (*open membership*)
- They mostly use **PoW** as a consensus protocol
- They are computation-intensive
- Identities of participants are not known
- They are fully decentralized

Permissioned Blockchains In these systems, the identities of nodes are hidden, so they are less centralized than the permissionless systems. Consider a use-case of a supply-chain among a set of companies. Participants, such as retailers, warehouses, and manufacturing are all known to the system; hence they are called closed membership. As the closed-membership property implies, permissioned blockchains are not as decentralized as the permissionless ones. The number of participants and users is constant as opposed to the permissionless systems. All nodes have their own copy of the ledger, and they could act Byzantine or maliciously.

Unlike permissionless blockchains that use computation-intensive proof (PoW) to reach consensus, permissioned ones seek agreement and ordering through a communication-intensive protocol. Using Byzantine Fault-Tolerant (BFT) protocols and voting mechanisms instead of solving a difficult puzzle makes permissioned blockchains faster than public ones. Traditional BFT protocols usually implement a primary-backup model, which relies on a primary or leader node to run the consensus.

1.2 Crash-Fault Tolerance

A fundamental problem in distributed computing is to provide a reliable and available system. This usually happens through the replication of processes or data. Replication requires agreement or consensus. The consensus is defined when there is a set of nodes, and they want to agree on a single value in the presence of different types of failure. **Paxos** [124] is a well-established consensus protocol, and most of the subsequent protocols are based on or improvement of Paxos. Availability is one of the crucial aspects of any distributed system in the presence of faults. Paxos can be used to implement *replicated state machines* which is a solution for providing a reliable and fault-tolerant system.

Replicated state machines They are usually implemented using a replicated log. Each replica in the system keeps a copy of every transition in the state machine. The goal is to agree on every single step among non-faulty replicas. Providing fault-tolerance requires replicas and servers to be physically separated and the failures to be independent and the execution of each step needs to be deterministic

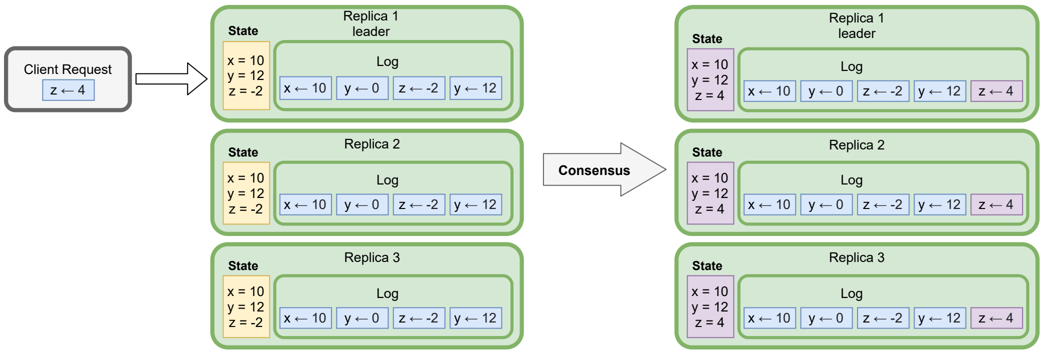


Figure 1: A demonstration of replicated state machine. A client request asks for a transition (Z = 4) and using an instance of consensus the log in all replicas gets updated and they transit to a new state

Raft [147] is a restructured replicated state machine based on Paxos and provides a better foundation for building practical systems. It also contains recovery protocols and leader election mechanisms. It works in a primary-backup manner. It only tolerates crash-fault or benign faults, and the leader election is based on timeouts. The role of the leader in Raft is to order and coordinate state transitions. The leader is equivalent to the proposer in Paxos.

1.3 Byzantine Fault Tolerance

PBFT [41] was the first practical consensus protocol that tolerates Byzantine faults. Similar to Paxos, it follows a primary-backup manner. PBFT is state machine replication of client transactions in the presence of malicious nodes. It is the responsibility of the primary node to order transactions and run a commit protocol among replicas. The primary can be malicious, and PBFT provides a sub-protocol called **View Change** to replace malicious primary. PBFT provides safety on a bounded number of faulty replicas and provides liveness in partial synchronous networks.

System and Threat Model: PBFT can tolerate up to one-third of replicas in the system. If the system has n nodes, f of them could be faulty or malicious where $\mathbf{f} < \lfloor \frac{\mathbf{n}-1}{3} \rfloor$. These faulty nodes can act Byzantine or malicious, meaning they may prevent sending messages, delay, or duplicate them. But they cannot forge digital signatures and break message encryption. PBFT assumes that the communication is authenticated. It supports both digital signatures and *Message Authentication Code* (MAC). For digital signatures, all replicas need to have a public and private key, and for MAC, each pair of nodes in the system has a common key. It assumes that node failures are independent, but the faulty nodes can collude with each other to disrupt the protocol.

Algorithm: PBFT is three-phase protocol. To reach a consensus on a request, first, the client sends a signed request $\langle T \rangle_c$ to the primary P. Then primary initiates the replication of this request by proposing it to all replicas via a PREPREPARE message. When a backup replica receives a PREPREPARE message from the primary, it agrees to participate in a two-phase Byzantine commit protocol. This commit protocol can succeed if at least $\mathbf{n} - 2\mathbf{f}$ non-faulty replicas receive the same PREPREPARE message.

In the first phase of the Byzantine commit protocol, each replica R responds to the PREPREPARE message m by broadcasting a PREPARE message in support of m. After broadcasting the PREPARE message, R waits until it receives $\mathbf{n} - \mathbf{f}$ PREPARE messages in support of m (indicating that at least $\mathbf{n} - 2\mathbf{f}$ non-faulty replicas support m).

Finally, after receiving these messages, R enters the second phase of the Byzantine commit protocol and broadcasts a COMMIT message in support of m. Once a replica R receives $\mathbf{n} - \mathbf{f}$ COMMIT messages in support of m, it has the guarantee that eventually all replicas will commit to $\langle T \rangle_c$. Figure 2 shows the flow of protocol for a system with 4 nodes.

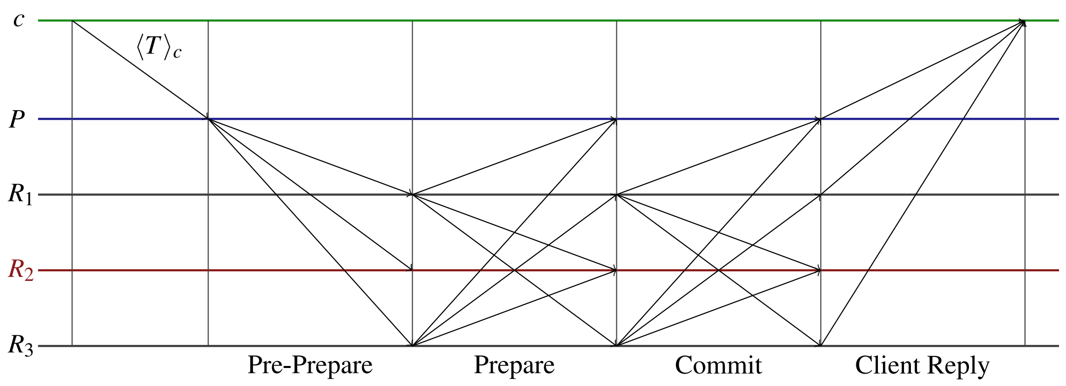


Figure 2: The flow of PBFT when the second replica (R2) is faulty and not sending any message

In this thesis, we investigate and explore various ways to improve permissioned blockchain from multiple perspectives. In the second chapter, we introduced GeoBFT, which improves the scalability of permissioned blockchains in wide-area networks [97]. In the third chapter, we focused on the sharded permissioned blockchains and presented a new Byzantine fault-tolerant sharding consensus [161]. In the fourth chapter, we introduced a serverless architecture using a blockchain shim to scale the execution of blockchains transparently [98]. In the last chapter, we designed and implemented a real-world permissioned blockchain and dissected the implementation's challenges [100].

2 Global Scale Byznatine Fault Tolerance

Recent interest in *blockchain technology* has renewed development of distributed *Byzantine fault-tolerant* (BFT) systems that can deal with failures and malicious attacks of some participants [15, 30, 39, 44, 52, 93, 101, 115, 139, 142, 155, 190]. Although these systems are safe, they attain low throughput, especially when the nodes are spread across a wide-area network (or *geographically large distances*). We believe this contradicts the central promises of blockchain technology: *decentralization* and *democracy*, in which arbitrary replicas at arbitrary distances can participate [65, 90, 101].

At the core of any blockchain system is a BFT consensus protocol that helps participating replicas to achieve resilience. Existing blockchain database systems and data-processing frameworks typically use *permissioned blockchain designs* that rely on traditional BFT consensus [91, 95, 109, 184, 148, 182]. These permissioned blockchains employ a *fully-replicated design* in which all replicas are known and each replica holds a full copy of the data (the blockchain).

2.1 Challenges for Geo-scale Blockchains

To enable geo-scale deployment of a permissioned blockchain system, we believe that the underlying consensus protocol must distinguish between *local* and *global* communication. This belief is easily supported in practice. For example, in Table 1 we illustrate the ping round-trip time and bandwidth measurements. These measurements show that global message latencies are at least 33–270 times higher than local latencies, while the maximum throughput is 10–151 times lower, both implying that communication between regions is *several orders of magnitude* more costly than communication within regions. Hence, a blockchain system needs to recognize and minimize global communication if it is to attain high performance in a geo-scale deployment.

	Ping round-trip times (ms)						*Bandwidth* (Mbit/s)					
	O	I	M	B	T	S	O	I	M	B	T	S
Oregon (O)	≤ 1	38	65	136	118	161	7998	669	371	194	188	136
Iowa (I)		≤ 1	33	98	153	172		10004	752	243	144	120
Montreal (M)			≤ 1	82	186	202			7977	283	111	102
Belgium (B)				≤ 1	252	270				9728	79	66
Taiwan (T)					≤ 1	137					7998	160
Sydney (S)						≤ 1						7977

Table 1: Real-world inter- and intra-cluster communication costs in terms of the ping round-trip times (which determines *latency*) and bandwidth (which determines *throughput*). These measurements are taken in Google Cloud using clusters of `n1` machines (replicas) that are deployed in six different regions.

In the design of geo-scale aware consensus protocols, this translates to two important properties. First, a geo-scale aware consensus protocol needs to be *aware of the network topology*. This can be achieved by clustering replicas in a region together and favoring communication within such clusters over global inter-cluster communication. Second, a geo-scale aware consensus protocol needs to be *decentralized*: no single replica or cluster should be responsible for coordinating all consensus decisions, as such a centralized design limits the throughput to the outgoing global bandwidth and latency of this single replica or cluster.

Existing state-of-the-art consensus protocols do not share these two properties. The influential Practical

Byzantine Fault Tolerance consensus protocol (PBFT) [40, 41] is centralized, as it relies on a single primary replica to coordinate all consensus decisions, and requires a vast amount of global communication (between all pairs of replicas). Protocols such as ZYZZYVA improve on this by reducing communication costs in the optimal case [17, 121, 122]. However, these protocols still have a highly centralized design and do not favor local communication. Furthermore, ZYZZYVA provides high throughput only if there are no failures and requires reliable clients [3, 48]. The recently introduced HOTSTUFF improves on PBFT by simplifying the recovery process on primary failure [200]. This allows HOTSTUFF to efficiently switch primaries for every consensus decision, providing the potential of decentralization. However, the design of HOTSTUFF does not favor local communication, and the usage of threshold signatures strongly centralizes all communication for a single consensus decision to the primary of that round. Another recent protocol POE provides better throughput than both PBFT and ZYZZYVA in the presence of failures, this without employing threshold signatures [91]. Unfortunately, also POE has a centralized design that depends on a single primary. Finally, the geo-aware consensus protocol STEWARD promises to do better [6], as it recognizes local clusters and tries to minimize inter-cluster communication. However, due to its centralized design and reliance on cryptographic primitives with high computational costs, STEWARD is unable to benefit from its topological knowledge of the network.

2.2 GeoBFT: Towards Geo-scale Consensus

In this work, we improve on the state-of-the-art by introducing GEOBFT, a topology-aware and decentralized consensus protocol. In GEOBFT, we group replicas in a region into clusters, and we let each cluster make consensus decisions independently. These consensus decisions are then shared via an optimistic low-cost communication protocol with the other clusters, in this way assuring that all replicas in all clusters are able to learn the same sequence of consensus decisions: if we have two clusters $\mathscr{C}_1$ and $\mathscr{C}_2$ with $\mathbf{n}$ replicas each, then our optimistic communication protocol requires only $\lceil \mathbf{n}/3 \rceil$ messages to be sent from $\mathscr{C}_1$ to $\mathscr{C}_2$ when $\mathscr{C}_1$ needs to share local consensus decisions with $\mathscr{C}_2$. In specific, we make the following contributions:

1. We introduce the GEOBFT consensus protocol, a novel consensus protocol that performs a topological-aware grouping of replicas into local clusters to minimize global communication. GEOBFT also decentralizes consensus by allowing each cluster to make consensus decisions independently.
2. To reduce global communication, we introduce a novel global sharing protocol that *optimistically* performs minimal inter-cluster communication, while still enabling reliable detection of communication failure.
3. The optimistic global sharing protocol is supported by a novel *remote view-change protocol* that deals with any malicious behavior and any failures.
4. We prove that GEOBFT guarantees *safety*: it achieves a unique sequence of consensus decisions among all replicas and ensures that clients can reliably detect when their transactions are executed, this independent of any malicious behavior by any replicas.
5. We show that GEOBFT guarantees *liveness*: whenever the network provides reliable communication, GEOBFT continues successful operation, this independent of any malicious behavior by any replicas.
6. We also implemented other state-of-the-art BFT protocols in ResilientDB (ZYZZYVA, PBFT, HOTSTUFF, and STEWARD), and evaluate GEOBFT against these BFT protocols using the YCSB benchmark [53]. We show that GEOBFT *achieves up-to-six times more throughput* than existing BFT protocols.

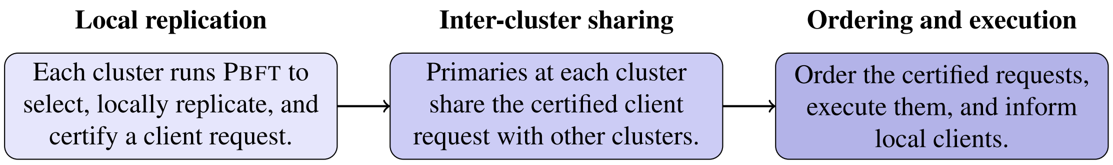

Local replication
Each cluster runs PBFT to select, locally replicate, and certify a client request.
Inter-cluster sharing
Primaries at each cluster share the certified client request with other clusters.
Ordering and execution
Order the certified requests, execute them, and inform local clients.

Figure 3: Steps in a round of the GEOBFT protocol.

In Table 2, we provide a summary of the complexity of the normal-case operations of GEOBFT and compare this to the complexity of other popular BFT protocols.

Protocol	Decisions	Communication		Centralized
		(Local)	*(Global)*	
GEOBFT (our protocol)	$\mathbf{z}$	$\mathcal{O}(2\mathbf{z}\mathbf{n}^2)$	$\mathcal{O}(\mathbf{f}\mathbf{z}^2)$	No
↳ *single decision*	1	$\mathcal{O}(4\mathbf{n}^2)$	$\mathcal{O}(\mathbf{f}\mathbf{z})$	No
STEWARD	1	$\mathcal{O}(2\mathbf{z}\mathbf{n}^2)$	$\mathcal{O}(\mathbf{z}^2)$	Yes
ZYZZYVA	1	$\mathcal{O}(\mathbf{z}\mathbf{n})$		Yes
PBFT	1	$\mathcal{O}(2(\mathbf{z}\mathbf{n})^2)$		Yes
POE	1	$\mathcal{O}((\mathbf{z}\mathbf{n})^2)$		Yes
HOTSTUFF	1	$\mathcal{O}(8(\mathbf{z}\mathbf{n}))$		Partly

Table 2: The normal-case metrics of BFT consensus protocols in a system with $\mathbf{z}$ clusters, each with $\mathbf{n}$ replicas of which at most $\mathbf{f}$, $\mathbf{n} > 3\mathbf{f}$, are Byzantine. GEOBFT provides the lowest global communication cost per consensus decision (transaction) and operates decentralized.

2.3 Geo-Scale Consensus

We now present our 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. Next, we sketch the high-level working of such a round of GEOBFT. Each round consists of the three steps sketched in Figure 3: *local replication*, *global sharing*, and *ordering and execution*, which we further detail next.

1. 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*.

2. 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. To deal with failures, the global sharing protocol utilizes a novel remote view-change protocol.

3. 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).

In Figure 4, we sketch a single round of GEOBFT in a setting of two clusters with four replicas each.

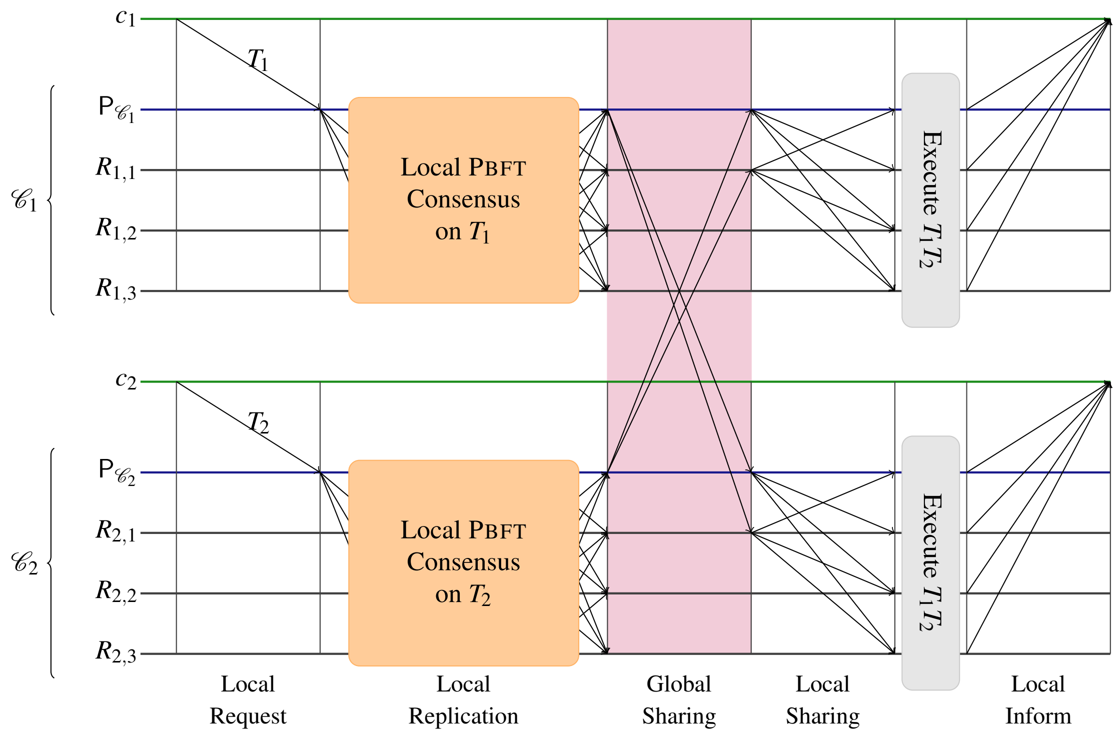


Figure 4: Representation of the normal-case algorithm of GEOBFT running on two clusters. Clients c_i, $i \in \{1,2\}$, request transactions T_i from their local cluster $\mathscr{C}_i$. The primary $\mathsf{P}_{\mathscr{C}_i} \in \mathscr{C}_i$ replicates this transaction to all local replicas using PBFT. At the end of local replication, the primary can produce a cluster certificate for T_i. These are shared with other clusters via inter-cluster communication, after which all replicas in all clusters can execute T_i and $\mathscr{C}_i$ can inform c_i.

2.3.1 Preliminaries

To present GEOBFT in detail, we first introduce the system model we use and the relevant notations.

Let $\mathfrak{R}$ be a set of replicas. We model a topological-aware *system* as a partitioning of $\mathfrak{R}$ into a set of clusters $\mathfrak{S} = \{\mathscr{C}_1, \dots, \mathscr{C}_{\mathbf{z}}\}$, in which each cluster $\mathscr{C}_i$, $1 \leq i \leq \mathbf{z}$, is a set of $|\mathscr{C}_i| = \mathbf{n}$ replicas of which at most $\mathbf{f}$ are *faulty* and can behave in *Byzantine*, possibly coordinated and malicious, manners. We assume that in each cluster $\mathbf{n} > 3\mathbf{f}$.

Remark 2.1. We assumed $\mathbf{z}$ clusters with $\mathbf{n} > 3\mathbf{f}$ replicas each. Hence, $\mathbf{n} = 3\mathbf{f} + j$ for some $j \geq 1$. We use the same failure model as STEWARD [6], but our failure model differs from the more-general failure model utilized by PBFT, ZYZZYVA, and HOTSTUFF [17, 40, 41, 121, 122, 200]. These protocols can each tolerate the failure of up-to-$\lfloor \mathbf{zn}/3 \rfloor = \lfloor (3\mathbf{fz} + \mathbf{z}j)/3 \rfloor = \mathbf{fz} + \lfloor \mathbf{z}j/3 \rfloor$ replicas, even if more than $\mathbf{f}$ of these failures happen in a single region; whereas GEOBFT and STEWARD can only tolerate $\mathbf{fz}$ failures, of which at most $\mathbf{f}$ can happen in a single cluster. E.g., if $\mathbf{n} = 13$, $\mathbf{f} = 4$, and $\mathbf{z} = 7$, then GEOBFT and STEWARD can tolerate $\mathbf{fz} = 28$ replica failures in total, whereas the other protocols can tolerate 30 replica failures. The failure model we use enables the efficient geo-scale aware design of GEOBFT, this without facing well-known communication bounds [60, 66, 67, 68, 78].

We write $\mathsf{f}(\mathscr{C}_i)$ to denote the Byzantine replicas in cluster $\mathscr{C}_i$ and $\mathsf{nf}(\mathscr{C}_i) = \mathscr{C}_i \setminus \mathsf{f}(\mathscr{C}_i)$ to denote the non-faulty replicas in $\mathscr{C}_i$. Each replica $R \in \mathscr{C}_i$ has a unique identifier $\mathrm{id}(R)$, $1 \leq \mathrm{id}(R) \leq \mathbf{n}$. We assume that non-faulty replicas 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 GEOBFT.

Some messages in GEOBFT are forwarded (for example, the client request and commit certificates during inter-cluster sharing). To ensure that malicious replicas do not tamper with messages while forwarding them, we sign these messages using digital signatures [120, 137]. We write $\langle m \rangle_u$ to denote a message signed by u. We assume that it is practically impossible to forge digital signatures. We also assume *authenticated communication*: Byzantine replicas can impersonate each other, but no replica can impersonate another non-faulty replica. Hence, on receipt of a message m from replica $R \in \mathscr{C}_i$, one can determine that R did send m if $R \notin \mathsf{f}(\mathscr{C}_i)$; and one can only determine that m was sent by a non-faulty replica if $R \in \mathsf{nf}(\mathscr{C}_i)$. In the permissioned setting, authenticated communication is a minimal requirement to deal with Byzantine behavior, as otherwise Byzantine replicas can impersonate all non-faulty replicas (which would lead to so-called Sybil attacks) [70]. For messages that are forwarded, authenticated communication is already provided via digital signatures. For all other messages, we use less-costly message authentication codes [120, 137]. Replicas will discard any messages that are not well-formed, have invalid message authentication codes (if applicable), or have invalid signatures (if applicable).

Next, we define the consensus provided by GEOBFT.

Definition 2.2. Let $\mathfrak{S}$ be a system over $\mathfrak{R}$. A single run of any *consensus protocol* should satisfy the following two requirements:

Termination Each non-faulty replica in $\mathfrak{R}$ 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*. A single round of our GEOBFT consists of $\mathbf{z}$ consecutive runs of the PBFT consensus protocol. Hence, in a single round of GEOBFT, all non-faulty replicas execute the same sequence of $\mathbf{z}$ transactions.

To provide *safety*, we do not need any other assumptions on communication or on the behavior of clients. Due to well-known impossibility results for asynchronous consensus [32, 33, 79, 83], we can only provide *liveness* 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.

2.3.2 Local Replication

In the first step of GEOBFT, the local replication step, each cluster will independently choose a client request to execute. Let $\mathfrak{S}$ be a system. Each round ρ of GEOBFT starts with each cluster $\mathscr{C} \in \mathfrak{S}$ replicating a client request T of client $c \in \mathrm{clients}(\mathscr{C})$. To do so, GEOBFT relies on PBFT [40, 41],[1] a primary-backup protocol

[1]Other consensus protocols such as ZYZZYVA [17, 121, 122] and HOTSTUFF [200] promise to improve on PBFT by sharply reducing communication. In our setting, where local communication is abundant (see Table 1), such improvements are unnecessary, and the costs

in which one replica acts as the *primary*, while all the other replicas act as *backups*. In PBFT, the primary is responsible for coordinating the replication of client transactions. We write $\mathsf{P}_{\mathscr{C}}$ to denote the replica in $\mathscr{C}$ that is the current *local primary* of cluster $\mathscr{C}$. The normal-case of PBFT operates in four steps which we sketch in Figure 5. Next, we detail these steps.

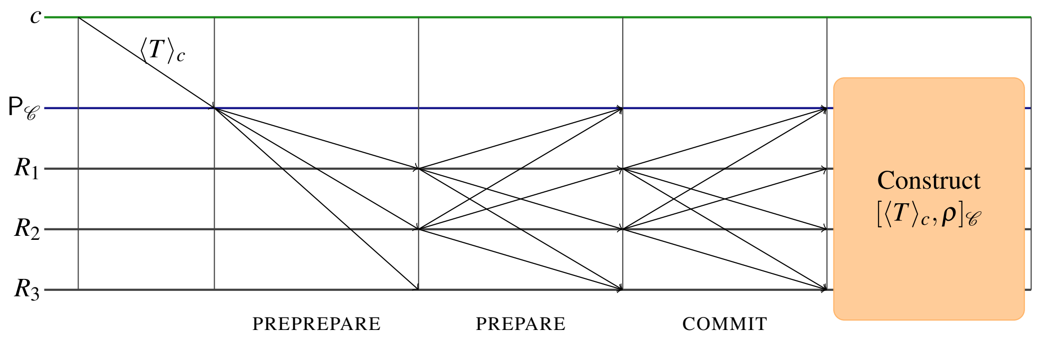
First, the primary $\mathsf{P}_{\mathscr{C}}$ receives client requests of the form $\langle T \rangle_c$, transactions T signed by a local client $c \in \text{clients}(\mathscr{C})$.

Then, in round ρ, $\mathsf{P}_{\mathscr{C}}$ chooses a request $\langle T \rangle_c$ and initiates the replication of this request by proposing it to all replicas via a PREPREPARE message. When a backup replica receives a PREPREPARE message from the primary, it agrees to participate in a two-phase Byzantine commit protocol. This commit protocol can succeed if at least $\mathbf{n} - 2\mathbf{f}$ non-faulty replicas receive the same PREPREPARE message.

In the first phase of the Byzantine commit protocol, each replica R responds to the PREPREPARE message m by broadcasting a PREPARE message in support of m. After broadcasting the PREPARE message, R waits until it receives $\mathbf{n} - \mathbf{f}$ PREPARE messages in support of m (indicating that at least $\mathbf{n} - 2\mathbf{f}$ non-faulty replicas support m).

Finally, after receiving these messages, R enters the second phase of the Byzantine commit protocol and broadcasts a COMMIT message in support of m. Once a replica R receives $\mathbf{n} - \mathbf{f}$ COMMIT messages in support of m, it has the guarantee that eventually all replicas will commit to $\langle T \rangle_c$.

This protocol exchanges sufficient information among all replicas to enable detection of malicious behavior of the primary and to recover from any such behavior. Moreover, on success, each non-faulty replica $R \in \mathscr{C}$ will be committed to the proposed request $\langle T \rangle_c$ and will be able to construct a *commit certificate* $[\langle T \rangle_c, \rho]_R$ that proves this commitment. In GEOBFT, this commit certificate consists of the client request $\langle T \rangle_c$ and $\mathbf{n} - \mathbf{f} > 2\mathbf{f}$ identical COMMIT messages for $\langle T \rangle_c$ signed by distinct replicas. Optionally, GEOBFT can use threshold signatures to represent these $\mathbf{n} - \mathbf{f}$ signatures via a single constant-sized threshold signature [175].



c
$\langle T \rangle_c$
$\mathsf{P}_{\mathscr{C}}$
R_1
R_2
R_3
Construct $[\langle T \rangle_c, \rho]_{\mathscr{C}}$
PREPREPARE
PREPARE
COMMIT

Figure 5: The normal-case working of round ρ of PBFT within a cluster $\mathscr{C}$: a client c requests transaction T, the primary $\mathsf{P}_{\mathscr{C}}$ proposes this request to all local replicas, which prepare and commit this proposal, and, finally, all replicas can construct a commit certificate.

In GEOBFT, we use a PBFT implementation that only uses digital signatures for client requests and COMMIT messages, as these are the only messages that need forwarding. In this configuration, PBFT provides the following properties:

Lemma 2.3 (Castro et al. [40, 41]). *Let $\mathfrak{S}$ be a system and let $\mathscr{C} \in \mathfrak{S}$ be a cluster with $\mathbf{n} > 3\mathbf{f}$. We have the*

of ZYZZYVA (reliable clients) and HOTSTUFF (high computational complexity) can be avoided.

following:

Termination *If communication is reliable, has bounded delay, and a replica $R \in \mathscr{C}$ is able to construct a commit certificate $[\langle T \rangle_c, \rho]_R$, then all non-faulty replicas $R' \in \mathsf{nf}(\mathscr{C})$ will eventually be able to construct a commit certificate $[\langle T' \rangle_{c'}, \rho]_{R'}$.*

Non-divergence *If replicas $R_1, R_2 \in \mathscr{C}$ are able to construct commit certificates $[\langle T_1 \rangle_{c_1}, \rho]_{R_1}$ and $[\langle T_2 \rangle_{c_2}, \rho]_{R_2}$, respectively, then $T_1 = T_2$ and $c_1 = c_2$.*

From Lemma 2.3, we conclude that all commit certificates made by replicas in $\mathscr{C}$ for round ρ show commitment to the same client request $\langle T \rangle_c$. Hence, we write $[\langle T \rangle_c, \rho]_{\mathscr{C}}$, to represent a commit certificate from some replica in cluster $\mathscr{C}$.

To guarantee the correctness of PBFT (Lemma 2.3), we need to prove that both non-divergence and termination hold. From the normal-case working outlined above and in Figure 5, PBFT guarantees non-divergence independent of the behavior of the primary or any malicious replicas.

To guarantee termination when communication is reliable and has bounded delay, PBFT uses *view-changes* and *checkpoints*. If the primary is faulty and prevents any replica from making progress, then the *view-change protocol* enables non-faulty replicas to reliably detect primary failure, recover a common non-divergent state, and trigger primary replacement until a non-faulty primary is found. After a successful view-change, progress is resumed. We refer to these PBFT-provided view-changes as *local view-changes*. The *checkpoint protocol* enables non-faulty replicas to recover from failures and malicious behavior that do not trigger a view-change.

2.3.3 Inter-Cluster Sharing

Once a cluster has completed local replication of a client request, it proceeds with the second step: sharing the client request with all other clusters. Let $\mathfrak{S}$ be a system and $\mathscr{C} \in \mathfrak{S}$ be a cluster. After $\mathscr{C}$ reaches local consensus on client request $\langle T \rangle_c$ in round ρ—enabling construction of the commit certificate $[\langle T \rangle_c, \rho]_{\mathscr{C}}$ that proves local consensus—$\mathscr{C}$ needs to exchange this client request and the accompanying proof with all other clusters. This exchange step requires global inter-cluster communication, which we want to minimize while retaining the ability to reliably detect failure of the sender. However, minimizing this inter-cluster communication is not as straightforward as it sounds, which we illustrate next:

Example 2.4. Let $\mathfrak{S}$ be a system with two clusters $\mathscr{C}_1, \mathscr{C}_2 \in \mathfrak{S}$. Consider a simple global communication protocol in which a message m is sent from $\mathscr{C}_1$ to $\mathscr{C}_2$ by requiring the primary $\mathsf{P}_{\mathscr{C}_1}$ to send m to the primary $\mathsf{P}_{\mathscr{C}_2}$ (which can then disseminate m in $\mathscr{C}_2$). In this protocol, the replicas in $\mathscr{C}_2$ cannot determine what went wrong if they do not receive any messages. To show this, we distinguish two cases:

(1) $\mathsf{P}_{\mathscr{C}_1}$ is Byzantine and behaves correctly toward every replica, except that it never sends messages to $\mathsf{P}_{\mathscr{C}_2}$, while $\mathsf{P}_{\mathscr{C}_2}$ is non-faulty.

(2) $\mathsf{P}_{\mathscr{C}_1}$ is non-faulty, while $\mathsf{P}_{\mathscr{C}_2}$ is Byzantine and behaves correctly toward every replica, except that it drops all messages sent by $\mathsf{P}_{\mathscr{C}_1}$.

In both cases, the replicas in $\mathscr{C}_2$ do not receive any messages from $\mathscr{C}_1$, while both clusters see correct behavior of their primaries with respect to local consensus. Indeed, with this little amount of communication, it is impossible for replicas in $\mathscr{C}_2$ to determine whether $\mathsf{P}_{\mathscr{C}_1}$ is faulty (and did not send any messages) or $\mathsf{P}_{\mathscr{C}_2}$ is faulty (and did not forward any received messages from $\mathscr{C}_1$).

In GEOBFT, we employ an *optimistic* approach to reduce communication among the clusters. Our optimistic approach consists of a low-cost normal-case protocol that will succeed when communication is reliable and the primary of the sending cluster is non-faulty. To deal with any failures, we use a *remote view-change* protocol that guarantees eventual normal-case behavior when communication is reliable. First, we describe the normal-case protocol, after which we will describe in detail the remote view-change protocol.

Optimistic inter-cluster sending In the optimistic case, where participants are non-faulty, we want to send a minimum number of messages while retaining the ability to reliably detect failure of the sender. In Example 2.4, we already showed that sending only a single message is not sufficient. Sending $\mathbf{f}+1$ messages is sufficient, however.

Let $m = (\langle T\rangle_c, [\langle T\rangle_c, \rho]_{\mathscr{C}_1})$ be the message that some replica in cluster $\mathscr{C}_1$ needs to send to some replicas $\mathscr{C}_2$. Note that m includes the request replicated in $\mathscr{C}_1$ in round ρ, and the commit-certificate, which is the proof that such a replication did take place. Based on the observations made above, we propose a two-phase normal-case global sharing protocol. We sketch this normal-case sending protocol in Figure 6 and present the detailed pseudo-code for this protocol in Figure 7.

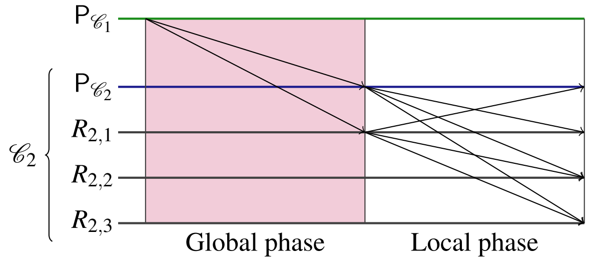


Figure 6: A schematic representation of the normal-case working of the global sharing protocol used by $\mathscr{C}_1$ to send $m = (\langle T\rangle_c, [\langle T\rangle_c, \rho]_{\mathscr{C}_1})$ to $\mathscr{C}_2$.

```
The global phase (used by the primary $\mathsf{P}_{\mathscr{C}_1}$) :
1: Choose a set $S$ of $\mathbf{f}+1$ replicas in $\mathscr{C}_2$.
2: Send $m$ to each replica in $S$.

The local phase (used by replicas $R \in \mathscr{C}_2$) :
3: event receive $m$ from a replica $Q \in \mathscr{C}_1$ do
4:     Broadcast $m$ to all replicas in $\mathscr{C}_2$.
```

Figure 7: The normal-case global sharing protocol used by $\mathscr{C}_1$ to send $m = (\langle T\rangle_c, [\langle T\rangle_c, \rho]_{\mathscr{C}_1})$ to $\mathscr{C}_2$.

In the *global phase*, the primary $\mathsf{P}_{\mathscr{C}_1}$ sends m to $\mathbf{f}+1$ replicas in $\mathscr{C}_2$. In the *local phase*, each non-faulty replica $R \in \mathsf{nf}(\mathscr{C}_2)$ that receives a well-formed m forwards m to all replicas in its cluster $\mathscr{C}_2$.

Proposition 2.5. *Let $\mathfrak{S}$ be a system, let $\mathscr{C}_1, \mathscr{C}_2 \in \mathfrak{S}$ be two clusters, and let $m = (\langle T\rangle_c, [\langle T\rangle_c, \rho]_{\mathscr{C}_1})$ be the message $\mathscr{C}_1$ sends to $\mathscr{C}_2$ using the normal-case global sharing protocol of Figure 7. We have the following:*

Receipt *If the primary $\mathsf{P}_{\mathscr{C}_1}$ is non-faulty and communication is reliable, then every replica in $\mathscr{C}_2$ will eventually receive m.*

Agreement *Replicas in $\mathcal{C}_2$ will only accept client request $\langle T \rangle_c$ from $\mathcal{C}_1$ in round ρ.*

Proof. If the primary $\mathsf{P}_{\mathcal{C}_1}$ is non-faulty and communication is reliable, then $\mathbf{f}+1$ replicas in $\mathcal{C}_2$ will receive m (Line 2). As at most $\mathbf{f}$ replicas in $\mathcal{C}_2$ are Byzantine, at least one of these receiving replicas is non-faulty and will forward this message m to all replicas in $\mathcal{C}_2$ (Line 4), proving termination.

The commit certificate $[\langle T \rangle_c, \rho]_{\mathcal{C}_1}$ cannot be forged by faulty replicas, as it contains signed COMMIT messages from $\mathbf{n}-\mathbf{f} > \mathbf{f}$ replicas. Hence, the integrity of any message m forwarded by replicas in $\mathcal{C}_2$ can easily be verified. Furthermore, Lemma 2.3 rules out the existence of any other messages $m' = [\langle T' \rangle_{c'}, \rho]_{\mathcal{C}_1}$, proving agreement. □

We notice that there are two cases in which replicas in $\mathcal{C}_2$ do not receive m from $\mathcal{C}_1$: either $\mathsf{P}_{\mathcal{C}_1}$ is faulty and did not send m to $\mathbf{f}+1$ replicas in $\mathcal{C}_2$, or communication is unreliable, and messages are delayed or lost. In both cases, non-faulty replicas in $\mathcal{C}_2$ initiate *remote view-change* to force primary replacement in $\mathcal{C}_1$ (causing replacement of the primary $\mathsf{P}_{\mathcal{C}_1}$).

Remote view-change The normal-case global sharing protocol outlined will only succeed if communication is reliable and the primary of the sending cluster is non-faulty. To recover from any failures, we provide a remote view-change protocol. Let $\mathfrak{S} = \{\mathcal{C}_1, \dots, \mathcal{C}_{\mathbf{z}}\}$ be a system. To simplify presentation, we focus on the case in which the primary of cluster $\mathcal{C}_1$ fails to send $m = (\langle T \rangle_c, [\langle T \rangle_c, \rho]_{\mathcal{C}_1})$ to replicas of $\mathcal{C}_2$. Our remote view-change protocol consists of four phases, which we detail next.

First, non-faulty replicas in cluster $\mathcal{C}_2$ detect the failure of the current primary $\mathsf{P}_{\mathcal{C}_1}$ of $\mathcal{C}_1$ to send m. Note that although the replicas in $\mathcal{C}_2$ have no information about the contents of message m, they are awaiting arrival of a well-formed message m from $\mathcal{C}_1$ in round ρ. Second, the non-faulty replicas in $\mathcal{C}_2$ initiate agreement on failure detection. Third, after reaching agreement, the replicas in $\mathcal{C}_2$ send their request for a remote view-change to the replicas in $\mathcal{C}_1$ in a reliable manner. In the fourth and last phase, the non-faulty replicas in $\mathcal{C}_1$ trigger a local view-change, replace $\mathsf{P}_{\mathcal{C}_1}$, and instruct the new primary to resume global sharing with $\mathcal{C}_2$. Next, we explain each phase in detail.

To be able to detect failure, $\mathcal{C}_2$ must assume reliable communication with bounded delay. This allows the usage of timers to detect failure. To do so, every replica $R \in \mathcal{C}_2$ sets a timer for $\mathcal{C}_1$ at the start of round ρ and waits until it receives a valid message m from $\mathcal{C}_1$. If the timer expires before R receives such an m, then R detects failure of $\mathcal{C}_1$ in round ρ. Successful detection will eventually lead to a remote view-change request.

From the perspective of $\mathcal{C}_1$, remote view-changes are controlled by external parties. This leads to several challenges not faced by traditional PBFT view-changes (the local view-changes used within clusters, e.g., as part of local replication):

(1) A remote view-change in $\mathcal{C}_1$ requested by $\mathcal{C}_2$ should only trigger at most a single local view-change in $\mathcal{C}_1$, otherwise remote view-changes enable *replay attacks*.

(2) While replicas in $\mathcal{C}_1$ detect failure of $\mathsf{P}_{\mathcal{C}_1}$ and initiate local view-change, it is possible that $\mathcal{C}_2$ detects failure of $\mathcal{C}_1$ and requests remote view-change in $\mathcal{C}_1$. In this case, only a single successful view-change in $\mathcal{C}_1$ is necessary.

(3) Likewise, several clusters $\mathcal{C}_2, \dots, \mathcal{C}_{\mathbf{z}}$ can simultaneously detect failure of $\mathcal{C}_1$ and request remote view-change in $\mathcal{C}_1$. Also in this case, only a single successful view-change in $\mathcal{C}_1$ is necessary.

Furthermore, a remote view-change request for cluster $\mathcal{C}_1$ cannot depend on any information only available to $\mathcal{C}_1$ (e.g., the current primary $\mathsf{P}_{\mathcal{C}_1}$ of $\mathcal{C}_1$). Likewise, the replicas in $\mathcal{C}_1$ cannot determine which messages (for which rounds) have already been sent by previous (possibly malicious) primaries of $\mathcal{C}_1$: remote view-change requests must include this information. Our remote view-change protocol addresses each of these concerns. In Figures 8 and 9, we sketch this protocol and its pseudo-code. Next, we describe the protocol in detail.

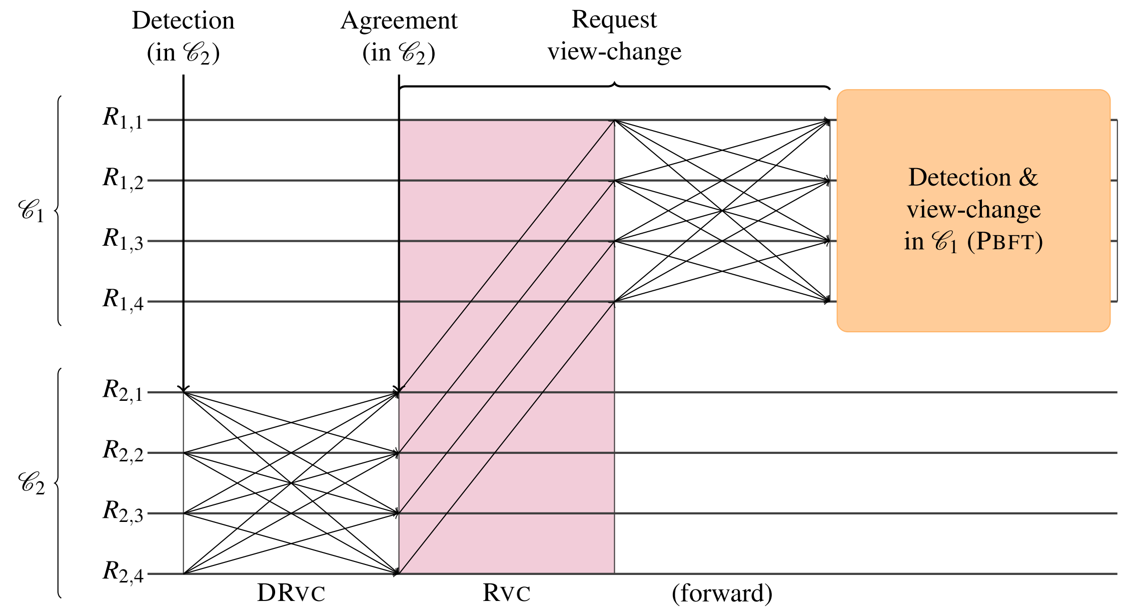


Figure 8: A schematic representation of the remote view-change protocol of GeoBFT running at a system $\mathfrak{S}$ over $\mathfrak{R}$. This protocol is triggered when a cluster $\mathcal{C}_2 \in \mathfrak{S}$ expects a message from $\mathcal{C}_1 \in \mathfrak{S}$, but does not receive this message in time.

```
**Initiation role** (used by replicas $R \in \mathscr{C}_2$) :
1: $v_1 := 0$ (number of remote view-changes in $\mathscr{C}_1$ requested by $R$).
2: event detect failure of $\mathscr{C}_1$ in round $\rho$ do
3:     Broadcast $\text{DRvc}(\mathscr{C}_1, \rho, v_1)$ to all replicas in $\mathscr{C}_2$.
4:     $v_1 := v_1 + 1$.
5: event $R$ receives $\text{DRvc}(\mathscr{C}_1, \rho, v_1)$ from $R' \in \mathscr{C}_2$ do
6:     if $R$ received $(\langle T \rangle_c, [\langle T \rangle_c, \rho]_{\mathscr{C}})$ from $Q \in \mathscr{C}_1$ then
7:         Send $(\langle T \rangle_c, [\langle T \rangle_c, \rho]_{\mathscr{C}})$ to $R'$.
8: event $R$ receives $\text{DRvc}(\mathscr{C}_1, \rho, v_1')$ from $\mathbf{f}+1$ replicas in $\mathscr{C}_2$ do
9:     if $v_1 \leq v_1'$ then
10:        $v_1 := v_1'$.
11:        Detect failure of $\mathscr{C}_1$ in round $\rho$ (if not yet done so).
12: event $R$ receives $\text{DRvc}(\mathscr{C}_1, \rho, v_1)$ from $\mathbf{n}-\mathbf{f}$ replicas in $\mathscr{C}_2$ do
13:     Send $\langle \text{Rvc}(\mathscr{C}_1, \rho, v_1) \rangle_R$ to $Q \in \mathscr{C}_1$, $\text{id}(R) = \text{id}(Q)$.

**Response role** (used by replicas $Q \in \mathscr{C}_1$) :
14: event $Q$ receives $\langle \text{Rvc}(\mathscr{C}_1, \rho, v) \rangle_R$ from $R$, $R \in (\mathfrak{R} \setminus \mathscr{C}_1)$ do
15:     Broadcast $\langle \text{Rvc}(\mathscr{C}_1, \rho, v) \rangle_R$ to all replicas in $\mathscr{C}_1$.
16: event $Q$ receives $\langle \text{Rvc}(\mathscr{C}_1, \rho, v) \rangle_{R_i}$, $1 \leq i \leq \mathbf{f}+1$, such that:
        1. $\{R_i \mid 1 \leq i \leq \mathbf{f}+1\} \subset \mathscr{C}'$, $\mathscr{C}' \in \mathfrak{S}$;
        2. $|\{R_i \mid 1 \leq i \leq \mathbf{f}+1\}| = \mathbf{f}+1$;
        3. no recent local view-change was triggered; and
        4. $\mathscr{C}'$ did not yet request a $v$-th remote view-change
    do
17:     Detect failure of $\mathsf{P}_{\mathscr{C}_1}$ (if not yet done so).
```

Figure 9: The remote view-change protocol of GeoBFT running at a system $\mathfrak{S}$ over $\mathfrak{R}$. This protocol is triggered when a cluster $\mathscr{C}_2 \in \mathfrak{S}$ expects a message from $\mathscr{C}_1 \in \mathfrak{S}$, but does not receive this message in time.

Let $R \in \mathscr{C}_2$ be a replica that detects failure of $\mathscr{C}_1$ in round ρ and has already requested v_1 remote view-changes in $\mathscr{C}_1$. Once a replica R detects a failure, it initiates the process of reaching an agreement on this failure among other replicas of its cluster $\mathscr{C}_2$. It does so by broadcasting message $\text{DRvc}(\mathscr{C}_1, \rho, v_1)$ to all replicas in $\mathscr{C}_2$ (Line 3 of Figure 9).

Next, R waits until it receives identical $\text{DRvc}(\mathscr{C}_1, \rho, v_1)$ messages from $\mathbf{n}-\mathbf{f}$ distinct replicas in $\mathscr{C}_2$ (Line 12 of Figure 9). This guarantees that there is agreement among the non-faulty replicas in $\mathscr{C}_2$ that $\mathscr{C}_1$ has failed. After receiving these $\mathbf{n}-\mathbf{f}$ messages, R requests a remote view-change by sending message $\langle \text{Rvc}(\mathscr{C}_1, \rho, v_1) \rangle_R$ to the replica $Q \in \mathscr{C}_1$ with $\text{id}(R) = \text{id}(Q)$ (Line 13 of Figure 9).

In case some other replica $R' \in \mathscr{C}_2$ received m from $\mathscr{C}_1$, then R' would respond with message m in response to the message $\text{DRvc}(\mathscr{C}_1, \rho, v)$ (Line 5 of Figure 9). This allows R to recover in cases where it could not reach an agreement on the failure of $\mathscr{C}_1$. Finally, some replica $R' \in \mathscr{C}_2$ may detect the failure of $\mathscr{C}_1$ later than R. To handle such a case, we require each replica R' that receives identical $\text{DR}\textsc{vc}(\mathscr{C}_1, \rho, v)$ messages from $\mathbf{f}+1$ distinct replicas in $\mathscr{C}_2$ to assume that the cluster $\mathscr{C}_1$ has failed. This assumption is valid as one of these $\mathbf{f}+1$ messages must have come from a non-faulty replica in $\mathscr{C}_2$, which must have detected the failure of cluster $\mathscr{C}_1$ successfully (Line 8 of Figure 9).

If replica $Q \in \mathscr{C}_1$ receives a remote view-change request $m_{\text{RCV}} = \langle \text{R}\textsc{vc}(\rho, v)\rangle_R$ from $R \in \mathscr{C}_2$, then Q verifies whether m_{RCV} is well-formed. If m_{RCV} is well-formed, Q forwards m_{RCV} to all replicas in $\mathscr{C}_1$ (Line 14 of Figure 9). Once Q receives $\mathbf{f}+1$ messages identical to m_{RCV}, signed by distinct replicas in $\mathscr{C}_2$, it concludes that at least one of these remote view-change requests must have come from a non-faulty replica in $\mathscr{C}_2$. Next, Q determines whether it will honor this remote view-change request, which Q will do when no concurrent local view-change is in progress and when this is the first v-th remote view-change requested by $\mathscr{C}_2$ (the lather prevents replay attacks). If these conditions are met, Q detects its current primary $\mathsf{P}_{\mathscr{C}_1}$ as faulty (Line 16 of Figure 9).

When communication is reliable, the above protocol ensures that all non-faulty replicas in $\mathscr{C}_1$ will detect failure of $\mathsf{P}_{\mathscr{C}_1}$. Hence, eventually a successful local view-change will be triggered in $\mathscr{C}_1$. When a new primary in $\mathscr{C}_1$ is elected, it takes one of the remote view-change requests it received and determines the rounds for which it needs to send requests (using the normal-case global sharing protocol of Figure 7). As replicas in $\mathscr{C}_2$ do not know the exact communication delays, they use exponential back off to determine the timeouts used while detecting subsequent failures of $\mathscr{C}_1$.

We are now ready to prove the main properties of remote view-changes.

Proposition 2.6. *Let $\mathfrak{S}$ be a system, let $\mathscr{C}_1, \mathscr{C}_2 \in \mathfrak{S}$ be two clusters, and let $m = (\langle T\rangle_c, [\langle T\rangle_c, \rho]_{\mathscr{C}})$ be the message $\mathscr{C}_1$ needs to send to $\mathscr{C}_2$ in round ρ. If communication is reliable and has bounded delay, then either every replica in $\mathscr{C}_2$ will receive m or $\mathscr{C}_1$ will perform a local view-change.*

Proof. Consider the remote view-change protocol of Figure 9. If a non-faulty replica $R' \in \mathsf{nf}(\mathscr{C}_2)$ receives m, then any replica in $\mathscr{C}_2$ that did not receive m will receive m from R' (Line 5). In all other cases, at least $\mathbf{f}+1$ non-faulty replicas in $\mathscr{C}_2$ will not receive m and will timeout. Due to exponential back-off, eventually each of these $\mathbf{f}+1$ non-faulty replicas will initiate and agree on the same v_1-th remote view-change. Consequently, all non-faulty replicas in $\mathsf{nf}(\mathscr{C}_2)$ will participate in this remote view-change (Line 8). As $|\mathsf{nf}(\mathscr{C}_2)| = \mathbf{n}-\mathbf{f}$, each of these $\mathbf{n}-\mathbf{f}$ replicas $R \in \mathsf{nf}(\mathscr{C}_2)$ will send $\langle \text{R}\textsc{vc}(\mathscr{C}_1, \rho, v)\rangle_R$ to some replica $Q \in \mathscr{C}_1$, $\text{id}(R) = \text{id}(Q)$ (Line 12). Let $S = \{Q \in \mathscr{C}_1 \mid R \in \mathsf{nf}(\mathscr{C}_2) \wedge \text{id}(R) = \text{id}(Q)\}$ be the set of receivers in $\mathscr{C}_1$ of these messages and let $T = S \cap \mathsf{nf}(\mathscr{C}_1)$. We have $|S| = \mathbf{n}-\mathbf{f} > 2\mathbf{f}$ and, hence, $|T| > \mathbf{f}$. Each replica $Q \in T$ will broadcast the message it receives to all replicas in $\mathscr{C}_1$ (Line 14). As $|T| > \mathbf{f}$, this eventually triggers a local view-change in $\mathscr{C}_1$ (Line 16). □

Finally, we use the results of Proposition 2.5 and Proposition 2.6 to conclude

Theorem 2.7. *Let $\mathfrak{S} = \{\mathscr{C}_1, \dots, \mathscr{C}_{\mathbf{z}}\}$ be a system over $\mathfrak{R}$. If communication is reliable and has bounded delay, then every replica $R \in \mathfrak{R}$ will, in round ρ, receive a set $\{(\langle T_i\rangle_{c_i}, [\langle T_i\rangle_{c_i}, \rho]_{\mathscr{C}_i}) \mid (1 \le i \le \mathbf{z}) \wedge (c_i \in \text{clients}(\mathscr{C}_i))\}$ of $\mathbf{z}$ messages. These sets all contain identical client requests.*

Proof. Consider cluster $\mathscr{C}_i \in \mathfrak{S}$. If $\mathsf{P}_{\mathscr{C}_i}$ behaves reliable, then Proposition 2.5 already proves the statement with respect to $(\langle T_i\rangle_{c_i}, [\langle T_i\rangle_{c_i}, \rho]_{\mathscr{C}_i})$. Otherwise, if $\mathsf{P}_{\mathscr{C}_i}$ behaves Byzantine, then then Proposition 2.6 guarantees that either all replicas in $\mathfrak{R}$ will receive $(\langle T_i\rangle_{c_i}, [\langle T_i\rangle_{c_i}, \rho]_{\mathscr{C}_i})$ or $\mathsf{P}_{\mathscr{C}_i}$ will be replaced via a local view-change. Eventually, these local view-changes will lead to a non-faulty primary in $\mathscr{C}_i$, after which Proposition 2.5 again proves the statement with respect to $(\langle T_i\rangle_{c_i}, [\langle T_i\rangle_{c_i}, \rho]_{\mathscr{C}_i})$. □

2.3.4 Ordering and Execution

Once replicas of a cluster have chosen a client request for execution and have received all client requests chosen by other clusters, they are ready for the final step: ordering and executing these client requests. In specific, in round ρ, any non-faulty replica that has valid requests from all clusters can move ahead and execute these requests.

Theorem 2.7 guarantees after the local replication step (Section 2.3.2) and the inter-cluster sharing step (Section 2.3.3) each replica in $\mathfrak{R}$ will receive the same set of $\mathbf{z}$ client requests in round ρ. Let $S_\rho = \{(\langle T_i \rangle_{c_i} \mid (1 \leq i \leq \mathbf{z}) \land (c_i \in \text{clients}(\mathscr{C}_i))\}$ be this set of $\mathbf{z}$ client requests received by each replica.

The last step is to put these client requests in a unique order, execute them, and inform the clients of the outcome. To do so, GEOBFT simply uses a pre-defined ordering on the clusters. For example, each replica executes the transactions in the order $[T_1, \dots, T_{\mathbf{z}}]$. Once the execution is complete, each replica $R \in \mathscr{C}_i$, $1 \leq i \leq \mathbf{z}$, informs the client c_i of any outcome (e.g., confirmation of execution or the result of execution). Note that each replica R only informs its local clients. As all non-faulty replicas are expected to act deterministic, execution will yield the same state and results across all non-faulty replicas. Hence, each client c_i is guaranteed to receive identical response from at least $\mathbf{f}+1$ replicas. As there are at most $\mathbf{f}$ faulty replicas per cluster and faulty replicas cannot impersonate non-faulty replicas, at least one of these $\mathbf{f}+1$ responses must come from a non-faulty replica. We conclude the following:

Theorem 2.8 (GEOBFT is a consensus protocol)**.** *Let $\mathfrak{S}$ be a system over $\mathfrak{R}$ in which every cluster satisfies* $\mathbf{n} > 3\mathbf{f}$. *A single round of* GEOBFT *satisfies the following two requirements:*

Termination *If communication is reliable and has bounded delay, then* GEOBFT *guarantees that each non-faulty replica in $\mathfrak{R}$ executes* $\mathbf{z}$ *transactions.*

Non-divergence GEOBFT *guarantees that all non-faulty replicas execute the same* $\mathbf{z}$ *transaction.*

Proof. Both *termination* and *non-divergence* are direct corollaries of Theorem 2.7. □

2.3.5 Final Remarks

Until now we have presented the design of GEOBFT using a strict notion of rounds. Only during the last step of each round of GEOBFT, which orders and executes client requests (Section 2.3.4), this strict notion of rounds is required. All other steps can be performed out-of-order. For example, local replication and inter-cluster sharing of client requests for future rounds can happen in parallel with ordering and execution of client requests. In specific, the replicas of a cluster $\mathscr{C}_i$, $1 \leq i \leq z$ can replicate the requests for round $\rho+2$, share the requests for round $\rho+1$ with other clusters, and execute requests for round ρ in parallel. Hence, GEOBFT needs minimal synchronization between clusters.

Additionally, we do not require that every cluster always has client requests available. When a cluster $\mathscr{C}$ does not have client requests to execute in a round, the primary $\mathsf{P}_{\mathscr{C}}$ can propose a no-op-request. The primary $\mathsf{P}_{\mathscr{C}}$ can detect the need for such a no-op request in round ρ when it starts receiving client requests for round ρ from other clusters. As with all requests, also such no-op requests requires commit certificates obtained via local replication.

To prevent that $\mathsf{P}_{\mathscr{C}}$ can indefinitely ignore requests from some or all clients in clients($\mathscr{C}$), we rely on standard PBFT techniques to detect and resolve such attacks during local replication. These techniques effectively allow clients in clients($\mathscr{C}$) to force the cluster to process its request, ruling out the ability of faulty primaries to indefinitely propose no-op requests when client requests are available.

Furthermore, to simplify presentation, we have assumed that every cluster has exactly the same size and that the set of replicas never change. These assumptions can be lifted, however. GEOBFT can easily be extended to also work with clusters of varying size, this only requires minor tweaks on the remote view-change protocol of Figure 9 (the conditions at Line 16 rely on the cluster sizes, see Proposition 2.6). To deal with faulty replicas that eventually recover, we can rely on the same techniques as PBFT [40, 41]. Full dynamic membership, in which replicas can join and leave GEOBFT via some vetted automatic procedure, is a challenge for any permissioned blockchain and remains an open problem for future work [29, 164].

2.4 Implementation and Evaluation

GEOBFT is designed to enable geo-scale deployment of a permissioned blockchain. We will present our ResilientDB fabric [99] in the next section. A permissioned blockchain fabric that can use GEOBFT to provide such a geo-scale aware high-performance permissioned blockchain. ResilientDB is especially tuned to enterprise-level blockchains in which (i) replicas can be dispersed over a wide area network; (ii) links connecting replicas at large distances have low bandwidth; (iii) replicas are untrusted but known; and (iv) applications require high throughput and low latency. These four properties are directly motivated by practical properties of geo-scale deployed distributed systems (see Table 1 in Section 2.1). In Figure 10, we present the architecture of GEOBFTinside ResilientDB.

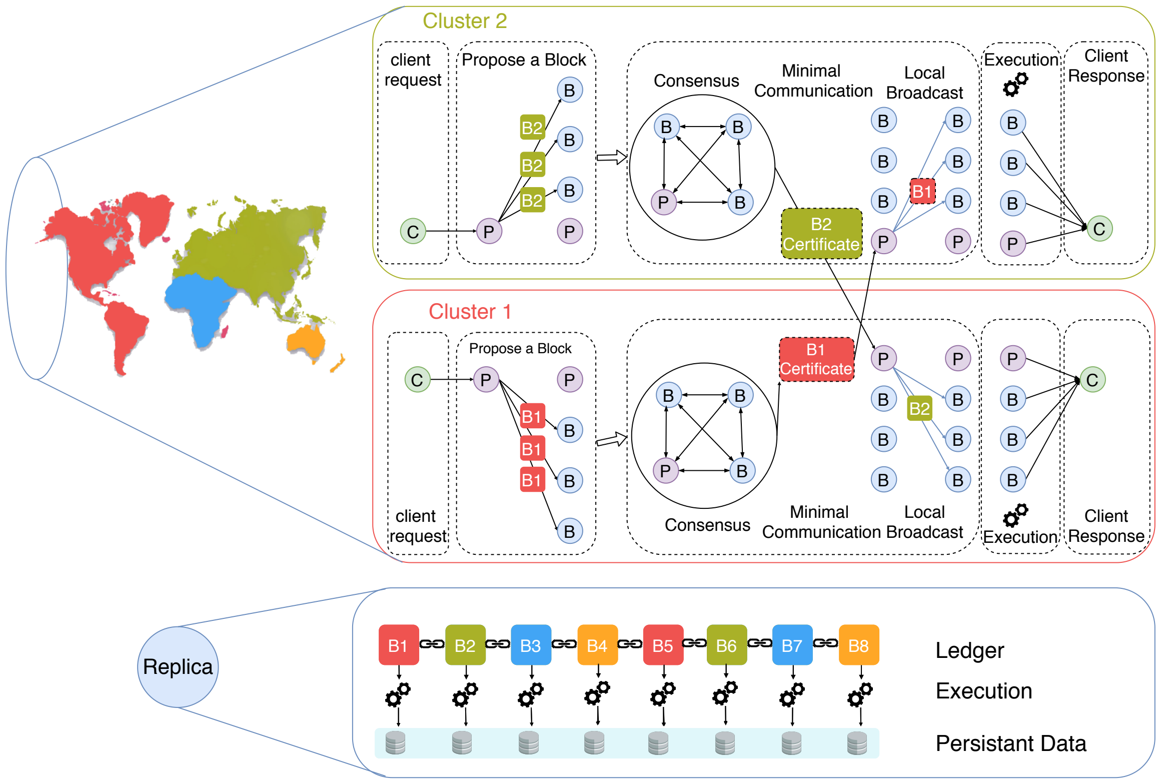


Figure 10: Architecture of GEOBFT inside ResilientDB Fabric.

2.4.1 Evaluation

To showcase the practical value of GEOBFT, we now use our ResilientDB fabric to evaluate GEOBFT against four other popular state-of-the-art consensus protocols (PBFT, ZYZZYVA, HOTSTUFF, and STEWARD). We deploy ResilientDB on the Google Cloud using `N1` machines that have 8-core Intel Skylake CPUs and 16 GB of main memory. Additionally, we deploy 160 k clients on eight 4-core machines having 16 GB of main memory. We equally distribute the clients across all the regions used in each experiment.

In each experiment, the workload is provided by the *Yahoo Cloud Serving Benchmark* (YCSB) [53]. Each client transaction queries a YCSB table with an active set of 600 k records. For our evaluation, we use *write queries*, as those are typically more costly than read-only queries. Prior to the experiments, each replica is initialized with an identical copy of the YCSB table. The client transactions generated by YCSB follow a uniform Zipfian distribution. Clients and replicas can batch transactions to reduce the cost of consensus. In our experiments, we use a *batch size* of 100 requests per batch (unless stated otherwise).

With a batch size of 100, the messages have sizes of 5.4 kB (PREPREPARE), 6.4 kB (commit certificates containing seven COMMIT messages and a PREPREPARE message), 1.5 kB (client responses), and 250 B (other messages). The size of a commit certificate is largely dependent on the size of the PREPREPARE message, while the total size of the accompanying COMMIT messages is small. Hence, the inter-cluster sharing of these certificates is not a bottleneck for GEOBFT: existing BFT protocols send PREPREPARE messages to all replicas irrespective of their region. Further, if the size of COMMIT messages starts dominating, then threshold signatures can be adopted to reduce their cost [175].

To perform geo-scale experiments, we deploy replicas across *six* different regions, namely Oregon, Iowa, Montreal, Belgium, Taiwan, and Sydney. In Table 1, we present our measurements on the inter-region network latency and bandwidth. We run each experiment for 180 s: first, we allow the system to warm-up for 60 s, after which we collect measurement results for the next 120 s. We average the results of our experiments over three runs.

For PBFT and ZYZZYVA, centralized protocols in which a single primary replica coordinates consensus, we placed the primary in Oregon, as this region has the highest bandwidth to all other regions (see Table 1). For HOTSTUFF, our implementation permits all replicas to act as both primary and non-primary at the same time. For both GEOBFT and STEWARD, we group replicas in a single region into a single cluster. In each of these protocols, each cluster has its own local primary. Finally, for STEWARD, a centralized protocol in which the primary cluster coordinates the consensus, we placed the primary cluster in Oregon. We focus our evaluation on answering the following four research questions:

(1) What is the impact of geo-scale deployment of replicas in distant clusters on the performance of GEOBFT, as compared to other consensus protocols?

(2) What is the impact of the size of local clusters (relative to the number of clusters) on the performance of GEOBFT, as compared to other consensus protocols?

(3) What is the impact of failures on the performance of GEOBFT, as compared to other consensus protocols?

(4) Finally, what is the impact of request batching on the performance of GEOBFT, as compared to other consensus protocols, and under which batch sizes can GEOBFT already provide good throughput?

2.4.2 Impact of Geo-Scale deployment

First, we determine the impact of geo-scale deployment of replicas in distant regions on the performance of GeoBFT and other consensus protocols. To do so, we measure the throughput and latency attained by ResilientDB as a function of the number of regions, which we vary between 1 and 6. We use 60 replicas evenly distributed over the regions, and we select regions in the order Oregon, Iowa, Montreal, Belgium, Taiwan, and Sydney. E.g., if we have four regions, then each region has 15 replicas, and we have these replicas in Oregon, Iowa, Montreal, and Belgium. The results of our measurements can be found in Figure 11.

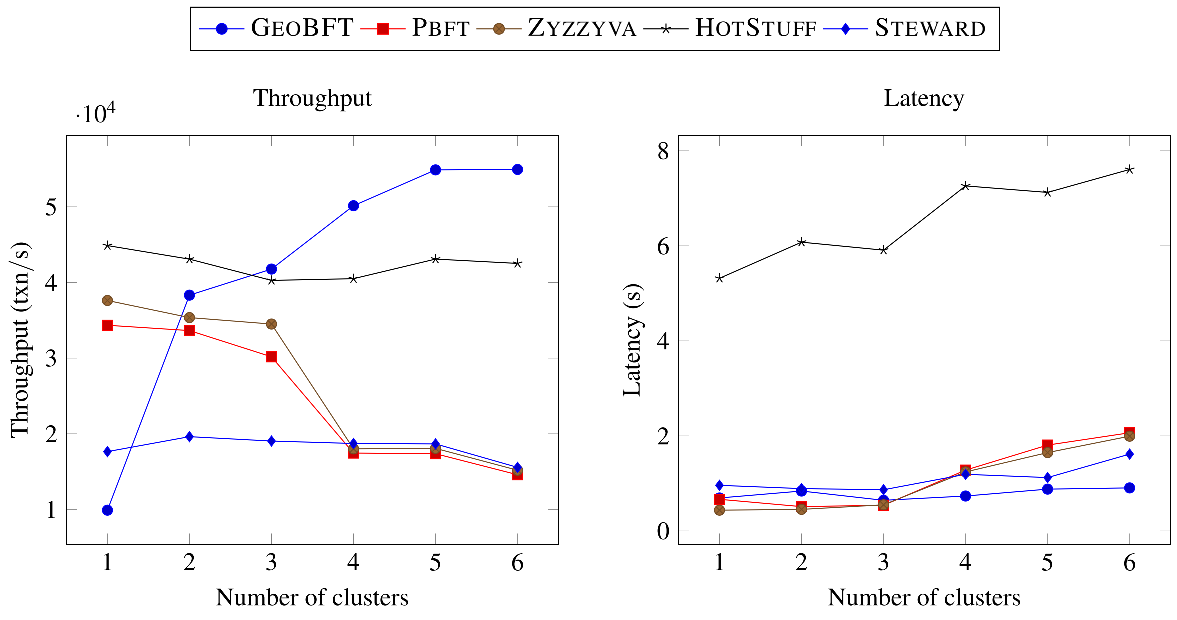


Figure 11: Throughput and latency as a function of the number of clusters; **zn** = 60 replicas.

From the measurements, we see that Steward is unable to benefit from its topological knowledge of the network: in practice, we see that the high computational costs and the centralized design of Steward prevent high throughput in all cases. Both Pbft and Zyzzyva perform better than Steward, especially when ran in a few well-connected regions (e.g., only the North-American regions). The performance of these protocols falls when inter-cluster communication becomes a bottleneck, however (e.g., when regions are spread across continents). HotStuff, which is designed to reduce communication compared to Pbft, has reasonable throughput in a geo-scale deployment, and sees only a small drop in throughput when regions are added. The high computational costs of the protocol prevent it from reaching high throughput in any setting, however. Additionally, HotStuff has very high latencies due to its 4-phase design. As evident from Figure 2, HotStuff clients face severe delay in receiving a response for their client requests.

Finally, the results clearly show that GeoBFT scales well with an increase in regions. When running at a single cluster, the added overhead of GeoBFT (as compared to Pbft) is high, which limits its throughput in this case. Fortunately, GeoBFT is the only protocol that actively benefits from adding regions: adding regions implies adding clusters, which GeoBFT uses to increase parallelism of consensus and decrease centralized communication. This added parallelism helps offset the costs of inter-cluster communication, even when remote regions are added. Similarly, adding remote regions only incurs a low latency on GeoBFT.

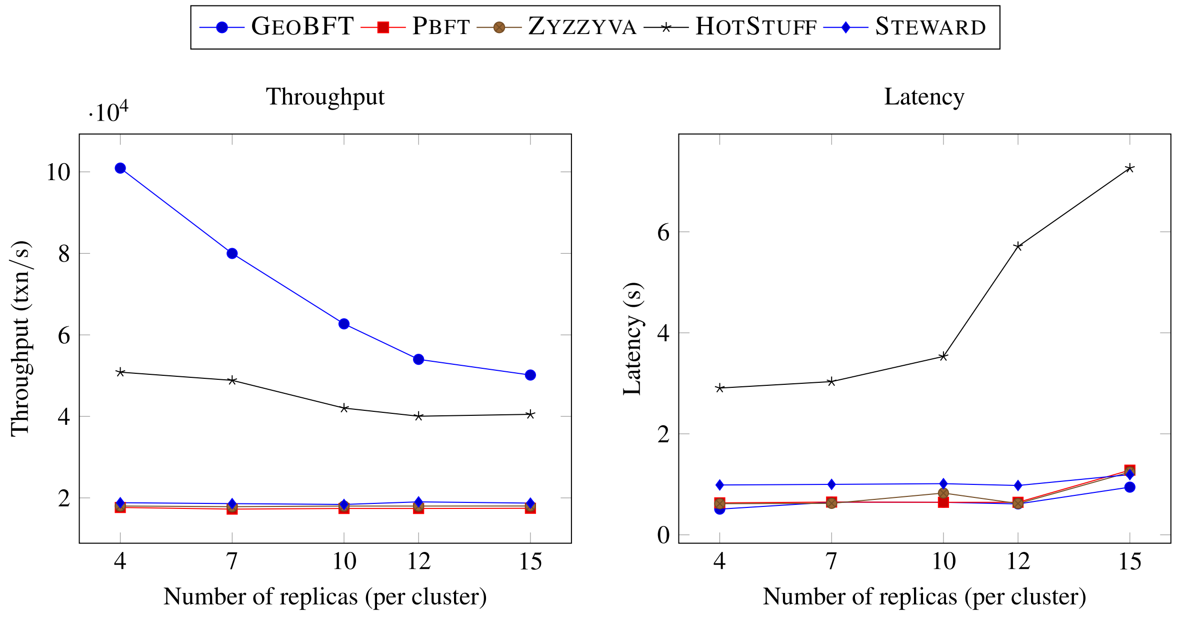


Figure 12: Throughput and latency as a function of the number of replicas per cluster; $\mathbf{z} = 4$.

Recall that GeoBFT sends only $\mathbf{f}+1$ messages between any two clusters. Hence, a total of $\mathcal{O}(\mathbf{zf})$ inter-cluster messages are sent, which is much less than the number of messages communicated across clusters by other protocols (see Figure 2). As the cost of communication between remote clusters is high (see Figure 1), this explains why other protocols have lower throughput and higher latencies than GeoBFT. Indeed, when operating on several regions, GeoBFT is able to outperform Pbft by a factor of up-to-3.1$\times$ and outperform HotStuff by a factor of up-to-1.3$\times$.

2.4.3 Impact of Local Cluster Size

Next, we determine the impact of the number of replicas per region on the performance of GeoBFT and other consensus protocols. To do so, we measure the throughput and latency attained by ResilientDB as a function of the number of replicas per region, which we vary between 4 and 15. We have replicas in four regions (Oregon, Iowa, Montreal, and Belgium). The results of our measurements can be found in Figure 12.

The measurements show that increasing the number of replicas only has minimal negative influence on the throughput and latency of Pbft, Zyzzyva, and Steward. As seen in the previous Section, the inter-cluster communication cost for the primary to contact individual replicas in other regions (and continents) is the main bottleneck. Consequently, the number of replicas used only has minimal influence. For HotStuff, which does not have such a bottleneck, adding replicas does affect throughput and—especially—latency, this due to the strong dependence between latency and the number of replicas in the design of HotStuff.

The design of GeoBFT is particularly tuned toward a large number of regions (clusters), and not toward a large number of replicas per region. We observe that increasing the replicas per cluster also allows each cluster to tolerate more failures (increasing $\mathbf{f}$). Due to this, the performance drop-off for GeoBFT when increasing the replicas per region is twofold: first, the size of the certificates exchanged between clusters is a function of $\mathbf{f}$; second, each cluster sends their certificates to $\mathbf{f}+1$ replicas in each other cluster. Still, the parallelism incurred by running in four clusters allows GEOBFT to outperform all other protocols, even when scaling up to fifteen replicas per region, in which case it is still 2.9× faster than PBFT and 1.2× faster than HOTSTUFF.

2.4.4 Impact of Failures

In our third experiment, we determine the impact of replica failures on the performance of GEOBFT and other consensus protocols. To do so, we measure the throughput attained by ResilientDB as a function of the number of replicas, which we vary between 4 and 12. We perform the measurements under three failure scenarios: a single non-primary replica failure, up to **f** simultaneous non-primary replica failures per region, and a single primary failure. As in the previous experiment, we have replicas in four regions (Oregon, Iowa, Montreal, and Belgium). The results of our measurements can be found in Figure 13.

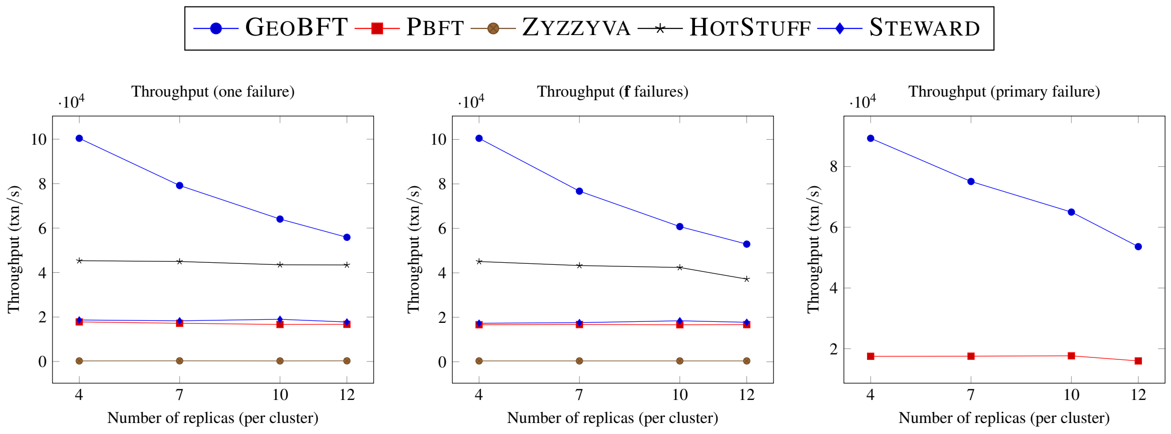


Figure 13: Throughput as a function of the number of replicas per cluster; $\mathbf{z} = 4$. *Left*, throughput with one non-primary failure. *Middle*, throughput with **f** non-primary failures. *Right*, throughput with a single primary failure.

Single non-primary replica failure The measurements for this case show that the failure of a single non-primary replica has only a small impact on the throughput of most protocols. The only exception being ZYZZYVA, for which the throughput plummets to zero, as ZYZZYVA is optimized for the optimal non-failure case. The inability of ZYZZYVA to effectively operate under any failures is consistent with prior analysis of the protocol [3, 48].

f non-primary replica failures per cluster In this experiment, we measure the performance of GEOBFT in the worst case scenario it is designed for: the simultaneous failure of **f** replicas in each cluster (**fz** replicas in total). This is also the worst case STEWARD can deal with, and is close to the worst case the other protocols can deal with (see Remark 2.1).

The measurements show that the failures have a moderate impact on the performance of all protocols (except for ZYZZYVA which, as in the single-failure case, sees its throughput plummet to zero). The reduction in throughput is a direct consequence of the inner working of the consensus protocols. Consider, e.g., GEOBFT. In GEOBFT, replicas in each cluster first choose a local client request and replicate this request locally using PBFT (see Section 2.3.2). In each such local replication step, each replica will have two phases in which it needs to receive $\mathbf{n}-\mathbf{f}$ identical messages before proceeding to the next phase (namely, PREPARE and COMMIT messages). If there are no failures, then each replica only must wait for the $\mathbf{n}-\mathbf{f}$ fastest messages and can proceed to the next phase as soon as these messages are received (ignoring any delayed messages). However, if there are $\mathbf{f}$ failures, then each replica must wait for all messages of the remaining non-failed replicas to arrive before proceeding, including the slowest arriving messages. Consequently, the impact of temporary disturbances causing random message delays at individual replicas increases with the number of failed replicas, which negatively impacts performance. Similar arguments also hold for PBFT, STEWARD, and HOTSTUFF.

Single primary failure In this experiment, we measure the performance of GEOBFT if a single primary fails (in one of the four regions). We compare the performance of GEOBFT with PBFT under failure of a single primary, which will cause primary replacement via a view-change. For PBFT, we require checkpoints to be generated and transmitted after every 600 client transactions. Further, we perform the primary failure after 900 client transactions have been ordered.

For GEOBFT, we fail the primary of the cluster in Oregon once each cluster has ordered 900 transactions. Similarly, each cluster exchanges checkpoints periodically, after locally replicating every 600 transactions. In this experiment, we have excluded ZYZZYVA, as it already fails to deal with non-primary failures, HOTSTUFF, as it utilizes rotating primaries and does not have a notion of a fixed primary, and STEWARD, as it does not provide a readily-usable and complete view-change implementation. As expected, the measurements show that recovery from failure incurs a small reduction in overall throughput in both protocols, as both protocols are able to recover to normal-case operations after failure.

2.4.5 Impact of Request Batching

We now determine the impact of the batch size—the number of client transactions processed by the consensus protocols in a single consensus decision—on the performance of various consensus protocols. To do so, we measure the throughput attained by ResilientDB as a function of the batch size, which we vary between 10 and 300. For this experiment, we have replicas in four regions (Oregon, Iowa, Montreal, and Belgium), and each region has seven replicas. The results of our measurements can be found in Figure 14.

The measurements show a clear distinction between, on the one hand, PBFT, ZYZZYVA, and STEWARD, and, on the other hand, GEOBFT and HOTSTUFF. Note that in PBFT, ZYZZYVA, and STEWARD a single primary residing in a single region coordinates all consensus. This highly centralized communication limits throughput, as it is bottlenecked by the bandwidth of the single primary. GEOBFT—which has primaries in each region—and HOTSTUFF—which rotates primaries—both distribute consensus over several replicas in several regions, removing bottlenecks due to the bandwidth of any single replica. Hence, these protocols have sufficient bandwidth to support larger batch sizes (and increase throughput). Due to this, GEOBFT is able to outperform PBFT by up-to-6.0$\times$. Additionally, as the design of GEOBFT is optimized to minimize global bandwidth usage, GEOBFT is even able to outperform HOTSTUFF by up-to-1.6$\times$.

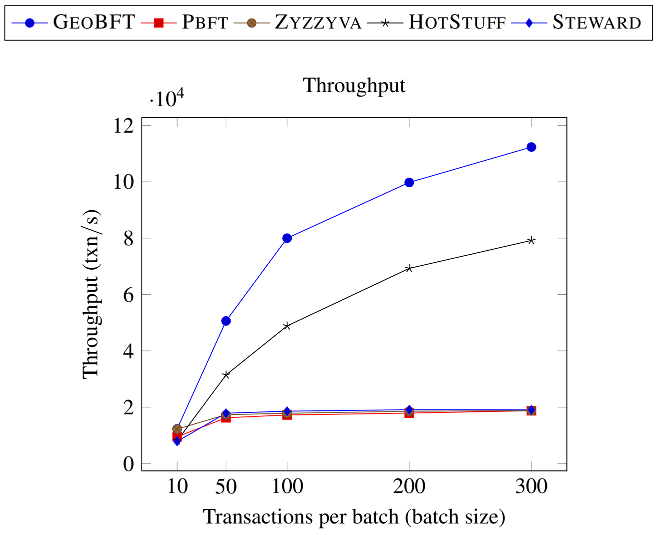


Figure 14: Throughput as a function of the batch size; $\mathbf{z} = 4$ and $\mathbf{n} = 7$.

3 RingBFT: High Throughput Resilient Sharding

Recent works have illustrated a growing interest in *federated data management* [173, 35, 61, 188, 186, 76]. In a federated system, a common database is maintained by several parties. These parties need to reach a *consensus* on the fate of every transaction that is committed to database. Such a database managed by multiple parties raises concerns for data-privacy, data-quality, resistance against adversaries, and database availability and consistency [102, 162, 130].

To facilitate secure multi-party data-management, prior works have proposed the use of blockchain technology [102, 130, 11, 129]. Blockchain technology employs age-old database principles to facilitate a democratic and failure-resilient consensus among several participants. Such a democratic system allows all parties to maintain a *copy* of the common database–act as a *replica*–and cast a *vote* on the fate of any transaction. Hence, at the core of any federated data management system is a *Byzantine-Fault Tolerant* (BFT) consensus protocol that aims to order all the client transactions in the same manner across all the replicas, this despite of any malicious attacks. Once a transaction is ordered, it is recorded in a block, and this block is linked to the previous block, essentially making this ever-growing chain of blocks as immutable–blockchain.

At a closer look, BFT consensus protocols are resilient counterparts of the Two-phase commit and Three-phase commit protocols [87, 177, 103]. Commit protocols robustly handle failure of few participants but block under delay or loss of communicating messages. To eliminate these limitations, prior works have employed crash-fault tolerant protocols such as Paxos and Raft [124, 147]. Although these crash-fault tolerant protocols ensure that replicas reach a safe consensus under *crash* failures, they are unable to shield the system against byzantine attacks. In federated systems, byzantine attacks are common as malicious participants may wish: (i) to exclude transactions of some clients, (ii) to make system unavailable to clients, and (iii) to make replicas inconsistent. Hence, the use of a BFT protocol is order.

In this chapter, we present a novel BFT protocol RINGBFT that aims to be secure against byzantine attacks, achieves high throughputs, and incurs low latencies. Our RINGBFT protocol explores the landscape of sharded-replicated databases, and helps to scale *permissioned* blockchains. RINGBFT aims to make consensus inexpensive even when transactions require access to *multiple shards*. In the rest of this section, we motivate the need for our design choices.

3.1 Challenges for Efficient BFT Consensus

Over the past two decades, BFT protocols have also gone through a series of modifications to guarantee resilience against byzantine attacks, while ensuring high throughputs and low latency. The seminal work by Castro and Liskov [40, 41] led to the design of first practical BFT protocol, PBFT, which advocates a *primary-backup* paradigm where primary initiates the consensus and all the backups follow primary's lead. PBFT achieves consensus among the replicas in three phases, of which two require quadratic communication complexity. Since PBFT's inception, several exciting primary-backup protocols, such as ZYZZYVA [122], SBFT [84], and POE [91], have been proposed. Prior works [6, 97] have illustrated that these *single* primary protocols are essentially *centralized* and prevent scaling the system to a large number of replicas. Further, if the replicas are separated by geographically large distances, then these protocols incur low throughput and high latencies due to low bandwidth and high round-trip time.

These challenges with single primary protocols led to the introduction of modern consensus protocols, such as *Proof-of-Work* [113, 72], *Proof-of-Stake* [24, 23], and *Proof-of-Capacity* [73, 16], which are employed by *permissionless* blockchain applications. As the name suggest, permissionless applications allow any participant to act as a replica without disclosing its identity to other replicas. Hence, the protocols employed by these applications are either compute-expensive or permit state of the replicas to diverge[2] until recovery. Hence, in this work, we focus on permissioned blockchain applications where identity of the participating replicas are known prior to the start of the consensus.

An alternate solution in permissioned spaced is to employ *multi-primary* protocols like RCC [94, 95] that permit all replicas to act as primaries by running multiple consensuses concurrently. However, multi-primary protocols also face scalability limitations as despite concurrent consensuses, each transaction requires communication between all the replicas. This had led to the design of *topology-aware* protocols, such as STEWARD [6] and GEOBFT [97], which cluster replicas based on their geographical distances. GEOBFT expects each cluster to first locally order its client transactions by running the PBFT protocol, and then exchange this ordered transaction with all the other clusters. Although GEOBFT is highly scalable, it necessitates total replication, which forces communicating large messages among geographically distant replicas.

3.2 The Landscape for Sharding

To mitigate the costs associated with replicated databases, a common strategy is to employ *sharded-replicated* paradigm [149]. In a sharded-replicated database, the data is distributed across a set of *shards* where each shard manages a unique *partition* of the data. Further, each shard replicates its partition of data to ensure availability under failures. If each transaction accesses only one shard, then these sharded systems can fetch high throughputs as consensus is restricted to a subset of replicas.

AHL [58] was the first permissioned blockchain system to employ principles of sharding. AHL's seminal design helps to scale blockchain systems to hundreds of replicas across the globe and achieve high throughputs for single-shard transactions. To tackle *cross-shard* transactions that require access to data in multiple shards, AHL designates a set of replicas as a *reference committee*, which globally orders all such transactions. Following AHL's design, SHARPER [8] presents a sharding protocol that eliminates the need for a reference committee by necessitating global communication among all the participating shards.

Decades of research in database community has illustrated that *cross-shard* transactions are common [63, 54, 203, 183, 105, 140]. In fact, heavy presence of these cross-shard transactions has led to development of several concurrency control [25, 26, 105] and commit protocols [87, 177, 103]. Hence, in this chapter, we present our RINGBFT protocol that significantly reduces the costs associated with cross-shard transactions.

RINGBFT envisions each shard participating in *multiple circular flows*, simultaneously. For each cross-shard transaction, RINGBFT follows the principle of *process and forward*. This implies that each shard performs consensus on the transaction and forwards it to the next shard. This flow continues until each shard is aware of the fate of the transaction. However, the real challenge with cross-shard transactions is to manage *conflicts* and to prevent *deadlocks*, which RINGBFT achieves by *storing multiple sequences*.

3.2.1 Cross-Shard Dilemma

For any sharding system, ordering a single-shard is trivial. As a single-shard transaction requires access to only one shard, ordering such a transaction across all the replicas of the shard requires running a standard

[2]In blockchain terminology, such divergence is stated as a fork in the chain where two or more replicas become temporarily inconsistent

BFT protocol. Further, single-shard transactions support *parallelism* as each shard can order its transaction in parallel, this without any communication between shards.

On the other hand, cross-shard transactions are complex. Not only do they require communication between shards but also their fate depends on the consent of each of the involved shards. Further, two or more cross-shard transactions can *conflict* if they require access to same data. Such conflicts can cause one or more transactions to abort or worse can create a *deadlock*. Hence, we need an efficient protocol to order these cross-shard transactions, which ensures that the system is both *safe* and *live*.

Designated Committee. One of the ways to order cross-shard transactions is to designate a set of replicas with this task. AHL [58] defines a reference committee that assigns an order to each cross-shard transaction, which requires running PBFT protocol among all the members of the reference committee. Next, reference committee members run the Two-phase commit (2PC) protocol with all the replicas of involved shard. Notice that the 2PC protocol requires: (1) each shard to send a vote to the reference committee, (2) reference committee collects these votes and takes a decision (abort or commit), and (3) each shard implements the decision. Firstly, this solution requires each shard to run the PBFT protocol to decide on the vote. Secondly, reference committee needs to again run PBFT to reach a common decision. Finally, not only there are multiple phases of communication between geo-distributed shards, but also quadratic communication as each member of reference committee is transmitting messages to all the replicas of each shard.

3.3 System Model

To explain our RINGBFT protocol in detail, we first lay down some notations and assumptions. Our system comprises of a set $\mathfrak{S}$ of shards where each shard S provides a replicated service. In specific, each shard S manages a *unique partition of the data*, which is replicated by a set $\mathfrak{R}\mathrm{S}$ of *replicas*.

In each shard S, there are $\mathsf{f}(\subseteq)\mathfrak{R}\mathrm{S}$ *byzantine* replicas, of which $\mathsf{nf}(=)\mathfrak{R}\mathrm{S} \setminus \mathsf{f}()$ are *non-faulty* replicas. We expect non-faulty replicas to follow the protocol and act deterministic, that is, on identical inputs, all non-faulty replicas must produce identical outputs. We write $\mathbf{z} = |\mathfrak{S}|$ to denote the total number of shards and $\mathbf{n} = |\mathfrak{R}\mathrm{S}|$, $\mathbf{f} = |\mathsf{f}(|)$, and $\mathbf{nf} = |\mathsf{nf}(|)$ to denote the number of replicas, faulty replicas, and non-faulty replicas, respectively, in each shard.

Fault-Tolerance Requirement. Traditional, BFT protocols such as PBFT, ZYZZYVA, and SBFT expect a total replicated system where the total number of byzantine replicas are less than *one-third* of the total replicas in the system. In our sharded-replicated model, we adopt a slightly weaker setting where at each shard the total number of byzantine replicas are less than *one-third* of the total replicas in that shard. In specific, at each shard S, we have $\mathbf{n} \geq 3\mathbf{f}+1$. Notice that this requirement is in accordance with existing works in byzantine sharding space [58, 8, 187, 202].

Cross-Shard Transactions. Each shard $\mathrm{S} \in \mathfrak{S}$ can receive a *single-shard* or cross-shard transaction. A single-shard transaction for S leads to *intra-shard* communication, that is, all the messages necessary to order this transaction are exchanged among the replicas of S. Each cross-shard transaction requires access to data from a subset of shards (henceforth we use the abbreviation CST to refer to a cross-shard transaction). We denote this subset of shards as $\mathfrak{I}$ where $\mathfrak{I} \subseteq \mathfrak{S}$, and refer to it as *involved shards*. Each CST can be termed as *simple* or *complex*. A simple CST is a collection of fragments where each shard can independently run consensus and execute its fragment. On the other hand, a complex CST includes dependencies, that is, an involved shard may require access to data from other involved shards to execute its fragment.

Ring Order. We assumes shards in set $\mathfrak{S}$ are *logically* arranged in a *ring topology*. In specific, each shard

$S \in \mathfrak{S}$ has a position in the ring, which we denote by $\mathrm{id}(S)$, $1 \leq \mathrm{id}(S) \leq |\mathfrak{S}|$. RINGBFT employs these identifiers to specify the flow of a CST or ***ring order***. For instance, a simple ring policy can be that each CST is processed by the involved shards in the increasing order of their identifiers. RINGBFT can also adopt other complex permutations of these identifiers for determining the flow across the ring.

Authenticated Communication. We assume that each message exchanged among clients and replicas is *authenticated*. Further, we assume that byzantine replicas are unable to impersonate non-faulty replicas. Notice that authenticated communication is a minimal requirement to deal with Byzantine behavior. For intra-shard communication, we employ cheap *message authentication codes* (MACs), while for cross-shard communication we employ digital signatures (DS) to achieve authenticated communication. MACs are a form of symmetric cryptography where each pair of communicating nodes shares a *secret key*. We expect non-faulty replicas to keep their *secret keys* hidden. DS follow asymmetric cryptography. In specific, prior to signing a message, each replica generates a pair of *public-key* and *private-key*. The signer keeps the private-key hidden and uses it to sign a message. Each receiver authenticates the message using the corresponding public-key.

In the rest of this manuscript, if a message m is signed by a replica Rr using DS, we represent it as $\langle Rr \rangle_m$ to explicitly identify replica Rr. Otherwise, we assume that the message employs MAC.

To ensure message integrity, we employ a *collision-resistant cryptographic hash function* $\mathtt{D}(\cdot)$ that maps an arbitrary value v to a constant-sized digest $\mathtt{D}(v)$ [120]. We assume that there is a negligible probability to find another value v', $v \neq v'$, such that $\mathtt{D}(v) = \mathtt{D}(v')$. Further, we refer to a message as *well-formed* if a non-faulty receiver can validate the DS or MAC, verify the integrity of message digest, and determine that the sender of the message is also the creator.

3.4 RingBFT Consensus Protocol

To achieve efficient consensus in sharded-replicated databases, we employ our RINGBFT protocol. While designing our RINGBFT protocol, we set following goals:

(G1) Inexpensive consensus of single-shard transactions.
(G2) Flexibility of employing different existing consensus protocols for intra-shard consensus.
(G3) Efficient consensus of cross-shard transactions.
(G4) Cheap communication between globally-distributed shards.

Next, we define the *safety* and *liveness* guarantees provided by our RINGBFT protocol.

Definition 3.1. Let $\mathfrak{S}$ be a system of shards and $\mathfrak{R}S$ be a set of replicas in some shard $S \in \mathfrak{S}$. Each run of a *consensus protocol* in this system should satisfy the following two requirements:

Involvement Each $S \in \mathfrak{S}$ only processes a transaction if $S \in \mathfrak{I}$.
Termination Each non-faulty replica in $\mathfrak{R}S$ executes a transaction.
Non-divergence All non-faulty replicas in $\mathfrak{R}S$ execute the same transaction.
Consistence Each non-faulty replica in $\mathfrak{S}$ executes a conflicting transaction in same order.

In traditional replicated systems, non-divergence implies *safety*, while termination implies *liveness*. For a sharded-replicated system like RINGBFT, we need stronger guarantees. If a transaction requires access to only one shard, safety is provided by involvement and non-divergence, while termination sufficiently guarantees liveness. For a cross-shard transaction, to guarantee safety, we also need consistence apart from involvement and non-divergence, while liveness is provided using involvement and termination.

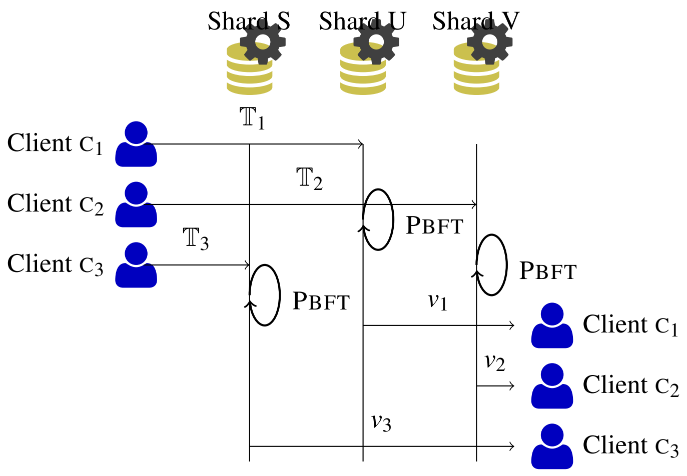


Figure 15: An illustration of how RINGBFT manages single-shard transactions. Each of the three shards S, U, and V receive transactions $\mathbb{T}_1$, $\mathbb{T}_2$, and $\mathbb{T}_3$ from their respective clients c_1, c_2, and c_3 to execute. Each shard independently run PBFT consensus, and sends responses to respective clients.

RINGBFT guarantees safety in asynchronous setting. In such a setting, messages may get lost, delayed or duplicated, and up to **f** replicas may act byzantine. However, RINGBFT can only provide liveness during periods of synchrony. Notice that these assumptions are no harder than those required by existing protocols [40, 58, 8].

3.4.1 Consensus of Single-Shard Transactions

To order and execute single-shard transactions is trivial. For this task, RINGBFT employs one of the many available primary-backup consensus protocols and runs them at each shard. In the rest of this section, for the sake of explanation, we assume that RINGBFT employs the PBFT consensus protocol to order single-shard transactions. We use the following example to explain RINGBFT's single-shard consensus.

Example 3.2. Assume a system that comprises of three shards S, U, and V. Say client c_1 sends $\mathbb{T}_1$ to S, c_2 sends $\mathbb{T}_2$ to U, and client c_3 sends $\mathbb{T}_3$ to V. On receiving the client transaction, the primary of each shard initiates the PBFT consensus protocol among its replicas. Once each replica successfully orders the transaction, it sends a response to the client. Such a flow is depicted in Figure 15.

It is evident from Example 3.2 that there is no communication among the shards. This is the case because each transaction requires access to data available inside only one shard. Hence, ordering single-shard transactions for shard S requires running the PBFT protocol among the replicas of S without any synchronization with other shards. For the sake of completeness, we present the single-shard consensus based on PBFT protocol in brief next.

Request. When client c wants to execute a transaction $\mathbb{T}$, it creates a $\langle c \rangle_{\mathbb{T}}$ and sends it to the primary p of shard S that has access to corresponding data.

Pre-prepare. When p receives message $m := \langle c \rangle_{\mathbb{T}}$ from the client, it checks if the message is well-formed. If this is the case, p creates a message PREPREPARE(m, Δ, k) and broadcasts it to all the replicas of shard . This PREPREPARE message includes: (1) sequence number k that specifies the order for this transaction, and

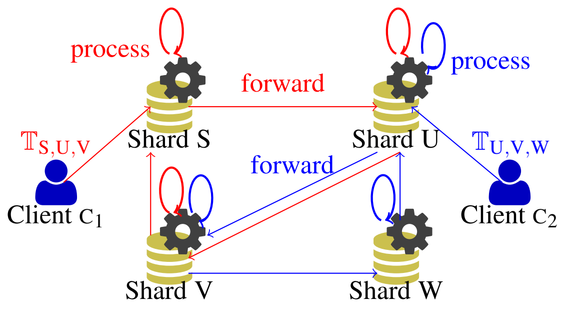


Figure 16: An illustration of how RINGBFT employs the principle of Process and Forward to order two cross-shard transactions $\mathbb{T}_{S,U,V}$ and $\mathbb{T}_{U,V,W}$ across four shards.

(2) digest $\Delta = D(\langle c \rangle_{\mathbb{T}})$ of the client transaction which will be used in future communication to reduce data communicated across network.

Prepare. When a replica R of shard S receives a PREPREPARE message from its primary, it checks if the message is well-formed. If this is the case, the replica R agrees to support P's order for m by sending PREPARE(Δ, k) to all the replicas of S.

Commit. When R receives identical PREPARE messages (and are also well-formed) from at least **nf** replicas of S, it achieves a weak guarantee that majority of non-faulty replicas have also agreed to support P's order for m. Hence, it marks this request as *prepared*, creates a COMMIT(Δ, k) message, and broadcasts this message.

Reply. When R receives identical COMMIT messages (and are also well-formed) from at least **nf** replicas of S, it achieves a strong guarantee that majority of non-faulty replicas have also *prepared* this request. Hence, it executes transaction $\mathbb{T}_T$, after all the $(k-1)$-th transactions have been executed and replies to the client C.

We use the following lemma to illustrate the safety of the above protocol for ordering single-shard transactions:

Proposition 3.3. *Let* R_i, $i \in \{1,2\}$, *be two non-faulty replicas in shard* S *that committed to* $\langle c_i \rangle_{\mathbb{T}_i}$ *as the k-th transaction sent by* P. *If* $\mathbf{n} > 3\mathbf{f}$, *then* $\langle c_1 \rangle_{\mathbb{T}_1} = \langle c_2 \rangle_{\mathbb{T}_2}$.

Proof. Replica R_i only committed to $\langle c_i \rangle_{\mathbb{T}}$ after R_i received identical COMMIT(Δ, k) messages from **nf** distinct replicas in S. Let X_i be the set of such **nf** replicas and $Y_i = X_i \setminus f()$ be the non-faulty replicas in X_i. As $|f(|) = \mathbf{f}$, so $|Y_i| \geq \mathbf{nf} - \mathbf{f}$. We know that each non-faulty replica only supports one transaction from primary P as the k-th transaction, and it will send only one PREPARE message. This implies that sets Y_1 and Y_2 must not overlap. Hence, $|X_1 \cup X_2| \geq 2(\mathbf{nf} - \mathbf{f})$. As $|X_1 \cup X_2| = \mathbf{nf}$, the above inequality simplifies to $3\mathbf{f} \geq \mathbf{n}$, which contradicts $\mathbf{n} > 3\mathbf{f}$. Thus, we conclude $\langle c_1 \rangle_{\mathbb{T}_1} = \langle c_2 \rangle_{\mathbb{T}_2}$. □

3.4.2 Consensus of Cross-Shard Transactions: Process and Forward

In this section, we illustrate how RINGBFT guarantees efficient consensus of cross-shard transactions. An efficient solution to cross-shard transactions not only helps us to meet the key goal for this work, but also permits scaling a sharded-replicated system across wide-area network. In principle, RINGBFT introduces a *process-and-forward* paradigm to order cross-shard transactions. We use the following example to illustrate RINGBFT's cross-shard consensus.

Example 3.4. Assume a system that comprises of four shards S, U, V, and W. Say client c_1 wants to process a transaction $\mathbb{T}_{S,U,V}$ that requires access to data from shards S, U, and V. Similarly, Say client c_2 wants to process a transaction $\mathbb{T}_{U,V,W}$ that requires access to data from shards U, V, and W. In this case, client c_1 sends its transaction to primary of shard S while c_2 sends its transaction to primary of U. On receiving $\mathbb{T}_{S,U,V}$, replicas of S process the transaction and forward it to replicas of U. Next, replicas of U process $\mathbb{T}_{S,U,V}$ and forward it to replicas of V. Finally, replicas of V process $\mathbb{T}_{S,U,V}$ and send it back to replicas of S, which reply to client c_1. Similar flow takes place while ordering transaction $\mathbb{T}_{U,V,W}$. We illustrate these flows in Figure 16.

Although Example 3.4 illustrates RINGBFT's process-and-forward paradigm, it raises following important questions:

(Q1) Can a shard concurrently order multiple cross-shard transactions?
(Q2) How does RINGBFT handle conflicting transactions?
(Q3) Can shards running RINGBFT protocol deadlock?
(Q4) How much communication is required between two shards?

To answer these questions, we explain RINGBFT's process-and-forward paradigm next.

Cross-shard Transactional Flow RINGBFT assumes shards are arranged in a logical ring. For the sake of explanation, we assume the ring order of *lowest to highest identifier*. We denote the first shard in a ring order as the ***initiator shard***, which is responsible for starting consensus on the client transaction. To safely execute a cross-shard transaction (CST), each shard may have to perform the tasks of process-and-forward multiple times. This implies that each CST may require one or more *rotations* across the ring. If a CST is simple, then a single rotation is sufficient to ensure each involved shard $S \in \mathfrak{I}$ safely executes its fragment. Otherwise, a CST may require at least *two* rotations across the ring.

Prior to presenting the RINGBFT's consensus protocol that safely orders each CST, we sketch the flow of a CST in Figure 18. In this figure, we assume a system of four shards: S, U, V, and W where $\text{id}(S) < \text{id}(U) < \text{id}(V) < \text{id}(W)$. The client creates a transaction $\mathbb{T}_{S,U,W}$ that requires access to data in shards S, U, and W and sends this transaction to the primary P_S of S. On receiving this transaction, P_S initiates the PBFT consensus protocol (*local replication*) among its replicas. If the local replication is successful, then all the replicas of S ***lock*** the corresponding data. This locking of data-items is the key to preventing deadlocks. Next, replicas of S forward the transaction to replicas of shard U. Notice that only *linear communication* takes place between replicas of S and U. Hence, to handle any failures, replicas of U share this message among themselves. Next, replicas of U also follow similar steps and forward transaction to W. As W is the last shard in the ring of involved shards, it goes ahead and executes the CST if all the dependencies are met. Finally, replicas of shards S and U also execute the transaction and replicas of S send the result of execution to the client.

```
Initialization:
// $k_{\max}$ :=0 (maximum sequence number in shard S)
// $\Sigma^{\mathfrak{I}} := \emptyset$ (set of data-fragments of each shard)
// $\pi := \emptyset$ (list of pending transactions at a replica)

Client-role (used by client $c$ to request transaction $\mathbb{T}_{\mathfrak{I}}$) :
1:  Sends $\langle c \rangle_{\mathbb{T}_{\mathfrak{I}}}$ to the primary $P_S$ of shard S.
2:  Awaits receipt of messages Response($\langle c \rangle_{\mathbb{T}_{\mathfrak{I}}}, k, r$) from $\mathbf{f}+1$ replicas of S.
3:  Considers $\mathbb{T}_{\mathfrak{I}}$ executed, with result $r$, as the $k$-th transaction.

Primary-role (running at the primary $P_S$ of shard S) :
4:  event $P_S$ receives $\langle c \rangle_{\mathbb{T}_{\mathfrak{I}}}$ do
5:    if $S \in \mathfrak{I} \wedge \mathrm{id}(S) = $ FirstInRingOrder($\mathfrak{I}$) then
6:      Calculate digest $\Delta := D(\langle c \rangle_{\mathbb{T}})$.
7:      Broadcast Preprepare($\langle c \rangle_{\mathbb{T}_{\mathfrak{I}}}, \Delta, k$) in shard S (order at sequence $k$).
8:    else
9:      Forward to primary $P_U$ of shard U, $U \in \mathfrak{S} \wedge \mathrm{id}(U) = $ FirstInRingOrder($\mathfrak{I}$)

Non-Primary Replica-role (running at the replica $R$ of shard S) :
10: event $R$ receives Preprepare($\langle c \rangle_{\mathbb{T}_{\mathfrak{I}}}, \Delta, k$) from $P_S$ such that:
        1. message is well-formed; and
        2. $R$ did not accept a $k$-th proposal from $P_S$.
    do
11:   Broadcast Prepare($\Delta, k$) to replicas in $\mathfrak{R}S$.

Replica-role (running at the replica $R$ of shard S) :
12: event $R$ receives nf message Prepare($\Delta, k$) such that:
        1. each message is well-formed and is sent by a distinct replica, $Q \in \mathfrak{R}S$.
    do
13:   Broadcast $\langle R \rangle_{\text{Commit}(\Delta,k)}$ to replicas in $\mathfrak{R}S$.

14: event $R$ receives nf $m := \langle Q \rangle_{\text{Commit}(\Delta,k)}$ messages such that:
        1. each message $m$ is well-formed and is sent by a distinct replica $Q \in \mathfrak{R}S$.
    do
15:   U be the shard to forward such that $\mathrm{id}(U) = $ NextInRingOrder($\mathfrak{I}$).
16:   $A$ := set of DS of these nf messages.
17:   if $k = k_{\max} + 1$ // Forward to next shard then
18:     Lock data-fragment corresponding to $\langle c \rangle_{\mathbb{T}_{\mathfrak{I}}}$.
19:     Send $\langle R \rangle_{\text{Forward}(\langle c \rangle_{\mathbb{T}_{\mathfrak{I}}}, A, m, \Delta,)}$ to replica $O$, where $O \in \mathfrak{R}U \wedge \mathrm{id}(R) = \mathrm{id}(O)$
20:   else
21:     Store $\langle R \rangle_{\text{Forward}(\langle c \rangle_{\mathbb{T}_{\mathfrak{I}}}, A, m, \Delta,)}$ in $\pi$.
22:   while $\pi! = \emptyset$ // Pop out waiting transaction. do
23:     Extract transaction at $k_{\max} + 1$ from $\pi$ (if any).
24:     if Corresponding data-fragment is not locked then
25:       $k_{\max} = k_{\max} + 1$
26:       Follow lines 18 and 19.
27:     else
28:       Store transaction at $k_{\max}$ in $\pi$.
29:       Exit the loop.

    // Locally share any message from previous shard.
30: event $R$ receives message $m := \langle Q \rangle_{\text{message-type}}$ such that:
        1. $m$ is well-formed and sent by replica $Q$, where
               $\mathrm{id}(U) = $ PrevInRingOrder($\mathfrak{I}$), $Q \in \mathfrak{R}U \wedge \mathrm{id}(R) = \mathrm{id}(Q)$
    do
31:   Broadcast $m$ to all replicas in S.

    // Forward message from previous shard.
32: event $R$ receives $\mathbf{f}+1$ $m' := \langle Q \rangle_{\text{Forward}(\langle c \rangle_{\mathbb{T}_{\mathfrak{I}}}, A, m, \Delta)}$ such that:
        1. $m'$ is well-formed.
        2. set $A$ includes valid DS from nf replicas corresponding to $m$.
    do
33:   if Data-fragment corresponding to $\langle c \rangle_{\mathbb{T}_{\mathfrak{I}}}$ is locked // Second Rotation then
34:     Execute data-fragment of $\langle c \rangle_{\mathbb{T}_{\mathfrak{I}}}$ and add to log.
35:     Push result to set $\Sigma^{\mathfrak{I}}$.
36:     Release the locks from corresponding data-fragment.
37:     V be the shard to forward such that $\mathrm{id}(V) = $ NextInRingOrder($\mathfrak{I}$).
38:     Send $\langle R \rangle_{\text{Execute}(\Delta, \Sigma^{\mathfrak{I}})}$ to replica $O$, where $O \in \mathfrak{R}V \wedge \mathrm{id}(R) = \mathrm{id}(O)$.
39:   else if $R = P_S$ // Primary initiates consensus then
40:     Broadcast Preprepare($\langle c \rangle_{\mathbb{T}_{\mathfrak{I}}}, \Delta, k'$) in shard S (order at sequence $k'$).

41: event $R$ receives $m' := \langle Q \rangle_{\text{Execute}(\Delta, \Sigma^{\mathfrak{I}})}$ such that:
        1. $m'$ is sent by replica $Q$, where $Q \in \mathfrak{R}U \wedge \mathrm{id}(R) = \mathrm{id}(Q)$
    do
42:   if Already executed $\langle c \rangle_{\mathbb{T}_{\mathfrak{I}}}$ // Reply to client then
43:     Send client $c$ the result $r$.
44:   else
45:     Follow lines 34 to 38.
```

Figure 17: The normal-case algorithm of RingBFT.

3.4.3 Cross-Shard Consensus Algorithm

We use Figure 17 to present RINGBFT's algorithm for ordering cross-shard transactions. Next, we discuss these steps in detail.

Client Request When a client C wants to process a cross-shard transaction $\mathbb{T}_{\mathfrak{I}}$, it creates a $\langle \text{C} \rangle_{\mathbb{T}_{\mathfrak{I}}}$ message and sends it to the primary of the *first shard in ring order*. As part of this transaction, the client C specifies the information regarding all the involved shards ($\mathfrak{I}$), such as their identifiers and the necessary *read-write* sets of each shard.

Client Request Reception When the primary P_{S} of shard S receives a client request $\mathbb{T}_{\mathfrak{I}}$, it first checks if the message is well-formed. If this is the case, then the primary S checks if among the set of involved shards $\mathfrak{I}$, S is the first shard in ring order. If this condition is met, then P assigns this request a linearly increasing sequence number k, calculates the digest Δ, and broadcasts a PREPREPARE message to all the replicas $\mathfrak{R}$S of its shard. In the case when S is not the first shard in the ring order, P_{S} forwards the transaction to the primary of the appropriate shard.

Pre-prepare Phase When a replica $\text{R} \in \mathfrak{R}\text{S}$ receives the PREPREPARE message from P_{S}, it checks if the request is well-formed. If this is the case and if R has not agreed to support any other request from P_{S} as the k-th request, then it broadcasts a PREPARE message in its shard S.

Prepare Phase When a replica R receives identical PREPARE messages from **nf** distinct replicas, it gets an assurance that a majority of non-faulty replicas are supporting this request. At this point, each replica R broadcasts a COMMIT message to all the replicas in S. Once a transaction passes this phase, the replica R marks it *prepared*.

Commit and Data Locking When a replica R receives well-formed identical COMMIT messages from **nf** distinct replicas in S, it checks if it also prepared this transaction at same sequence number. If this is the case, RINGBFT requires each replica R to *lock* all the read-write sets that transaction $\mathbb{T}_{\mathfrak{I}}$ needs to access in shard S.

In RINGBFT, we allow replicas to process and broadcast PREPARE and COMMIT messages *out-of-order*, but require each replica to acquire locks on data in transactional sequence order. This out-of-ordering helps replicas to continuously perform useful work by concurrently participating in consensus of several transactions. To achieve these tasks, each replica R tracks the maximum sequence number ($k_{\max}$), which indicates the sequence number of the last transaction to lock data. If sequence number k for a transaction $\mathbb{T}_{\mathfrak{I}}$ is greater than $k_{\max} + 1$, we store the transaction in a list π until transaction at $k_{\max} + 1$ has acquired the locks. Once the k-th transaction has acquired locks, we gradually release transactions in π until there is a transaction that wishes to lock already locked data-fragments. We illustrate this through the following example.

Example 3.5. Assume we use the following notations to represent four transactions and the data-fragments they access at shard S: $\mathbb{T}_{1,a}$, $\mathbb{T}_{2,b}$, $\mathbb{T}_{3,a}$, and $\mathbb{T}_{4,c}$. For instance $\mathbb{T}_{1,a}$ implies that transaction at sequence 1 requires access to data-item a. Next, due to out-of-order message processing, assume a replica R in S receives **nf** COMMIT messages for $\mathbb{T}_{2,b}$, $\mathbb{T}_{3,a}$, and $\mathbb{T}_{4,c}$ before $\mathbb{T}_{1,a}$. Hence, $\pi = \{\mathbb{T}_{2,b}, \mathbb{T}_{3,a}, \mathbb{T}_{4,c}\}$. Once R locks data-item a for transaction $\mathbb{T}_{1,a}$, it extracts $\mathbb{T}_{2,b}$ from π. As $\mathbb{T}_{2,b}$ wishes to lock a distinct data-item, so R continues processing $\mathbb{T}_{2,b}$. Next, R moves to $\mathbb{T}_{3,a}$ but it cannot process $\mathbb{T}_{3,a}$ due to lock-conflicts. Hence, it places back $\mathbb{T}_{3,a}$ in π and stops processing transactions in π until lock is available for $\mathbb{T}_{3,a}$.

Forward to next Shard via Linear Communication Once the data corresponding to transaction $\mathbb{T}_{\mathfrak{I}}$ is locked, each R in S sends a FORWARD message to some replica Q of the next shard in ring order. Notice that one of the key goals of RINGBFT is to ensure that this communication between two shards is linear. Hence, we design a communication primitive that builds on top of the optimal bound for communication between two shards [106, 97].

RINGBFT's cross-shard communication primitive can be stated as follows: *if each replicas from shard* S *communicates with a distinct replica in shard* U*, then at least* $\mathbf{f}+1$ *non-faulty replicas from* S *will communicate with* $\mathbf{f}+1$ *non-faulty replicas in* U. How does this help? This communication primitive requires exchanging a total of $\mathbf{n}$ messages (linear) between two shards. Further, it guarantees message delivery to at least $\mathbf{f}+1$ non-faulty replicas, which helps receiving replicas to determine the fate of the transaction.

In specific, we require each of the $\mathbf{n}$ replicas of S to initiate communication with the replicas of U having the same identifier. Hence, replica R of shard S sends a FORWARD message to replica Q in shard U such that $\text{id}(\text{R}) = \text{id}(\text{Q})$. By transmitting a FORWARD message, R is requesting Q to initiate consensus on $\langle \text{C} \rangle_{\mathbb{T}_{\mathfrak{I}}}$. For Q to support such a request, it needs a proof that $\langle \text{C} \rangle_{\mathbb{T}_{\mathfrak{I}}}$ was successfully ordered in shard S. Hence, R includes the DS on COMMIT messages from $\mathbf{nf}$ distinct replicas (Figure 17, Line 16).

Execution and Next Rotation Once a client request has been ordered on all the involved shards, we call it *one complete rotation* around the ring. This is a significant event because it implies that all the necessary data-fragments have been locked by each of the involved shards. In such a case, if the first shard in ring order (S) receives a FORWARD message, replicas of S will attempt to execute parts of transaction, which are a responsibility of S. On successful completion of the execution, replicas in S send EXECUTE messages to replicas in the next shard using the optimal communication primitive. This message includes the updated write sets ($\Sigma^{\mathfrak{I}}$), which can help in resolving any dependencies during execution.

Once the execution is completed across all the shards, the first shard in ring order may again receive an EXECUTE message. At this point, the replicas of S reply to the client with identical responses.

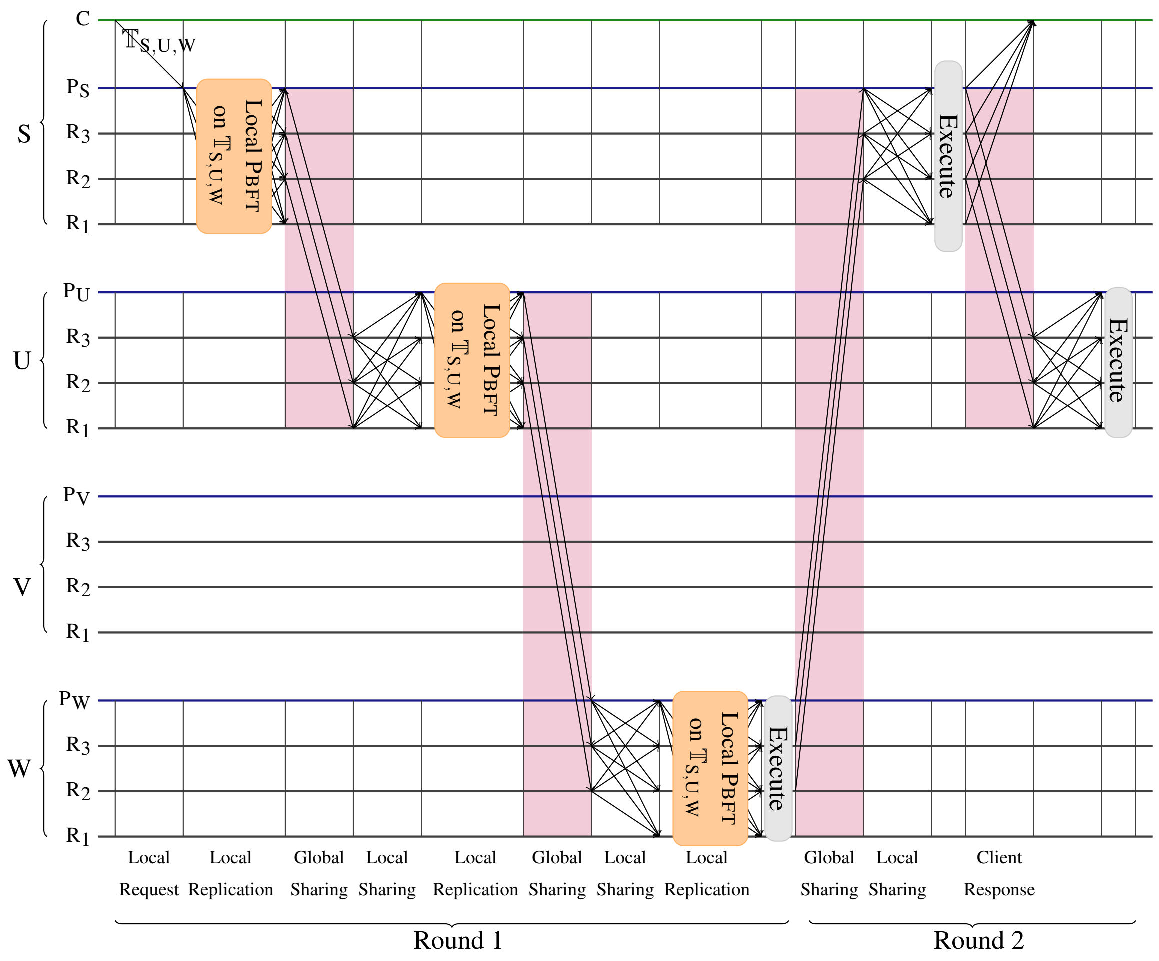


Figure 18: Representation of the normal-case flow of RINGBFT in a system of four shards where client sends a cross-shard transaction $\mathbb{T}_{S,U,W}$ that requires access to data in *three* shards: S, U, and W.

Unified Single and Cross-Shard Consensus Until now, we have presented RINGBFT protocols for handling single-shard and cross-shard transactions, respectively. However, any sharded-replicated system is expected to serve a mixture of transactions, which may include both single-shard and cross-shard transactions. Hence, by making *minor* modifications to Figure 17, we can provide a unified protocol to handle either type of transactions.

In specific, for a single-shard transaction $\mathbb{T}$, when a replica R receives identical COMMIT messages from **nf** distinct replicas in its shard, it attempts to acquire the locks. This only happens if its sequence number k is equal to $k_{\max}+1$, otherwise $\mathbb{T}$ is stored in π. When $k = k_{\max}+1$, then R executes $\mathbb{T}$ and replies to the client. Following this, the loop at Figure 17, Line 22 is run to extract any pending transactions.

The consensus provided by our RINGBFT protocol helps to achieve following guarantees:

Lemma 3.6. *In a system $\mathfrak{S}$ of shards, if at most* **f** *replicas out of* **n** *in each shard $S \in \mathfrak{S}$ are byzantine, then each replica $R \in \mathfrak{R}S$ will execute each transaction $\mathbb{T}_{\mathfrak{I}}$ in the same order if $S \in \mathfrak{I}$.*

Lemma 3.7. *In a system $\mathfrak{S}$ of shards, during periods of synchrony, if at most* **f** *replicas out of* **n** *in each shard*

$S \in \mathfrak{S}$ *are byzantine then each replica* $\text{R} \in \Re S$ *will continue to make progress. In specific, requests from a good client will be processed by* R *if* $S \in \mathfrak{I}$.

Lemma 3.8. *No two replicas* $\text{R} \in S$ *and* $\text{Q} \in U$, $S \neq U$ *that order two conflicting transactions* $\mathbb{T}_{\mathfrak{I}_1}$ *and* $\mathbb{T}_{\mathfrak{I}_2}$ *such that* $\text{R}, \text{Q} \in \mathfrak{I}_1 \cap \mathfrak{I}_2$ *will execute* $\mathbb{T}_{\mathfrak{I}_1}$ *and* $\mathbb{T}_{\mathfrak{I}_2}$ *in different orders.*

Lemma 3.6 and Lemma 3.7 state the *safety* and *liveness* guarantees provided by RINGBFT, respectively. Further, we use Lemma 3.8 to illustrate the *no deadlock* assurance of RINGBFT.

3.5 RINGBFT Evaluation

To evaluate and compare our RINGBFT protocol against various modern sharding protocols, we deployed implementation of RINGBFT in ResilientDB on Google Clout Platform (GPC) in 15 different regions as it is shown in table 3.

Table 3: Regions

	Country/City	**Region**
0	oregon	us-west1
1	iowa	us-central1
2	montreal	northamerica-northeast1
3	netherland	europe-west4
4	taiwan	asia-east1
5	sydney	australia-southeast1
6	singapore	asia-southeast1
7	south-carolina	us-east1
8	north-virginia	us-east4
9	los angeles	us-west2
10	las vegas	us-west4
11	london	europe-west2
12	belgium	europe-west1
13	tokyo	asia-northeast1
14	hong-kong	asia-east2

In experiments with less than 15 shards for we used the first n shard from this table. We used 16-core **N1** machines with 32GB of RAM for the replicas and 4-core **N1** machines with 16GB of RAM for the clients. These machines have Intel Broadwell CPUs with a 2.2GHz clock. We equally distributed clients in regions and shards based on the number of clients and shards.

In each experiment, the workload is provided by the *Yahoo Cloud Serving Benchmark* (YCSB) [53]. Each client transaction queries a YCSB table with an active set of 600 k records. For our evaluation, we use *write queries*, as those are typically more costly than read-only queries. Prior to the experiments, each replica is initialized with an identical copy of the YCSB table. The client transactions generated by YCSB follow a uniform Zipfian distribution. Clients and replicas can batch transactions to reduce the cost of consensus. In our experiments, we use a *batch size* of 100 requests per batch (unless stated otherwise).

Since the experiments and evaluation are in a geo-scaled WAN environment in different countries, the bottleneck for all protocols is inter-region communication in bandwidth and round-trip time. For RINGBFT, the inter-region communication is sending commit-certificates and final commit messages to the next shard in the ring using the reliable inter-cluster communication protocol. Table 4 shows the size of messages when we are using 100 transactions per batch and one operation per transaction.

Table 4: Message Sizes

Message	**Size in Bytes**
Pre-Prepare	5408
Prepare	216
Commit	269
Commit Certificate	6147
Checkpoint	164
Final Commit	1732

For AHL and Sharper, the proximity of shards in their order doesn't matter; however, for RingBFT, since the pattern of communication is a ring, it is crucial how close is the next shard. In order to have a fair comparison, we shuffled the regions to remove the advantage of RingBFT, although, in a real-world setting, RingBFT can benefit from the orientation of shards. According to Table 3 the first shard is always in Oregon, the second one in Iowa, and so on.

We didn't compare RINGBFT with all modern BFT protocols. The main reason for that is the performance gap between sharding and full replicated systems. We have implemented several BFT modern protocols such as RCC[95], SBFT[84], HotStuff[200], Steward[6], POE[91], Zyzzyva[121], and PBFT[40]. We had a brief comparison of these protocols against RINGBFT. The gap was so large, so we skipped having these protocols in the evaluation part and assumed AHL and Sharper, Which are cutting edge sharding protocols, would suffice. The scaling comparison of mentioned protocols is shown in figure 19.

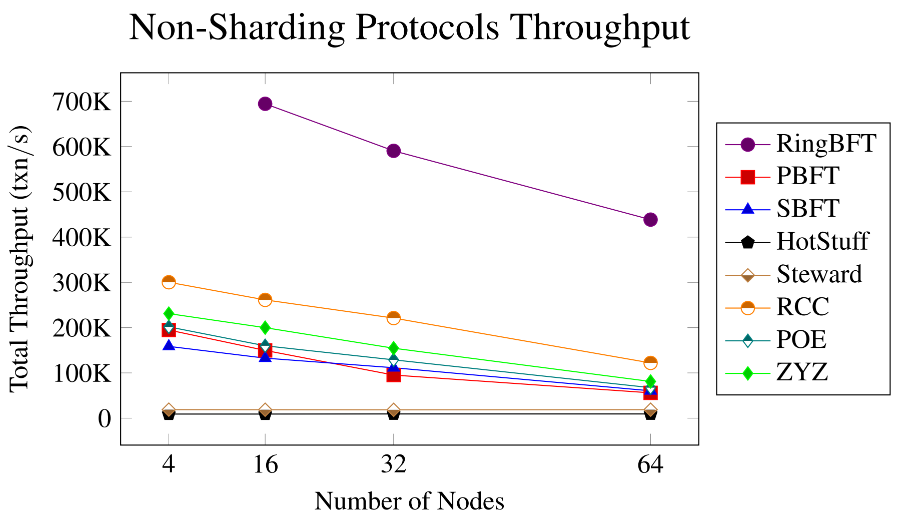


Figure 19: Evaluating Non-Sharding Protocols throughput and average latency

In this section, we will focus on the effect of different setup parameters on throughput and latency; including:

1. Number of shards.
2. Number of replicas per shard.

3. Percentage of coss-shard transactions.
4. Number of involved shards in cross-shard transactions.
5. Batch Size.
6. Number of simultaneous clients (Max Transactions Inflight).

The values of parameters described above used in our experiments are summarized in Table 5. Max Transaction Inflight's value is fine-tuned to prevent overwhelming the system by adding artificial queue time in replicas.

Table 5: Experiments

Number of Shards	**Number of Nodes**	**Cross Shard %**	**Batch size**	**Involved shard**	**MAX INF**
3, 5, 7, 9, 11, 15	28	30	100	All	Fine-Tuned
15	10, 16, 22, 28	30	100	All	Fine-Tuned
15	28	0, 5, 10, 15, 30, 60, 100	100	All	Fine-Tuned
15	28	30	10, 50, 100, .5k, 1k, 5k	All	Fine-Tuned
15	28	30	100	1, 3, 6, 9, 15	Fine-Tuned
15	28	30	100	All	5K, 10K, 15K, 20K

3.5.1 Impact of Increasing Number of Shards

In this experiment, we fix the replication level, which is the number of nodes in each shard. We ran this experiment with 28 replicas in each shard and scale from 3 shards to 15 shards in different geo distant regions.

The impact of adding shards on AHL's performance would be drastic in AHL since the reference committee needs to send all the messages for the 2PC protocol, including vote-request and global commit to every node in all involved shards, and this number is scaling from 56(2* 28) to 392(14 * 28). Processing these messages will affect AHL's throughput severely as it scales to 15 shards. AHL's latency will also increase because of the dense communication of a single point, which is the reference committee. However, Sharper will perform better because there is no single point of communication and message processing like the AHL's reference committee. The throughput in sharper decreases by increasing the number of shards because, for cross-shard transactions in Sharper, two rounds of quadratic communication between involved shards' nodes are required. Adding shard increases this communication both in terms of network and message processing (signing, validating).

Unlike Sharper and AHL, adding shards will not affect the amount of communication in RingBFT. The only change in RingBFT is the ring's length in the chain of shards in the RingBFT protocol. Because the communication pattern won't change in RingBFT, the throughput will not decrease by adding shards, and the latency will rise because it will take more time to go around the ring and reach consensus in all shards. As you can see in Figure 20 RingBFT is amazingly scalable compare to AHL and Sharper.

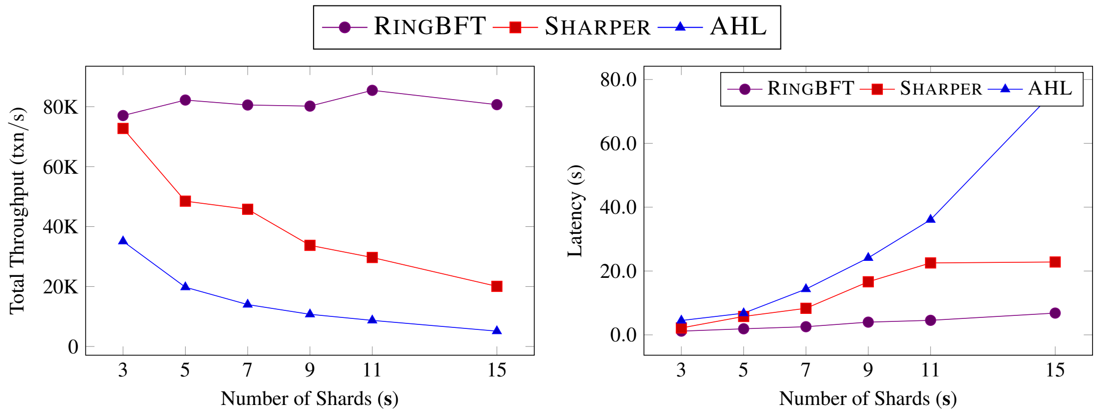


Figure 20: Impact of Increasing Number of Shards

3.5.2 Impact of Number Nodes in each Shard

In this experiment, we will focus on the effect of increasing the replication level in each shard by fixing the number of shards to 15 and 30% cross-shard workload and increasing the number of nodes in shards from 10 to 28.

The performance of AHL is already dropped because of having 15 shards and a 30% cross-shard workload. Increasing the number of nodes in shards again will put pressure on the reference committee, and getting at most 16K throughput with ten nodes in each shard does not give AHL room to fall by increasing the replication level. The performance of Sharper and RingBFT in non-cross-shard transactions is closely dependent on the replication level because they both use PBFT like consensus protocol with two rounds of quadratic communication, and increasing the number of replicas in each shard directly increases the communication and computation in replicas. The decrease in performance of RingBFT is because of increased communication local consensus and larger commit-certificates for inter-shard communication. In addition to local consensus cost in Sharper for cross-shard transactions, more communication and message processing is needed because of all-to-all communication between involved shards replicas. Figure 21 shows the result of this experiment. RinigBFT gained 4x TP in comparison to sharper and 16x compared to AHL. RingBFT is 2.3x faster than Sharper and 10.23x better than AHL.

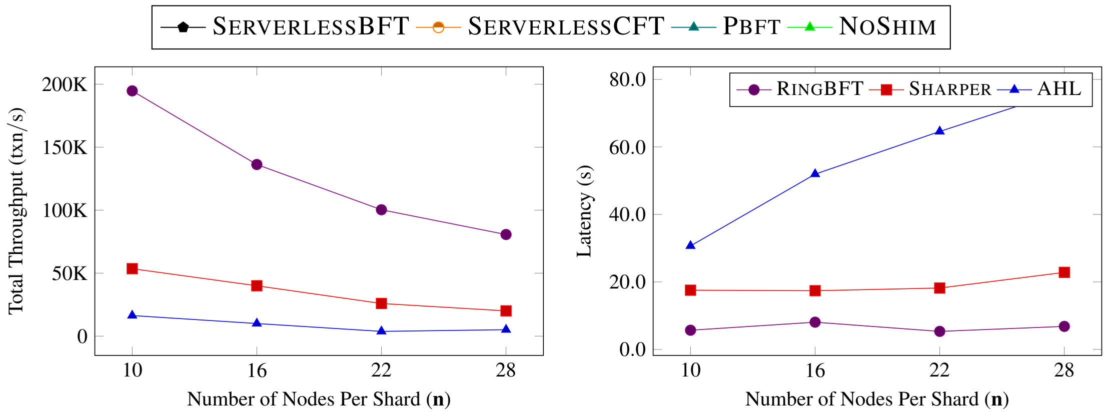


Figure 21: Impact of Number Nodes in each Shard

3.5.3 Impact of the Cross-Shard Workload

In this experiment, we measure the effect of the cross-shard workload rate, which is the number of transactions that touch data in all shards. Tweaking this rate has different impacts on different protocols. When the cross-shard rate is zero, all transactions are processed locally; therefore, the throughput and latency would be nearly the same because all three of them are doing a PBFT-like consensus in local shards. AHL, RingBFT, and Sharper achieve tremendously high performance in zero cross-shard workloads (A million transactions per second) because there is no inter-shard/region communication. As soon as we add a 5 percent cross-shard workload, the throughput drops significantly, and latency doubles, as shown in Figure X.

The performance drop in AHL is more severe than RingBFT and Sharper, and it is all because of the reference committee; while other replicas in AHL doesn't have so much to do, the reference committee is getting saturated in even small cross-shard workloads, and in 100 percent cross-shard workload its throughput falls down to 2000 transaction per second. The trend in RingBFT and Sharper is also the same, but the drop is lighter than AHL so that sharper is twice better than AHL in most cases, and RingBFT is four times better than Sharper after 15% cross-workload. One interesting observation in this experiment is that cross-shard throughput becomes constant after some point. It happens to AHL and sharper after five and RingBFT after 15. It shows that communication is the bottleneck here for all protocols, and based on the amount of communication, these protocols are performing differently. With our setup capacity in 15 regions, RingBFT can do 22K, Sharper 7K, and AHL 2K cross-shard transactions per second at most. The question is that why the non-cross-shard throughput is going down after coss-shard throughput becomes constant? The answer is that when the clients send more cross-shard transactions, processing and doing consensus for them will take more of the pipeline in the system and less room for non-cross ones; hence the non-cross throughput decreases too. Figure 22 shows total throughput, cross-throughout, and latency for different cross-shard workload rate.

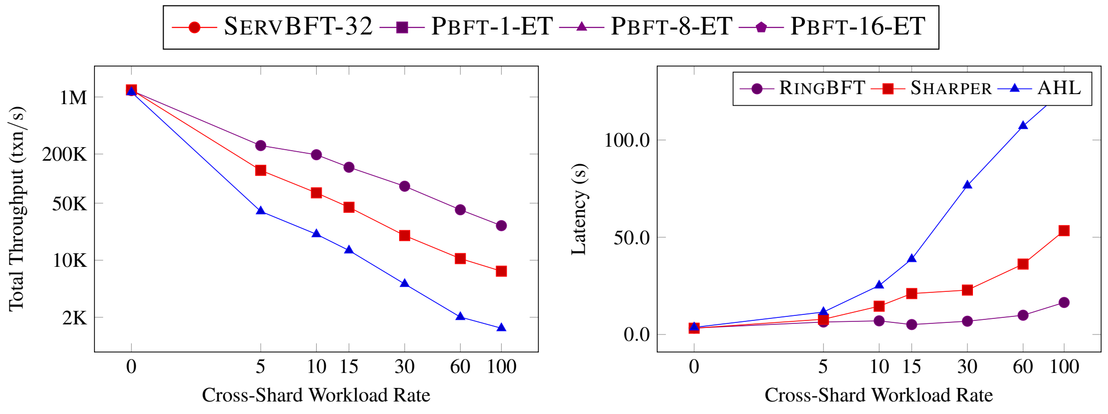


Figure 22: Impact of the Cross-Shard Workload

3.5.4 Impact of the Batch Size

Batch size is the number of transactions that a single instance of consensus contains, and replicas agree on the whole batch. It has upsides and downsides. Using large batches reduces the number of consensuses but increases the size of the messages. Large batch sizes also increase the latency because all transactions in a batch need to get done before sending client responses.

We analyze this experiment in terms of latency and throughput separately. Starting from the batch size of one, by increasing batch size, the throughput increases because, with less communication and fewer messages, we are processing more transactions. This trend lasts until the system reaches its saturation point in terms of communication and computation. Once the system is at filling its network bandwidth, adding more transactions to the batch won't increase the throughput because it cannot process more, and sending those batches will be a bottleneck for the system. Ideally, it should get constant after some point but because of implementation details and queuing; it drops slightly after some time.

Talking about latency starting from a batch size of one, increasing the batch size will decrease the /textbfAvarage Latency. For example, with x number of the round trip time, we were doing one transaction, now with the same number of round-trip times, we are doing ten transactions. This decrease in latency doesn't last long because of two reasons: First, in large batch size, all transactions should get finished to send client responses. Second, once the system reaches its saturation point of the network and computation, as we describe in the throughput part, adding transactions to batch will only increase the wait time behind the bottleneck, and latency starts to rise. Figure 23 shows the results of this experiment for throughput and latency.

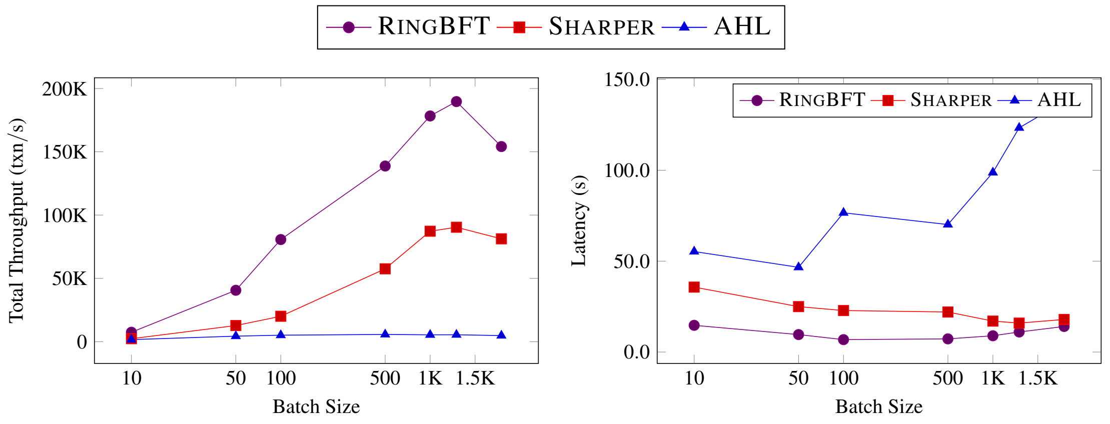


Figure 23: Impact of the Batch Size

3.5.5 Impact of Involved Shards

In this experiment, we fixed the cross-shard rate to 30 percent in 15 shards and varied the number of involved shards in the transactions. Since we shuffled the regions, they don't have proximity based on their shard id for each involved shard value; we create a window around each and include them in the transactions. For example, when we set the value of involved shards to three. the pattern of involved shards for transaction in shard x would be: (x-2,x-1,x), (x-1,x,x+1), (x,x+1,x+2).

As figure 24 shows, all three protocols observe a drop in performance by increasing the involved shards. The reason for the drop in performance is like previous experiments. RingBFT and Sharper perform around the same with three shards involved, but as soon as we go to 6 involved shards, the Sharper becomes half of RingBFT. At the same time, AHL doesn't perform well in comparison to the rest.

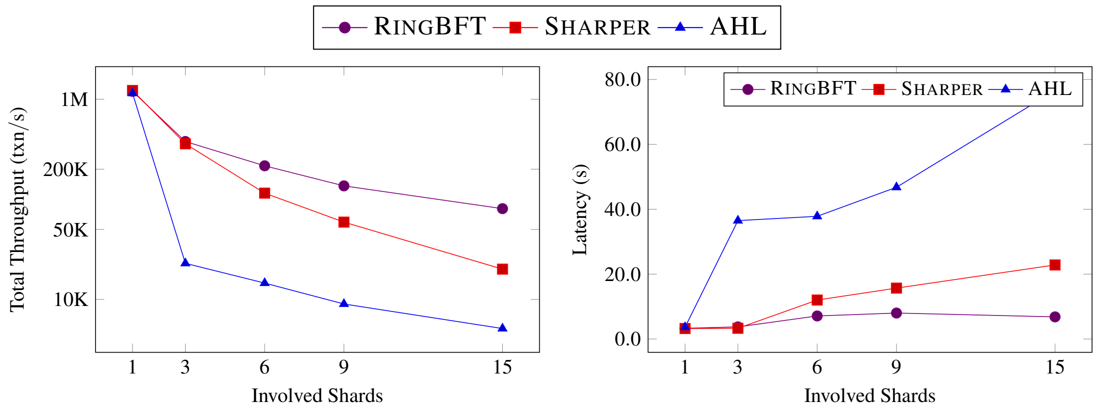


Figure 24: Impact of Involved Shards

3.5.6 Impact of Number of Inflight Transactions

When clients send transactions to the replicas, they should not overwhelm the system. It should be at a reasonable rate. In our system, we define a parameter called MAX_INF: it is the number of ongoing transactions for each client. A client sends MAX_INF transactions and waits to get a response. Once it receives the response for some of them, it will again fill up the MAX_INF ongoing transactions. This value parameter is sensitive and depends on the performance of each protocol.

Figure 25 shows the performance of three protocols with different MAX_INF values. The throughput trend for all protocols is the same. They increase and become constant after some time. The reason is that having a small MAX_INF will not fill the pipeline, so the system is not fully utilized. After some number, it reaches a saturation point, and throughput becomes constant. On the other hand, latency always increases by increasing MAX_INF because when the pipeline is filled, by increasing the MAX_INF, you are just making the queues longer and more wait time for the transactions to get processed.

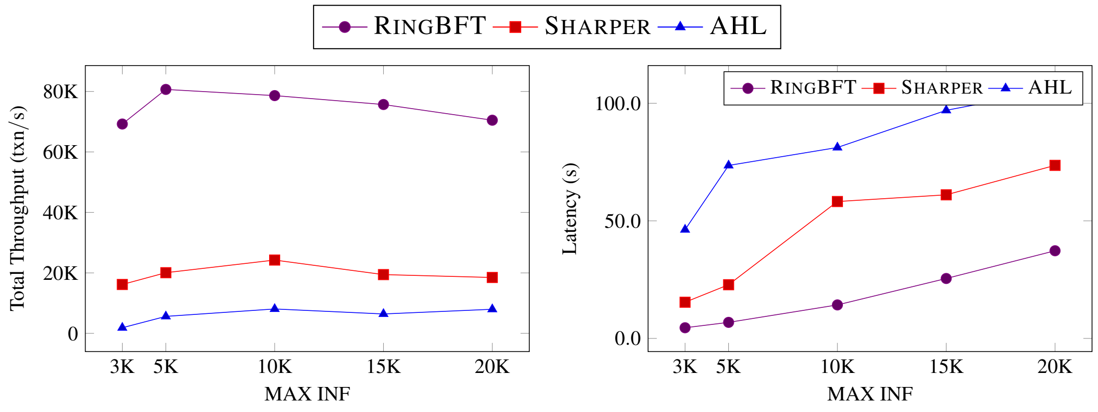


Figure 25: Impact of Number of Inflight Transactions

4 Reliable Transactions in Serverless-Edge Co-design

Modern edge applications demand novel solutions where edge applications do not have to rely on a single cloud provider (which cannot be in the vicinity of every edge device) or dedicated edge servers (which cannot scale as clouds) for processing compute-intensive tasks. A recent computing philosophy, Sky computing [179, 42], proposes giving each user ability to select between available cloud providers.

In this chapter, we present our serverless-edge co-design, which extends the Sky computing vision. In our serverless-edge co-design, we expect edge devices to collaborate and spawn required number of serverless functions. This raises several key challenges: (1) how will this collaboration take place, (2) what if some edge devices are compromised, and (3) what if a selected cloud provider is malicious. Hence, we design SERVERLESSBFT, the first protocol to guarantee Byzantine fault-tolerant (BFT) transactional flow between edge devices and serverless functions. We present an exhaustive list of attacks and their solutions on our serverless-edge co-design. Further, we extensively benchmark our architecture on a variety of parameters.

4.1 Challenges for Serverless-Edge Co-design

We introduces SERVERLESSBFT, the first protocol to guarantee Byzantine fault-tolerant (BFT) transactional flow between edge devices and serverless functions. The design of SERVERLESSBFT is motivated from the recent introduction of *Sky Computing*, which envisages utility computing in a multi-cloud environment [179, 42]. Sky computing propounds the design of an *inter-cloud broker* that takes as input a client program and output specifications and selects the best cloud providers to execute the client program. Such a broker is extremely desirable for the edge and Internet of Things (IoT) applications, which run on edge devices, such as smart devices, sensors, UAVs, and phones, that have limited compute power and memory [46].

On the one hand, existing edge applications expect response latency in the order of tens of milliseconds [151, 123, 43, 145]. On the other hand, they are forced to delegate compute-intensive tasks to a specific third-party cloud provider such as AWS and Azure [150, 169, 180]. A recent way to solve this dilemma is to install dedicated *edge-servers* that are closer to the edge devices [28, 193, 13]. These edge servers are installed and maintained by the enterprise behind the application [28, 134, 80]. If any server hardware crashes, then the enterprise may need to purchase new hardware.

Moreover, with ever-growing application needs, these servers are unlikely to scale seamlessly as third-party clouds.

SERVERLESSBFT realizes the Sky computing vision in edge computing by giving the edge applications flexibility to select any of the available cloud providers. As a result, the edge application can select different cloud providers based on the location of its users.[3] However, moving data across cloud providers degrades system performance and is expensive. So, we take a step further and permit edge applications to make use of serverless technology, which (i) decouples storage, compute, and network, (ii) supports pay-as-you-go model where the enterprise pays only for the resources used, and (iii) supports auto-scaling policies [178, 20, 194]. We refer to this interaction as *serverless-edge co-design* as it promotes light-weight tasks at the edge while compute-intensive tasks are done at the serverless cloud. Our serverless-edge co-design targets low latency by allowing edge devices to spawn serverless functions at the nearest cloud.

Our serverless-edge co-design also presents several research challenges, which we present next.

[3]At present, switching cloud providers is common for most applications due to geo-political reasons and government regulations [146].

4.1.1 Task distribution between edge and serverless.

Our SERVERLESSBFT protocol requires an edge application to push its compute-intensive task to the cloud by spawning serverless functions (for simplicity, we refer to these functions as *executors*). To do so, we need to design a *compatibility layer*. We build this compatibility layer on top of edge devices and refer to it as a *shim*. At shim, the edge devices collaborate and spawn serverless executors for executing compute-intensive client requests.

4.1.2 Lack of Trust at Shim.

As edge devices may belong to different parties, which may not trust each other, it is hard for these devices to collaborate. Hence, our SERVERLESSBFT protocol runs a traditional BFT protocol to allow these edge devices reach a consensus [40, 97]. This consensus decides which edge device will spawn the desired number of executors and the order in which client requests are processed. For consensus, we opt for BFT protocols as they are resilient to malicious attacks. Further, depending on the location and nature of edge devices, SERVERLESSBFT permits various shim designs: a single shim of all devices running PBFT [40] consensus, multiple dependent shims of devices spread globally, running GeoBFT [97], and multiple independent shims running AHL [58], Sharper [8], or RingBFT [161]. For simplicity, in this work, we assume a single shim of $3\mathbf{f}+1$ devices where up to $\mathbf{f}$ devices can act malicious.

4.1.3 Lack of Trust at Serverless cloud.

Depending on the application requirement, shim may spawn serverless executors at one or more available cloud providers in the vicinity. Hence, there is again a lack of trust: some cloud providers may have mal-intent or may have poor QoS (crashed or failed executors) [47, 110]. As a result, SERVERLESSBFT requires the shim to spawn $2\mathbf{f}+1$ executors and permits up to $\mathbf{f}$ of them to fail. This extra spawning is not new; Yahoo's Hadoop also executes the same code multiple times to reduce latency due to stragglers [176].

4.1.4 Private Data access and retrieval.

Recent reports illustrate that around 90% of the industries are not only sticking with their existing on-premise servers, but also scaling them up [119, 74]. For at least 65% of these industries, the key reason for maintaining on-premise servers is to protect their consumer data from data-breaches and attacks [119]. In our serverless-edge co-design, we adhere to this design choice and assume that all the client data is stored in an on-premise storage at the enterprise. As a result, the enterprise can control access to the data. Hence, edge devices or executors lack rights to update the storage, but may request read access to the same. For updates to the storage, we write a lightweight wrapper (*verifier*) around the storage that collects execution results, updates the data-store, and forwards the results to the clients.

Furthermore, we observe several other new challenges with our architecture: (i) Byzantine shim devices may spawn fewer executors, for which we need to hold them accountable. (ii) During execution, executors may need to read data from the storage. (iii) If the client transactions are conflicting and their read-write sets are unknown until execution, we may have to abort such transactions.

We envision our serverless-edge architecture to seamlessly integrate with existing edge applications. To realize this goal experimentally, we design a shim of nodes and require them to spawn AWS Lambda functions as executors. On each shim node, we install ResilientDB's light-weight and multi-threaded consensus framework [97, 96, 100]. We evaluate our SERVERLESSBFT protocol on *eight* distinct parameters. Our results illustrate that SERVERLESSBFT can facilitate shims of up to 128 devices in 11 global regions. Further, in our experiments, we are easily able to spawn 21 executors in parallel (could not scale further due to limits by cloud provider), and the peak throughput achieved by our SERVERLESSBFT protocol is 240 k txns/s while the minimum latency incurred is 30 ms.

We make the following *contributions*:

• We design of a novel serverless-edge co-design that meets the vision of Sky computing and helps design low latency reliable edge applications where edge devices can select cloud providers based on desired output specifications.

• In our serverless-edge architecture, we neither trust the edge devices nor the serverless executors. Hence, we introduce a novel protocol SERVERLESSBFT that manages the flow of a client request in our serverless-edge architecture and shields the system against arbitrary results and malicious attacks.

• We enlist possible attacks in our serverless-edge architecture and present solutions to recover the system.

• Our SERVERLESSBFT protocol presents algorithms to handle conflicting transactions with or without the knowledge of read-write available to shim nodes prior to execution.

4.2 Motivation and Use Case

The motivations behind our serverless-edge co-design are the emerging use cases of edge-computing, such as AR/VR video-streaming and Unmanned Aerial Vehicles (UAVs). These applications require massive data-processing as they need to run ML models to train data on-flight or provide the user useful insights. The key challenge these applications face is the rapidly changing user characteristics.

We consider a real-world use case of UAVs as a motivating example for this work [5]. In recent years, UAVs have been adopted by e-commerce industries, such as Amazon and Walmart, for product deliveries. These UAVs help to securely and quickly transport user goods in a cost-efficient manner. During the delivery process, each UAV travels over multiple geographical locations and performs an array of tasks, such as navigation, image recognition, and live video-streaming.

In Figure 26(b), we illustrate the traditional way of computing for UAVs, where each UAV offloads all the collected data to the dedicated edge servers for processing. In this model, UAVs are forced to communicate with dedicated servers. When the server is in the vicinity, the communication round-trip costs are low; otherwise, they are high. Each edge server executes the requests from various UAVs in an ordered-fashion. Moreover, these servers need to be continuously scaled, new software needs to be installed, and OS needs to be updated, which makes them a financially expensive choice.

In Figure 26(a), we reimagine the UAV delivery operation in our serverless-edge co-design. Switching to our serverless-edge model, allows UAVs in the vicinity to interact with each other and act as a shim that spawns serverless executors to process collected data. To alleviate concerns regarding round-trip costs, the shim is permitted to opt for services from local cloud providers. In fact, shim can spawn executors at multiple clouds and wait for whichever responds earliest. As there is a lack of trust among shim devices and executors, we have our SERVERLESSBFT protocol to manage all the transactional flow in a byzantine fault-tolerant manner.

Byzantine failures in the wild. Do real-world systems face more than just crash failures? Unfortunately,

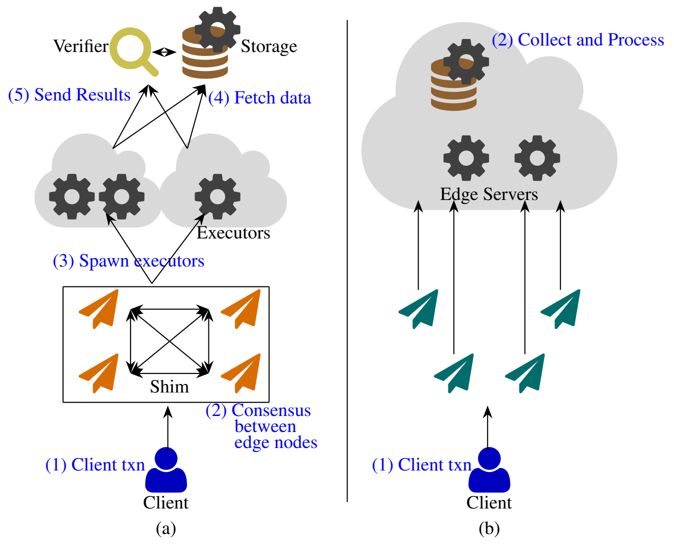


Figure 26: Illustration of (a) Serverless-Edge architecture employing the SERVERLESSBFT protocol and (b) architecture prevalent in existing edge applications.

yes. Existing systems suffer from *omission* failures where nodes can crash [54], and *arbitrary* failures where nodes can act in an unexpected manner [47]. Almost all real-world applications handle omission failures using protocols based on Paxos-family [124, 147]. However, the true challenge is to bulwark the system system against often overlooked arbitrary failures: Google's UpRight [47] provides fault-tolerance against byzantine failures, Google has also observed corrupt execution errors [110], and Cloudflare observed a misbehaving switch sending incorrect messages [50]. Hence, it is better to guard system against these failures.

4.3 Preliminaries

We make standard assumptions as made by any BFT system [40, 97, 8, 96]. We represent our serverless-edge architecture $\mathscr{A}$ through a quintuple, $\mathscr{A} = \{\text{clients}(,)\mathfrak{R}, \mathscr{E}, \mathscr{S}, \mathscr{V}\}$, where we use clients() to denote the set of clients, $\mathfrak{R}$ to denote the shim of edge devices or nodes, $\mathscr{E}$ to denote the serverless executors, $\mathscr{V}$ and $\mathscr{S}$ to denote the verifier and data-store. As described in Section 4.1, we assume an on-premise data-store maintained by the enterprise, while the verifier is a lightweight wrapper around the data-store. Hence, both verifier and storage are assumed to be honest and trusted.

Fault-Tolerance Requirement at Shim. We use the notation $\mathbf{n}\mathfrak{R} = |\mathfrak{R}|$ to represent total number of edge nodes in $\mathscr{A}$. At most $\mathbf{f}\mathfrak{R}$ of these nodes are byzantine and can crash-fail or act arbitrarily; $\mathbf{n}\mathfrak{R} \geq 3\mathbf{f}\mathfrak{R} + 1$. The remaining $2\mathbf{f}\mathfrak{R} + 1$ nodes are honest and follow the protocol.

Authenticated Communication. To exchange messages among different components, we employ Digital Signatures (DS) and Message Authentication Codes (MAC) [120]. To represent a message m signed by a component R using DS, we use the notation $\langle R \rangle_m$. Anyone who has the signer's public-key can verify this signature. One of the common ways to exchange public-keys is through a public-key certificates [38]. For MACs, signer and verifier use a common key, which is kept secret. We use Diffie-Hellman key exchange for securely sharing secret keys. In rest of the text, any message m that does not indicate the identity of the signer implies the use of MAC. Although MACs offer higher throughput than DS, DS guarantee *non-repudiation* [40, 102]. We also employ a *collision-resistant hash function* $\mathtt{D}(\cdot)$ to map a value v to a constant-sized digest $\mathtt{D}(v)$. We use a function id() to assign an identifier to each node $R \in \mathfrak{R}$ and each executor $\mathtt{E} \in \mathscr{E}$. We assume that byzantine components can neither impersonate honest components, nor subvert cryptographic constructs. We *do not* make any assumptions on the behavior of the clients. We term a message as *well-formed* if it passes all the cryptographic and other necessary checks.

4.3.1 Serverless Cloud Assumptions

We expect access to one or more serverless clouds such as AWS Lambda and Google Functions. These serverless cloud should permit edge nodes to seamlessly upload the desirable code or transactions for processing as per the application specifications. For simplicity, in rest of the text, we assume that the shim nodes access only one cloud provider for *spawning executors* to execute client transactions. However, there is *no free food* as these serverless clouds follow a *pay-per-use* model where whoever spawns executors also pays for their use [114]. We expect these clouds to meet the following:

• *Fault-Tolerance:* To handle arbitrary faults at the serverless cloud, we spawn $\mathbf{n}\mathscr{E} \geq 2\mathbf{f}\mathscr{E} + 1$ executors, and assume that at most $\mathbf{f}\mathscr{E}$ are byzantine. Prior works have shown that $2\mathbf{f}\mathscr{E} + 1$ executors guarantee successful execution of a transaction in the byzantine setting [199]. This leads us to observe the following:

1. The values for $\mathbf{f}\mathscr{E}$ and $\mathbf{f}\mathfrak{R}$ may or may not be same.
2. In Section 4.6, we illustrate that if the transactions are conflicting, then we need an additional $\mathbf{f}\mathscr{E}$ executors to prevent an indistinguishable byzantine attack.

• *Identity:* We expect each spawned executor to be assigned a unique pair of public-private key, which it uses to digitally sign a message.

• *Accountability:* Each executor is spawned by some shim node that pays for this service. Hence, we expect that no executor can spawn more executors. Further, the expected number of executors *to be spawned* by shim nodes is known to all the components of our architecture.

• *Payment.* As executors are spawned by shim nodes, it implies that the spawner will be billed by the cloud provider. Hence, post successful consensus of a transaction, the edge application's enterprise pays the spawner a fixed amount to cover its expenses.

4.4 Architecture

We now discuss in detail the BFT transactional flow guaranteed by our SERVERLESSBFT protocol in the serverless-edge co-design. In Figure 27, we schematically present this flow; the shim consists of $\mathbf{n}\mathfrak{R} = 4$ edge nodes and $\mathbf{n}\mathscr{E} = 3$ executors are spawned per transaction. For understandability, we will periodically refer to the UAV use case of Section 4.2.

As stated earlier, shim can have different abstractions and can run any BFT protocol. In this work, we assume a single shim of $3\mathbf{f}\mathfrak{R} + 1$ and require shim nodes to run the PBFT [40] protocol. PBFT is considered as a representative BFT protocol as all the other protocols follow its design. PBFT protocol works in *views*. For each view, one node is designated as the *primary* and is responsible for successful completion of consensuses in that view. If the primary acts malicious, the view is changed and the primary is replaced.

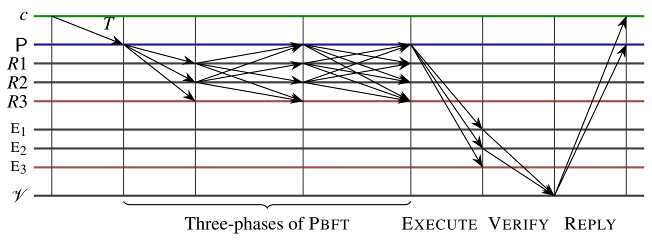


Figure 27: Schematic representation of the transactional flow in SERVERLESSBFT protocol. Given a client transaction T, the nodes of the shim work together to order this transaction, following which the primary P invokes the executors at the serverless cloud to execute T. Post execution, the executors send their results to the verifier, which replies to the client.

4.4.1 Client Request and Response

Any user that accesses the edge application becomes a client in our system. E.g., each UAV that requires data-processing from the cloud acts as a client and packages its request as a transaction. A client c send a message $\langle c \rangle_T$ to the primary node P[4] of the current view v of the shim when it wants to process a transaction T. Notice that c employs DS to sign this message (refer to Figure 28, Line 1). The client c marks $\langle c \rangle_T$ 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 (Line 3).

4.4.2 Shim Ordering

SERVERLESSBFT assigns each shim node (e.g. UAV) an identifier, $0, 1, 2, ..., \mathbf{n}\mathfrak{R}$. Initially, the shim node with identifier 0 is designated as the primary P of the shim. On receiving a client request $\langle c \rangle_T$, P checks if $\langle c \rangle_T$ is well-formed. If this is the case, P initiates the PBFT protocol as follows.

- *Pre-prepare.* The primary P assigns a sequence number k to the well-formed client message $m := \langle c \rangle_T$ and sends it as a PREPREPARE message to all the nodes of the shim. This PREPREPARE message also includes a digest $\Delta = \mathrm{D}(m)$, which is used in future communication to save space. Notice that the primary signs this message using MAC, which provide sufficient guarantees for this phase. When a node $R \in \mathfrak{R}$ receives a PREPREPARE message from the primary P of view v, it runs the message through a series of checks. If the checks are successful, then R agrees to support the order k for this client request by broadcasting a PREPARE message.

- *Prepare.* When a node R receives identical PREPARE messages from $2\mathbf{f}\mathfrak{R}+1$ distinct nodes (can include its own message to reach the count), it marks the request m as *prepared* and broadcasts a COMMIT message. We require each node R to use DS to sign the COMMIT message.

- *Commit.* When R receives identical COMMIT messages from $2\mathbf{f}\mathfrak{R}+1$ nodes, it marks m as *committed*.

Remark. PBFT requires two phases of quadratic communication complexity. Instead, shim can employ BFT protocols like PoE [91] and SBFT [84] that guarantee linear communication with the help of advanced cryptographic schemes like threshold signatures. Note: in our architecture, the edge devices are acting as both clients and shim nodes.

[4]Some BFT protocols require a client request to be sent to all the nodes.

4.4.3 Serverless Optimistic Execution

Once P commits a request, SERVERLESSBFT requires P to connect with the serverless cloud and spawn $\mathbf{n}\mathscr{E}$ executors. P sends each of these executors an EXECUTE message (Line 9), which includes a *certificate* $\mathfrak{C}$; a set of signatures of $2\mathbf{f}\mathfrak{R}+1$ distinct shim nodes and proves that these nodes agreed to order this request (Line 8). Prior to executing the transaction T, each executor E $\in \mathscr{E}$ checks if the certificate $\mathfrak{C}$ is valid.

During execution, E may need to access the value of read-write sets ($\mathbf{rw}$). Hence, it connects with the storage $\mathscr{S}$ and *fetches* the required data (Lines 17-18). However, executors *do not write* to the storage. Any intermediate results are stored locally. Further, these executors do not communicate with each other. Post execution, each executor E sends a VERIFY message to the verifier $\mathscr{V}$, which includes the computed result r, certificate $\mathfrak{C}$, and accessed read-write sets $\mathbf{rw}$.

Remark. We allow shim to spawn either stateless or stateful executors [114, 136]. Stateful executors have memory and remember the results of last execution. By definition, severless executors are "fleeting" and return after execution; a common way to assign these executors memory is by having a layer that stores computed results [178]. To employ stateful executors in our model, we would need BFT guarantees on the additional layer. Hence, we focus on stateless executors. Including $\mathfrak{C}$ in the EXECUTE and VERIFY messages helps to detect byzantine attacks (§ 4.6.2). Further, by employing threshold signatures, we can reduce the size of the certificate. Threshold signatures allow combining $2\mathbf{f}\mathfrak{R}+1$ signatures into a single signature.

4.4.4 Verifier and Concurrency Control

The verifier $\mathscr{V}$ is a lightweight wrapper around the data-store $\mathscr{S}$ and is assumed to be *correct and trusted*. The verifier collects well-formed VERIFY messages from the executors (in set $\mathbf{V}$) and once it has a quorum of matching results that do not violate the *concurrency control constraints*, it updates the data-store. It performs these tasks in the following *order*:

1. If set $\mathbf{V}$ has at least $\mathbf{f}\mathscr{E}+1$ matching VERIFY messages, $\mathscr{V}$ marks the transaction as *matched*. Following this, $\mathscr{V}$ ignores any other VERIFY message for $\langle c \rangle_T$ (Line 23).
2. If k is the sequence number for $\langle c \rangle_T$ and k_{max} is the sequence number of last *validated* request, then if $k_{\text{max}} \neq k$, $\mathscr{V}$ places the k-th request in the list π (Line 29).
3. If $k_{\text{max}} = k$, $\mathscr{V}$ checks if the value of the read-write sets $\mathbf{rw}$ of the k-th request is same as that in the data-store $\mathscr{S}$ (Lines 31-32). If the *read sets match*, $\mathscr{V}$ sends the client and the shim primary RESPONSE messages and updates the write sets at the storage in accordance with the result r (Lines 33-34). Note: matching read-write sets is only required when the transactions are conflicting. We discuss this in Section 4.6.
4. Next, $\mathscr{V}$ increments k_{max} and checks if π includes the transaction with sequence number k_{max}. If so, it removes the k_{max}-th transaction from π and runs steps in Lines 26-27. These concurrency control checks ensure that consistent updates are written to the storage.

```
Initialization:
// $k_{\max}$ := sequence number of the next request to be verified by $\mathscr{V}$.
// $\pi := \emptyset$ (list of requests marked matched at verifier)

Client-role (used by client $c$ to request transaction $T$) :
1:  Sends $\langle c\rangle_T$ to the primary $\mathsf{P}$.
2:  Awaits receipt of message $\text{RESPONSE}(\langle c\rangle_T, k, r)$ from $\mathscr{V}$.
3:  Considers $T$ executed, with result $r$, as the $k$-th transaction.

Primary-role (running at the primary node $\mathsf{P}$) :
4:  event $\mathsf{P}$ receives $\langle c\rangle_T$ do
5:      Calculate digest $\Delta := \mathrm{D}(\langle c\rangle_T)$.
6:      Broadcast $\text{PREPREPARE}(\langle c\rangle_T, \Delta, k)$ to all nodes (order at sequence $k$).

7:  event $\mathsf{P}$ receives $\mathbf{nf}\mathfrak{R}$ $m := \langle R\rangle_{\text{COMMIT}(\Delta,k)}$ messages such that:
        1. each message $m$ is well-formed and is sent by a distinct node $R \in \mathfrak{R}$.
    do
8:      $\mathfrak{C}$ := set of DS of these $\mathbf{nf}\mathfrak{R}$ messages. // Certificate
9:      Send $\langle \mathsf{P}\rangle_{\text{EXECUTE}(\langle c\rangle_T, \mathfrak{C}, m, \Delta)}$ to all executors $\mathrm{E} \in \mathscr{E}$. // Serverless access

Non-Primary role (running at a node $R \in \mathfrak{R}$) :
10: event $R$ receives $\text{PREPREPARE}(\langle c\rangle_T, \Delta, k)$ from $\mathsf{P}$ such that:
        1. message is well-formed, and $R$ did not accept a $k$-th proposal from $\mathsf{P}$.
    do
11:     Broadcast $\text{PREPARE}(\Delta, k)$ to all nodes in $\mathfrak{R}$.

All nodes role (running at the node $R$) :
12: event $R$ receives $\text{PREPARE}(\Delta, k)$ 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 $\langle R\rangle_{\text{COMMIT}(\Delta,k)}$ to all nodes in $\mathfrak{R}$.

Executor-role (running at the executor $\mathrm{E} \in \mathscr{E}$) :
14: event $\mathrm{E}$ receives $\langle \mathsf{P}\rangle_{\text{EXECUTE}(\langle c\rangle_T, \mathfrak{C}, m, \Delta)}$ from $\mathsf{P}$ such that:
        1. message is well-formed,
        2. $m := \text{COMMIT}(\Delta, k)$, and
        3. Certificate $\mathfrak{C}$ includes $\mathbf{nf}R$ distinct DS on $m$.
    do
15:     while $T$ not executed do
16:         $\mathbf{rw}$ := Read-write sets for $T$.
17:         if Need the current state of $\mathbf{rw}$ // Storage access then
18:             Fetch $\mathbf{rw}$ state (values) from storage $\mathscr{S}$
19:     $r$ := Result of executing $T$
20:     Send $\text{VERIFY}(\langle c\rangle_T, \mathfrak{C}, m, \mathbf{rw}, r)$ to verifier $\mathscr{V}$. // Communication with verifier

Verifier-role (running at the verifier $\mathscr{V}$) :
21: event $\mathscr{V}$ receives $m' := \text{VERIFY}(\langle c\rangle_T, A, m, \mathbf{rw}, r)$ message from an executors such that:
        1. $m'$ is well-formed and is sent by a distinct executor $\mathrm{E} \in \mathscr{E}$, and
        2. $m := \text{COMMIT}(\Delta, k)$.
    do
22:     Add $m'$ to $\mathbf{V}$.

23: event Set $\mathbf{V}$ has $\mathbf{f}\mathscr{E}+1$ identical $m' := \text{VERIFY}(\langle c\rangle_T, A, m, \mathbf{rw}, r)$ messages do
24:     if $k = k_{\max}$ // Next request in order. then
25:         Run function ccheck($\pi$)
26:         while $k_{\max}$-th transaction is in $\pi$ // Other requests do
27:             Run function ccheck($\pi$)
28:     else
29:         Store $m'$ in $\pi$.

30: function ccheck (list: $\pi$)
31:     $\mathbf{rw}'$ := Current state of $\mathbf{rw}$ fetched from storage $\mathscr{S}$.
32:     if $\mathbf{rw}' = \mathbf{rw}$ // Concurrency control check then
33:         Send $\langle \mathscr{V}\rangle_{\text{RESPONSE}(\Delta,r)}$ to the client $c$ and primary $\mathsf{P}$. // Reply to client.
34:         Update corresponding $\mathbf{rw}$ with $r$ at the storage $\mathscr{S}$.
35:     $k_{\max} = k_{\max} + 1$.
```

Figure 28: Byzantine Fault-Tolerant transaction processing by SERVERLESSBFT protocol in the serverless-edge architecture.

4.4.5 System Guarantees

We now state the guarantees offered by our different components of our serverless-edge architecture.

Shim Consistency. If an honest node commits a transaction T, then all the honest nodes commit T.

Shim Non-Divergence. If two honest nodes order a transaction T at sequence number k and k', then $k = k'$.

Shim Termination. If an honest client sends a transaction T, then an honest node will eventually commit T.

Executor Termination. If an honest primary sends an EXECUTE message for transaction T, then an honest executor will execute T.

Verifier Non-Divergence. If the shim commits a transaction T at sequence k, then the verifier will eventually update the corresponding result at the storage at order k.

Together, shim consistency, shim non-divergence, and verifier non-divergence imply *safety*, while shim termination and executor termination imply *liveness*. Our SERVERLESSBFT protocol guarantees safety in an asynchronous environment where the messages can get lost, delayed, or duplicated, and byzantine components can collude or act arbitrarily. To guarantee liveness, our SERVERLESSBFT protocol expects periods of synchrony. Note: our SERVERLESSBFT offers standard safety and liveness guarantees, also offered by other systems [40, 91, 97, 58, 8].

4.5 Tackling Byzantine Attacks

In our architecture, at most $\mathbf{f}\mathfrak{R}$ shim nodes and $\mathbf{f}\mathscr{E}$ serverless executors can act byzantine. If the primary of shim is honest, then byzantine nodes cannot affect the ongoing transactional flow. Similarly, byzantine executors can either provide incorrect result or ignore execution, but as there are at least $\mathbf{f}\mathscr{E}+1$ honest executors, EXECUTE messages sent by honest primaries will be processed. Hence, following is an exhaustive list of attacks on our design.

(i) *Request Suppression.* If the primary of shim is byzantine, it can try to prevent consensus on some client requests.

(ii) *Nodes in Dark.* If shim's primary is byzantine, it can keep up to $\mathbf{f}\mathfrak{R}$ honest shim nodes in *dark* by not involving them in consensuses.

(iii) *Verifier Flooding.* Byzantine components can flood the verifier with requests that have been already verified.

Next, we present algorithms to recover from these attacks.

4.5.1 Request Suppression

In the serverless-edge architecture, byzantine components can work together to deny service to one or more clients. This request suppression attack can take three different forms:

(i) *Request Ignorance.* If the shim's primary node P is byzantine, it can willfully drop a request m from a client c, or indefinitely delay consensus on m.

(ii) *Unsuccessful Consensus.* A byzantine primary P may involve less than $2\mathbf{f}\mathfrak{R}+1$ nodes in consensus on a client request m. As a result, these nodes will not reach consensus on m.

(iii) *Less Executors.* A byzantine primary P may permit consensus on a client request m, but disallow its execution by spawning less than $\mathbf{n}\mathscr{E}$ serverless executors. In such a case, the verifier $\mathscr{V}$ will not receive $\mathbf{f}\mathscr{E}+1$ matching execution results.

To detect these attacks, we setup three distinct *timers* at various components of our architecture.

• *Client timer.* Our SERVERLESSBFT protocol requires each client c to start a timer τ_m prior to sending its request m to the primary P. When c receives a RESPONSE message for m from the verifier $\mathscr{V}$, it stops τ_m.

• *Node timer.* Our SERVERLESSBFT protocol requires each node $R \in \mathfrak{R}$ to start a timer τ_m when it receives a well-formed PREPREPARE message for a client request m from the primary P. When R marks m as committed, it stops τ_m.

• *Node re-transmission timer.* If a non-primary node $R \in \mathfrak{R}$ receives an ERROR message from the verifier $\mathscr{V}$ (see Section 4.5.1) then R forwards the ERROR message to the primary P and starts the re-transmission timer Υ. When R receives a corresponding ACK message from $\mathscr{V}$, it stops Υ.

In the case the timers of c or R expire, the respective component detects a request suspension attack and initiates the following mechanisms for recovery from this attack.

```
Client-role (running at the client $c$) :
event $c$'s timer $\tau_m$ for request $m := \langle c \rangle_T$ timeouts do
    Sends $\langle c \rangle_T$ to the verifier $\mathscr{V}$.
    Restarts $\tau_m$.
    if Figure 28, Lines 2 and 3 are successful // Receives $\mathbf{f}\mathfrak{R}+1$ matching responses then
        Cancel $\tau_m$

Verifier-role (running at the verifier $\mathscr{V}$) :
event $\mathscr{V}$ receives a well-formed request $m := \langle c \rangle_T$ from client $c$ do
    if Previously sent RESPONSE for $m$ then
        Resends message $\langle \mathscr{V} \rangle_{\text{RESPONSE}(\Delta,r)}$ to $c$.
    else if $m$ exists in list $\pi$ // Waiting for consensus of $k_{\max}$-th request then
        Broadcasts $\langle \mathscr{V} \rangle_{\text{ERROR}(k_{\max})}$ to all shim nodes.
    else if Did not receive any VERIFY message for $\langle c \rangle_T$ then
        Broadcasts $\langle \mathscr{V} \rangle_{\text{ERROR}(\langle c \rangle_T)}$ to all shim nodes. // Missing Request
    else
        Broadcasts $\langle \mathscr{V} \rangle_{\text{REPLACE}(\langle c \rangle_T)}$ to all shim nodes. // Byzantine Primary

Node-role (running at the node $R$) :
event $R$ receives $\langle \mathscr{V} \rangle_{\text{ERROR}(\langle c \rangle_T)}$ or $\langle \mathscr{V} \rangle_{\text{ERROR}(k_{\max})}$ from $\mathscr{V}$ do
    Start a timer $\Upsilon$.
    Forward the ERROR message to the primary $\mathsf{P}$.

event $R$'s timer $\tau_m$ or $\Upsilon_m$ timeout or $R$ receives $\langle \mathscr{V} \rangle_{\text{REPLACE}(\langle c \rangle_T)}$ from $\mathscr{V}$ do
    Run the view-change protocol to replace $\mathsf{P}$
```

Figure 29: Actions performed by various participants of the serverless-edge infrastructure in response to a request suppression attack.

Client action on timeout If a client c's timer τ_m timeouts, then c forwards its request to the verifier $\mathscr{V}$ and restarts its timer (refer to Figure 29). In specific, each time c's timer expires, after some exponential backoff, it re-sends its request to $\mathscr{V}$ until it receives a RESPONSE message from $\mathscr{V}$.

Verifier action on receiving client request When the verifier $\mathscr{V}$ receives a request $m := \langle c \rangle_T$ from client c, it first determines if it has seen $\langle c \rangle_T$ till now or not. If $\mathscr{V}$ has not received any VERIFY messages for $\langle c \rangle_T$, it sends $\langle \mathscr{V} \rangle_{\text{ERROR}(\langle c \rangle_T)}$ message to all the nodes in the shim. Otherwise, there can be only three cases:

(i) $\mathscr{V}$ did send a RESPONSE message for $\langle c \rangle_T$, so it simply resends the RESPONSE message.

(ii) $\langle c \rangle_T$ resides in π. Further, assume that it was ordered by shim at some sequence number k. So $k_{\max} < k$, and $\mathscr{V}$ is waiting for the request with sequence number $k_{\max}$. Unless the $k_{\max}$-th request is validated by $\mathscr{V}$, succeeding requests cannot be processed. So, $\mathscr{V}$ needs to notify shim nodes about the missing request at sequence $k_{\max}$, and it does so by sending $\langle \mathscr{V} \rangle_{\text{ERROR}(k_{\max})}$ to all the shim nodes. Note: this gap between $k_{\max}$ and k could have been created by byzantine primary.

(iii) $\mathscr{V}$ did not receive $\mathbf{f}\mathscr{E}+1$ matching VERIFY messages for $\langle c \rangle_T$. This can only occur if the primary is byzantine. So, $\mathscr{V}$ sends $\langle \mathscr{V} \rangle_{\text{REPLACE}(\langle c \rangle_T)}$ to all the shim nodes.

Once $\mathscr{V}$ successfully verifies the request at sequence number $k_{\max}$ or $\langle c\rangle_T$, $\mathscr{V}$ creates a corresponding $\langle\mathscr{V}\rangle_{\text{ACK}(k_{\max})}$ or $\langle\mathscr{V}\rangle_{\text{ACK}(\langle c\rangle_T)}$ message and broadcasts it to shim.

Node action on ERROR message When a shim node $R \in \mathfrak{R}$ receives an ERROR message from the verifier, it can only conclude the following:

- R received $\langle\mathscr{V}\rangle_{\text{ERROR}(k_{\max})}$ message and has either committed or not committed the request at sequence number $k_{\max}$.
- R received $\langle\mathscr{V}\rangle_{\text{ERROR}(\langle c\rangle_T)}$ message and has either committed or not committed the request $\langle c\rangle_T$.

Irrespective of these cases, the node R starts a re-transmit timer Υ to track the behavior of the primary. Next, it forwards the received ERROR message to the primary. If the timer Υ expires before R receives a corresponding acknowledgment message ($\langle\mathscr{V}\rangle_{\text{ACK}(k_{\max})}$ or $\langle\mathscr{V}\rangle_{\text{ACK}(\langle c\rangle_T)}$) from the verifier $\mathscr{V}$, R concludes that the primary is byzantine and requests a view-change. Hence, the onus is on the primary to guarantee consensus and execution.

Node action on timeout When the timer τ_m for a node $R \in \mathfrak{R}$ expires, R concludes that the shim's primary for view v is byzantine, and it requests primary replacement by broadcasting a VIEWCHANGE message. We employ PBFT's *view-change* protocol to replace a byzantine primary. A node R's request for change of view from v to $v+1$ is only successful if it receives support of at least $2\mathbf{f}\mathfrak{R}+1$ nodes, that is, at least $2\mathbf{f}\mathfrak{R}+1$ shim nodes must broadcast VIEWCHANGE messages. Replacing the current primary requires designating another shim node as the next primary. Like PBFT, we assume nodes have a pre-decided order of becoming the primary. As a result, when the replica designated as the primary for view $v+1$ receives VIEWCHANGE requests from at least $2\mathbf{f}\mathfrak{R}+1$ nodes, it assumes the role of the primary and broadcasts a NEWVIEW message to bring all the nodes to the same state. Similarly, when a node R receives a REPLACE message from the verifier $\mathscr{V}$, it initiates the view-change protocol to replace the primary P to view v. We defer the details for the exact view-change protocol to the original PBFT paper [40].

4.5.2 Shim Nodes in Dark

If the primary P is byzantine, it may attempt to only include $2\mathbf{f}\mathfrak{R}+1$ nodes in consensus as only $2\mathbf{f}\mathfrak{R}+1$ nodes are needed to mark any request as prepared and committed. As a result, the remaining $\mathbf{f}\mathfrak{R}$ nodes will be in dark. Next, we explain what we mean by being in dark.

(i) *Node Exclusion.* A byzantine primary P can exclude up to $\mathbf{f}\mathfrak{R}$ honest nodes from consensuses by not sending them the PREPREPARE messages for client requests.

(ii) *Equivocation.* A byzantine primary P can equivocate by associating two client requests with the same sequence number k. If P is clever, it will ensure that one of these client requests is committed by at least $\mathbf{f}\mathfrak{R}+1$ honest nodes while the remaining $\mathbf{f}\mathfrak{R}$ honest nodes do not commit any request at sequence number k.

The key challenge to resolving the attack (i) is that it is *impossible to detect*. In this attack, the byzantine primary P is clever and does not want to risk replacement. Hence, P facilitates continuous consensus on incoming client requests by at least $\mathbf{f}\mathfrak{R}+1$ honest nodes. As a result, the remaining $\mathbf{f}\mathfrak{R}$ nodes are unable to trigger view-change by themselves.

Lemma 4.1. *If at most $\mathbf{f}\mathfrak{R}$ shim nodes are in dark, then it is impossible to detect such an attack and replace the primary.*

Proof. Let D be the set of shim nodes in dark, such that $|D| \leq \mathbf{f}\mathfrak{R}$. We start with the assumption that the nodes in D are able to prove that they are under an attack by the byzantine primary P and ensure P's replacement by convincing a majority of nodes to participate in the view-change protocol.

For a view-change to take place at least $2\mathbf{f}\mathfrak{R}+1$ nodes need to support such an event. As P is clever, it ensures that at least $U \geq \mathbf{f}\mathfrak{R}+1$ honest nodes continuously participate in consensus. Clearly, $U > D$, which implies that a majority of honest nodes will not request view-change. The remaining $\mathbf{n}\mathfrak{R} - U - D = \mathbf{f}\mathfrak{R}$ nodes are byzantine and will support the primary in this attack. Moreover, the nodes in set U cannot distinguish between the nodes in set D and the up to $\mathbf{f}\mathfrak{R}$ actual byzantine nodes, as the byzantine nodes can always request a view-change in an attempt to derail the system progress by replacing an honest primary. Hence, the view-change request by nodes in D will never be successful. □

Featherweight Checkpoints. To recover from nodes in dark attacks, we design a featherweight variant of existing checkpoint protocols [40, 121]. Existing BFT protocols require nodes to periodically construct and exchange CHECKPOINT messages, but these messages are *expensive* as they include all the client requests and the proof that they are committed (COMMIT messages from $2\mathbf{f}\mathfrak{R}+1$ distinct nodes) since the last checkpoint. As our shim nodes neither execute client requests nor store any data, during our featherweight checkpoint protocol, these nodes only send the signed proofs (certificates) for each committed request since last checkpoint.

Remark. The nodes in dark attacks do not make the system unsafe but put it at the mercy of the byzantine nodes, which can stop responding after several consensuses have passed; the system suffers from massive communication during recovery.

4.5.3 Verifier Flooding

As the verifier manages all updates to the data-store, it is a desirable target by byzantine components. Specifically, byzantine components can try the following ways to disrupt the system by flooding the verifier with redundant requests.

(i) *Duplicate Spawning by Primary.* If the shim's primary node is byzantine, it can spawn more executors than necessary.

(ii) *Duplicate Spawning by Non-primary.* A byzantine non-primary node that was once the primary node of shim has access to old certificates and EXECUTE messages. It can use these messages to spawn new executors at the serverless cloud.

(iii) *Duplicate Messages by Executors.* A byzantine executor can send duplicate VERIFY messages to the verifier.

Although flooding attacks seem trivial to perform, they have monetary impacts on the byzantine components. Spawning each serverless executor requires the spawner to pay a fixed amount of money. As a result, any flooding attack performed by a byzantine component will be *self-penalizing*. For example, in our architecture, each primary is paid a fixed amount per consensus by the edge application organization. Hence, a rational byzantine component will avoid this attack.

Moreover, all of these attacks are trying to flood the verifier with the VERIFY messages. To mitigate the impact of these flooding attacks: we require the verifier $\mathscr{V}$ to ignore any VERIFY message for a client request m, once it has received matching VERIFY messages for m from $\mathbf{f}\mathscr{E}+1$ executors. Finally, it is a common practice to connect different entities on the network via sockets. If flooding attacks take place, the verifier can block communication from such connections.

4.6 Transactional Conflicts

Two client transactions T and T' are termed as *conflicting* if T and T' require access to a common data-item x and at least one of these operations writes to x [149]. In our SERVERLESSBFT protocol, transactional conflicts arise from the following set of transactions: two transactions T and T' ordered at sequences k and k', respectively, and $k < k'$ and T writes to x, which T' reads.

Example 4.2. For ensuing discussions, we assume two conflicting transactions T and T'. Let the sequence number for T be 3 and sequence number for T' be 4. Further, assume T needs to write to data-item x and T' needs to read x.

4.6.1 Concurrent Spawning

On a close inspection of Figure 28, one can observe that the primary P does not wait for consensus of the k-th request to finish before initiating consensus for the $(k+1)$-th request. This process of concurrently invoking multiple consensuses has been employed by prior works to increase the system throughput as it reduces the idle times for nodes [97, 143].

To further boost the throughput, we permit the primary to spawn the $\mathbf{n}\mathscr{E}$ executors for the $(k+1)$-th request prior to spawning executors for k-th request. We term this as *concurrent spawning*. If the client requests are non-conflicting, concurrent spawning helps to parallelize execution.

In the case transactions are conflicting, like T and T' of Example 4.2, we can have two cases: the read-write sets a transaction accesses are either *known or unknown* to the shim nodes prior to execution. Depending on the knowledge of read-write sets, transactions *may or may not abort* in our architecture. A naive way would be to ask the shim primary to sequentially spawn executors for each client request, but that will significantly reduce the throughput attained by our SERVERLESSBFT protocol. Hence, we design algorithms to handle either cases, which we discuss next.

4.6.2 Unknown Read-Write Sets

If the shim nodes cannot determine the read-write sets of a transaction during consensus, we require the shim nodes to continue following the algorithm in Figure 28. The only change is that the shim's primary should spawn an *additional* $\mathbf{f}\mathscr{E}$ executors; the shim primary now spawns $\mathbf{n}\mathscr{E} \geq 3\mathbf{f}\mathscr{E}+1$ executors instead of $\mathbf{n}\mathscr{E} \geq 2\mathbf{f}\mathscr{E}+1$ as stated earlier. We prove the need for these additional executors later.

However, due to the conflicting transactions like T and T' of Example 4.2, the verifier $\mathscr{V}$ may observe the following: (i) it did not receive $\mathbf{f}\mathscr{E}+1$ matching VERIFY messages for T', or (ii) the read sets of T' are stale. In such cases, the verifier would have to abort transaction T'.

Byzantine Aborts and Decentralized Spawning. A big challenge to permitting the verifier to abort transactions is a byzantine primary that can intentionally delay spawning executors for some of the committed transactions to get them aborted. Moreover, this attack is impossible to detect by other shim nodes or the verifier. Prior works have shown that there are no easy solutions to prevent byzantine aborts for conflicting transactions with unknown read-write sets [181]. One way to prevent this attack in our serverless-edge architecture is to require each node of the shim to spawn some executors at the serverless cloud. In specific, once a node $R \in \mathfrak{R}$ commits a client request m, it spawns $\mathbf{e}$ executors.

$$\mathbf{e} = \begin{cases} 1, & \text{if } \mathbf{n}\mathscr{E} \leq \mathbf{n}\mathfrak{R} \\ \left\lceil \dfrac{\mathbf{n}\mathscr{E}}{2\mathbf{f}\mathfrak{R}+1} \right\rceil, & \text{otherwise} \end{cases} \tag{1}$$

If $\mathbf{n}\mathscr{E}$ is less than $\mathbf{n}\mathfrak{R}$, then each node $R \in \mathfrak{R}$ needs to spawn only one executor. This will guarantee that of all the spawned executors at least $\mathbf{f}\mathscr{E}+1$ are honest. Otherwise, each node R needs to spawn $\left\lceil \dfrac{\mathbf{n}\mathscr{E}}{2\mathbf{f}\mathfrak{R}+1} \right\rceil$ executors. *Why?* Because up to $\mathbf{f}\mathfrak{R}$ nodes are byzantine and may avoid spawning any executors. Hence, the remaining $2\mathbf{f}\mathfrak{R}+1$ honest nodes need to spawn $\mathbf{n}\mathscr{E}$ executors. Clearly, the total number of spawned executors ($\mathbf{e} \times \mathbf{n}\mathfrak{R}$) is much larger than the required number of executors $\mathbf{n}\mathscr{E}$. This is a trade-off we need to pay if we want to *decentralize the spawning of serverless executors*. Another major trade-off of this decentralized spawning is that if the read-write sets are known, then each node needs to sequentially spawn executors. Hence, like primary (refer to Section 4.6.3), each node has to track the dependencies. Moreover, the proposed value of $\mathbf{e}$ is only valid if each honest node commits the client request. If up to $\mathbf{f}\mathfrak{R}$ honest nodes are in dark, then $\mathbf{e}$ changes as follows:

$$\mathbf{e} = \begin{cases} 1, & \text{if } \mathbf{n}\mathscr{E} \leq \mathbf{n}\mathfrak{R} \\ \left\lceil \dfrac{\mathbf{n}\mathscr{E}}{\mathbf{f}\mathfrak{R}+1} \right\rceil, & \text{otherwise} \end{cases} \tag{2}$$

Conservatively, we can set $\mathbf{e} = \mathbf{n}\mathscr{E}$, but that will lead to spawning $\mathbf{n}\mathscr{E} \times \mathbf{n}\mathfrak{R}$ executors in the worst case.

Verifier Abort Detection. With the addition of byzantine aborts, the verifier needs to determine when to abort a transaction T' and if possible, the cause for abort. As a result, the verifier needs to wait for $\mathbf{f}\mathscr{E}+1$ matching VERIFY messages for T'.

For this purpose, our SERVERLESSBFT protocol requires the verifier $\mathscr{V}$ to start a timer τ_m when it receives the first VERIFY message for the transaction $m := T'$. $\mathscr{V}$ stops τ_m when it receives $\mathbf{f}\mathscr{E}+1$ matching VERIFY messages, or it receives VERIFY messages from all the $3\mathbf{f}\mathscr{E}+1$ executors.

Like in Figure 28, say the verifier collects all the incoming VERIFY messages for m in a set $\mathbf{V}$. If the verifier's timer expires while waiting, it takes one of the following actions:

- $|\mathbf{V}| < 2\mathbf{f}\mathscr{E}+1$: This case implies that the verifier $\mathscr{V}$ received less than $2\mathbf{f}\mathscr{E}+1$ VERIFY messages for transaction T'. As a result, $\mathscr{V}$ concludes that the primary P is byzantine and it creates and broadcasts a REPLACE message to the shim nodes. Receiving less than $2\mathbf{f}\mathscr{E}+1$ VERIFY messages implies that either the primary P spawned less than $\mathbf{n}\mathscr{E}$ executors or some messages got dropped; at most $\mathbf{f}\mathscr{E}$ executors can act byzantine and can decide to not send VERIFY messages to $\mathscr{V}$, In either case, it is safe to conservatively blame the primary. Note: even existing BFT protocols decide to blame the primary if messages get dropped [40, 84, 121].

- $\mathbf{n}\mathscr{E} > |\mathbf{V}| \geq 2\mathbf{f}\mathscr{E}+1$: This case implies that the verifier $\mathscr{V}$ received more than $2\mathbf{f}\mathscr{E}+1$ VERIFY messages for transaction T'. As a result, the verifier $\mathscr{V}$ cannot conclude that the shim's primary is byzantine as $\mathscr{V}$ has received VERIFY message from at least $2\mathbf{f}\mathscr{E}+1$ distinct executors. Observing responses from at least $2\mathbf{f}\mathscr{E}+1$ executors is a guarantee that at least $\mathbf{f}\mathscr{E}+1$ honest executors tried to execute T' to the best of their ability. Hence, even if the shim's primary is byzantine and intentionally delays spawning executors for T', there is no way that the verifier can prove this (due to concurrent spawning).

This forces the verifier to abort this transaction. Assume k is the sequence number for T'. If $k_{\max} = k$, then $\mathscr{V}$ sends the client an $\langle \mathscr{V} \rangle_{\text{ABORT}(T')}$ message. Otherwise, $\mathscr{V}$ adds T' to the list π, but ***tags*** it as abort. Later, when T' is extracted from the list π, the verifier $\mathscr{V}$ aborts it.

We now describe the indistinguishable attack, which forces us to require primary to spawn $\mathbf{n}\mathscr{E} \geq 3\mathbf{f}\mathscr{E} + 1$.

Theorem 4.3. *If client transactions are conflicting and the primary* P *spawns* $\mathbf{n}\mathscr{E} < 3\mathbf{f}\mathscr{E} + 1$ *executors, then the* SERVERLESSBFT *protocol faces an indistinguishable attack.*

Proof. Assume that P spawns only $2\mathbf{f}\mathscr{E} + 1$ executors. We know that up to $\mathbf{f}\mathscr{E}$ of these executors can act byzantine. As a result, for any client request, the verifier $\mathscr{V}$ may receive only $\mathbf{f}\mathscr{E} + 1$ VERIFY messages. Further, due to transactional conflicts, these $\mathbf{f}\mathscr{E} + 1$ VERIFY messages may not match. Eventually, $\mathscr{V}$'s timer will expire and it needs to take some action. $\mathscr{V}$ can decide to abort this transaction, but this would lead to a new problem—a byzantine primary P may never spawn more than $\mathbf{f}\mathscr{E} + 1$ executors and up to $\mathbf{f}\mathscr{E}$ of those executors may be byzantine. Hence, all subsequent conflicting transactions may abort.

Alternatively, $\mathscr{V}$ can blame the primary for receiving less than $\mathbf{f}\mathscr{E} + 1$ matching VERIFY messages, but such a decision could be wrong as P may not be byzantine and the lack of sufficient matching messages could be a result of conflicts and byzantine executors. □

4.6.3 Best Effort Conflict Avoidance

In database literature, several works have employed the concept of deterministic databases for efficient conflict resolution [183, 105, 157]. In these databases, the order in which transactions are applied to the database is determined prior to its execution, which is only possible if the read-write sets of the transactions are known to the participating nodes.

In our SERVERLESSBFT protocol, we learn from these databases. If the primary has any knowledge of the read-write sets, it uses the *queuing strategy* of these databases, to create plans that allow running non-conflicting transactions in parallel [183, 105, 158, 157, 118]. Such a strategy would require us to make straightforward modifications to the algorithm presented in Figure 28. We would need the shim primary to maintain a *logical map* of all data-items. This map does not store any values of the data-items, but helps the primary to *locally lock* different data-items. Further, the primary can no longer concurrently spawn executors for a transaction until it has determined its conflicts. Next, we list the steps.

1. The primary P adds the k'-th transaction to the execution queue after it has added (or spawned executors for) all the k-th transactions in the queue, where $k < k'$.
2. If the k'-th transaction does not conflict with any k-th transaction ($k < k'$), P spawns serverless executors for the k'-th transaction after it has logically locked all the data-items that are *written* by the k'-th transaction.
3. Next, P dequeues a non-conflicting transaction at the head of some queue and repeats Step 2.
4. When P is notified by the verifier $\mathscr{V}$ that T has been executed, it unlocks the data-items accessed by T and follows Step 3. We believe these steps can help to reduce aborts.

4.7 Safety and Liveness Guarantees

We now prove that SERVERLESSBFT guarantees safety and liveness. As the shim nodes employ PBFT protocol, we borrow the following proposition guaranteed by PBFT.

Proposition 4.4. *Let Ri, $i \in \{1,2\}$, be two honest shim nodes that committed $\langle ci \rangle_{T_i}$ as the k-th transaction of view v. If* $\mathbf{n}\mathfrak{R} > 3\mathbf{f}\mathfrak{R}$*, then* $\langle c1 \rangle_{T_1} = \langle c2 \rangle_{T_2}$.

Theorem 4.5. *Given an architecture* $\mathscr{A} = \{\text{clients}(,)\mathfrak{R}, \mathscr{E}, \mathscr{S}, \mathscr{V}\}$*, if the number of byzantine shim nodes and byzantine serverless executors are bounded by* $\mathbf{f}\mathfrak{R}$ *and* $\mathbf{f}\mathscr{E}$*, respectively, then* SERVERLESSBFT *protocol guarantees safety.*

Proof. Prior to proving this, we note that as the verifier $\mathscr{V}$ is trusted, storage $\mathscr{S}$ will be updated in the order agreed by $2\mathbf{f}\mathfrak{R} + 1$ of shim nodes. We prove the rest as follows:

Non-conflicting transactions. If the primary P is honest, then from Proposition 4.4, we can conclude that no two shim nodes will commit different transactions at the same sequence number and P will spawn $2\mathbf{f}\mathscr{E} + 1$ executors. These transactions will persist across views as in any view-change quorum of $2\mathbf{f}\mathfrak{R} + 1$ replicas, there will be one honest replica that has executed this request. If P is byzantine and assigns two or more requests the same sequence number k, then from Proposition 4.4, we know that P will not be successful. If the byzantine P sends the PREPREPARE for some T to less than $2\mathbf{f}\mathfrak{R} + 1$ replicas, this transaction will not commit. As a result, at least $\mathbf{f}\mathfrak{R} + 1$ replicas will timeout and a VIEWCHANGE will take place. The new primary waits for VIEWCHANGE messages from $2\mathbf{f}\mathfrak{R} + 1$ replicas, and uses these messages to create a NEWVIEW message. This NEWVIEW message includes a list of requests for each sequence number present in the VIEWCHANGE message. Each replica on receiving the NEWVIEW message can verify its contents and update its state.

Conflicting transactions with unknown read-write sets. In the case of conflicting transactions, the only additional attack a byzantine primary P can do is to get a transaction aborted by delaying spawning executors. However, as P does not know, which transactions are conflicting, this is all based on a guess. Note: this attack does not make the data-store unsafe. □

Theorem 4.6. *Given an architecture* $\mathscr{A} = \{\text{clients}(,)\mathfrak{R}, \mathscr{E}, \mathscr{S}, \mathscr{V}\}$*, if the network is reliable and the number of byzantine shim nodes and serverless executors are bounded by* $\mathbf{f}\mathfrak{R}$ *and* $\mathbf{f}\mathscr{E}$*, respectively, then* SERVERLESSBFT *guarantees liveness.*

Proof. Prior to proving this, we note that as the verifier $\mathscr{V}$ is trusted, so if it receives $2\mathbf{f}\mathscr{E} + 1$ matching VERIFY messages with correct read-write sets, it will send a reply to the client. We prove the rest as follows:

Non-conflicting transactions. If the primary P is honest, then every transaction will be committed by at least $2\mathbf{f}\mathfrak{R} + 1$ shim nodes. P will use this to create a certificate and spawn $2\mathbf{f}\mathscr{E} + 1$ executors and $\mathscr{V}$ will receive $\mathbf{f}\mathscr{E} + 1$ matching responses.

If P is byzantine, it can perform one of the many types of request suspension attacks described in Section 4.5.1. For each such attack, either the client c or the nodes in $\mathfrak{R}$ will timeout. This will force P to either ensure consensus of c's transaction, or be replaced through the view-change protocol. Post view-change, if the subsequent primary is also byzantine, then it will also be eventually replaced. This process can happen at most $\mathbf{f}\mathfrak{R}$ consecutive times, after which the system will be live. In the case a byzantine primary P attempts to keep up to $\mathbf{f}\mathfrak{R}$ nodes in dark, then using the featherweight checkpoint protocol these nodes will be brought to the same state.

Conflicting transactions with unknown read-write sets. In the case of conflicting transactions, the only additional attack a byzantine primary P can do is to spawn less than $3\mathbf{f}\mathscr{E} + 1$. In such a case, if the verifier $\mathscr{V}$ receives less than $2\mathbf{f}\mathscr{E} + 1$ VERIFY messages, such that less than $\mathbf{f}\mathscr{E} + 1$ are matching, $\mathscr{V}$'s timer τ_m will timeout and it will send a REPLACE message to the shim nodes. For other cases, $\mathcal{V}$ will send the client a RESPONSE or ABORT message depending on if it receives $\mathbf{f}\mathcal{E}+1$ matching VERIFY messages. □

4.8 Implementation

To gauge the practicality of our vision of a BFT serverless-edge architecture, we implement and evaluate our design.

Shim. As the shim nodes represent edge devices, which may have access to limited resources, we want the shim nodes to have a lightweight BFT implementation. So, on each shim node, we install ResilientDB's node architecture [97, 96, 100, 91, 159, 92]. ResilientDB provides access to a multi-threaded, pipelined, and modular architecture for designing BFT applications.[5] The codebase is written in C++ and we deploy ResilientDB's PBFT protocol at the shim. Clients also employ C++ to create YCSB transactions (refer Section 4.9) and use NNG [57] sockets for communication.

Invoker. At each shim node, we deploy an invoker to spawn $\mathbf{n}\mathcal{E}$ executors when indicated by the node's consensus instance. ResilientDB provides at each node an *execute-thread*, which calls invoker as soon as a request is committed. Our implementation of the invoker is written in Go [69] using the AWS SDK for Go. Further, our invoker does not wait for the spawned executors to finish and proceeds to spawn the executors for the next client request.

Serverless Function. Each AWS Lambda executor receives a function written in C++ that includes the client transaction. This function instructs the executor to: (i) verify the certificate $\mathfrak{C}$, (ii) execute the transaction, (iii) fetch necessary read-write sets from the storage database, and (iv) send the result to the verifier. We encode the communication between the Lambda function and the verifier in a stateless HTTP request. We use CryptoPP[51] library for digital signatures and verification and use CPR[170] to create and send HTTP requests.

Verifier. We implement the verifier in Go and install a simple HTTP/Net webserver at the verifier for receiving the executor responses. Further, our verifier includes a *hashmap* to count the matching responses for each transaction. Post validation, the verifier uses NNG to send a response to the client.

4.9 Evaluation

Our evaluation aims to answer following questions regarding our SERVERLESSBFT protocol.

(Q1) Impact of client congestion?
(Q2) Impact of increasing the number of executors?
(Q3) Impact of batching client requests?
(Q4) Impact of expensive execution?
(Q5) Impact of spawning executors across globe?
(Q6) Impact of resource limitations at edge devices?
(Q7) Impact of conflicting transactions?
(Q8) Baseline comparison of SERVERLESSBFT?
(Q9) Impact of task offloading?

Setup. We deploy the verifier, shim nodes, and clients on the Oracle Cloud Infrastructure (OCI). These components use *VM.Standard.E3.Flex* architecture with 10 GiB NICs. Each shim node has 16 cores and

[5]ResilientDB is open-sourced at *https://resilientdb.com/*.

16 GiB RAM and the verifier has 8 cores. We use AWS Lambda Functions for spawning serverless executors in up to 11 regions in the following order: North California, Oregon, Ohio, Canada, Frankfurt, Ireland, London, Paris, Stockholm, Seoul, and Singapore. In our experiments, we use up to 128 shim nodes and 21 executors. We run each experiment for 180 seconds with 60 seconds warmup time and report the average results over three runs.

Unless *explicitly* stated, we use the following setup. We require the primary node to spawn 3 AWS Lambda executors, each of which is spawned in a distinct region. Further, we deploy up to 80 k clients on 4 OCI machines to concurrently issue requests. Each client waits for a response prior to sending its next request. We also require clients and edge nodes to employ *batching* and run consensuses on batches of 100 client transactions. The size of each type of message communicated is: PREPREPARE (5392 B), PREPARE (216 B), COMMIT (220 B), EXECUTE (3320 B), and RESPONSE (2270 B).

Benchmark. To evaluate our serverless-edge architecture across different parameters, for some experiments, we need to fix the number of shim nodes. We learn from existing database literature, specifically the Blockbench [65] paper, and select two configurations. **SERVBFT-8**: Medium size shim with 8 nodes. **SERVBFT-32**: Large size shim with 32 nodes (maximum number of nodes in any Blockbench experiment).

Similarly, we adopt the popular *Yahoo Cloud Serving Benchmark* (YCSB) from Blockbench suite, which has also been used by several prior works in database literature for designing transactions [97, 183, 105, 157, 65, 53, 103]. We use YCSB to create key-value transactions that access a database of 600 k records. Specifically, our transactions perform read and write operations. With regards to edge applications, these transactions represent user transactions that require access to existing records in the storage.

A. Impact of Client Congestion. In Figure 32, we vary the number of deployed clients from 2 k to 88 k. For the first five data-points on the graph, we double the number of clients and for succeeding points, we increase the number of clients by 8 k. Initially, an increase in the number of clients causes an increase in system throughput, post which the throughput saturates. This happens because *each entity in our serverless-edge architecture has to now do more work than before, which causes an increase in computational and communication costs.* As a result, the latency keeps increasing as each request spends a longer time in the architecture. Hence, SERVBFT-8 outperforms SERVBFT-32 as *fewer nodes are involved in each consensus, which implies smaller wait time for each request. Summary:* We observe that initially SERVBFT-8 attains up to 1.6× more throughput and 1.2× less latency than SERVBFT-32. However, on increasing the number of clients, the gap increases to 2.8× more throughput and 2.71× less latency.

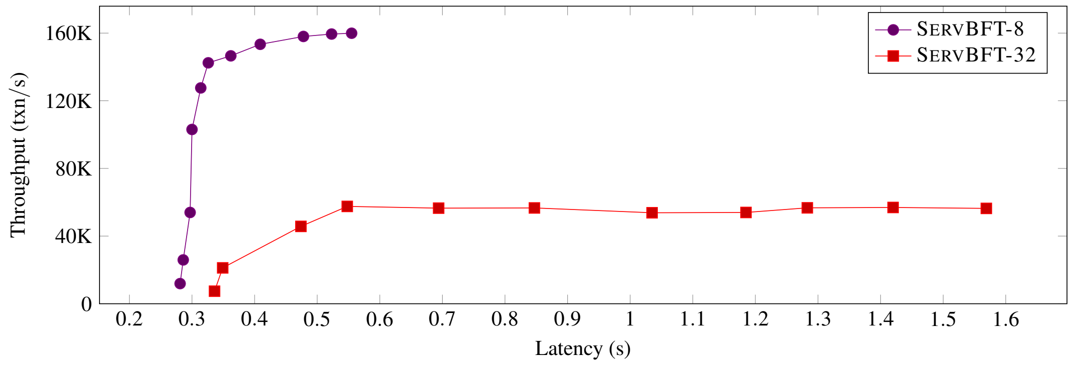


Figure 32: Comparing latency against throughput on varying the number of clients sending requests to the shim.

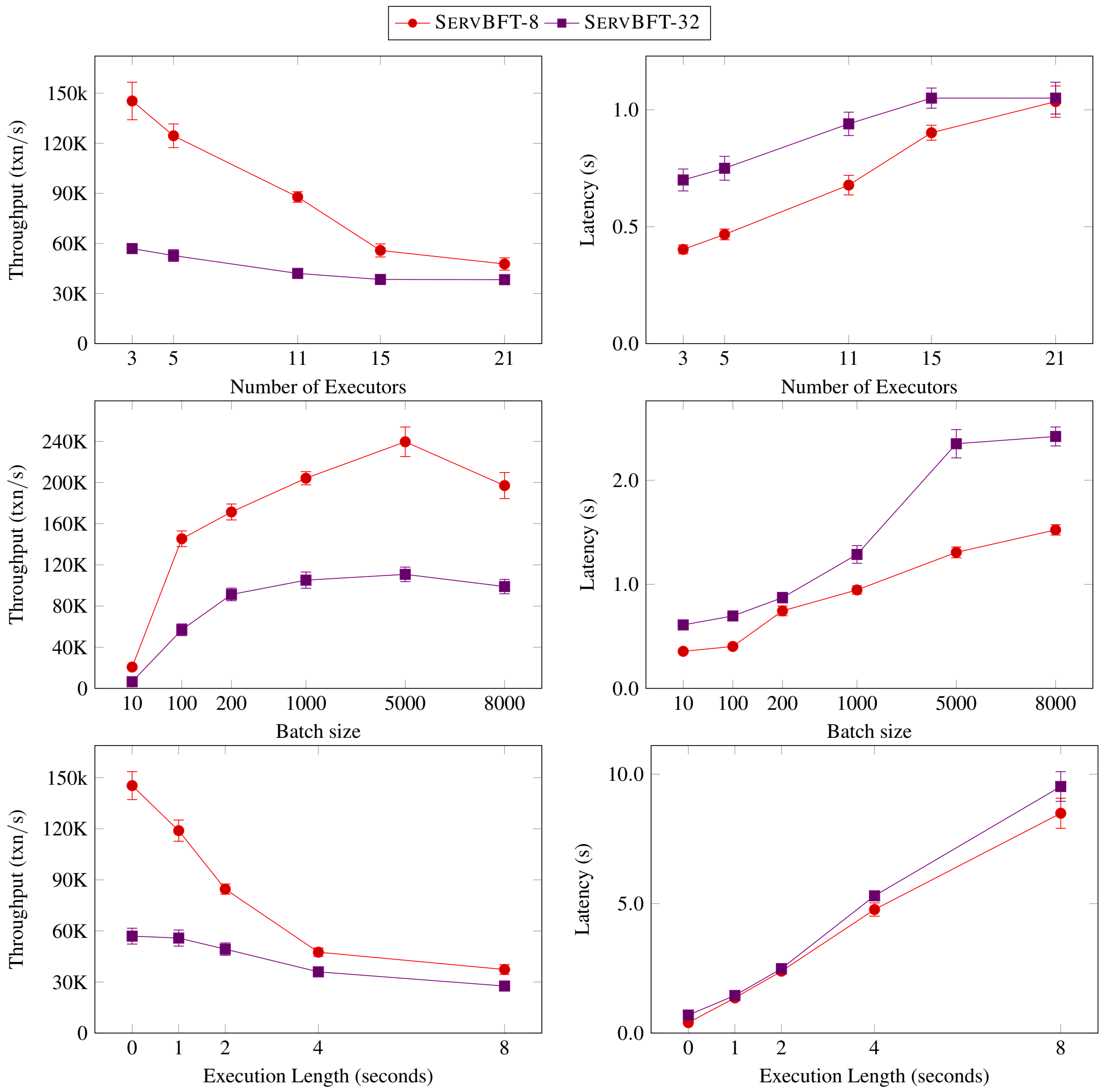


Figure 30: Benchmarking throughput attained and latency incurred by the BFT Serverless-Edge architecture (1).

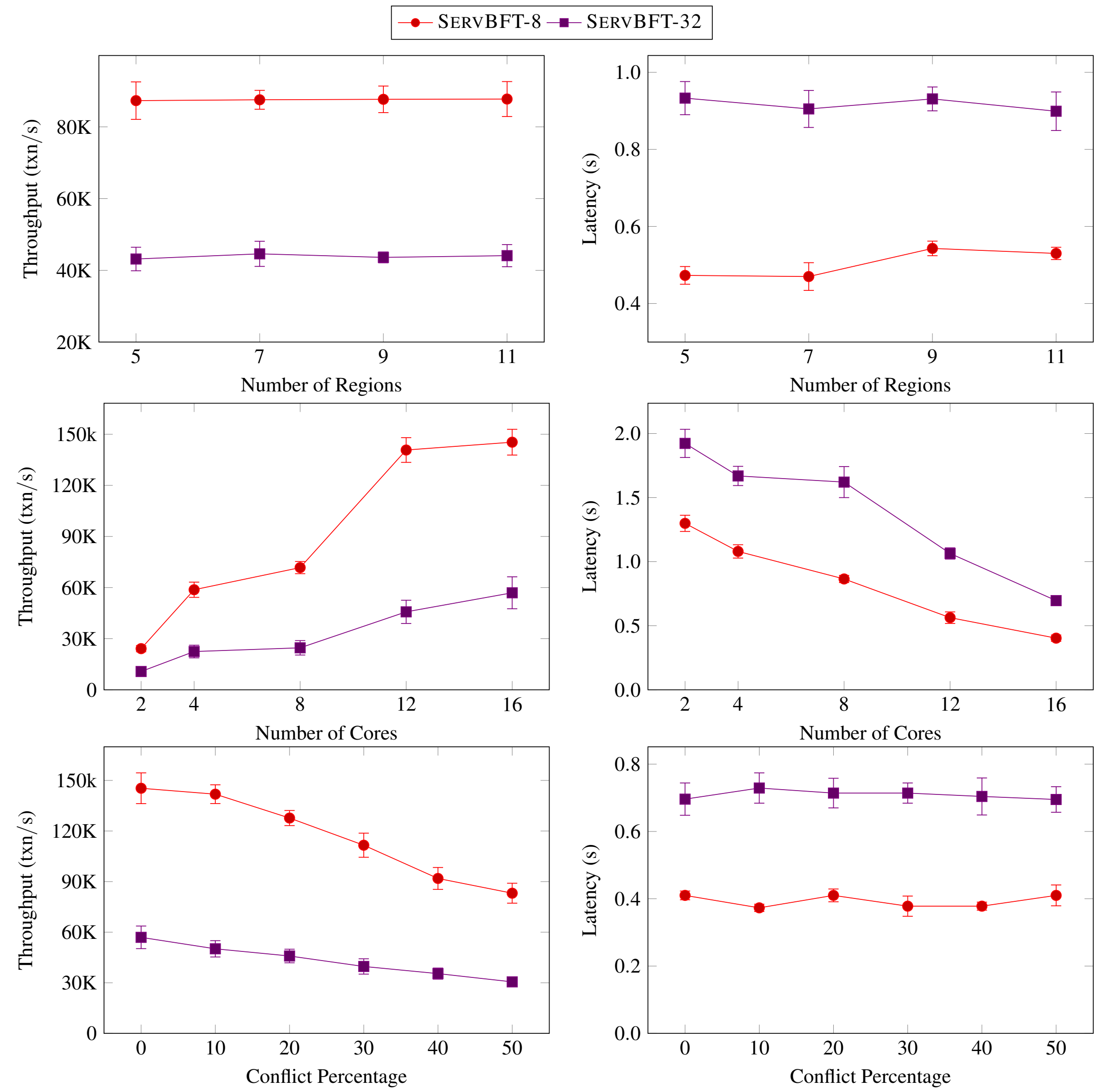


Figure 31: Benchmarking throughput attained and latency incurred by the BFT Serverless-Edge architecture (2).

B. Impact of Executors. In Figure 33, we vary the number of serverless executors spawned by the primary node: 3, 5, 11, 15, and 21. For these experiments, we spawned executors in up to *seven* regions and tried to evenly split these executors across these regions. These figures illustrate that an increase in the number of executors causes a decrease in throughput and an increase in latency. Although all the executors process the requests in parallel, *there is an increase in the task of spawning at the primary and increase in validation at the verifier.* Further, as executors are spread across distinct regions, *the reduced bandwidth and increased ping costs* delays communication. *Summary:* At 3 executors, SERVBFT-8 attains 2.59× more throughput and 43% less latency than SERVBFT-32, while at 15 executors, 47% more throughput and 5% less latency.

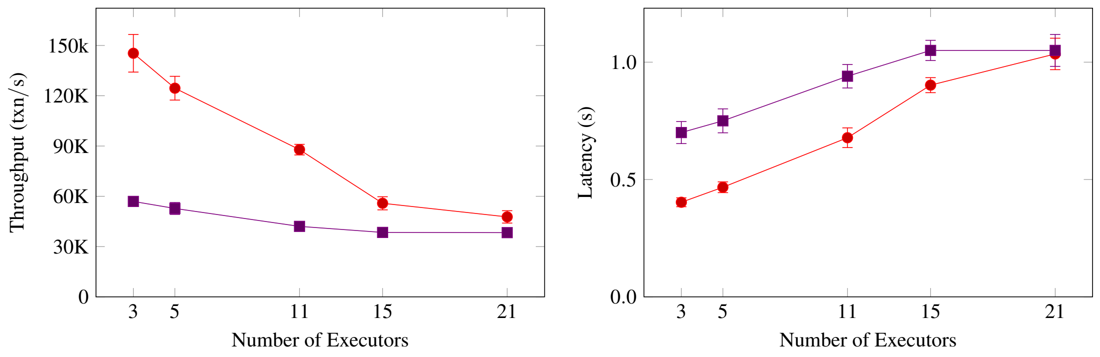


Figure 33: Impact of Scaling Executors

C. Impact of Batching. In Figure 34, we vary the size of batch of client requests from 10 to 8 k. With an increase in batch size, we first observe an increase in the system throughput followed by an eventual decrease. Although larger batches imply a corresponding decrease in the number of runs of the SERVERLESSBFT protocol, *it substantially increases the costs of communicating batches across the shim nodes and executors.* Further, larger batches are much more expensive to process for shim nodes and executors.

Summary: From batch size 10 to 5 k, SERVBFT-8 observes an increase in throughput by 11.42× and SERVBFT-32 observes an increase in throughput by 18.5×.

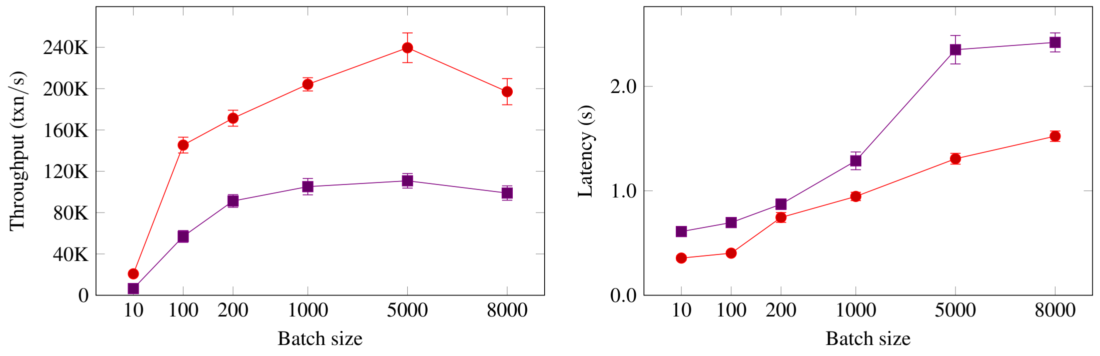


Figure 34: Impact of Batching

D. Impact of Expensive Execution. In Figure 35, we test with transactions that require large execution time; we vary the time required for execution from few milliseconds to 8 seconds. As the time required to execute a transaction increases, the time required by the shim and the verifier to *process this request becomes insignificant.* Prior works show that such transactions or codes, which bottleneck the system throughput and latency are prevalent [10]. This experiment also proves that *our serverless-edge architecture introduces minimal costs* to the applications that require large execution times. *Summary:* From execution length of few milliseconds to 8 seconds, SERVBFT-8's throughput reduces by 74.5% and latency increases by 21×, while SERVBFT-32's throughput reduces by 51% and latency increases by 13.6×.

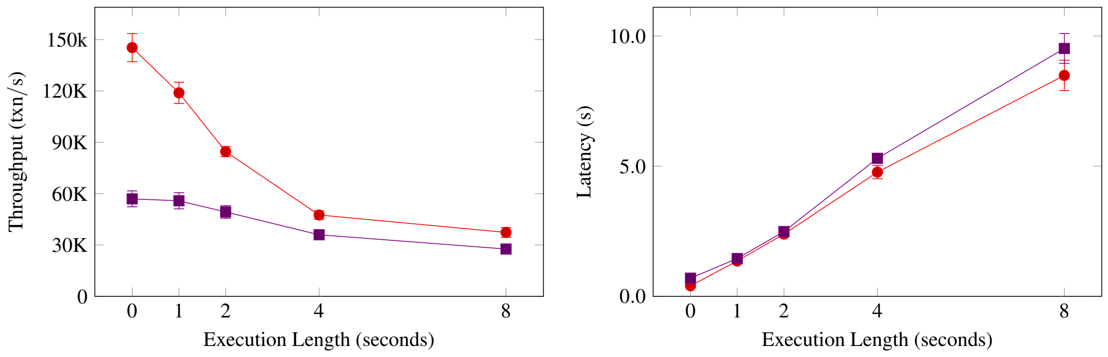


Figure 35: Impact of Expensive Execution

E. Impact of Spawning Executors across Globe. In Figure 36, we require the primary node to spawn 11 executors in 5, 7, 9, and 11 regions; we vary the number of regions while spawning same number of executors. The primary node uses the round-robin protocol to spawn executors in each region. In this experiment, we want to observe the impact of system performance on increasing the number of regions. We observe that the *throughput and latency remain constant.* The primary node spawns 11 executors ($\mathbf{f}\mathscr{E} = 5$), so the verifier needs to wait for only $\mathbf{f}\mathscr{E} + 1 = 6$ matching VERIFY messages. The first 6 messages received by the verifier (deployed at North California) are from nearby regions: North American and European.

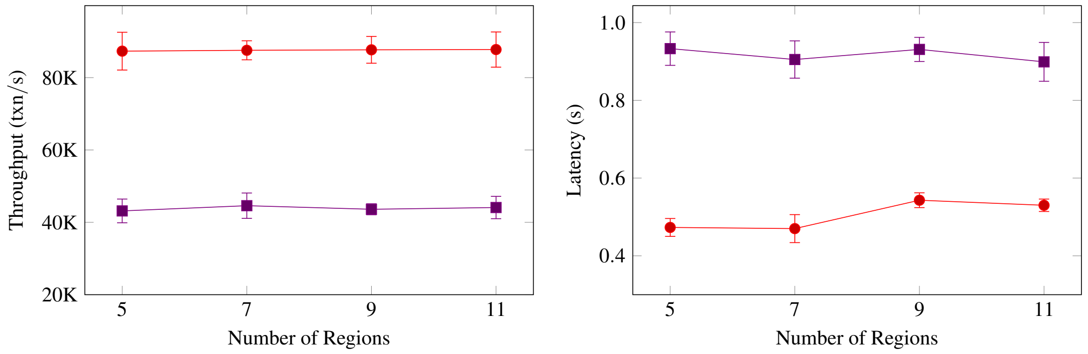


Figure 36: Impact of Executor Distribution

F. Impact of Computing Power. We use Figure 37 to limit the available computing resources at shim nodes. As shim nodes represent edge devices, these devices may have limited cores and memory. So we test the impact of this restricted hardware om SERVERLESSBFT. Unsurprisingly, as we increase the number of available cores, the protocols achieve higher throughputs and lower latencies. This is the case because *our* *shim nodes adopt the multi-threaded pipelined architecture* of ResilientDB, which performs better with an increase in available cores.

Summary: From experiments at 2 cores to 16 cores, SERVBFT-8's throughput increases by $6\times$ and latency decreases by 70%, while SERVBFT-32's throughput increases by $5\times$ and latency decreases by 64%.

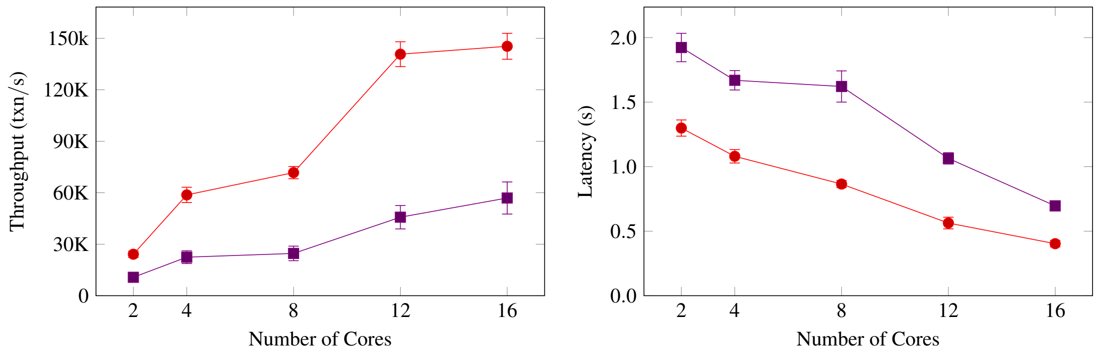


Figure 37: Impact of Computing Power

G. Impact of Conflicting Transactions. We now vary the degree of transactional conflicts from 0 to 50% and illustrate our findings in Figure 38. *As the read-write sets are unknown, the primary node cannot logically lock these transactions, so they will get aborted at the verifier.* Hence, we observe a decrease in throughput with an increase in the rate of conflicts. However, the latency remains unchanged as the response time for the client remains the same.

Summary: From 0% conflicting transactions to 50% conflicting, SERVBFT-8's throughput decreases by 43%, and SERVBFT-32's throughput decreases by 46%.

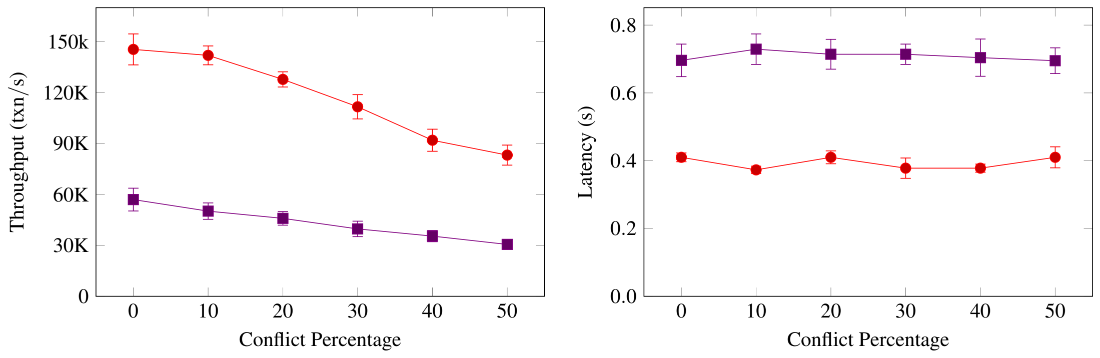


Figure 38: Impact of Conflicting Transactions.

H. Shim Scalability. Until now, in all the experiments, we ran the PBFT protocol at the shim. So, we create *three baseline designs* to compare against SERVERLESSBFT:

(a) **NOSHIM**– Represents the experiment where there is no shim; no BFT consensus takes place. All the clients send their requests to a node, which instantaneously spawns executors.

(b) **SERVERLESSCFT**– Represents the experiment where the shim nodes employ a crash fault-tolerant (CFT) like Paxos [124] for consensus. As CFT protocols do not protect against byzantine attacks, they do not require cryptographic signatures, which in turn reduces the amount of work done per consensus. Further, unlike PBFT, Paxos is linear.

(c) PBFT– We also test our SERVERLESSBFT protocol against a BFT system (e.g. ResilientDB) running the PBFT protocol. In this system, we assume each node is a replica and executes the request in the agreed order post consensus [40, 97]. As a result, there are no costs associated with spawning executors and waiting for verifier to validate the requests.

In these experiments, we also gauge how the shim scales with an increase in the number of edge devices. For this purpose, we vary the number of shim nodes from 4 to 128. We use Figure 39 to illustrate the throughput and latency metrics and observe the following order for throughput attained:

$$\text{SERVERLESSBFT} < \text{PBFT} < \text{SERVERLESSCFT} < \text{NOSHIM}$$

NOSHIM has a constant throughput because there is no change in the number of shim nodes. Moreover, PBFT performs slightly better than our SERVERLESSBFT protocol. This implies that the *verifier and executors do not adversely impact the throughput of* PBFT. Finally, SERVERLESSCFT outperforms PBFT, which implies that *the throughput of the serverless-edge architecture can be increased by replacing* PBFT *with faster consensus protocols. Summary:* SERVERLESSBFT and SERVERLESSCFT achieve up to 22% less throughput and 1.25× more throughput than PBFT, respectively.

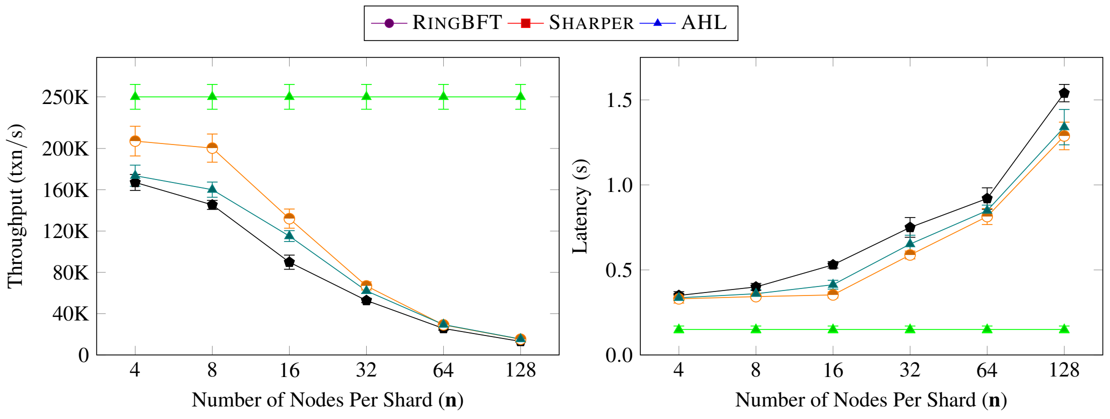


Figure 39: Comparing SERVERLESSBFT against our three baseline designs: SERVERLESSCFT, PBFT and NOSHIM.

I. Impact of Task Offloading. We use Figure 40 to illustrate the benefits of employing our serverless-edge model. Specifically, we introduce compute-intensive tasks (increasing execution time) and compare the peak throughput and monetary costs against setups where all the computations (PBFT consensus and transaction execution) are done on the edge devices (no serverless). We make two observations: (1) If transactions can be executed in parallel, our serverless-edge model is only bounded by the rate of consensus and the number of executors that can be spawned in parallel. This is in contrast to setups where shim performs all tasks and becomes resource-bounded, which adversely decreases the throughput. To further validate this resource-boundedness, we calculate monetary costs of these experiments (in cents/ktxn) and use the precise costs for spawning serverless executors at AWS Lambda and running machines on OCI. Resource-boundedness increases monetary costs as machines need to be run for a larger period of time to complete the same set of transactions. (2) Serverless clouds permit selecting optimal hardware. To illustrate this, for experiments where shim does all tasks, we vary the number of execution threads (ET) at shim nodes $(1, 8, 16)$. If the available hardware has few cores, then a smaller set of transactions (1 or 8) can execute in parallel, which impacts throughput. Alternatively, an enterprise can require edge devices to have more cores (16), which may be underutilized if there is less available parallelism.

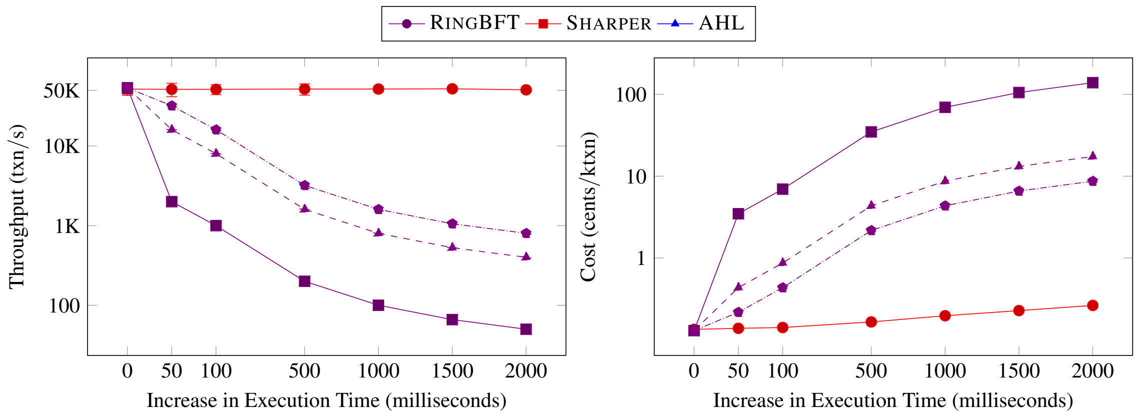


Figure 40: Comparing our serverless-edge model against PBFT. Here, ET refers to number of execution threads assigned to specific PBFT implementation.

4.10 Serverless Related Work

Edge computing is a decade old problem for which prior works have presented several interesting solutions [193, 197, 206, 141, 152, 191]. These solutions aim to reduce latency for edge applications, but they cannot handle byzantine attacks and require developers to perform managerial tasks.

In recent years, *Serverless computing* has also gained a lot of interest with the aim of offloading the managerial tasks such as server provisioning and resource scaling to the cloud provider while the developer only uploads the code required to be executed [20, 194, 114]. Prior works have presented novel solutions in this direction: AFT [178] introduces a shim to make stateful executors consistent; PolarDB [37] presents a serverless database; and Faasm [174] aims to design efficient stateful executors. However, neither these works target edge applications, nor they consider byzantine attacks.

To design applications that can handle byzantine attacks, existing works have employed *Byzantine Fault-Tolerant* consensus protocols in the context of *blockchain technology* [8, 4, 75, 204, 195, 18, 131, 165, 31, 198, 86, 128, 201, 9, 132, 205]. These applications assume that a set of nodes holding the same data run a BFT protocol. Each committed transaction is noted in an append-only ledger, blockchain, which can be queried in future to track transactions. EdgeChain [151] introduces a blockchain layer in the edge-compute model, which allocates the resources to edge devices. However, it does not tackle byzantine attacks from edge-clouds. Bajoudah et al [19] introduce a blockchain-based edge model where IOT devices maintain the blockchain. Blockene [168] wants to allow mobile devices to participate in blockchain consensus by delegating all the storage, computation, and communication tasks to a set of powerful servers.

ChainFaas designs a volunteer-based serverless cloud that reduces the work of existing serverless providers by allowing existing devices to contribute to serverless computing. To securely log each transaction, ChainFaas maintains a blockchain network. However, ChainFass assumes that devices will be non-faulty and does not tackle byzantine failures.

Aslanpur et al. [12] present the vision of a serverless-edge framework. Their proposal does not assign tasks to edge devices and delegates all jobs to the serverless cloud. Further, there is no discussion on handling byzantine failures. Moreover, their vision is neither implemented nor does their paper present any evaluation. Baresi et al. [21] present a similar design, but their design focusses on mobile computing. They do present a small evaluation of their design, but neither is their code available, nor do they make use of actual serverless cloud providers (like AWS). Their design delegates everything to the mobile edge servers (where they create a serverless cloud) and does not handle byzantine failures. Our NoShim experiment (Figure 39) approximates their architecture.

In comparison, our serverless-edge co-design handles byzantine attacks, permits edge devices to select any serverless provider in vicinity, offloads compute-intensive tasks to cloud while allowing light-weight ordering on edge devices.

4.11 Conclusions

In this chapter, we presented SERVERLESSBFT, the first protocol to guarantee Byzantine Fault-Tolerant transactional flow among edge devices and serverless functions. SERVERLESSBFT facilitates collaboration among edge devices, which spawn serverless executors at one or more cloud providers in their vicinity to process compute-intensive operations. Our proposed architecture ensures that only consistent updates are written to the database. We also present solutions to resolve various attacks on our proposed architecture. Our extensive evaluation illustrates that our architecture is scalable and is a good fit for the emerging edge applications.

5 ResilientDB, Design and Implementation

Since the inception of *blockchain* [102, 65], the distributed systems community has renewed its interest in the age-old design of Byzantine-Fault Tolerant (BFT) systems. At the core of any blockchain applications is a BFT algorithm that ensures all the replicas of this blockchain application reach a *consensus*, that is, agree on the order for a given client request, even if some of the replicas are byzantine [40, 121, 95, 82, 10]. Surprisingly, even after a decade of its introduction and publication of several prominent research works, the major use-case of blockchain technology 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. Prior works [103, 158, 157, 104] have shown that the traditional distributed systems can achieve throughputs of the order 100K transactions per second while the initial blockchain applications, such as Bitcoin [139] and Ethereum [190], have throughputs of at most ten transactions per second. Such low throughputs do not affect the users of these applications, as the aim of these applications is to promote an alternative currency, which is unregulated by any large corporation, that is, anyone can join, and the identities of the participants are kept hidden (*open membership*). Evidently, this open-membership property has also led to several attacks on these applications [65, 156, 71].

This led to industry-grade *permissioned* blockchain systems, where only a select group of users, some of which may be untrusted, can participate [10]. However, the throughputs of current permissioned blockchain applications are still of the order 10K transactions per second [10, 7, 58]. Several prior works blame the low throughput and scalability of a permissioned blockchain system on to its underlying BFT consensus algorithm [65, 121, 200, 58]. 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 [40], if implemented on skillfully-optimized blockchain fabric, can outperform a fast niche-case and optimized for fault-free consensus, BFT protocol, such as ZYZZYVA [121]. We use Figure 41 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 [27] 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.

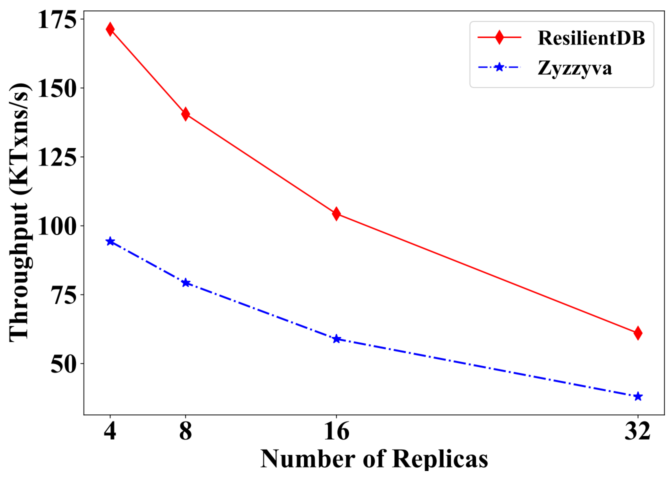


Figure 41: Two permissioned applications employing distinct BFT consensus protocols.

This work is aimed at illustrating that the design and architecture of the underlying system are as important as optimizing BFT consensus. Decades of academic research and industry experience has helped the community in designing efficient distributed applications [36, 81, 144, 166]. We use these principles to illustrate the 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 by 10%.
- Out-of-order processing of client transactions can help gain 60% more throughput.
- Protocols optimized for fault-free cases 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. [6] We now enlist our contributions:

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

[6] ResilientDB is available and open-sourced at https://resilientdb.com.

5.1 Dissecting Permissioned Blockchain

Most of the strategies we discussed in the previous section focussed at: (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 [7, 58, 202]. Due to the sustained reduction in hardware cost (as a consequence of Moore's Law [138]), 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 [7, 112], while others promote aggregating client requests into large batches [10, 139]. 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 [40, 108]. 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 [107]. 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 blockchain application:

- Sign a message.
- Verify an incoming message.
- Generate the digest of a client request.

• Hash a record or data.

To sign and verify a message, a blockchain application can employ either symmetric-key cryptography or asymmetric-key cryptography [120]. 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 [120]. Hence, several works suggest using DSs [10, 7, 58, 202]. 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 Implementation

5.2.1 Architecture

In this section, we present the architecture and capabilities of our ResilientDB fabric. ResilientDB is written entirely in C++ and provides a GUI to ease user interaction with the system. Further, we also provide a *Dockerized* deployment that allows any user to experience and test the ResilientDB fabric (comprising of multiple replicas and clients) on its local machine. In Figure 42, we illustrate the overall architecture, which we describe in detail next.

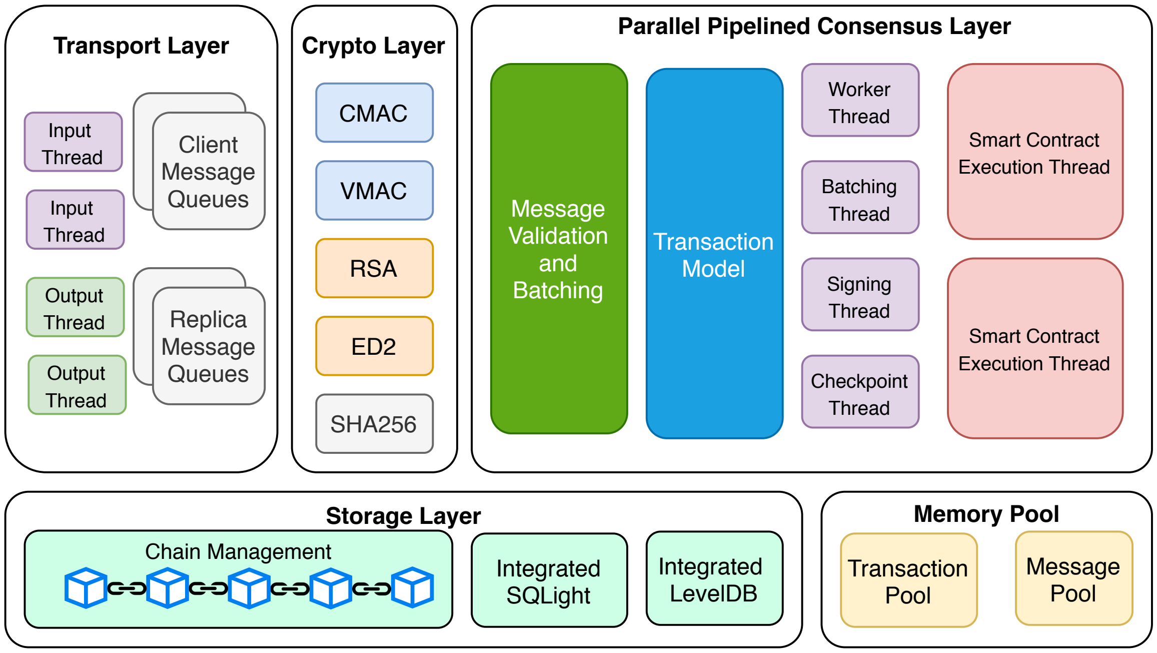


Figure 42: Architecture of the ResilientDB fabric.

• **Transport Layer.** Permissioned blockchains use *com-munication-intensive* BFT consensus protocols. Hence, they expect an efficient transport layer to facilitate exchange of messages between replicas.[7] ResilientDB employs Nanomsg sockets to facilitate communication among clients and replicas via TCP or UDP

[7]Permissionless systems are compute-intensive as they run consensus by solving a complex cryptographic puzzle.

(depending on the choice of the developer). We also provide support for fast *RDMA* communication for replicas with RDMA capabilities.

To facilitate efficient communication, ResilientDB employs multiple input/output threads with dedicated sockets. Note that the number of input/output threads can be readily adjusted based on the network requirements and buffering bottlenecks. ResilientDB also provides access to distinct message queues. Depending on the type of a message, these queues can be used by different threads to communicate with each other and to place the message on the network.

• **Crypto Layer.** Blockchains typically are designed to deal with malicious adversaries. To secure communication and prevent message tampering, ResilientDB employs NIST-recommended cryptographic constructs from the *Crypto*++ library. Depending on specific needs, replicas and clients can digitally sign their messages using either (i) asymmetric-key cryptographic schemes such as ED25519 or RSA; or (ii) symmetric-key cryptographic schemes such as CMAC and AES [120]. ResilientDB also provides message digests via either SHA256 or SHA3 hashes.

• **Parallel Pipelined Consensus Layer.** At the core of any permissioned blockchain application lies a BFT consensus protocol that safely replicates client transactions among all replicas. Decades of research has brought forth several such protocols. No one protocol is the best-fit in all environments, however. For example, ZYZZYVA achieves high throughput if none of the replicas are faulty, HOTSTUFF [200] works well if latency is not critical, GEOBFT [97] scales well when replicas are geographically distant, and PBFT, although typically-considered too slow, is most robust against failures. These characteristics of existing BFT protocols permit us to conclude that any resilient permissioned blockchain fabric should *facilitate testing and implementation of different* BFT *protocols*. ResilientDB's consensus layer allows this and to support this claim we provide implementation of all of the aforementioned protocols (among many others).

Furthermore, as we argued in previous sections, there is more to a blockchain system than just its BFT protocol. In specific, we showed that a permissioned blockchain fabric adopting a system-centric design and employing a slow BFT protocol outperforms a protocol-centric fabric that uses a fast protocol. To yield such a system-centric design, ResilientDB employs transaction batching, multi-threading, pipelining, out-of-order processing, and memory pooling.

In Figure 43, 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 work 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.

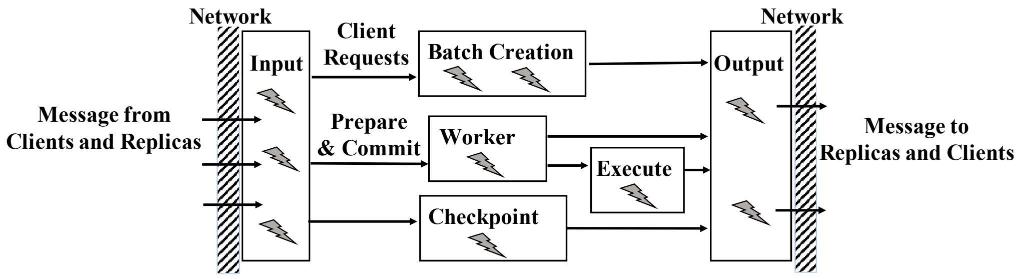


Figure 43: Multi-threaded deep-pipelines for the ResilientDB replicas.

Batching. Prior works have batched client transactions to reduce the cost of consensus [40, 121]. We permit batching of transactions at both clients and replicas and developers can specify any size for such batches. Batching reduces both communication costs and computation costs by reducing the number of messages that are exchanged (which also reduced the number of message signatures necessary).

Transactions and Smart Contracts. ResilientDB supports YCSB transactions and customized Smart Contracts. YCSB transactions can be used for benchmarking performance and developers can easily vary the skew (read/write percentage) of these transaction. ResilientDB also provides APIs for designing and testing Smart Contracts, which are similar to stored procedures in databases [64]. To demonstrate this, we implemented Ethereum's account-based smart contracts for banking applications [190] (see Section 5.2.2).

We associate each transaction with a *transaction manager* that manages the resources required for handling transactions. We provide fast lookup of transaction managers via indices on transaction identifier and batch identifier. Furthermore, transaction managers are pooled and reused to save on allocation and deallocation costs.

Order-Execute vs. Execute-Order Paradigm. Traditional permissioned blockchain systems employ the order-execute paradigm, which states that a transaction needs to be ordered across all replicas prior to its execution [34, 88]. This is in contrast with the execute-order paradigm proposed by Hyperledger [10], which advocates to first execute and then order the transaction. Both of these paradigms have their pros and cons. Our ResilientDB fabric provides support for both paradigms and allows developers to select the paradigm that best fits their applications.

Multi-Threaded Deep Pipeline. As stated before, permissioned blockchain systems are communication-intensive. Hence, we ensure that our ResilientDB fabric is not underutilizing hardware and will only be limited by network capacity. To do so, we designed the consensus protocols in ResilientDB such that the critical path is as simple as possible and all other tasks are split of in their separate threads. E.g., threads to deal with message sending, with message receiving, signing messages, verifying signatures, creating transaction batches, performing checkpoints, and executing transactions. Users can easily adjust the number of required threads depending on the specific needs of their applications.

• **Memory Pool.** Blockchain systems that process thousands of transactions, smart contracts, and messages per second require high-performance management of memory resources. For ResilientDB's memory management, we employ Jemalloc. Further, we minimize memory allocation and deallocation by using distinct memory pools for messages, transaction managers, and smart contracts. Depending on the size of an allocation, each thread accesses the required pool and fetches an unallocated memory object. At the time of garbage collection, obsolete objects are marked as released and placed back in the respective memory pool for reuse. The best practice to overcome memory management for the transactions instances, smart contracts, and messages is to have memory pool with efficient data structure to fetch and put back these memory units.

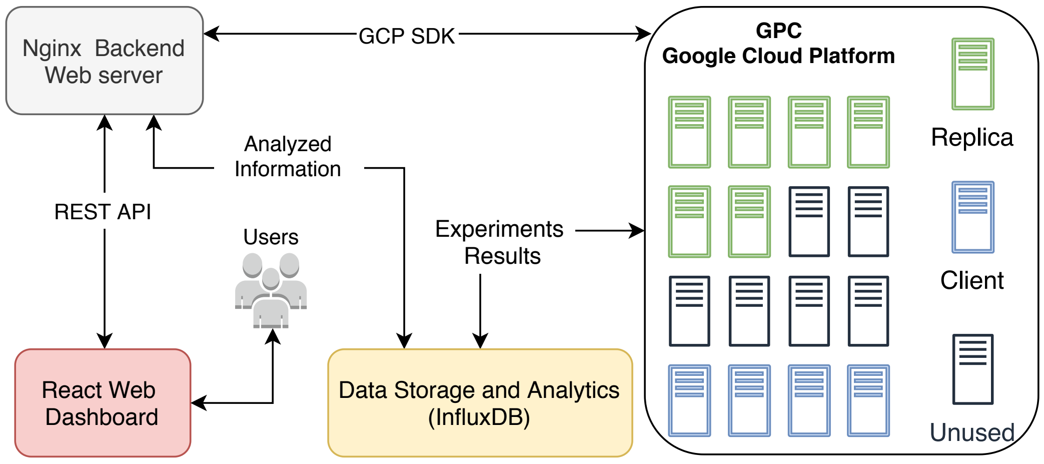


Figure 44: Demo Platform Architecture.

ResilientDB provides multiple pools for the transaction workloads, messages, and smart contract to avoid initializing and allocating memory for every instance. It is shown that memory pools improve the throughput of whole system by 2 order of magnitude.

- **Chain Storage.** ResilientDB provides support for secure ledger (blockchain) management. To enable efficient execution of client transactions, we also support efficient read and write accesses to client records. In specific, each replica can use popular databases such as SQLite and LevelDB to achieve data persistence for the ledger and client records under failure. ResilientDB provides several simple APIs that allow developers to read and write to these databases and modify their schema if necessary.

5.2.2 Demonstration Scenario

During our demonstration, each user will get access to a graphical web-based interface of ResilientDB. Figure 44 illustrates the architecture of our demonstration environment. We provide a web-based UI for specifying experiment parameters, for monitoring the real-time throughput and latency of the system, and for the analysis of collected data.

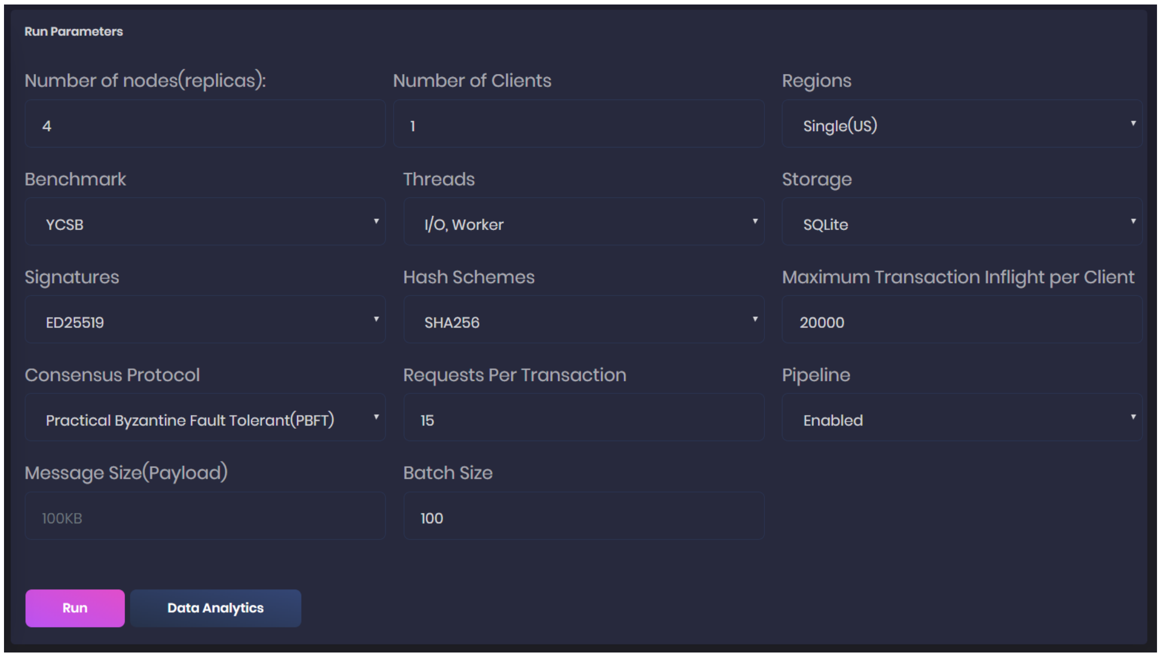
In specific, users can specify their choice of parameters on our React Web Dashboard, which uses REST APIs to forward these parameters to our Nginx back end. The back end compiles the code and deploys the executables on the Google Cloud Platform (GCP). Once the executables start running, any emitted result is continuously stored in InfluxDB. Throughout this process, our dashboard shows the user the current state of the system. If the user wants to visualize the ongoing results, our dashboard asks the back end to fetch the data from InfluxDB and plot the required graphs. This allows us to show the user real-time system throughput and latency metrics. We employ React (open-source), Nginx (performance), and InfluxDB (eases management of time-series data) for their associated advantages.

We provide our users access to *two* demonstration scenarios. The first demonstration scenario focuses on making users understand the different parameters that affect the performance of a blockchain fabric. The second demonstration scenario allows interested users to create and deploy their own smart contracts on-the-fly. We explain these next.

Mix-and-Match The key ***takeaway*** of the mix-and-match demonstration is to make users experiment and observe the different parameters that affect throughput (transactions per second) and latency (time from the client request to the response) of a permissioned blockchain application. We give users a GUI (see Figure 46) and ask them to *mix-and-match* the parameters listed in Figure 45.

Parameter	**Options**	
BFT Protocol	PBFT, Zyzzyva, GeoBFT, HotStuff	
Transactions	YCSB, Banking Smart Contracts	
Requests/Txn	1, 5, 15, 50	
Batch Size	1, 10, 100, 500, 1000, 4000	
Message Size	0 kB, 100 kB, 200 kB, 400 kB	
Pipeline	Enable or Disable	
Threads	I/O, Worker, Execute Sign, Checkpoint	
Storage	In-memory, SQLite, LevelDB	
Signatures	Disable	
	Only Asymmetric	ED25519, RSA
	Only Symmetric	CMAC, VMAC
	Mix	Use both
Hash Schemes	SHA256, SHA3	

Figure 45: ResilientDB parameters.


Run Parameters

Number of nodes(replicas): 4

Number of Clients: 1

Regions: Single(US)

Benchmark: YCSB

Threads: I/O, Worker

Storage: SQLite

Signatures: ED25519

Hash Schemes: SHA256

Maximum Transaction Inflight per Client: 20000

Consensus Protocol: Practical Byzantine Fault Tolerant(PBFT)

Requests Per Transaction: 15

Pipeline: Enabled

Message Size(Payload): 100KB

Batch Size: 100

Figure 46: The interactive WebUI dashboard.

We first require the user to ***Sign-up/Sign-in*** to our ResilientDB portal. Next, the user can ***Configure*** experiments of its choice. To do so, the user first selects a ***BFT*** protocol. At present, we have already implemented four state-of-the-art protocols. Next, the user decides whether it wants clients to send YCSB transactions or to run Banking Smart Contracts. The user can set the number of *requests* each client includes in its transactions and the *size of the batch* (if batching is employed by replicas). To test the limit of the network, we also provide capability to add a *predefined load* to messages.

We allow our users to select whether they want to *enable or disable pipelining*. Enabling pipelining in ResilientDB allows the various phases of a BFT protocol to be executed in parallel. For example, PBFT is a three-phase *preprepare-prepare-commit* protocol. If we pipeline PBFT, then one transaction is *prepared* while previous transactions are *committed* and *executed*. To ensure safety, the ordering is delayed until execution [40, 97]. In similar ways, the phases of other protocols can be pipelined. As stated earlier, ResilientDB also divides tasks across threads. We allow users to *choose the number of threads* needed to create batches, to sign messages, to fetch data from the network and to place output on the network. Further, we allow users to select the *type of storage* for their blockchain ledger and client records. At present, we support the in-memory databases SQLite and LevelDB for storing the ledger and client records. Finally, users can decide the type of *cryptographic constructs*, signatures and digests, they want to employ. Note that a user need not specify all parameters. In such a case, the system will proceed with the *default parameters*.

Finally, the user can deploy the experiment via the ***Run*** button, which initiates the script that will compile, deploy, and run the experiment. The user is presented with a *web-page* that tracks the experiment progress. The user also has an option of *monitoring* the results in real-time. Once the experiment completes, the user can query the InfluxDB database holding all results.

Deploying Smart-Contract The key takeaway of the *Deploying Smart-Contract* demonstration is to show how users can design their own applications in ResilientDB. We believe that demonstrating users the ease with which they can use ResilientDB to create new applications illustrates ResilientDB's general applicability.

Say we want to design a banking application. The *transfer* transaction is a key utility, as it allows movement of money from one account to another. To support transfers, we create a smart contract that allows a user *Bob* to transfer an amount *X* from his account to the account of *Alice* (see Figure 47). Prior to transferring *X*, the smart contract also needs to check if *Bob* (`source`) has at least amount *X* (`source_bal`) in his account. The smart contract needs access to the database with client records, for which we use `GET` and `PUT` APIs.

```
/* return: 1 for commit, 0 for abort */

int TransferMoney::execute()
{
  int source_bal = db->Get(this->source);
  int dest_bal = db->Get(this->dest);
  if (this->amount <= source) {
    db->Set(this->source, source_bal - amount);
    db->Set(this->dest, dest_bal + amount);
    return 1;
  }
  return 0;
}
```

Figure 47: Transfer Smart Contract in ResilientDB.

We provide a *base class* (`SmartContract`) that developers can inherit to define their functionalities. Further, the client needs to provide the required parameters for the new smart contract. For example, the client specifies the source, destination and the amount. Note that these changes do not affect the process of compiling, deploying, and running the code, which can still be done through our GUI. Hence, with simple changes, users can build their own applications using our ResilientDB fabric.

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 simple PBFT protocol to ensure 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 (and latency) 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.

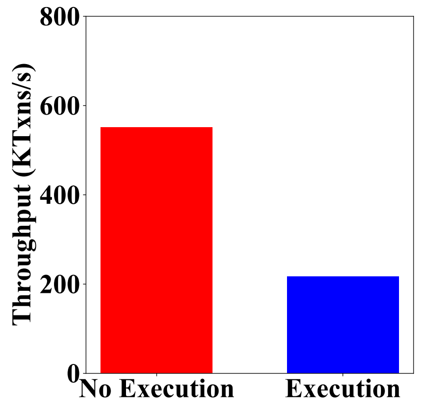
We use YCSB [65, 53] for generating workload for client requests. For creating a request, each client indexes a YCSB table with an active set of 600K records. In our evaluation, we require client requests to contain only write accesses, as a majority of blockchain requests are updates to the existing 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 [120]. At each replica, we permit one worker-thread, one execute-thread and two batch-threads

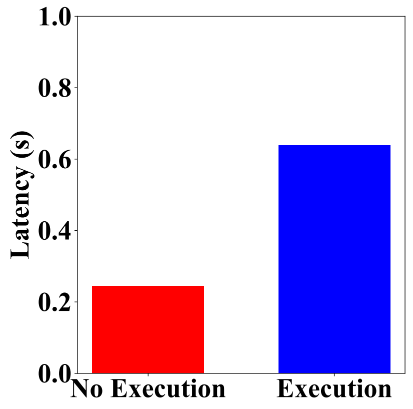
5.3.2 Effect of Threading and Pipelining

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.

We first try to gauge the upper bound performance of our system. In Figures 48a and 48b, we measure the maximum throughput and latency a system can achieve, when there is no communication among the replicas or any consensus protocol. We use the term *No Execution* to refer to the case where all the clients send their requests 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. In both of these experiments, we allowed two threads to work independently at the primary replica, that is, no ordering is maintained. Clearly, the system can attain high throughputs (up to $500K$ txns/s) and has low latency (up to 0.25s).



(a) System throughput.

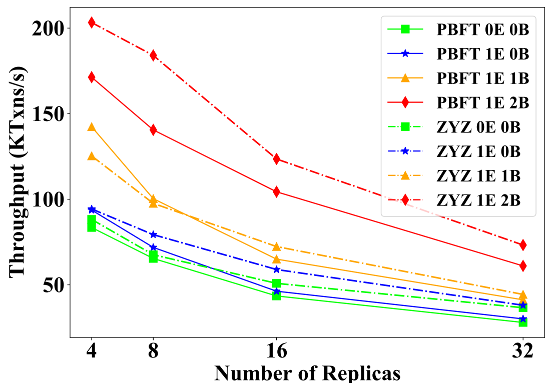


(b) Latency.

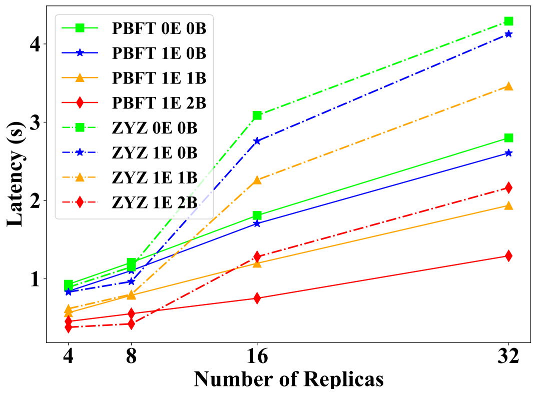
Figure 48: Upper bound measurements: (i) Primary responds back to the client without Execution, and (ii) executes and then reply.

Next, 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 43. In Figures 49a and 49b, 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 49a, we can trivially highlight the benefits of a good implementation. Further, these plots help to confirm our intuition that a multi-threaded pipelined 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.



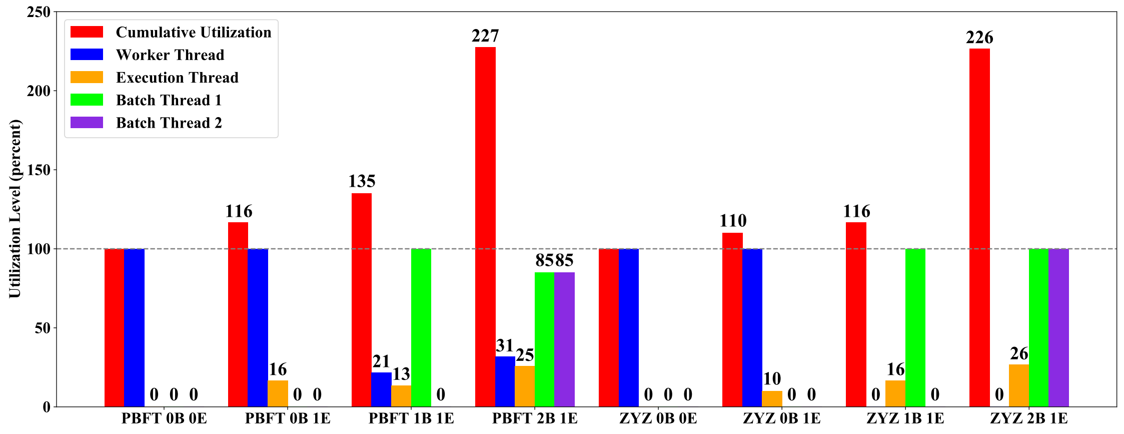
(a) System throughput.



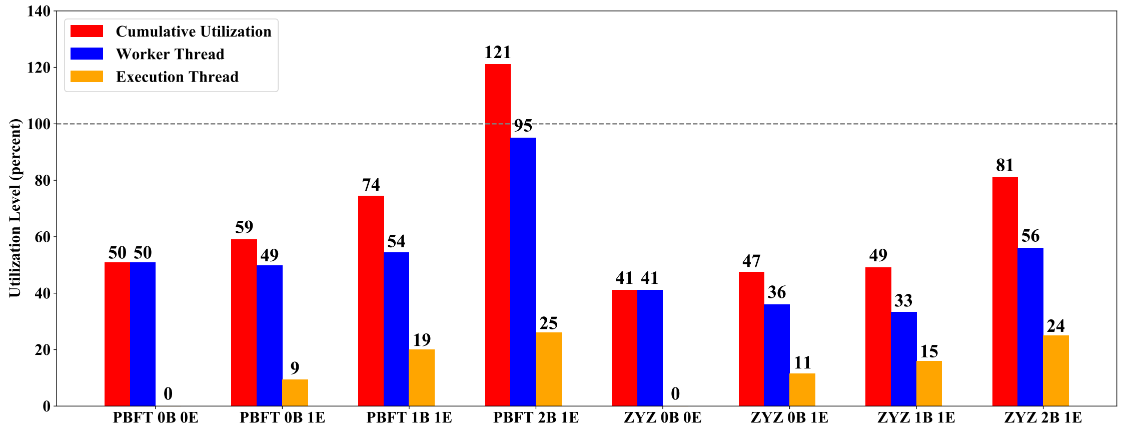
(b) Latency.

Figure 49: 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.

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 saturated. 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 saturated. 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 saturating. Figures 50a and 50b show the saturation level for different threads at a replica. In this figure, we mark 100% as the maximum saturation for any thread. Using the bar for *cumulative saturation*, we show a summation of the saturations 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.



(a) Primary Replica.



(b) Backup Replica.

Figure 50: Saturation level of different threads at a replica. The mean is at 100%, which implies the thread is completely saturated.

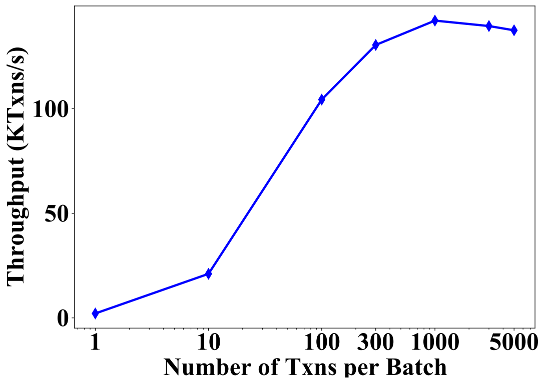
It can be observed that if PBFT is given benefit of ResilientDB's standard pipeline (1E 2B), then it can attain higher throughput than all but one ZYZZYVA implementations. The only ZYZZYVA implementation (1E 2B) that outperforms PBFT is the one that employs ResilientDB's standard threaded-pipeline. Further,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

We have always stated that 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.

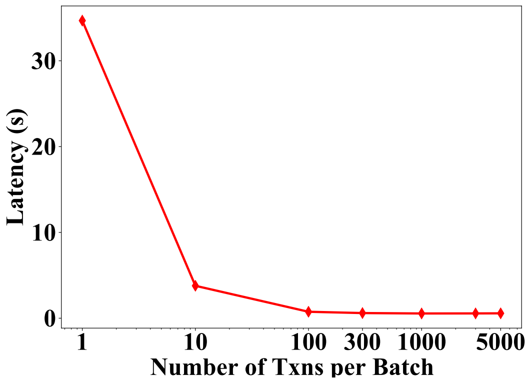
5.3.3 Effect of Transaction Batching

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.

Using Figures 51a and 51b, 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%.



(a) System throughput.



(b) Latency.

Figure 51: System throughput and latency on varying number of transactions per batch. Here, 16 replicas participate in consensus.

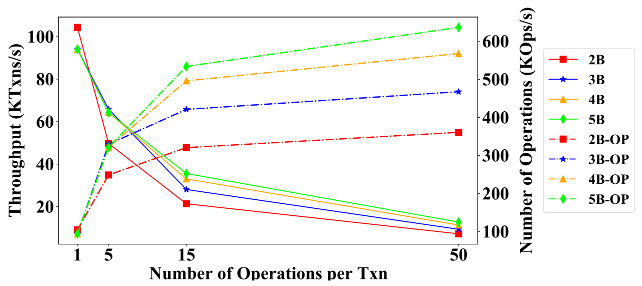
5.3.4 Effect of Multi-Operation Transactions

We now answer question Q5, that is, understand how multi-operation transactions affect the throughput of a system? In Figures 52a and 52b, 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.

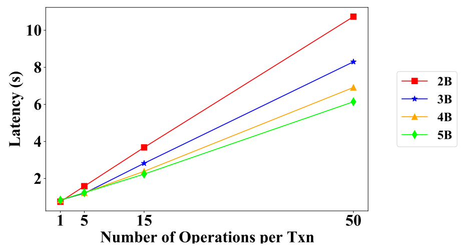
It is evident from these figures that on increasing the number of operations per transaction, the system throughput decreases. This decrease is a consequence of batch-threads getting saturated as they perform task of batching and allocating resources for transaction. Hence, we ran several experiments 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.

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%.



(a) System throughput.



(b) Latency.

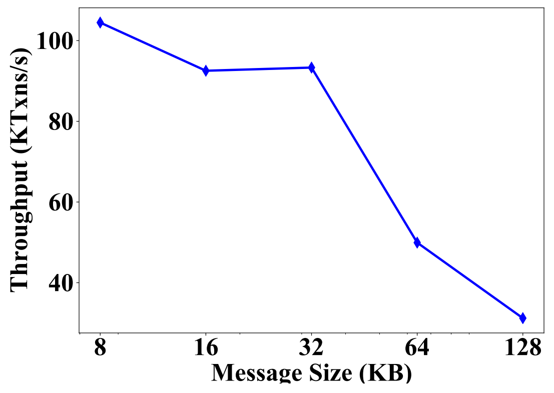
Figure 52: System throughput and latency on varying the number of operations per transaction; B denotes the number of batch-threads.

5.3.5 Effect of Message Size

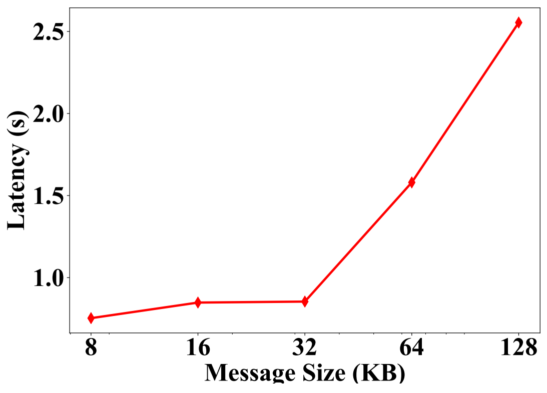
We now try to answer question Q6 by increasing the size of the PRE-PREPARE message in each consensus. The key intuition behind this experiment is to gauge how well a PBC system performs when the requests sent by a client are large. Although each batch includes only 100 client transactions, individually, these requests can be large. Hence, these experiments are aimed at exploiting a different system parameter than the plots of Figure 51.

In Figures 53a and 53b, 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.



(a) System throughput.



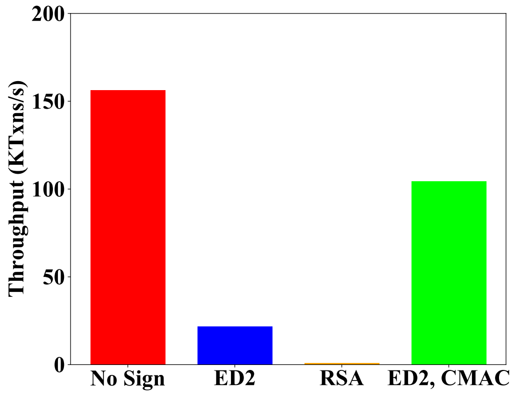
(b) Latency.

Figure 53: System throughput and latency on varying the message size. Here, 16 replicas participate in consensus.

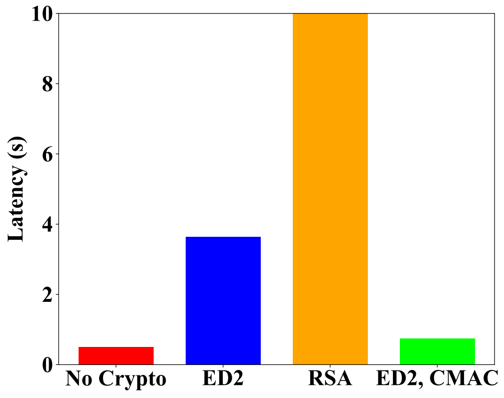
5.3.6 Effect of Cryptographic Signatures

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.

Figures 54a and 54b help 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×.



(a) System throughput.



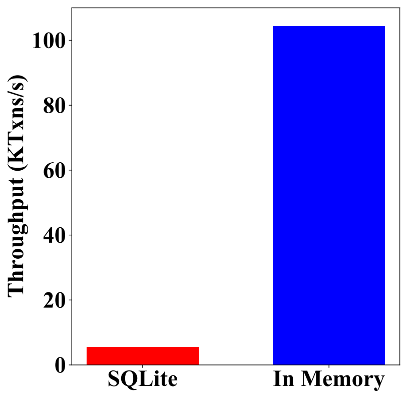
(b) Latency.

Figure 54: System throughput and latency with different signature schemes. Here, 16 replicas participate in consensus.

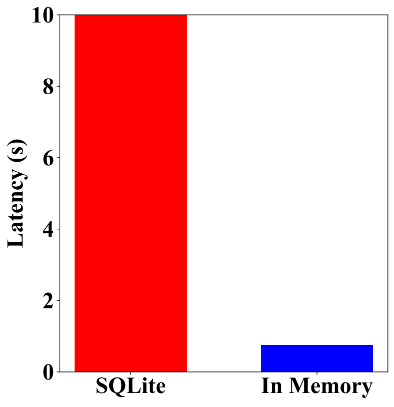
5.3.7 Effect of Memory Storage

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

In Figures 55a and 55b, 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×.



(a) System throughput.



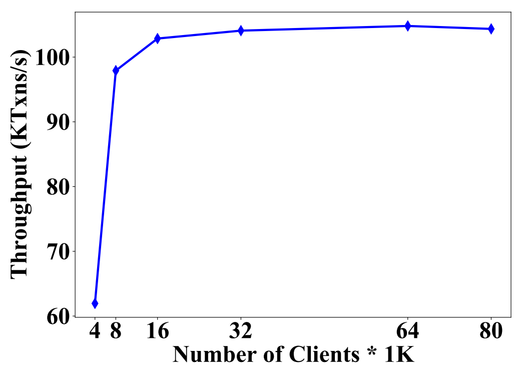
(b) Latency.

Figure 55: System throughput and latency for in-memory storage vs. off-memory storage. Here, 16 replicas used for consensus.

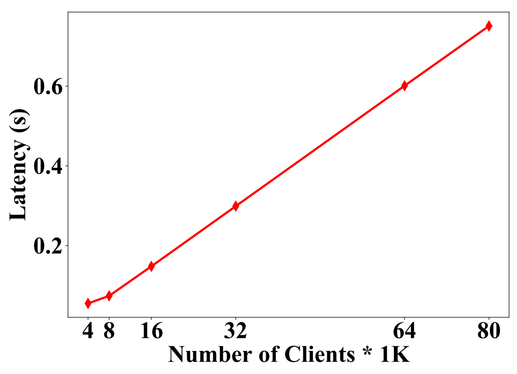
5.3.8 Effect of Clients

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 56a, 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 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 56b). 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.



(a) System throughput.

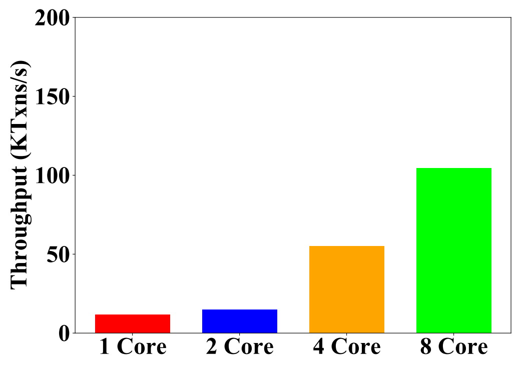


(b) Latency.

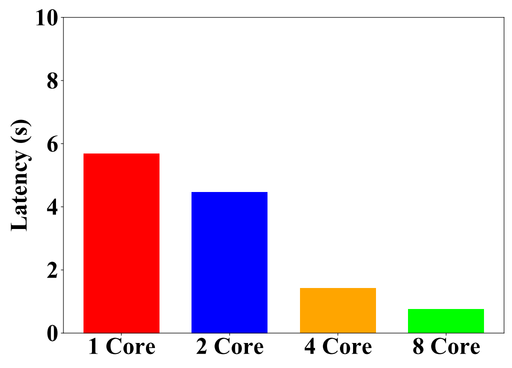
Figure 56: System throughput and latency on varying the number of clients. Here, 16 replicas participate in consensus.

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 Figures 57a and 57b 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 43) 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.



(a) System throughput.



(b) Latency.

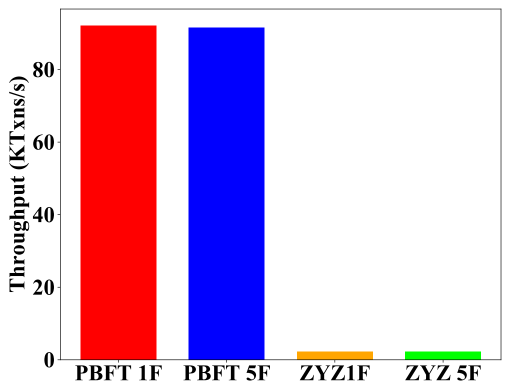
Figure 57: System throughput and latency on varying the number of hardware cores. Here, 16 replicas participate in consensus.

5.3.10 Effect of Replica Failures

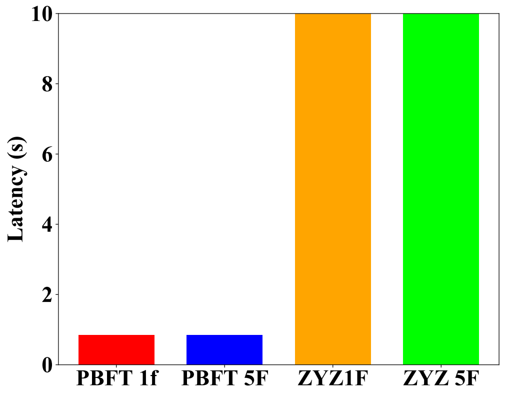
We now try to answer question Q11 by analyzing whether a fast BFT consensus protocol can withstand replica failures. This experiment also illustrates the impact of failures on a PBC. In specific, we perform a head-on comparison of ZYZZYVA against PBFT, while allowing some backup replicas to fail.

In Figures 58a and 58b, 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. On the other hand, ZYZZYVA observes 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. Hence, we approximate this by requiring clients to wait for only a small time.



(a) System throughput.



(b) Latency.

Figure 58: System throughput and latency on failing non-primary replicas. Here, 16 replicas participate in consensus.

6 Future Works

Until now, we have explored the scalability aspect of Fault-Tolerant protocols in a Byzantine environment. GeoBFT scales BFT protocols in the WAN setting, and RingBFT improves performance further, using the sharding technique. RingBFT scales up to 500 replicas that are located in different geographic locations. ServerlessBFT provides Byzantine fault-tolerant (BFT) transactional flow between edge devices and serverless functions and allows blockchain applications to push their compute-intensive workloads to cloud. In our future work, we will focus on Hybrid-Blockchains and blockchain with trusted components.

6.1 Hybrid Blockchains

All Pbft-based protocols are closed membership, meaning that the members of the system are known from the creation of the genesis block. This is one of the limitations of most permissioned blockchain systems. In a permissionless environment like Bitcoin, nodes can join and leave anytime during the consensus. In such a setting, there should be a mechanism to prevent Sybil Attacks, and that comes with a cost which is expensive Nakatomo Consensus (PoW). Sybil attack is defined as an attack where an attacker generates many identities to outvote honest replicas. Using PoW to overcome Sybil attacks changes the voting method from one vote per identity to one vote per hash-power. All PoW blockchains are still vulnerable to 51% and selfish mining attacks, where more than half of the hash-power in the system is controlled by an adversary. Since PoW is computation-intensive and requires a notable amount of resources, Bitcoin suffers from poor performance. It takes 10 minutes for a transaction to be included in a block, and to get confirmed, it needs to be deep in the chain, which makes it even slower.

The poor performance of permissionless blockchains and the closed-membership requirements of permissioned blockchains led us to explore blockchain systems with the following properties:

- **Open membership**: A system where nodes' identities are not known as a priori. Nodes can join and leave the system at some specific times.
- **Responsiveness**: We call a system responsive if it can commit its transactions at the speed of network delay or a factor of it.

To gain responsiveness in a system with open membership, we consolidate both permissionless and blockchains to introduce **Hybrid-Blockchains**. Previous works have proposed such systems. Pas and Shi [154] proposed a hybrid consensus protocol that using PoW to agree on committee members, and this subset acts as a permissioned blockchain to process transactions responsively. Periodically the committee members change based on the PoW puzzle.

Ittai Abraham et al. [1] introduced a reconfigurable Byzantine consensus called Solida. They use a modified version of Pbft among its committee to reach consensus over transactions and use a PoW puzzle to elect new committee members. Upon receiving a solution for the PoW puzzle, The nodes inside the committee try to reach a consensus to hand over the ledger to the new committee.

6.2 Byzantine Fault Tolerant Consensus with Trusted Components

Bft protocols tolerate a subset of participants behaving arbitrarily: a malicious actor can delay, reorder or drop messages (*omission faults*); it can also send conflicting information to participants (*equivocation*). As a result, BFT consensus protocols are costly: maintaining correctness in the presence of such attacks requires a minimum of $3\mathbf{f}+1$ participants, where $\mathbf{f}$ participants can be malicious. This is in contrast to crash-fault tolerant protocols (CFT), where participants can fail by crashing, which require $2\mathbf{f}+1$ participants only for correctness [124, 147].

To minimise this additional cost, some existing BFT protocols leverage trusted components to curb the degree to which participants can behave arbitrarily [55, 56, 125, 172]. A trusted component provably performs a specific computation: incrementing a counter [126, 185], appending to a log [45], or more advanced options like executing a complex algorithm [127, 117, 167]. While there exists a large number of BFT protocols that leverage these trusted components [45, 126, 196] (we refer to these protocol as TRUST-BFT protocols for simplicity), they all proceed in a similar fashion: they *force* each replica to commit to a single order for each request by having the trusted component sign each message it sends. In turn, each trusted component either: (1) records the chosen order for a client request in an append-only log, or (2) binds this order for the request with the value of a monotonically increasing counter. Committing to an order in this way allows these protocols to remain safe with $2\mathbf{f}+1$ replicas only, bringing them in line with their CFT counterparts.

While reducing replication cost is a significant benefit, we argues that current TRUST-BFT protocols place *too much trust in trusted components*. Our analysis uncovers three fundamental issues with existing TRUST-BFT implementations that preclude most practical deployments: (i) limited responsiveness for clients, (ii) safety concerns associated with trusted components, and (iii) inability to perform multiple consensuses in parallel.

Responsiveness We observe that byzantine replicas can successfully prevent a client from receiving a response for its transactions. While the transaction will still commit (consensus liveness), the system will appear to clients as stalled and thus appear non-responsive to clients.

TRUST-BFT protocols allow a reduced quorum size of $\mathbf{f}+1$ to commit a request. As $\mathbf{f}$ of those may be Byzantine, only one honest replica is guaranteed to execute the operation. This is insufficient to guarantee that a client will receive the necessary $\mathbf{f}+1$ matching responses post operation execution and thus validate that the response is indeed valid.

Loss of Safety under Rollback. Existing TRUST-BFT protocols consider an idealised model of trusted computations. They assume that the trusted components cannot be compromised and that their data remains persistent in the presence of a malicious host. This assumption does not yet align with current hardware functionality. A large number of these protocols employ Intel SGX enclaves for trusted computing [59, 22, 171]. Unfortunately, SGX-based designs have been shown to suffer from rollback attacks [189, 135, 111], and the solutions to mitigate these attacks lack practical deployments [77]. Hardware enclaves that do provably defend against rollback attacks, such as persistent counters [55] and TPMs [89], have prohibitively high latencies (tens of milliseconds) [133, 126, 153].

Sequential Consensus. Existing TRUST-BFT protocols are inherently sequential as they require each outgoing message to be ordered and attested by trusted components. While recent work mitigates this issue by pipelining consensus phases [59] or running multiple independent consensus invocations [22], their performance remains fundamentally limited by the RTT of each protocol phase. In fact, despite their lower replication factor, we observe that TRUST-BFT protocols achieve lower throughput than traditional parallel BFT protocols [41, 97, 181, 143].

We think that TRUST-BFT protocols have targeted the wrong metric: while reducing the replication factor to $2\mathbf{f}+1$ may seem appealing from a resource efficiency or management overhead standpoint, it, paradoxically, comes at a significant performance cost. Nonetheless, trusted components can still bring huge benefits to BFT consensus *when they use* $3\mathbf{f}+1$ *replicas*. In our future work we plan to propose a novel suite of consensus algorithms (**Flexi**ble **Trust**ed BFT (FLEXITRUST)), which address the aforementioned limitations. These protocols are always responsive and achieve high throughput as they (1) make minimal use of trusted components (once per client operation and at the primary replica only), and (2) support parallel consensus invocations. Both these properties are made possible by the ability to use large quorums (of size $2\mathbf{f}+1$) when using $3\mathbf{f}+1$ replicas. Our techniques can be used to convert any TRUST-BFT protocol into a FLEXITRUST protocol. We provide as examples two such conversions: FLEXI-BFT and FLEXI-ZZ, two protocols based on PBFT [41]/ MINBFT [185] and ZYZZYVA [121]/ MINZZ [185], respectively. FLEXI-BFT follows a similar structure to PBFT, but requires one less communication phase. FLEXI-ZZ is, we believe, of independent interest: the protocol achieves consensus in a single linear phase without using expensive cryptographic constructs such as threshold signatures. Crucially, unlike the ZYZZYVA and MINZZ protocols, FLEXI-ZZ continues to commit in a single-round even when a single participant misbehaves, thus maintaining high-throughput [97, 47, 49]. Further, FLEXI-ZZ does not face the safety bug like ZYZZYVA [2].

References

[1] I Abraham, D Malkhi, K Nayak, L Ren, and A Spiegelman. A blockchain protocol based on reconfigurable byzantine consensus. *OPODIS, Solida*, 2017.

[2] Ittai Abraham, Guy Gueta, Dahlia Malkhi, Lorenzo Alvisi, Rama Kotla, and Jean-Philippe Martin. Revisiting fast practical byzantine fault tolerance, 2017. URL: https://arxiv.org/abs/1712.01367.

[3] Ittai Abraham, Guy Gueta, Dahlia Malkhi, Lorenzo Alvisi, Ramakrishna Kotla, and Jean-Philippe Martin. Revisiting fast practical byzantine fault tolerance, 2017. URL: https://arxiv.org/abs/1712.01367.

[4] Shreya Agrawal and Khuzaima Daudjee. A Performance Comparison of Algorithms for Byzantine Agreement in Distributed Systems. In *12th European Dependable Computing Conference, EDCC*, pages 249–260. IEEE Computer Society, 2016. doi:10.1109/EDCC.2016.17.

[5] Amazon Staff. Amazon Prime Air prepares for drone deliveries, 2022. URL: https://www.aboutamazon.com/news/transportation/amazon-prime-air-prepares-for-drone-deliveries.

[6] Yair Amir, Claudiu Danilov, Danny Dolev, Jonathan Kirsch, John Lane, Cristina Nita-Rotaru, Josh Olsen, and David Zage. Steward: Scaling byzantine fault-tolerant replication to wide area networks. *IEEE Transactions on Dependable and Secure Computing*, 7(1):80–93, 2010. doi:10.1109/TDSC.2008.53.

[7] Mohammad Javad Amiri, Divyakant Agrawal, and Amr El Abbadi. CAPER: A cross-application permissioned blockchain. *Proc. VLDB Endow.*, 12(11):1385–1398, 2019. doi:10.14778/3342263.3342275.

[8] Mohammad Javad Amiri, Divyakant Agrawal, and Amr El Abbadi. SharPer: Sharding permissioned blockchains over network clusters, 2019. URL: https://arxiv.org/abs/1910.00765v1.

[9] Mohammad Javad Amiri, Sujaya Maiyya, Divyakant Agrawal, and Amr El Abbadi. Seemore: A fault-tolerant protocol for hybrid cloud environments. In *36th IEEE International Conference on Data Engineering, ICDE 2020, Dallas, TX, USA, April 20-24, 2020*, pages 1345–1356. IEEE, 2020. doi:10.1109/ICDE48307.2020.00120.

[10] Elli Androulaki et al. Hyperledger fabric: A distributed operating system for permissioned blockchains. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, 2018.

[11] L. Aniello, R. Baldoni, E. Gaetani, F. Lombardi, A. Margheri, and V. Sassone. A Prototype Evaluation of a Tamper-Resistant High Performance Blockchain-Based Transaction Log for a Distributed Database. In *2017 13th European Dependable Computing Conference*, pages 151–154, 2017. doi:10.1109/EDCC.2017.31.

[12] Mohammad S. Aslanpour, Adel N. Toosi, Claudio Cicconetti, Bahman Javadi, Peter Sbarski, Davide Taibi, Marcos Assuncao, Sukhpal Singh Gill, Raj Gaire, and Schahram Dustdar. Serverless edge computing: Vision and challenges. In *2021 Australasian Computer Science Week Multiconference*,

ACSW '21, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3437378.3444367.

[13] Mohammad Sadegh Aslanpour, Adel Nadjaran Toosi, Raj Gaire, and Muhammad Aamir Cheema. Wattedge: A holistic approach for empirical energy measurements in edge computing. In *Service-Oriented Computing - 19th International Conference, ICSOC*, volume 13121 of *Lecture Notes in Computer Science*, pages 531–547. Springer, 2021. doi:10.1007/978-3-030-91431-8_33.

[14] Diem Association. Welcome to diem: Diem documentation. URL: https://developers.libra.org/docs/welcome-to-libra.

[15] GSM 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.

[16] Giuseppe Ateniese, Ilario Bonacina, Antonio Faonio, and Nicola Galesi. Proofs of space: When space is of the essence. In *Security and Cryptography for Networks*, pages 538–557. Springer, 2014. doi:10.1007/978-3-319-10879-7_31.

[17] Pierre-Louis Aublin, Rachid Guerraoui, Nikola Knežević, Vivien Quéma, and Marko Vukolić. The next 700 BFT protocols. *ACM Transactions on Computer Systems*, 32(4):12:1–12:45, 2015. doi:10.1145/2658994.

[18] Furqan Baig and Fusheng Wang. Blockchain enabled distributed data management - A vision. In *35th IEEE International Conference on Data Engineering Workshops, ICDE Workshops*, pages 28–30. IEEE, 2019. doi:10.1109/ICDEW.2019.00-39.

[19] Shaimaa Bajoudah, Changyu Dong, and Paolo Missier. Toward a decentralized, trust-less marketplace for brokered iot data trading using blockchain. In *IEEE International Conference on Blockchain, Blockchain 2019, Atlanta, GA, USA, July 14-17, 2019*, pages 339–346. IEEE, 2019. doi:10.1109/Blockchain.2019.00053.

[20] Ioana Baldini, Paul Castro, Kerry Chang, Perry Cheng, Stephen Fink, Vatche Ishakian, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Aleksander Slominski, and Philippe Suter. *Serverless Computing: Current Trends and Open Problems*, pages 1–20. Springer Singapore, Singapore, 2017. doi:10.1007/978-981-10-5026-8_1.

[21] Luciano Baresi, Danilo Filgueira Mendonça, and Martín Garriga. Empowering low-latency applications through a serverless edge computing architecture. In *European Conference on Service-Oriented and Cloud Computing*, 2017.

[22] Johannes Behl, Tobias Distler, and Rüdiger Kapitza. Hybrids on steroids: SGX-based high performance BFT. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 222–237. ACM, 2017. doi:10.1145/3064176.3064213.

[23] Iddo Bentov, Ariel Gabizon, and Alex Mizrahi. Cryptocurrencies without proof of work. In *Financial Cryptography and Data Security*, pages 142–157. Springer, 2016. doi:10.1007/978-3-662-53357-4_10.

[24] Iddo Bentov, Rafael Pass, and Elaine Shi. Snow white: Robustly reconfigurable consensus and applications to provably secure proof of stake, 2016. URL: https://eprint.iacr.org/2016/919.

[25] P. A. Bernstein and N. Goodman. Concurrency Control in Distributed Database Systems. *ACM Comput. Surv.*, 13(2):185–221, 1981.

[26] P. A. Bernstein and N. Goodman. Multiversion Concurrency Control - Theory and Algorithms. *ACM TODS*, 8(4):465–483, 1983.

[27] A. Bessani, J. Sousa, and E. E. P. Alchieri. State machine replication for the masses with bft-smart. In *DSN*, 2014.

[28] Romil Bhardwaj, Zhengxu Xia, Ganesh Ananthanarayanan, Junchen Jiang, Yuanchao Shu, Nikolaos Karianakis, Kevin Hsieh, Paramvir Bahl, and Ion Stoica. Ekya: Continuous learning of video analytics models on edge compute servers. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 119–135, Renton, WA, April 2022. USENIX Association.

[29] Kenneth Birman, André Schiper, and Pat Stephenson. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems*, 9(3):272–314, 1991. doi:10.1145/128738.128742.

[30] Burkhard 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.

[31] Manuel Bravo, Zsolt István, and Man-Kit Sit. Towards improving the performance of BFT consensus for future permissioned blockchains. *CoRR*, abs/2007.12637, 2020. arXiv:2007.12637.

[32] Eric Brewer. CAP twelve years later: How the "rules" have changed. *Computer*, 45(2):23–29, 2012. doi:10.1109/MC.2012.37.

[33] Eric A. Brewer. Towards robust distributed systems (abstract). In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 7–7. ACM, 2000. doi:10.1145/343477.343502.

[34] Ethan Buchman, Jae Kwon, and Zarko Milosevic. Revisiting fast practical byzantine fault tolerance, 2018. URL: https://arxiv.org/abs/1807.04938.

[35] Matthias Butenuth, Guido v. Gösseln, Michael Tiedge, Christian Heipke, Udo Lipeck, and Monika Sester. Integration of heterogeneous geospatial data in a federated database. *ISPRS Journal of Photogrammetry and Remote Sensing*, 62(5):328 – 346, 2007. Theme Issue: Distributed Geoinformatics. doi:https://doi.org/10.1016/j.isprsjprs.2007.04.003.

[36] Rajkumar Buyya et al. A manifesto for future generation cloud computing: Research directions for the next decade. *ACM Comput. Surv.*, 2018.

[37] Wei Cao, Yingqiang Zhang, Xinjun Yang, Feifei Li, Sheng Wang, Qingda Hu, Xuntao Cheng, Zongzhi Chen, Zhenjun Liu, Jing Fang, Bo Wang, Yuhui Wang, Haiqing Sun, Ze Yang, Zhushi Cheng, Sen Chen, Jian Wu, Wei Hu, Jianwei Zhao, Yusong Gao, Songlu Cai, Yunyang Zhang, and Jiawang Tong. Polardb serverless: A cloud native database for disaggregated data centers. In *SIGMOD '21: International Conference on Management of Data*, pages 2477–2489. ACM, 2021. doi:10.1145/3448016.3457560.

[38] S. Capkun, L. Buttyan, and J.-P. Hubaux. Self-organized public-key management for mobile ad hoc networks. *IEEE Transactions on Mobile Computing*, 2(1):52–64, 2003. doi:10.1109/TMC.2003.1195151.

[39] Michael Casey, Jonah Crane, Gary Gensler, Simon Johnson, and Neha 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.

[40] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, pages 173–186, USA, 1999. USENIX.

[41] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems*, 20(4):398–461, 2002. doi:10.1145/571637.571640.

[42] Sarah Chasins, Alvin Cheung, Natacha Crooks, Ali Ghodsi, Ken Goldberg, Joseph E. Gonzalez, Joseph M. Hellerstein, Michael I. Jordan, Anthony D. Joseph, Michael W. Mahoney, Aditya Parameswaran, David Patterson, Raluca Ada Popa, Koushik Sen, Scott Shenker, Dawn Song, and Ion Stoica. The Sky Above The Clouds, 2022. doi:10.48550/ARXIV.2205.07147.

[43] Xusheng Chen, Haoze Song, Jianyu Jiang, Chaoyi Ruan, Cheng Li, Sen Wang, Gong Zhang, Reynold Cheng, and Heming Cui. Achieving low tail-latency and high scalability for serializable transactions in edge computing. In *EuroSys '21: Sixteenth European Conference on Computer Systems*, pages 210–227. ACM, 2021. doi:10.1145/3447786.3456238.

[44] 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.

[45] Byung-Gon Chun, Petros Maniatis, Scott Shenker, and John 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.

[46] Cisco. Cisco Annual Internet Report (2018–2023), 2020. URL: https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.html.

[47] Allen Clement, Manos Kapritsos, Sangmin Lee, Yang Wang, Lorenzo Alvisi, Mike Dahlin, and Taylor 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.

[48] Allen Clement, Edmund Wong, Lorenzo Alvisi, Mike Dahlin, and Mirco 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.

[49] Allen Clement, Edmund L. Wong, Lorenzo Alvisi, Michael Dahlin, and Mirco 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.

[50] Cloudflare. A byzantine failure in the real world, 2020. URL: https://blog.cloudflare.com/a-byzantine-failure-in-the-real-world/.

[51] The Crypto++ community. Crypto++ library 8.1, 2019. URL: https://www.cryptopp.com/.

[52] Cindy Compert, Maurizio Luinetti, and Bertrand Portier. Blockchain and GDPR: How blockchain could address five areas associated with gdpr compliance. Technical report, IBM Security, 2018. URL: https://public.dhe.ibm.com/common/ssi/ecm/61/en/61014461usen/security-ibm-security-solutions-wg-white-paper-external-61014461usen-20180319.pdf.

[53] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell 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.

[54] J. C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale 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.

[55] Victor Costan and Srinivas Devadas. Intel sgx explained. Cryptology ePrint Archive, Report 2016/086, 2016. https://ia.cr/2016/086.

[56] Victor Costan, Ilia Lebedev, and Srinivas Devadas. Sanctum: Minimal Hardware Extensions for Strong Software Isolation. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 857–874, Austin, TX, August 2016. USENIX Association.

[57] Garrett D'Amore. NNG Reference Manual, 2018.

[58] Hung Dang, Tien Tuan Anh Dinh, Dumitrel Loghin, Ee-Chien Chang, Qian Lin, and Beng Chin 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.

[59] Jérémie Decouchant, David Kozhaya, Vincent Rahli, and Jiangshan Yu. Damysus: Streamlined bft consensus leveraging trusted components. In *Proceedings of the Seventeenth European Conference on Computer Systems*, page 1–16, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3492321.3519568.

[60] Richard A. DeMillo, Nancy A. Lynch, and Michael J. Merritt. Cryptographic protocols. In *Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing*, pages 383–400. ACM, 1982. doi:10.1145/800070.802214.

[61] A. Deshpande and J. M. Hellerstein. Decoupled query optimization for federated database systems. In *Proceedings 18th International Conference on Data Engineering*, pages 716–727, 2002. doi:10.1109/ICDE.2002.994788.

[62] SQLite Developers. Sqlite home page, 2019.

[63] 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. doi:10.1145/2463676.2463710.

[64] Tien Tuan Anh Dinh, Rui Liu, Meihui Zhang, Gang Chen, Beng Chin Ooi, and Ji Wang. Untangling blockchain: A data processing view of blockchain systems. *IEEE Trans. Knowl. Data Eng.*, 30(7):1366–1385, 2018. doi:10.1109/TKDE.2017.2781227.

[65] Tien Tuan Anh Dinh, Ji Wang, Gang Chen, Rui Liu, Beng Chin Ooi, and Kian-Lee 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.

[66] D. Dolev. Unanimity in an unknown and unreliable environment. In *22nd Annual Symposium on Foundations of Computer Science*, pages 159–168. IEEE, 1981. doi:10.1109/SFCS.1981.53.

[67] D. Dolev and H. Strong. Authenticated algorithms for byzantine agreement. *SIAM Journal on Computing*, 12(4):656–666, 1983. doi:10.1137/0212045.

[68] Danny Dolev. The byzantine generals strike again. *Journal of Algorithms*, 3(1):14–30, 1982. doi:10.1016/0196-6774(82)90004-9.

[69] Alan AA Donovan and Brian W Kernighan. *The Go programming language*. Addison-Wesley Professional, 2015.

[70] John R. Douceur. The sybil attack. In *Peer-to-Peer Systems*, pages 251–260. Springer Berlin Heidelberg, 2002. doi:10.1007/3-540-45748-8_24.

[71] John R. Douceur. The Sybil Attack. In *First International Workshop on Peer-to-Peer Systems*, IPTPS '01, 2002.

[72] Cynthia Dwork and Moni Naor. Pricing via processing or combatting junk mail. In *Advances in Cryptology — CRYPTO' 92*, pages 139–147. Springer, 1992. doi:10.1007/3-540-48071-4_10.

[73] Stefan Dziembowski, Sebastian Faust, Vladimir Kolmogorov, and Krzysztof Pietrzak. Proofs of space. In *Advances in Cryptology – CRYPTO 2015*, pages 585–605. Springer, 2015. doi:10.1007/978-3-662-48000-7_29.

[74] Naomi Eide. On-premise servers persist in 98% of businesses, even with cloud hype, 2019. URL: https://www.ciodive.com/news/on-premise-servers-persist-in-98-of-businesses-even-with-cloud-hype/549643/.

[75] Muhammad El-Hindi, Carsten Binnig, Arvind Arasu, Donald Kossmann, and Ravi Ramamurthy. BlockchainDB: A shared database on blockchains. *PVLDB*, 12(11):1597–1609, 2019. doi:10.14778/3342263.3342636.

[76] Franz Färber, Sang Kyun Cha, Jürgen Primsch, Christof Bornhövd, Stefan Sigg, and Wolfgang Lehner. Sap hana database: Data management for modern business applications. *SIGMOD Rec.*, 40(4):45–51, January 2012. doi:10.1145/2094114.2094126.

[77] Shufan Fei, Zheng Yan, Wenxiu Ding, and Haomeng Xie. Security vulnerabilities of sgx and countermeasures: A survey. *ACM Comput. Surv.*, 54(6), jul 2021. doi:10.1145/3456631.

[78] Michael J. Fischer and Nancy A. Lynch. A lower bound for the time to assure interactive consistency. *Information Processing Letters*, 14(4):183–186, 1982. doi:10.1016/0020-0190(82)90033-3.

[79] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, 1985. doi:10.1145/3149.214121.

[80] Michelle Froese. How the industrial internet of things is benefiting wind operators, 2019. URL: https://www.windpowerengineering.com/how-the-industrial-internet-of-things-is-benefiting-wind-operators/.

[81] Jérôme Gallard and Adrien Lebre et al. Architecture for the next generation system management tools. *Future Generation Comp. Syst.*, 2012.

[82] Vijay K. Garg, John Bridgman, and Bharath Balasubramanian. Accurate byzantine agreement with feedback. In *OPODIS*, 2011.

[83] Seth Gilbert and Nancy 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.

[84] Guy Golan Gueta, Ittai Abraham, Shelly Grossman, Dahlia Malkhi, Benny Pinkas, Michael Reiter, Dragos-Adrian Seredinschi, Orr Tamir, and Alin Tomescu. SBFT: A scalable and decentralized trust infrastructure. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 568–580. IEEE, 2019. doi:10.1109/DSN.2019.00063.

[85] William J. Gordon and Christian Catalini. Blockchain technology for healthcare: Facilitating the transition to patient-driven interoperability. *Computational and Structural Biotechnology Journal*, 16:224–230, 2018. URL: https://www.sciencedirect.com/science/article/pii/S200103701830028X, doi:https://doi.org/10.1016/j.csbj.2018.06.003.

[86] Christian Gorenflo, Stephen Lee, Lukasz Golab, and Srinivasan Keshav. Fastfabric: Scaling hyperledger fabric to 20, 000 transactions per second. In *IEEE International Conference on Blockchain and Cryptocurrency, ICBC*, pages 455–463. IEEE, 2019. doi:10.1109/BLOC.2019.8751452.

[87] Jim 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.

[88] Gideon Greenspan. Multichain private blockchain, 2015. URL: https://www.multichain.com/download/MultiChain-White-Paper.pdf.

[89] Trusted Computing Group. Trusted Platform Module Library, 2019. URL: https://trustedcomputinggroup.org/resource/tpm-library-specification/.

[90] Suyash Gupta, Jelle Hellings, Sajjad Rahnama, and Mohammad Sadoghi. An in-depth look of BFT consensus in blockchain: Challenges and opportunities. In *Proceedings of the 20th International Middleware Conference Tutorials*, pages 6–10, 2019. doi:10.1145/3366625.3369437.

[91] Suyash Gupta, Jelle Hellings, Sajjad Rahnama, and Mohammad Sadoghi. Proof-of-execution: Reaching consensus through fault-tolerant speculation, 2019. URL: http://arxiv.org/abs/1911.00838.

[92] Suyash Gupta, Jelle Hellings, Sajjad Rahnama, and Mohammad Sadoghi. Building high throughput permissioned blockchain fabrics: Challenges and opportunities. *Proc. VLDB Endow.*, 13(12):3441–3444, 2020. doi:10.14778/3415478.3415565.

[93] Suyash Gupta, Jelle Hellings, and Mohammad Sadoghi. Brief announcement: Revisiting consensus protocols through wait-free parallelization. In *33rd International Symposium on Distributed Computing (DISC 2019)*, volume 146 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 44:1–44:3. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019. doi:10.4230/LIPIcs.DISC.2019.44.

[94] Suyash Gupta, Jelle Hellings, and Mohammad 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.

[95] Suyash Gupta, Jelle Hellings, and Mohammad Sadoghi. RCC: Resilient Concurrent Consensus for High-Throughput Secure Transaction Processing. In *37th IEEE International Conference on Data Engineering (ICDE)*, 2021. URL: http://arxiv.org/abs/1911.00837, arXiv:1911.00837.

[96] Suyash Gupta, Jelle Hellings, and Mohammad 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.

[97] Suyash Gupta, Sajjad Rahnama, Jelle Hellings, and Mohammad Sadoghi. ResilientDB: Global scale resilient blockchain fabric. *Proc. VLDB Endow.*, 13(6):868–883, 2020. doi:10.14778/3380750.3380757.

[98] Suyash Gupta, Sajjad Rahnama, Erik Linsenmayer, Faisal Nawab, and Mohammad Sadoghi. Reliable transactions in serverless-edge architecture. *CoRR*, abs/2201.00982, 2022. URL: https://arxiv.org/abs/2201.00982, arXiv:2201.00982.

[99] Suyash Gupta, Sajjad Rahnama, and Mohammad Sadoghi. Revisiting fast practical byzantine fault tolerance, 2019. URL: https://arxiv.org/abs/1911.09208.

[100] Suyash Gupta, Sajjad Rahnama, and Mohammad Sadoghi. Permissioned blockchain through the looking glass: Architectural and implementation lessons learned. In *Proceedings of the 40th IEEE International Conference on Distributed Computing Systems*, 2020.

[101] Suyash Gupta and Mohammad Sadoghi. *Blockchain Transaction Processing*, pages 1–11. Springer International Publishing, 2018. doi:10.1007/978-3-319-63962-8_333-1.

[102] Suyash Gupta and Mohammad Sadoghi. *Blockchain Transaction Processing*. Springer International Publishing, 2018.

[103] Suyash Gupta and Mohammad 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.

[104] Suyash Gupta and Mohammad Sadoghi. Efficient and non-blocking agreement protocols. *Distributed and Parallel Databases*, 2019. doi:10.1007/s10619-019-07267-w.

[105] Rachael Harding, Dana Van Aken, Andrew Pavlo, and Michael Stonebraker. An evaluation of distributed concurrency control. *PVLDB*, 10(5):553–564, 2017. doi:10.14778/3055540.3055548.

[106] Jelle Hellings and Mohammad Sadoghi. Brief announcement: The fault-tolerant cluster-sending problem. In *33rd International Symposium on Distributed Computing (DISC 2019)*, volume 146 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 45:1–45:3. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019. doi:10.4230/LIPIcs.DISC.2019.45.

[107] John L. Hennessy and David A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. 5th edition, 2011.

[108] M. P. Herlihy and J. M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM TOPLAS*, 1990.

[109] Maurice Herlihy. Blockchains from a distributed computing perspective. *Communications of the ACM*, 62(2):78–85, 2019. doi:10.1145/3209623.

[110] Peter H. Hochschild, Paul Turner, Jeffrey C. Mogul, Rama Govindaraju, Parthasarathy Ranganathan, David E. Culler, and Amin Vahdat. Cores that don't count. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, page 9–16, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3458336.3465297.

[111] Manuel Huber, Julian Horsch, and Sascha Wessel. Protecting suspended devices from memory attacks. In *Proceedings of the 10th European Workshop on Systems Security*, EuroSec'17, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3065913.3065914.

[112] Zsolt István, Alessandro Sorniotti, and Marko Vukolić. Streamchain: Do blockchains need blocks? SERIAL'18, 2018.

[113] Markus Jakobsson and Ari 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.

[114] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, Joseph E. Gonzalez, Raluca Ada Popa, Ion Stoica, and David A. Patterson. Cloud Programming Simplified: A Berkeley View on Serverless Computing, 2019. arXiv:1902.03383.

[115] Maged N. Kamel Boulos, James T. Wilson, and Kevin 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.

[116] Andreas Kamilaris, Agusti Fonts, and Francesc X. Prenafeta-Boldu. The rise of blockchain technology in agriculture and food supply chains. *Trends in Food Science & Technology*, 91:640–652, 2019. URL: https://www.sciencedirect.com/science/article/pii/S0924224418303686, doi:https://doi.org/10.1016/j.tifs.2019.07.034.

[117] Luyi Kang, Yuqi Xue, Weiwei Jia, Xiaohao Wang, Jongryool Kim, Changhwan Youn, Myeong Joon Kang, Hyung Jin Lim, Bruce L. Jacob, and Jian Huang. Iceclave: A trusted execution environment for in-storage computing. In *MICRO '21: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 199–211. ACM, 2021. doi:10.1145/3466752.3480109.

[118] Manos Kapritsos, Yang Wang, Vivien Quema, Allen Clement, Lorenzo Alvisi, and Mike 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 Association, 2012.

[119] Karthik Kashyap. 92% of Companies Say On-Premises Software Sales Are Growing: Dimensional Research Report, 2021. URL: https://www.spiceworks.com/tech/enterprise-software/news/92-of-companies-say-on-premises-software-sales-are-growing-dimensional-research-rep

[120] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography*. Chapman and Hall/CRC, 2nd edition, 2014.

[121] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: Speculative byzantine fault tolerance. *SIGOPS Oper. Syst. Rev.*, 41(6):45–58, 2007. doi:10.1145/1323293.1294267.

[122] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: Speculative byzantine fault tolerance. *ACM Trans. Comput. Syst.*, 27(4), 2010. doi:10.1145/1658357.1658358.

[123] Nicolas Kourtellis, Herodotos Herodotou, Maciej Grzenda, Piotr Wawrzyniak, and Albert Bifet. S2CE: a hybrid cloud and edge orchestrator for mining exascale distributed streams. In *DEBS '21: The 15th ACM International Conference on Distributed and Event-based Systems*, pages 103–113. ACM, 2021. doi:10.1145/3465480.3466926.

[124] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998. doi:10.1145/279227.279229.

[125] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanović, and Dawn 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.

[126] Dave Levin, John R. Douceur, Jacob R. Lorch, and Thomas Moscibroda. TrInc: Small Trusted Hardware for Large Distributed Systems. In *6th USENIX Symposium on Networked Systems Design and Implementation*. USENIX Association, April 2009.

[127] Wenhao Li, Yubin Xia, and Haibo Chen. Research on ARM trustzone. *GetMobile Mob. Comput. Commun.*, 22(3):17–22, 2018. doi:10.1145/3308755.3308761.

[128] Joshua Lind, Oded Naor, Ittay Eyal, Florian Kelbert, Emin Gün Sirer, and Peter Pietzuch. Teechain: A secure payment network with asynchronous blockchain access. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 63–79. ACM, 2019. doi:10.1145/3341301.3359627.

[129] Marta Lokhava, Giuliano Losa, David Mazières, Graydon Hoare, Nicolas Barry, Eli Gafni, Jonathan Jove, Rafał Malinowsky, and Jed McCaleb. Fast and secure global payments with stellar. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 80–96. ACM, 2019. doi:10.1145/3341301.3359636.

[130] Y. Lu, X. Huang, Y. Dai, S. Maharjan, and Y. Zhang. Blockchain and Federated Learning for Privacy-Preserved Data Sharing in Industrial IoT. *IEEE Transactions on Industrial Informatics*, 16(6):4177–4186, 2020. doi:10.1109/TII.2019.2942190.

[131] Shuaicheng Ma, Tamraparni Dasu, Yaron Kanza, Divesh Srivastava, and Li Xiong. Fraud buster: Tracking IRSF using blockchain while protecting business confidentiality. In *11th Conference on Innovative Data Systems Research, CIDR*. www.cidrdb.org, 2021.

[132] Sujaya Maiyya, Danny Hyun Bum Cho, Divyakant Agrawal, and Amr El Abbadi. Fides: Managing data on untrusted infrastructure. In *40th IEEE International Conference on Distributed Computing Systems, ICDCS 2020, Singapore, November 29 - December 1, 2020*, pages 344–354. IEEE, 2020. doi:10.1109/ICDCS47774.2020.00053.

[133] André Martin, Cong Lian, Franz Gregor, Robert Krahn, Valerio Schiavoni, Pascal Felber, and Christof Fetzer. Adam-cs: Advanced asynchronous monotonic counter service. In *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 426–437, 2021. doi:10.1109/DSN48987.2021.00053.

[134] Scott Martin. Flying high again: Airplane turnarounds take a spin with ai, 2018. URL: https://blogs.nvidia.com/blog/2018/12/21/airplane-turnarounds-gpu-ai-assaia/.

[135] Sinisa Matetic, Mansoor Ahmed, Kari Kostiainen, Aritra Dhar, David Sommer, Arthur Gervais, Ari Juels, and Srdjan Capkun. ROTE: Rollback Protection for Trusted Execution. In *Proceedings of the 26th USENIX Conference on Security Symposium*, SEC'17, page 1289–1306, USA, 2017. USENIX Association.

[136] Garrett McGrath and Paul 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.

[137] Alfred J. Menezes, Scott A. Vanstone, and Paul C. Van Oorschot. *Handbook of Applied Cryptography*. CRC Press, Inc., 1st edition, 1996.

[138] G. E. Moore. Cramming more components onto integrated circuits, Reprinted from Electronics, volume 38, number 8, 1965, pp.114 ff. *IEEE Solid-State Circuits Society Newsletter*, 2006.

[139] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2009. URL: https://bitcoin.org/bitcoin.pdf.

[140] F. Nawab, D. Agrawal, and A. El Abbadi. The Challenges of Global-scale Data Management. In *Proceedings of the 2016 International Conference on Management of Data*, pages 2223–2227. ACM, 2016. doi:10.1145/2882903.2912571.

[141] Faisal Nawab, Divy Agrawal, and Amr El Abbadi. Nomadic datacenters at the network edge: Data management challenges for the cloud with mobile infrastructure. In *Proceedings of the 21st International Conference on Extending Database Technology*, pages 497–500. OpenProceedings.org, 2018. doi:10.5441/002/edbt.2018.56.

[142] Faisal Nawab and Mohammad Sadoghi. Blockplane: A global-scale byzantizing middleware. In *35th International Conference on Data Engineering*, pages 124–135. IEEE, 2019. doi:10.1109/ICDE.2019.00020.

[143] Ray Neiheiser, Miguel Matos, and Luís Rodrigues. Kauri: Scalable BFT Consensus with Pipelined Tree-Based Dissemination and Aggregation. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 35–48, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3477132.3483584.

[144] Zhaojie Niu and Bingsheng He. A study of big data computing platforms: Fairness and energy consumption. In *IC2E Workshop*, 2016.

[145] Dan O'Keeffe, Theodoros Salonidis, and Peter R. Pietzuch. Frontier: Resilient edge processing for the internet of things. *Proc. VLDB Endow.*, 11(10):1178–1191, 2018. doi:10.14778/3231751.3231767.

[146] OneTrust. French Government Announce National Strategy for Cloud Technology, 2021. URL: https://www.onetrust.com/blog/french-government-announce-national-strategy-for-cloud-technology/.

[147] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, pages 305–320. USENIX Association, 2014.

[148] M. Tamer Özsu and Patrick Valduriez. *Principles of Distributed Database Systems*. Springer New York, 3th edition, 2011.

[149] M. Tamer Özsu and Patrick Valduriez. *Principles of Distributed Database Systems*. Springer, 2020. doi:10.1007/978-3-030-26253-2.

[150] Jianli Pan, Lin Ma, Ravishankar Ravindran, and Peyman TalebiFard. Homecloud: An edge cloud framework and testbed for new application delivery. In *23rd International Conference on Telecommunications, ICT 2016, Thessaloniki, Greece, May 16-18, 2016*, pages 1–6. IEEE, 2016. doi:10.1109/ICT.2016.7500391.

[151] Jianli Pan, Jianyu Wang, Austin Hester, Ismail Alqerm, Yuanni Liu, and Ying Zhao. Edgechain: An edge-iot framework and prototype based on blockchain and smart contracts. *IEEE Internet of Things Journal*, 6(3):4719–4732, 2019. doi:10.1109/JIOT.2018.2878154.

[152] John Paparrizos, Chunwei Liu, Bruno Barbarioli, Johnny Hwang, Ikraduya Edian, Aaron J. Elmore, Michael J. Franklin, and Sanjay Krishnan. Vergedb: A database for iot analytics on edge devices. In *11th Conference on Innovative Data Systems Research, CIDR*. www.cidrdb.org, 2021.

[153] Bryan Parno, Jacob R. Lorch, John R. Douceur, James W. Mickens, and Jonathan M. McCune. Memoir: Practical state continuity for protected modules. In *32nd IEEE Symposium on Security and Privacy, S&P 2011, 22-25 May 2011, Berkeley, California, USA*, pages 379–394. IEEE Computer Society, 2011. doi:10.1109/SP.2011.38.

[154] Rafael Pass and Elaine Shi. Hybrid consensus: Efficient consensus in the permissionless model, 2016. URL: https://eprint.iacr.org/2016/917.

[155] Michael Pisa and Matt 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.

[156] Nathaniel Popper. Worries grow that the price of bitcoin is being propped up, January 2018. URL: https://www.nytimes.com/2018/01/31/technology/bitfinex-bitcoin-price.html.

[157] Thamir Qadah, Suyash Gupta, and Mohammad Sadoghi. Q-Store: Distributed, Multi-partition Transactions via Queue-oriented Execution and Communication. In *Proceedings of the 23nd International Conference on Extending Database Technology, EDBT*, 2020.

[158] Thamir M. Qadah and Mohammad Sadoghi. QueCC: A queue-oriented, control-free concurrency architecture. In *Proceedings of the 19th International Middleware Conference*, pages 13–25, 2018. doi:10.1145/3274808.3274810.

[159] Sajjad Rahnama, Suyash Gupta, Thamir Qadah, Jelle Hellings, and Mohammad Sadoghi. Scalable, resilient and configurable permissioned blockchain fabric. *Proc. VLDB Endow.*, 13(12):2893–2896, 2020. doi:doi.org/10.14778/3415478.3415502.

[160] Sajjad Rahnama, Suyash Gupta, Thamir M. Qadah, Jelle Hellings, and Mohammad Sadoghi. Scalable, resilient, and configurable permissioned blockchain fabric. *Proc. VLDB Endow.*, 13(12):2893–2896, August 2020. doi:10.14778/3415478.3415502.

[161] Sajjad Rahnama, Suyash Gupta, Rohan Sogani, Dhruv Krishnan, and Mohammad Sadoghi. RingBFT: Resilient Consensus over Sharded Ring Topology. In *Proceedings of the 25th International Conference on Extending Database Technology, EDBT*, pages 2:298–2:311. OpenProceedings.org, 2022. doi:10.48786/edbt.2022.17.

[162] Thomas C. Redman. The impact of poor data quality on the typical enterprise. *Commun. ACM*, 41(2):79–82, 1998.

[163] Abderahman Rejeb, John G. Keogh, Suhaiza Zailani, Horst Treiblmaier, and Karim Rejeb. Blockchain technology in the food industry: A review of potentials, challenges and future research directions. *Logistics*, 4(4), 2020. URL: https://www.mdpi.com/2305-6290/4/4/27, doi:10.3390/logistics4040027.

[164] Aleta Ricciardi, Kenneth Birman, and Patrick Stephenson. The cost of order in asynchronous systems. In *Distributed Algorithms*, pages 329–345. Springer Berlin Heidelberg, 1992. doi:10.1007/3-540-56188-9_22.

[165] Pingcheng Ruan, Tien Tuan Anh Dinh, Dumitrel Loghin, Meihui Zhang, Gang Chen, Qian Lin, and Beng Chin Ooi. Blockchains vs. distributed databases: Dichotomy and fusion. In *SIGMOD '21: International Conference on Management of Data*, pages 1504–1517. ACM, 2021. doi:10.1145/3448016.3452789.

[166] Mohammad Sadoghi and Spyros Blanas. *Transaction Processing on Modern Hardware*. Synthesis Lectures on Data Management. Morgan & Claypool, 2019. doi:10.2200/S00896ED1V01Y201901DTM058.

[167] Vasily A. Sartakov, Stefan Brenner, Sonia Ben Mokhtar, Sara Bouchenak, Gaël Thomas, and Rüdiger Kapitza. Eactors: Fast and flexible trusted computing using SGX. In Paulo Ferreira and Liuba Shrira, editors, *Proceedings of the 19th International Middleware Conference*, pages 187–200. ACM, 2018. doi:10.1145/3274808.3274823.

[168] Sambhav Satija, Apurv Mehra, Sudheesh Singanamalla, Karan Grover, Muthian Sivathanu, Nishanth Chandran, Divya Gupta, and Satya Lokam. Blockene: A high-throughput blockchain over mobile devices. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 567–582. USENIX Association, November 2020.

[169] Mahadev Satyanarayanan, Paramvir Bahl, Ramon Caceres, and Nigel Davies. The case for vm-based cloudlets in mobile computing. *IEEE Pervasive Computing*, 8(4):14–23, 2009. doi:10.1109/MPRV.2009.82.

[170] Fabian Sauter and Kilian Traub. *C++ Requests: Curl for People*. 2021. URL: https://docs.libcpr.org/.

[171] Alex Shamis, Peter Pietzuch, Burcu Canakci, Miguel Castro, Cedric Fournet, Edward Ashton, Amaury Chamayou, Sylvan Clebsch, Antoine Delignat-Lavaud, Matthew Kerner, Julien Maffre, Olga Vrousgou, Christoph M. Wintersteiger, Manuel Costa, and Mark Russinovich. IA-CCF: Individual accountability for permissioned ledgers. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 467–491, Renton, WA, April 2022. USENIX Association.

[172] Youren Shen, Hongliang Tian, Yu Chen, Kang Chen, Runji Wang, Yi Xu, Yubin Xia, and Shoumeng Yan. *Occlum: Secure and Efficient Multitasking Inside a Single Enclave of Intel SGX*, page 955–970. Association for Computing Machinery, New York, NY, USA, 2020.

[173] Amit P. Sheth and James A. Larson. Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases. *ACM Comput. Surv.*, 22(3):183–236, September 1990. doi:10.1145/96602.96604.

[174] Simon Shillaker and Peter R. Pietzuch. Faasm: Lightweight isolation for efficient stateful serverless computing. In Ada Gavrilovska and Erez Zadok, editors, *2020 USENIX Annual Technical Conference, USENIX ATC*, pages 419–433. USENIX Association, 2020.

[175] Victor Shoup. Practical threshold signatures. In *Advances in Cryptology — EUROCRYPT 2000*, pages 207–220. Springer Berlin Heidelberg, 2000. doi:10.1007/3-540-45539-6_15.

[176] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–10, 2010. doi:10.1109/MSST.2010.5496972.

[177] Dale Skeen. A quorum-based commit protocol. Technical report, Cornell University, 1982.

[178] Vikram Sreekanti, Chenggang Wu, Saurav Chhatrapati, Joseph E. Gonzalez, Joseph M. Hellerstein, and Jose M. Faleiro. A Fault-Tolerance Shim for Serverless Computing. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3342195.3387535.

[179] Ion Stoica and Scott Shenker. From Cloud Computing to Sky Computing. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '21, page 26–32, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3458336.3465301.

[180] Salvatore J. Stolfo, Malek Ben Salem, and Angelos D. Keromytis. Fog computing: Mitigating insider data theft attacks in the cloud. In *2012 IEEE Symposium on Security and Privacy Workshops*, pages 125–128, 2012. doi:10.1109/SPW.2012.19.

[181] Florian Suri-Payer, Matthew Burke, Zheng Wang, Yunhao Zhang, Lorenzo Alvisi, and Natacha Crooks. Basil: Breaking up bft with acid (transactions). In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 1–17, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3477132.3483552.

[182] Gerard Tel. *Introduction to Distributed Algorithms*. Cambridge University Press, 2nd edition, 2001.

[183] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. Calvin: Fast Distributed Transactions for Partitioned Database Systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD, pages 1–12. ACM, 2012. doi:10.1145/2213836.2213838.

[184] Maarten van Steen and Andrew S. Tanenbaum. *Distributed Systems*. Maarten van Steen, 3th edition, 2017. URL: https://www.distributed-systems.net/.

[185] Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, Lau Cheuk Lung, and Paulo Verissimo. Efficient byzantine fault-tolerance. *IEEE Trans. Comput.*, 62(1):16–30, 2013. doi:10.1109/TC.2011.221.

[186] Hoang Tam Vo, Ashish Kundu, and Mukesh K. Mohania. Research directions in blockchain data management and analytics. In *Proceedings of the 21st International Conference on Extending Database Technology*, pages 445–448. OpenProceedings.org, 2018. doi:10.5441/002/edbt.2018.43.

[187] Jiaping Wang and Hao Wang. Monoxide: Scale out blockchains with asynchronous consensus zones. In *Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation*, pages 95–112. USENIX Association, 2019.

[188] Jingjing Wang, Tobin Baker, Magdalena Balazinska, Daniel Halperin, Brandon Haynes, Bill Howe, Dylan Hutchison, Shrainik Jain, Ryan Maas, Parmita Mehta, Dominik Moritz, Brandon Myers, Jennifer Ortiz, Dan Suciu, Andrew Whitaker, and Shengliang Xu. The Myria Big Data Management and Analytics System and Cloud Services. In *8th Biennial Conference on Innovative Data Systems Research*. www.cidrdb.org, 2017.

[189] Wenbin Wang, Chaoshu Yang, Runyu Zhang, Shun Nie, Xianzhang Chen, and Duo Liu. Themis: Malicious wear detection and defense for persistent memory file systems. In *2020 IEEE 26th International Conference on Parallel and Distributed Systems (ICPADS)*, pages 140–147, 2020. doi:10.1109/ICPADS51040.2020.00028.

[190] Gavin Wood. Ethereum: a secure decentralised generalised transaction ledger, 2016. EIP-150 revision. URL: https://gavwood.com/paper.pdf.

[191] Di Wu, He Xu, Zhongkai Jiang, Weiren Yu, Xuetao Wei, and Jiwu Lu. Edgelstm: Towards deep and sequential edge computing for iot applications. *IEEE/ACM Trans. Netw.*, 29(4):1895–1908, 2021. doi:10.1109/TNET.2021.3075468.

[192] Mingli Wu, Kun Wang, Xiaoqin Cai, Song Guo, Minyi Guo, and Chunming Rong. A comprehensive survey of blockchain: From theory to iot applications and beyond. *IEEE Internet of Things Journal*, 6(5):8114–8154, 2019. doi:10.1109/JIOT.2019.2922538.

[193] Wentai Wu, Ligang He, Weiwei Lin, and Rui Mao. Accelerating federated learning over reliability-agnostic clients in mobile edge computing systems. *IEEE Trans. Parallel Distributed Syst.*, 32(7):1539–1551, 2021. doi:10.1109/TPDS.2020.3040867.

[194] Yuncheng Wu, Tien Tuan Anh Dinh, Guoyu Hu, Meihui Zhang, Yeow Meng Chee, and Beng Chin Ooi. Serverless model serving for data science. *CoRR*, abs/2103.02958, 2021. arXiv:2103.02958.

[195] Cheng Xu, Ce Zhang, Jianliang Xu, and Jian Pei. Slimchain: Scaling blockchain transactions through off-chain storage and parallel processing. *Proc. VLDB Endow.*, 14(11):2314–2326, 2021.

[196] Sravya Yandamuri, Ittai Abraham, Kartik Nayak, and Michael 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.

[197] Jiannan Yang, Tiantian Qian, Fan Zhang, and Samee U. Khan. Real-time facial expression recognition based on edge computing. *IEEE Access*, 9:76178–76190, 2021. doi:10.1109/ACCESS.2021.3082641.

[198] Xinying Yang, Yuan Zhang, Sheng Wang, Benquan Yu, Feifei Li, Yize Li, and Wenyuan Yan. Ledgerdb: A centralized ledger database for universal audit and verification. *Proc. VLDB Endow.*, 13(12):3138–3151, 2020. doi:10.14778/3415478.3415540.

[199] Jian Yin, Jean-Philippe Martin, Arun Venkataramani, Lorenzo Alvisi, and Mike Dahlin. Separating Agreement from Execution for Byzantine Fault Tolerant Services. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, page 253–267, 2003. doi:10.1145/945445.945470.

[200] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan Gueta, and Ittai Abraham. HotStuff: BFT consensus with linearity and responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pages 347–356. ACM, 2019. doi:10.1145/3293611.3331591.

[201] Victor Zakhary, Divy Agrawal, and Amr El Abbadi. Atomic commitment across blockchains. *Proc. VLDB Endow.*, 13(9):1319–1331, 2020. doi:10.14778/3397230.3397231.

[202] Mahdi Zamani, Mahnush Movahedi, and Mariana 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.

[203] Erfan Zamanian, Carsten Binnig, Tim Harris, and Tim Kraska. The End of a Myth: Distributed Transactions Can Scale. *Proc. VLDB Endow.*, 10(6):685–696, 2017. doi:10.14778/3055330.3055335.

[204] Ce Zhang, Cheng Xu, Jianliang Xu, Yuzhe Tang, and Byron Choi. Gem^2-tree: A gas-efficient structure for authenticated range queries in blockchain. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 842–853, 2019. doi:10.1109/ICDE.2019.00080.

[205] Gengrui Zhang, Fei Pan, Michael Dang'ana, Yunhao Mao, Shashank Motepalli, Shiquan Zhang, and Hans-Arno Jacobsen. Reaching consensus in the byzantine empire: A comprehensive review of BFT consensus algorithms. *CoRR*, abs/2204.03181, 2022. arXiv:2204.03181, doi:10.48550/arXiv.2204.03181.

[206] Xiaoyu Zhang, Shixun Huang, Hai Dong, and Zhifeng Bao. Edge node placement with minimum costs: When user tolerance on service delay matters. In *Service-Oriented Computing - 19th International Conference, ICSOC*, volume 13121 of *Lecture Notes in Computer Science*, pages 765–772. Springer, 2021. doi:10.1007/978-3-030-91431-8_53.