ResilientDB: Global Scale Resilient Blockchain Fabric

Suyash Gupta* Sajjad Rahnama* Jelle Hellings Mohammad Sadoghi
Exploratory Systems Lab
Department of Computer Science
University of California, Davis
{sgupta,srahnama,jhellings,msadoghi}@ucdavis.edu

ABSTRACT

Recent developments in blockchain technology have inspired innovative new designs in resilient distributed systems and database systems. At their core, these blockchain applications typically use Byzantine fault-tolerant consensus protocols to maintain a common state across all replicas, even if some replicas are faulty or malicious. Unfortunately, existing consensus protocols are not designed to deal with *geo-scale deployments* in which many replicas spread across a geographically large area participate in consensus.

To address this, we present the Geo-Scale Byzantine Fault-Tolerant consensus protocol (GeoBFT). GeoBFT is designed for excellent scalability by using a topological-aware grouping of replicas in local clusters, by introducing parallelization of consensus at the local level, and by minimizing communication between clusters. To validate our vision of high-performance geo-scale resilient distributed systems, we implement GeoBFT in our efficient ResilientDB permissioned blockchain fabric. We show that GeoBFT is not only sound and provides great scalability, but also outperforms state-of-the-art consensus protocols by a factor of six in geo-scale deployments.

PVLDB Reference Format:
Suyash Gupta, Sajjad Rahnama, Jelle Hellings, and Mohammad Sadoghi. ResilientDB: Global Scale Resilient Blockchain Fabric. *PVLDB*, 13(6): xxxx-yyyy, 2020.
DOI: https://doi.org/10.14778/3380750.3380757

1. INTRODUCTION

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 [8, 14, 17, 21, 24, 46, 49, 56, 74, 75, 79, 93]. 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 [33, 44, 49].

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 [45, 47, 55, 88, 76, 87]. 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).

*Both authors have equally contributed to this work.

Proceedings of the VLDB Endowment, Vol. 13, No. 6
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3380750.3380757

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.

	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

1.1 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.

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 coordi-

nating 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) [18, 19] 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 [9, 62, 63]. 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, 23]. The recently introduced HotStuff improves on Pbft by simplifying the recovery process on primary failure [94]. 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 [45]. Unfortunately, also PoE has a centralized design that depends on a single primary. Finally, the geo-aware consensus protocol Steward promises to do better [5], 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.

1.2 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 $\mathcal{C}_1$ and $\mathcal{C}_2$ with $\mathbf{n}$ replicas each, then our optimistic communication protocol requires only $\lceil \mathbf{n}/3 \rceil$ messages to be sent from $\mathcal{C}_1$ to $\mathcal{C}_2$ when $\mathcal{C}_1$ needs to share local consensus decisions with $\mathcal{C}_2$. 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.

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

Protocol	Decisions	Communication		Centralized
		(Local)	*(Global)*	
GeoBFT (our paper)	$\mathbf{z}$	$\mathcal{O}(2\mathbf{zn}^2)$	$\mathcal{O}(\mathbf{fz}^2)$	No
↳ *single decision*	1	$\mathcal{O}(4\mathbf{n}^2)$	$\mathcal{O}(\mathbf{fz})$	No
Steward	1	$\mathcal{O}(2\mathbf{zn}^2)$	$\mathcal{O}(\mathbf{z}^2)$	Yes
Zyzzyva	1	$\mathcal{O}(\mathbf{zn})$		Yes
Pbft	1	$\mathcal{O}(2(\mathbf{zn})^2)$		Yes
PoE	1	$\mathcal{O}((\mathbf{zn})^2)$		Yes
HotStuff	1	$\mathcal{O}(8(\mathbf{zn}))$		Partly

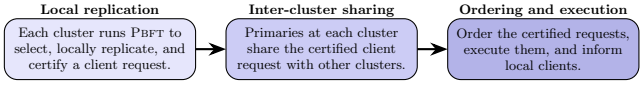

Figure 1: Steps in a round of the GeoBFT protocol.

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. To validate our vision of using GeoBFT in geo-scale settings, we present our ResilientDB fabric [48] and implement GeoBFT in this fabric.[1]

7. 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 [25]. We show that GeoBFT *achieves up-to-six times more throughput* than existing Bft protocols.

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.

2. GeoBFT: 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 1: *local replication*, *global sharing*, and *ordering and execution*, which we further detail next.

[1]We have open-sourced our ResilientDB fabric at `https://resilientdb.com/`.

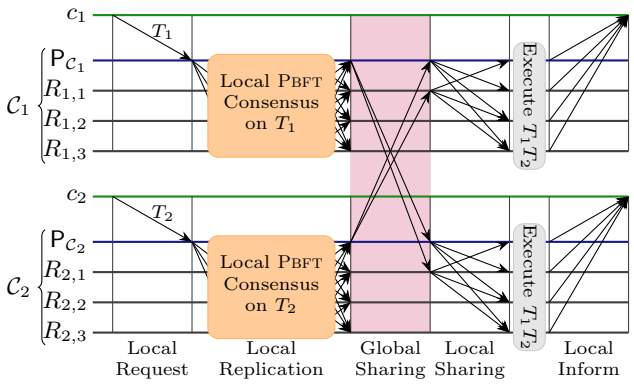

Figure 2: Representation of the normal-case algorithm of GeoBFT running on two clusters. Clients c_i, $i \in \{1, 2\}$, request transactions T_i from their local cluster $\mathcal{C}_i$. The primary $\mathsf{P}_{\mathcal{C}_i} \in \mathcal{C}_i$ replicates this transaction to all local replicas using 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 $\mathcal{C}_i$ can inform c_i.

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 2, we sketch a single round of GeoBFT in a setting of two clusters with four replicas each.

2.1 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 $\mathfrak{S} = \{\mathcal{C}_1, \ldots, \mathcal{C}_{\mathbf{z}}\}$, in which each cluster $\mathcal{C}_i$, $1 \leq i \leq \mathbf{z}$, is a set of $|\mathcal{C}_i| = \mathbf{n}$ 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 [5], but our failure model differs from the more-general failure model utilized by Pbft, Zyzzyva, and HotStuff [9, 18, 19, 62, 63, 94]. 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 [32, 35, 36, 37, 41].

We write $\mathsf{f}(\mathcal{C}_i)$ to denote the Byzantine replicas in cluster $\mathcal{C}_i$ and $\mathsf{nf}(\mathcal{C}_i) = \mathcal{C}_i \setminus \mathsf{f}(\mathcal{C}_i)$ to denote the non-faulty replicas in $\mathcal{C}_i$. Each replica $R \in \mathcal{C}_i$ 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 [58, 72]. 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 $R \in \mathcal{C}_i$, one can determine that R did send m if $R \notin \mathsf{f}(\mathcal{C}_i)$; and one can only determine that m was sent by a non-faulty replica if $R \in \mathsf{nf}(\mathcal{C}_i)$. 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) [39]. For messages that are forwarded, authenticated communication is already provided via digital signatures. For all other messages, we use less-costly message authentication codes [58, 72]. 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 [15, 16, 42, 43], 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.

2.2 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 $\mathcal{C} \in \mathfrak{S}$ replicating a client request T of client $c \in \text{clients}(\mathcal{C})$. To do so, GeoBFT relies on Pbft [18, 19],[2] a primary-backup protocol 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 $\mathsf{P}_\mathcal{C}$ to denote the replica in $\mathcal{C}$ that is the current *local primary* of cluster $\mathcal{C}$. The normal-case of Pbft operates in four steps which we sketch in Figure 3. Next, we detail these steps.

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

Then, in round ρ, $\mathsf{P}_\mathcal{C}$ 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 $R \in \mathcal{C}$ 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 [85].

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. [18, 19]). *Let $\mathfrak{S}$ be a system and let $\mathcal{C} \in \mathfrak{S}$ be a cluster with $\mathbf{n} > 3\mathbf{f}$. We have the following:*

[2]Other consensus protocols such as Zyzzyva [9, 62, 63] and HotStuff [94] 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 of Zyzzyva (reliable clients) and HotStuff (high computational complexity) can be avoided.

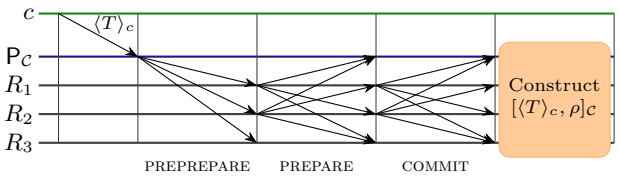

Figure 3: The normal-case working of round ρ of Pbft within a cluster $\mathcal{C}$: a client c requests transaction T, the primary $\mathsf{P}_\mathcal{C}$ proposes this request to all local replicas, which prepare and commit this proposal, and, finally, all replicas can construct a commit certificate.

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

Non-divergence *If replicas $R_1, R_2 \in \mathcal{C}$ 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 $\mathcal{C}$ for round ρ show commitment to the same client request $\langle T \rangle_c$. Hence, we write $[\langle T \rangle_c, \rho]_\mathcal{C}$, to represent a commit certificate from some replica in cluster $\mathcal{C}$.

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 3, 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 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 $\mathcal{C} \in \mathfrak{S}$ be a cluster. After $\mathcal{C}$ reaches local consensus on client request $\langle T \rangle_c$ in round ρ—enabling construction of the commit certificate $[\langle T \rangle_c, \rho]_\mathcal{C}$ that proves local consensus—$\mathcal{C}$ 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 $\mathcal{C}_1, \mathcal{C}_2 \in \mathfrak{S}$. Consider a simple global communication protocol in which a message m is sent from $\mathcal{C}_1$ to $\mathcal{C}_2$ by requiring the primary $\mathsf{P}_{\mathcal{C}_1}$ to send m to the primary $\mathsf{P}_{\mathcal{C}_2}$ (which can then disseminate m in $\mathcal{C}_2$). In this protocol, the replicas in $\mathcal{C}_2$ cannot determine

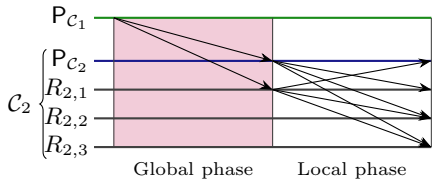

Figure 4: A schematic representation of the normal-case working of the global sharing protocol used by $\mathcal{C}_1$ to send $m = (\langle T \rangle_c, [\langle T \rangle_c, \rho]_{\mathcal{C}_1})$ to $\mathcal{C}_2$.

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

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

Figure 5: The normal-case global sharing protocol used by $\mathcal{C}_1$ to send $m = (\langle T \rangle_c, [\langle T \rangle_c, \rho]_{\mathcal{C}_1})$ to $\mathcal{C}_2$.

what went wrong if they do not receive any messages. To show this, we distinguish two cases:

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

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

In both cases, the replicas in $\mathcal{C}_2$ do not receive any messages from $\mathcal{C}_1$, 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 $\mathcal{C}_2$ to determine whether $\mathsf{P}_{\mathcal{C}_1}$ is faulty (and did not send any messages) or $\mathsf{P}_{\mathcal{C}_2}$ is faulty (and did not forward any received messages from $\mathcal{C}_1$).

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 $m = (\langle T \rangle_c, [\langle T \rangle_c, \rho]_{\mathcal{C}_1})$ be the message that some replica in cluster $\mathcal{C}_1$ needs to send to some replicas $\mathcal{C}_2$. Note that m includes the request replicated in $\mathcal{C}_1$ 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 4 and present the detailed pseudo-code for this protocol in Figure 5.

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

Proposition 2.5. *Let $\mathfrak{S}$ be a system, let $\mathcal{C}_1, \mathcal{C}_2 \in \mathfrak{S}$ be two clusters, and let $m = (\langle T \rangle_c, [\langle T \rangle_c, \rho]_{\mathcal{C}_1})$ be the message $\mathcal{C}_1$ sends to $\mathcal{C}_2$ using the normal-case global sharing protocol of Figure 5. We have the following:*

Receipt *If the primary $\mathsf{P}_{\mathcal{C}_1}$ is non-faulty and communication is reliable, then every replica in $\mathcal{C}_2$ 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 $\mathfrak{S} = \{\mathcal{C}_1, \ldots, \mathcal{C}_\mathbf{z}\}$ 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

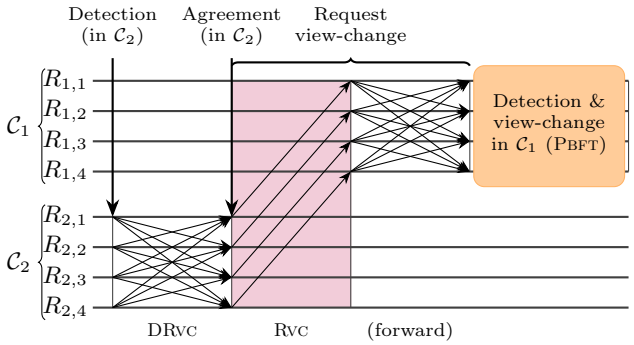

Figure 6: A schematic representation of the remote view-change protocol of GEOBFT running at a system $\mathfrak{S}$ over $\mathfrak{R}$. This protocol is triggered when a cluster $\mathcal{C}_2 \in \mathfrak{S}$ expects a message from $\mathcal{C}_1 \in \mathfrak{S}$, but does not receive this message in time.

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 $\mathcal{C}_2, \ldots, \mathcal{C}_\mathbf{z}$ 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 6 and 7, we sketch this protocol and its pseudo-code. Next, we describe the protocol in detail.

Let $R \in \mathcal{C}_2$ be a replica that detects failure of $\mathcal{C}_1$ in round ρ and has already requested v_1 remote view-changes in $\mathcal{C}_1$. Once a replica R detects a failure, it initiates the process of reaching an agreement on this failure among other replicas of its cluster $\mathcal{C}_2$. It does so by broadcasting message $\text{DRVC}(\mathcal{C}_1, \rho, v_1)$ to all replicas in $\mathcal{C}_2$ (Line 3 of Figure 7).

Next, R waits until it receives identical $\text{DRVC}(\mathcal{C}_1, \rho, v_1)$ messages from $\mathbf{n} - \mathbf{f}$ distinct replicas in $\mathcal{C}_2$ (Line 12 of Figure 7). This guarantees that there is agreement among the non-faulty replicas in $\mathcal{C}_2$ that $\mathcal{C}_1$ has failed. After receiving these $\mathbf{n} - \mathbf{f}$ messages, R requests a remote view-change by sending message $\langle \text{RVC}(\mathcal{C}_1, \rho, v_1) \rangle_R$ to the replica $Q \in \mathcal{C}_1$ with $\text{id}(R) = \text{id}(Q)$ (Line 13 of Figure 7).

In case some other replica $R' \in \mathcal{C}_2$ received m from $\mathcal{C}_1$, then R' would respond with message m in response to the message $\text{DRVC}(\mathcal{C}_1, \rho, v)$ (Line 5 of Figure 7). This allows R to recover in cases where it could not reach an agreement on the failure of $\mathcal{C}_1$. Finally, some replica $R' \in \mathcal{C}_2$ may detect the failure of $\mathcal{C}_1$ later than R. To handle such a case, we require each replica R' that receives identical $\text{DRVC}(\mathcal{C}_1, \rho, v)$ messages from $\mathbf{f} + 1$ distinct replicas in $\mathcal{C}_2$ to assume that

Initiation role (used by replicas $R \in \mathcal{C}_2$) **:**

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

Response role (used by replicas $Q \in \mathcal{C}_1$) **:**

14: **event** Q receives $\langle \text{RVC}(\mathcal{C}_1, \rho, v) \rangle_R$ from R, $R \in (\mathfrak{R} \setminus \mathcal{C}_1)$ **do**
15: Broadcast $\langle \text{RVC}(\mathcal{C}_1, \rho, v) \rangle_R$ to all replicas in $\mathcal{C}_1$.
16: **event** Q receives $\langle \text{RVC}(\mathcal{C}_1, \rho, v) \rangle_{R_i}$, $1 \leq i \leq \mathbf{f} + 1$, such that:
1. $\{R_i \mid 1 \leq i \leq \mathbf{f} + 1\} \subset \mathcal{C}'$, $\mathcal{C}' \in \mathfrak{S}$;
2. $|\{R_i \mid 1 \leq i \leq \mathbf{f} + 1\}| = \mathbf{f} + 1$;
3. no recent local view-change was triggered; and
4. $\mathcal{C}'$ did not yet request a v-th remote view-change

do
17: Detect failure of $\mathsf{P}_{\mathcal{C}_1}$ (if not yet done so).

Figure 7: 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.

the cluster $\mathcal{C}_1$ has failed. This assumption is valid as one of these $\mathbf{f} + 1$ messages must have come from a non-faulty replica in $\mathcal{C}_2$, which must have detected the failure of cluster $\mathcal{C}_1$ successfully (Line 8 of Figure 7).

If replica $Q \in \mathcal{C}_1$ receives a remote view-change request $m_{\text{RCV}} = \langle \text{RVC}(\rho, v) \rangle_R$ from $R \in \mathcal{C}_2$, then Q verifies whether m_{RCV} is well-formed. If m_{RCV} is well-formed, Q forwards m_{RCV} to all replicas in $\mathcal{C}_1$ (Line 14 of Figure 7). Once Q receives $\mathbf{f} + 1$ messages identical to m_{RCV}, signed by distinct replicas in $\mathcal{C}_2$, it concludes that at least one of these remote view-change requests must have come from a non-faulty replica in $\mathcal{C}_2$. 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 $\mathcal{C}_2$ (the lather prevents replay attacks). If these conditions are met, Q detects its current primary $\mathsf{P}_{\mathcal{C}_1}$ as faulty (Line 16 of Figure 7).

When communication is reliable, the above protocol ensures that all non-faulty replicas in $\mathcal{C}_1$ will detect failure of $\mathsf{P}_{\mathcal{C}_1}$. Hence, eventually a successful local view-change will be triggered in $\mathcal{C}_1$. When a new primary in $\mathcal{C}_1$ 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 5). As replicas in $\mathcal{C}_2$ do not know the exact communication delays, they use exponential back off to determine the timeouts used while detecting subsequent failures of $\mathcal{C}_1$.

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

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

Proof. Consider the remote view-change protocol of Figure 7. If a non-faulty replica $R' \in \mathsf{nf}(\mathcal{C}_2)$ receives m, then any replica in $\mathcal{C}_2$ 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 $\mathcal{C}_2$ 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 $\mathsf{nf}(\mathcal{C}_2)$ will participate in this remote view-change (Line 8). As $|\mathsf{nf}(\mathcal{C}_2)| = \mathbf{n}-\mathbf{f}$, each of these $\mathbf{n}-\mathbf{f}$ replicas $R \in \mathsf{nf}(\mathcal{C}_2)$ will send $\langle \text{RVC}(\mathcal{C}_1, \rho, v)\rangle_R$ to some replica $Q \in \mathcal{C}_1$, $\text{id}(R) = \text{id}(Q)$ (Line 12). Let $S = \{Q \in \mathcal{C}_1 \mid R \in \mathsf{nf}(\mathcal{C}_2) \wedge \text{id}(R) = \text{id}(Q)\}$ be the set of receivers in $\mathcal{C}_1$ of these messages and let $T = S \cap \mathsf{nf}(\mathcal{C}_1)$. 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 $\mathcal{C}_1$ (Line 14). As $|T| > \mathbf{f}$, this eventually triggers a local view-change in $\mathcal{C}_1$ (Line 16). □

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

Theorem 2.7. *Let $\mathfrak{S} = \{\mathcal{C}_1, \ldots, \mathcal{C}_\mathbf{z}\}$ 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 $\{(\langle T_i\rangle_{c_i}, [\langle T_i\rangle_{c_i}, \rho]_{\mathcal{C}_i}) \mid (1 \leq i \leq \mathbf{z}) \wedge (c_i \in \text{clients}(\mathcal{C}_i))\}$ of $\mathbf{z}$ messages. These sets all contain identical client requests.*

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

2.4 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 2.2) and the inter-cluster sharing step (Section 2.3) each replica in $\mathfrak{R}$ will receive the same set of $\mathbf{z}$ client requests in round ρ. Let $S_\rho = \{(\langle T_i\rangle_{c_i} \mid (1 \leq i \leq \mathbf{z}) \wedge (c_i \in \text{clients}(\mathcal{C}_i))\}$ 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 $[T_1, \ldots, T_\mathbf{z}]$. Once the execution is complete, each replica $R \in \mathcal{C}_i$, $1 \leq i \leq \mathbf{z}$, 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. □

2.5 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 2.4), 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 $\mathcal{C}_i$, $1 \leq i \leq z$ 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 $\mathcal{C}$ does not have client requests to execute in a round, the primary $\mathsf{P}_\mathcal{C}$ can propose a no-op-request. The primary $\mathsf{P}_\mathcal{C}$ 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 $\mathsf{P}_\mathcal{C}$ can indefinitely ignore requests from some or all clients in clients($\mathcal{C}$), we rely on standard Pbft techniques to detect and resolve such attacks during local replication. These techniques effectively allow clients in clients($\mathcal{C}$) 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 7 (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 [18, 19]. 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 [13, 83].

3. IMPLEMENTATION IN ResilientDB

GeoBFT is designed to enable geo-scale deployment of a permissioned blockchain. Next, we present our ResilientDB fabric [48], a permissioned blockchain fabric that uses GeoBFT to provide such a geo-scale aware high-performance

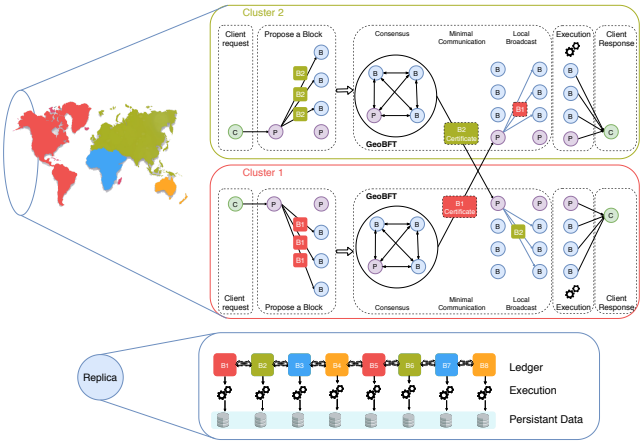

Figure 8: Architecture of our ResilientDB Fabric.

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 1). In Figure 8, we present the architecture of ResilientDB.

At the core of ResilientDB is the ordering of client requests and appending them to the *ledger*—the immutable append-only blockchain representing the ordered sequence of accepted client requests. The order of each client request will be determined via GeoBFT, which we described in Section 2. Next, we focus on how the ledger is implemented and on other practical details that enable geo-scale performance.

The ledger (blockchain). The key purpose of any blockchain fabric is the maintenance of the ledger: the immutable append-only blockchain representing the ordered sequence of client requests accepted. In ResilientDB, the i-th block in the ledger consists of the i-th executed client request. Recall that in each round ρ of GeoBFT, each replica executes $\mathbf{z}$ requests, each belonging to a different cluster $\mathcal{C}_i$, $1 \leq i \leq \mathbf{z}$. Hence, in each round ρ, each replica creates $\mathbf{z}$ blocks in the order of execution of the $\mathbf{z}$ requests. To assure immutability of each block, the block not only consists of the client request, but also contains a commit certificate. This prevents tampering of any block, as only a single commit certificate can be made per cluster per GeoBFT round (Lemma 2.3). As ResilientDB is designed to be a fully-replicated blockchain, each replica independently maintains a full copy of the ledger. The immutable structure of the ledger also helps when recovering replicas: tampering of its ledger by any replica can easily be detected. Hence, a recovering replica can simply read the ledger of any replica it chooses and directly verify whether the ledger can be trusted (is not tampered with).

Cryptography. The implementation of GeoBFT and the ledger requires the availability of strong cryptographic primitives, e.g., to provide digital signatures and authenticated communication (see Section 2.1). To do so, ResilientDB provides its replicas and clients access to NIST-recommended strong cryptographic primitives [10]. In specific, we use ED25519-based digital signatures to sign our messages and we use AES-CMAC message authentication codes to implement authenticated communication [58]. Further, we employ SHA256 to generate collision-resistant message digests.

Pipelined consensus. From our experience designing and implementing Byzantine consensus protocols, we know that throughput can be limited by waiting (e.g., due to message latencies) or by computational costs (e.g., costs of signing and verifying messages). To address both issues simultaneously, ResilientDB provides a multi-threaded pipelined architecture for the implementation of consensus protocols. In Figure 9, we have illustrated how GeoBFT is implemented in this multi-threaded pipelined architecture.

With each replica, we associate a set of *input threads* that receive messages from the network. The primary has one input thread dedicated to accepting client requests, which this input thread places on the batch queue for further processing. All replicas have two input threads for processing all other messages (e.g., those related to local replication and global sharing). Each replica has two *output threads* for sending messages. The ordering (consensus) and execution of client requests is done by the *worker*, *execute*, and *certify* threads. We provide details on these threads next.

Request batching. In the design of ResilientDB and GeoBFT, we support the grouping of client requests in *batches*. Clients can group their requests in batches and send these batches to their local cluster. Furthermore, local primaries can group requests of different clients into a single batch. Each batch is then processed by the consensus protocol (GeoBFT) as a single request, thereby sharing the cost associated with reaching consensus among each of the requests in the batch. Such request batching is a common practice in high-performance blockchain systems and can be applied to a wide range of workloads (e.g., processing financial transactions).

When an input thread at a local primary $\mathsf{P}_\mathcal{C}$ receives a batch of client requests from a client, the thread assigns it a linearly increasing number and places the batch in the *batch queue*. Via this assigned number, $\mathsf{P}_\mathcal{C}$ already decided the order in which this batch needs to be processed (and the following consensus steps are only necessary to communicate this order reliably with all other replicas). Next, the *batching thread* at $\mathsf{P}_\mathcal{C}$ takes request batches from the batch queue and initiates local replication. To do so, the batching thread initializes the data-structures used by the local replication step, creates a valid PRE-PREPARE message for the batch (see Section 2.2), and puts this message on the output queue. The output threads dequeue these messages from the output queue and send them to all intended recipients.

When a input thread at replica $R \in \mathcal{C}$ receives a message as part of local replication (e.g., PRE-PREPARE, PREPARE, COMMIT, see Section 2.2), then it places this message in the *work queue*. Next, the *worker thread* processes any incoming messages placed on the work queue by performing the steps of Pbft, the local replication protocol. At the end of the local replication step, when the worked thread has received $\mathbf{n} - \mathbf{f}$ identical COMMIT messages for a batch, it notifies both the *execution thread* and *certify thread*. The certify thread creates a commit certificate corresponding to this notification and places this commit certificate in the output queue, initiating inter-cluster sharing (see Section 2.3).

Figure 9: The multi-threaded implementation of GeoBFT in ResilientDB. *Left*, the implementation for local primaries. *Right*, the implementation for other replicas.

When a input thread at replica $R \in \mathcal{C}$ receives a message as part of global sharing (a commit certificate), then it places this message in the *certify queue*. Next, the certify thread processes any incoming messages placed on the certify queue by performing the steps of the global sharing protocol. If the certify thread has processed commit certificates for batches in round ρ from all clusters, then it notifies the execution thread that round ρ can be ordered and executed.

The execute thread at replica $R \in \mathcal{C}$ waits until it receives notification for all commit certificates associated with the next round. These commit certificates correspond to all the client batches that need to be ordered and executed. When the notifications are received, the execute thread simply performs the steps described in Section 2.4.

Other protocols. ResilientDB also provides implementations of *four* other state-of-the-art consensus protocols, namely, Pbft, Zyzzyva, HotStuff, and Steward (see Section 1.1). Each of these protocols use the multi-threaded pipelined architecture of ResilientDB and are structured similar to the design of GeoBFT. Next, we provide some details on each of these protocols.

We already covered the working of Pbft in Section 2.2. Next, Zyzzyva is designed with the most optimal case in mind: it requires non-faulty clients and depends on clients to aid in the recovery of *any* failures. To do so, clients in Zyzzyva require identical responses from all $\mathbf{n}$ replicas. If these are not received, the client initiates recovery of any requests with sufficient $\mathbf{n}-\mathbf{f}$ responses by broadcasting certificates of these requests. This will greatly reduce performance when *any* replicas are faulty. In ResilientDB, the certify thread at each replica processes these recovery certificates.

HotStuff is designed to reduce communication of Pbft. To do so, HotStuff uses threshold signatures to combine $\mathbf{n}-\mathbf{f}$ message signatures into a single signature. As there is no readily available implementation for threshold signatures available for the Crypto++ library, we skip the construction and verification of threshold signatures in our implementation of HotStuff. Moreover, we allow each replica of HotStuff to act as a primary in parallel without requiring the usage of pacemaker-based synchronization [94]. Both decisions give our HotStuff implementation a substantial performance advantage in practice.

Finally, we also implemented Steward. This protocol groups replicas into clusters, similar to GeoBFT. Different from GeoBFT, Steward designates one of these clusters as the *primary cluster*, which coordinates all operations. To reduce inter-cluster communication, Steward uses threshold signatures. As with HotStuff, we omitted these threshold signatures in our implementation.

4. 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) [25]. 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 [85].

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?

4.1 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 10.

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. 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× and outperform HotStuff by a factor of up-to-1.3×.

4.2 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 11.

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.

4.3 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 $\mathbf{f}$ 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 12.

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, 23].

$\mathbf{f}$ *non-primary replica failures per cluster.* In this experiment, we measure the performance of GeoBFT in the worst

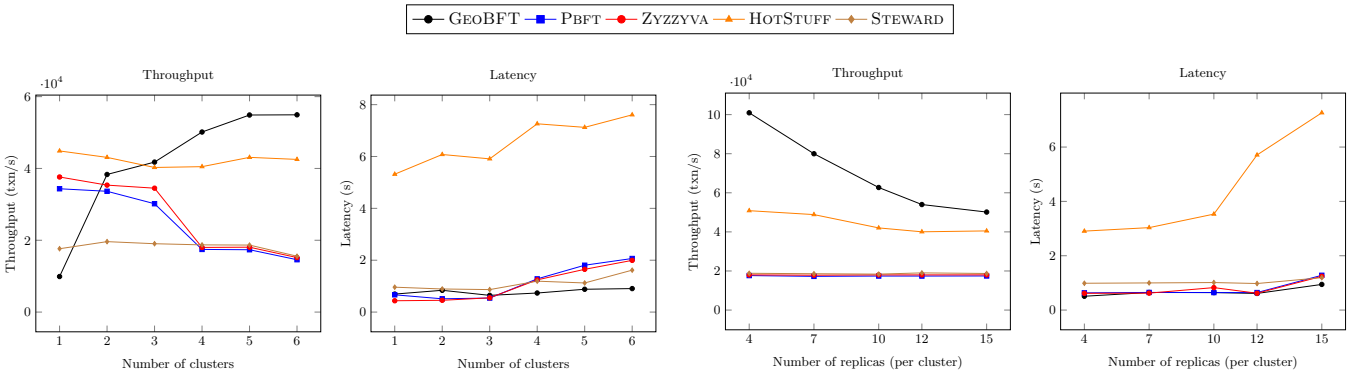

Figure 10: Throughput and latency as a function of the number of clusters; zn = 60 replicas.

Figure 11: Throughput and latency as a function of the number of replicas per cluster; z = 4.

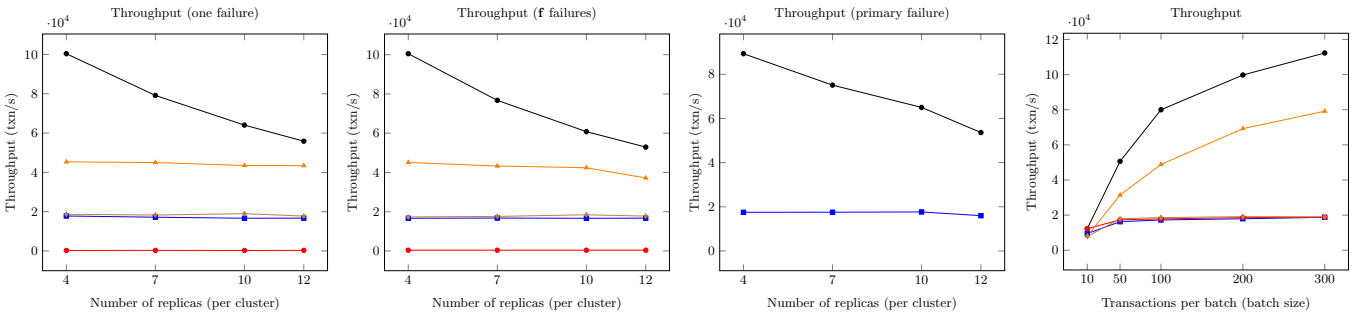

Figure 12: Throughput as a function of the number of replicas per cluster; z = 4. *Left*, throughput with one non-primary failure. *Middle*, throughput with f non-primary failures. *Right*, throughput with a single primary failure.

Figure 13: Throughput as a function of the batch size; z = 4 and n = 7.

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 2.2). 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 **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.

4.4 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 13.

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×. 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×.

5. RELATED WORK

Resilient systems and consensus protocols have been widely studied by the distributed computing community (e.g., [50, 51, 54, 76, 80, 82, 84, 87, 88]). Here, we restrict ourselves to works addressing some of the challenges addressed by GeoBFT: consensus protocols supporting high-performance or geo-scale aware resilient system designs.

Traditional Bft *consensus.* The consensus problem and related problems such as Byzantine Agreement and Interactive Consistency have been studied in detail since the late seventies [32, 35, 36, 37, 38, 41, 42, 64, 65, 73, 78, 86, 87]. The introduction of the Pbft-powered *BFS*—a fault-tolerant version of the networked file system [52]—by Castro et al. [18, 19] marked the first practical high-performance system using consensus. Since the introduction of Pbft, many consensus protocols have been proposed that improve on aspects of Pbft, e.g, Zyzzyva, PoE, and HotStuff, as discussed in the Introduction. To further improve on the performance of Pbft, some consensus protocols consider providing less failure resilience [2, 20, 29, 66, 68, 69], or rely on trusted components [11, 22, 28, 57, 89, 90]. Some recent protocols propose the notion of multiple parallel primaries [46, 47]. Although such designs have partial decentralization, these protocols are still not geo-scale aware. None of these protocols are fully geo-scale aware, however, making them unsuitable for the setting we envision for GeoBFT.

Protocols such as Steward, Blink, and Menicus improve on Pbft by partly optimizing for geo-scale aware deployments [4, 5, 70, 71]. In Section 4, we already showed that the design of Steward—which depends on a primary cluster—severely limits its performance. The Blink protocol improved the design of Steward by removing the need for a primary cluster, as it requires each cluster to order all incoming requests [4]. This design comes with high communication costs, unfortunately. Finally, Menicus tries to reduce communication costs for clients by letting clients only communicate with close-by replicas [70, 71]. By alternating the primary among all replicas, this allows clients to propose requests without having to resort to contacting replicas at geographically large distances. As such, this design focusses on the costs perceived by clients, whereas GeoBFT focusses on the overall costs of consensus.

Sharded Blockchains. Sharding is an indispensable tool used by database systems to deal with Big Data [1, 26, 27, 34, 76, 88]. Unsurprisingly, recent blockchain systems such as SharPer, Elastico, Monoxide, AHL, and RapidChain explore the use of sharding within the design of a replicated blockchain [6, 7, 30, 67, 92, 95]. To further enable sharded designs, also high-performance communication primitives that enable communication between fault-tolerant clusters (shards) have been proposed [53]. Sharding does not fully solve performance issues associated with geo-scale deployments of permissioned blockchains, however. The main limitation of sharding is the efficient evaluation of complex operations across shards [12, 76], and sharded systems achieve high throughputs only if large portions of the workload access single shards. For example, in SharPer [7] and AHL [30], two recent permissioned blockchain designs, consensus on the cross-shard transactions is achieved either by running Pbft among the replicas of the involved shards or by starting a two-phase commit protocol after running Pbft locally within each shard, both methods with significant cross-shard costs.

Our ResilientDB fabric enables geo-scale deployment of a fully replicated blockchain system that does not face such challenges, and can easily deal with any workloads. Indeed, the usage of sharding is orthogonal to the fully-replicated design we aim at with ResilientDB. Still, the integration of geo-scale aware sharding with the design of GeoBFT—in which local data is maintained in local clusters only—promises to be an interesting avenue for future work.

Permissionless Blockchains. Bitcoin, the first wide-spread permissionless blockchain, uses *Proof-of-Work* (PoW) to replicate data [49, 74, 93]. PoW requires limited communication between replicas, can support many replicas, and can operate in unstructured geo-scale peer-to-peer networks in which independent parties can join and leave at any time [81]. Unfortunately, PoW incurs a high computational complexity on all replicas, which has raised questions about the energy consumption of Bitcoin [31, 91]. Additionally, the complexity of PoW causes relative long transaction processing times (minutes to hours) and significantly limits the number of transactions a permissionless blockchain can handle: in 2017, it was reported that Bitcoin can only process 7 transactions per second, whereas Visa already processes 2000 transactions per second on average [79]. Since the introduction of Bitcoin, several PoW-inspired protocols and Bitcoin-inspired systems have been proposed [40, 59, 60, 61, 77, 93], but none of these proposals come close to providing the performance of traditional permissioned systems, which are already vastly outperformed by GeoBFT.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we present our Geo-Scale Byzantine Fault-Tolerant consensus protocol (GeoBFT), a novel consensus protocol with great scalability. To achieve great scalability, GeoBFT relies on a topological-aware clustering of replicas in local clusters to minimize costly global communication, while providing parallelization of consensus. As such, GeoBFT enables geo-scale deployments of high-performance blockchain systems. To support this vision, we implement GeoBFT in our permissioned blockchain fabric—ResilientDB—and show that GeoBFT is not only correct, but also attains up to *six times* higher throughput than existing state-of-the-art Bft protocols.

7. REFERENCES

[1] 2ndQuadrant. Postgres-XL: Open source scalable SQL database cluster. URL: https://www.postgres-xl.org/.

[2] Michael Abd-El-Malek, Gregory R. Ganger, Garth R. Goodson, Michael K. Reiter, and Jay J. Wylie. Fault-scalable byzantine fault-tolerant services. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, pages 59–74. ACM, 2005. doi:10.1145/1095810.1095817.

[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] Yair Amir, Brian Coan, Jonathan Kirsch, and John Lane. Customizable fault tolerance forwide-area replication. In *26th IEEE International Symposium on Reliable Distributed Systems*, pages 65–82. IEEE, 2007. doi:10.1109/SRDS.2007.40.

[5] 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.

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

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

[8] 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.

[9] 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.

[10] Elaine Barker. Recommendation for key management, part 1: General. Technical report, National Institute of Standards & Technology, 2016. Special Publication 800-57 Part 1, Revision 4. doi:10.6028/NIST.SP.800-57pt1r4.

[11] 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.

[12] Philip A. Bernstein and Dah-Ming W. Chiu. Using semi-joins to solve relational queries. *Journal of the ACM*, 28(1):25–40, 1981. doi:10.1145/322234.322238.

[13] 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.

[14] 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.

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

[16] 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.

[17] 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.

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

[19] 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.

[20] Gregory Chockler, Dahlia Malkhi, and Michael K. Reiter. Backoff protocols for distributed mutual exclusion and ordering. In *Proceedings 21st International Conference on Distributed Computing Systems*, pages 11–20. IEEE, 2001. doi:10.1109/ICDSC.2001.918928.

[21] 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.

[22] Byung-Gon Chun, Petros Maniatis, Scott Shenker, and John Kubiatowicz. Attested append-only memory: Making adversaries stick to their word. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, pages 189–204. ACM, 2007. doi:10.1145/1294261.1294280.

[23] 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.

[24] 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.

[25] 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.

[26] Oracle Corporation. MySQL NDB cluster: Scalability. URL: https://www.mysql.com/products/cluster/scalability.html.

[27] Oracle Corporation. Oracle sharding. URL: https://www.oracle.com/database/technologies/high-availability/sharding.html.

[28] Miguel Correia, Nuno Ferreira Neves, and Paulo Verissimo. How to tolerate half less one byzantine nodes in practical distributed systems. In *Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems*, pages 174–183. IEEE, 2004. doi:10.1109/RELDIS.2004.1353018.

[29] James Cowling, Daniel Myers, Barbara Liskov, Rodrigo Rodrigues, and Liuba Shrira. HQ replication: A hybrid quorum protocol for byzantine fault tolerance. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, pages 177–190. USENIX Association, 2006.

[30] 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.

[31] Alex de Vries. Bitcoin's growing energy problem. *Joule*, 2(5):801–805, 2018. doi:10.1016/j.joule.2018.04.016.

[32] 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.

[33] 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.

[34] Microsoft Docs. Sharding pattern. URL: https://docs.microsoft.com/en-us/azure/architecture/patterns/sharding.

[35] 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.

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

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

[38] Danny Dolev and Rüdiger Reischuk. Bounds on information exchange for byzantine agreement. *Journal of the ACM*, 32(1):191–204, 1985. doi:10.1145/2455.214112.

[39] 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.

[40] Ittay Eyal, Adem Efe Gencer, Emin Gün Sirer, and Robbert Van Renesse. Bitcoin-NG: A scalable blockchain protocol. In *13th USENIX Symposium on Networked Systems Design and Implementation*, pages 45–59, Santa Clara, CA, 2016. USENIX Association.

[41] 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.

[42] 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.

[43] 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.

[44] 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.

[45] Suyash Gupta, Jelle Hellings, Sajjad Rahnama, and Mohammad Sadoghi. Proof-of-Execution: Reaching Consensus through Fault-Tolerant Speculation, 2019. URL: https://arxiv.org/abs/1911.00838.

[46] 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.

[47] Suyash Gupta, Jelle Hellings, and Mohammad Sadoghi. Scaling blockchain databases through parallel resilient consensus paradigm, 2019. URL: https://arxiv.org/abs/1911.00837.

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

[49] 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.

[50] 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.

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

[52] Thomas Haynes and David Noveck. RFC 7530: Network file system (NFS) version 4 protocol, 2015. URL: https://tools.ietf.org/html/rfc7530.

[53] 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.

[54] Jelle Hellings and Mohammad Sadoghi. Coordination-free byzantine replication with minimal communication costs. In *Proceedings of the 23rd International Conference on Database Theory*, volume 155 of *Leibniz International Proceedings in Informatics (LIPIcs)*. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2020.

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

[56] 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.

[57] Rüdiger Kapitza, Johannes Behl, Christian Cachin, Tobias Distler, Simon Kuhnle, Seyed Vahid Mohammadi, Wolfgang Schröder-Preikschat, and Klaus Stengel. CheapBFT: Resource-efficient byzantine fault tolerance. In *Proceedings of the 7th ACM European Conference on Computer Systems*, pages 295–308. ACM, 2012. doi:10.1145/2168836.2168866.

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

[59] Aggelos Kiayias, Alexander Russell, Bernardo David, and Roman Oliynykov. Ouroboros: A provably secure Proof-of-Stake blockchain protocol. In *Advances in Cryptology – CRYPTO 2017*, pages 357–388. Springer International Publishing, 2017. doi:10.1007/978-3-319-63688-7_12.

[60] Sunny King and Scott Nadal. PPCoin: Peer-to-peer crypto-currency with proof-of-stake, 2012. URL: https://peercoin.net/assets/paper/peercoin-paper.pdf.

[61] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Nicolas Gailly, Ismail Khoffi, Linus Gasser, and Bryan Ford. Enhancing bitcoin security and performance with strong consistency via collective signing. In *Proceedings of the 25th USENIX Conference on Security Symposium*, pages 279–296. USENIX Association, 2016.

[62] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: Speculative byzantine fault tolerance. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, pages 45–58. ACM, 2007. doi:10.1145/1294261.1294267.

[63] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: Speculative byzantine fault tolerance. *ACM Transactions on Computer Systems*, 27(4):7:1–7:39, 2009. doi:10.1145/1658357.1658358.

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

[65] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, 1982. doi:10.1145/357172.357176.

[66] Barbara Liskov and Rodrigo Rodrigues. Byzantine clients rendered harmless. In *Distributed Computing*, pages 487–489. Springer Berlin Heidelberg, 2005. doi:10.1007/11561927_35.

[67] Loi Luu, Viswesh Narayanan, Chaodong Zheng, Kunal Baweja, Seth Gilbert, and Prateek Saxena. A secure sharding protocol for open blockchains. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 17–30. ACM, 2016. doi:10.1145/2976749.2978389.

[68] Dahlia Malkhi and Michael Reiter. Byzantine quorum systems. *Distributed Computing*, 11(4):203–213, 1998. doi:10.1007/s004460050050.

[69] Dahlia Malkhi and Michael Reiter. Secure and scalable replication in Phalanx. In *Proceedings Seventeenth IEEE Symposium on Reliable Distributed Systems*, pages 51–58. IEEE, 1998. doi:10.1109/RELDIS.1998.740474.

[70] Yanhua Mao, Flavio P. Junqueira, and Keith Marzullo. Mencius: Building efficient replicated state machines for WANs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, pages 369—384. USENIX Association, 2008.

[71] Yanhua Mao, Flavio P. Junqueira, and Keith Marzullo. Towards low latency state machine replication for uncivil wide-area networks. In *Fifth Workshop on Hot Topics in System Dependability*, 2009.

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

[73] Shlomo Moran and Yaron Wolfstahl. Extended impossibility results for asynchronous complete networks. *Information Processing Letters*, 26(3):145–151, 1987. doi:10.1016/0020-0190(87)90052-4.

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

[75] 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.

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

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

[78] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, 1980. doi:10.1145/322186.322188.

[79] 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.

[80] Dan R. K. Ports, Jialin Li, Vincent Liu, Naveen Kr. Sharma, and Arvind Krishnamurthy. Designing distributed systems using approximate synchrony in data center networks. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, pages 43–57. USENIX Association,

2015.

[81] Bitcoin Project. Bitcoin developer guide: P2P network, 2018. URL: https://bitcoin.org/en/developer-guide#p2p-network.

[82] 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.

[83] 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.

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

[85] 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.

[86] Gadi Taubenfeld and Shlomo Moran. Possibility and impossibility results in a shared memory environment. *Acta Informatica*, 33(1):1–20, 1996. doi:10.1007/s002360050034.

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

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

[89] Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, and Lau Cheuk Lung. EBAWA: Efficient byzantine agreement for wide-area networks. In *2010 IEEE 12th International Symposium on High Assurance Systems Engineering*, pages 10–19. IEEE, 2010. doi:10.1109/HASE.2010.19.

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

[91] Harald Vranken. Sustainability of bitcoin and blockchains. *Current Opinion in Environmental Sustainability*, 28:1–9, 2017. doi:10.1016/j.cosust.2017.04.011.

[92] 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.

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

[94] 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.

[95] 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.