Ziziphus: Scalable Data Management Across Byzantine Edge Servers

Mohammad Javad Amiri[1] Daniel Shu[2] Sujaya Maiyya[3] Divyakant Agrawal[2] Amr El Abbadi[2]
[1]*University of Pennsylvania,* [2]*University of California Santa Barbara,* [3]*University of Waterloo*
[1]mjamiri@seas.upenn.edu, [2]{danielshu, agrawal, amr}@cs.ucsb.edu, [3]smaiyya@uwaterloo.ca

***Abstract*—Edge computing while bringing computation and data closer to users in order to improve response time, distributes edge servers in wide area networks resulting in increased communication latency between the servers. Synchronizing globally distributed edge servers, especially in the presence of Byzantine servers, becomes costly due to the high communication complexity of Byzantine fault-tolerant consensus protocols. In this paper, we present Ziziphus, a geo-distributed system that partitions edge servers into fault-tolerant zones where each zone processes transactions initiated by nearby clients locally. Global synchronization among zones is required only in special situations, e.g., migration of clients from one zone to another. On the one hand, the two-level architecture of Ziziphus confines the malicious behavior of nodes within zones requiring a much cheaper protocol at the top level for global synchronization. On the other hand, Ziziphus processes local transactions within zones by edge servers closer to clients resulting in enhanced performance. Ziziphus further introduces zone clusters to enhance scalability where instead of running global synchronization among all zones, only zones of a single cluster are synchronized.**

***Index Terms*—Edge Computing, Scalability, Consensus**

I. Introduction

Edge computing shifts computation and data closer to users [9][23][28][33]. While this computational paradigm improves response time and saves bandwidth, edge servers may be distributed in a wide area network where communication among servers incurs high latency. Edge servers communicate mainly to establish agreement on the order of transactions using fault-tolerant protocols. While large-scale data management systems such as Google's Spanner [14], Amazon's Dynamo [16], and Facebook's Tao [11] rely on crash fault-tolerant (CFT) protocols, e.g., Paxos [27], consensus among edge servers requires Byzantine fault-tolerant (BFT) protocols, e.g., PBFT [13] due to the non-trustworthiness of edge infrastructures.

BFT protocols suffer from high communication and message complexity, especially in wide-area networks. For instance, PBFT [13], the most popular BFT protocol, requires $3f + 1$ servers (where f is the number of maximum simultaneous malicious servers), three communication phases, and quadratic message complexity in terms of the number of servers, which make it impractical in establishing consensus among edge servers distributed over wide area networks.

Various attempts have been made to reduce the complexity of BFT protocols in large-scale geo-distributed systems over wide area networks. These include both *hierarchical fully replicated* and *sharded partially replicated* solutions.

On the one hand, in Steward [2] and Blockplane [30], data is fully replicated across multiple fault-tolerant clusters. Both systems use a hierarchical approach where the maliciousness of Byzantine servers is confined within clusters and a CFT protocol is used to establish global consensus among clusters. Thus, every single transaction needs to be globally synchronized to ensure the mutual consistency of all copies of data and availability in case of data center-scale outages.

On the other hand, sharded distributed systems shard data and replicate a data shard on each cluster. A transaction that accesses a single shard is processed by the nodes of the specific cluster, while a global cross-shard transaction is executed on *all* shards. Since nodes may be malicious, clusters use BFT protocols to process their local transactions. While sharded distributed systems, e.g., Caper [3], and Qanaat [8], do not run a global consensus for every transaction, such systems use BFT protocols for global consensus, resulting in high latency.

In the context of edge computing networks, we use the notion of *zones* where each zone maintains the data of its nearby clients. This paper presents *Ziziphus*, a geo-distributed system that supports edge computing applications with possibly mobile edge clients that migrate from one zone to another. Ziziphus thus represents a trade-off between the full replication approach and the sharded approaches where each shard is replicated within a cluster. Ziziphus provides a zonal abstraction where client data is replicated on nodes of a single fault-tolerant zone and only made accessible in other zones when it is needed to process global transactions, e.g., client migration. This zonal abstraction confines the maliciousness of Byzantine servers within each zone while supporting large-scale geo-distributed applications.

In this paper, this abstraction is used to support edge computing applications where data accesses have an affinity towards locality. In Ziziphus, edge nodes are partitioned into Byzantine fault-tolerant zones consisting of $3f+1$ nodes where f is the maximum number of maliciously faulty nodes in a zone. At the local level, each zone processes local transactions initiated by its nearby clients independent of other zones. Thus, the global synchronization in Ziziphus is considerably reduced compared to geo-replicated systems.

A geo-distributed system might require to enforce network-wide policies, e.g., a zone cannot host more than 10000 clients, or a client can migrate at most 10 times a year. Ziziphus maintains *global system meta-data*, including the number of clients of each zone, the number of client migrations, application-

dependent data, etc. on all nodes. Global synchronization among *all* zones is only needed when the global system meta-data needs to be updated. The most common case of global synchronization occurs when a client migrates from one zone to another. In this case, Ziziphus runs a consensus protocol among all zones where, in contrast to sharded distributed systems, the protocol requires linear communication, and only the majority of zones to participate. Ziziphus presents a *data synchronization* protocol to support the global synchronization of zones and a *data migration* protocol to migrate the client data from the source to the destination zone.

Ziziphus is able to tolerate f failures within each zone and $\lfloor \frac{Z-1}{2} \rfloor$ entire zone failures out of Z zones, which might fail due to natural disasters, for global transactions. However, replicating local transactions on only the nodes of a single (nearby) zone weakens the availability guarantees of Ziziphus compared to geo-distributed systems in case an entire zone fails. To partially tolerate failures for local transactions, Ziziphus uses *lazy synchronization* where local updates are periodically shared with all other zones , e.g., when global synchronization is needed.

As the system scales, the number of zones might increase to hundreds or even thousands over wide area networks. Running global synchronization among all these zones for every global transaction results in high latency. To address this problem, Ziziphus defines *zone clusters* where each zone cluster consists of a set of zones in a region. Zones within a zone cluster maintain the same (regional) system meta-data, which is different from the system meta-data maintained by other zone clusters. Using zone clusters, instead of running global synchronization among all zones, only zones of a single cluster are synchronized. A *cross-cluster data synchronization* protocol is presented to handle migration cases where the source and destination zones are in two different zone clusters.

The contributions of this paper are three-fold:

- Ziziphus, a geo-distributed system that partitions Byzantine edge servers into fault-tolerant zones where each zone processes transactions initiated by nearby clients locally and global synchronization among zones is required only in special situations, e.g., the migration of nodes between zones;
- A data synchronization protocol to globally synchronize zones and a data migration protocol to migrate client data from a source to a destination zone; and
- Zone clusters to enhance the scalability of Ziziphus where instead of running global synchronization among all zones, only the zones of a single cluster are synchronized.

II. Motivation

In this section, we briefly describe *healthcare* as a practical edge computing application that can realize the full potential of Ziziphus. Healthcare applications are usually delay-sensitive; hence, deploying healthcare applications in cloud environments results in a high processing latency due to the large network distance between patients and cloud servers [21].

In an edge computing-enabled healthcare application, edge servers store and process the data collected from patients' devices such as sensors, cameras, medical devices, or smartphones to enable advanced remote-patient monitoring. A healthcare application might need to process the data of thousands or even millions of patients. Hence, only the nearby edge server(s) process each client's data to improve response time. Healthcare applications need to support patients' mobility as well when a patient moves from one spatial area (zone) to another. Furthermore, each healthcare application needs to enforce network-wide policies and regulations issued by official institutions, e.g., insurance companies.

In a nutshell, to support such edge computing applications, Ziziphus needs to deal with four main requirements.

Spatial locality. In a healthcare application, transactions on patient data are typically issued by nearby edge devices, e.g., patients, doctors, etc. The co-location of processing infrastructure and edge devices significantly reduces latency, and alleviates the cost and congestion of network use [36].

Edge device mobility. While the data of each patient is maintained by its nearby edge servers, the mobility of patients across zones needs to be supported.

Global synchronization. An edge computing application requires enforcing network-wide policies across all zones. These policies might affect the migration of patients from one zone to another, e.g., insurance policies in the healthcare application.

Scalability. An edge computing application might need to process the data of millions of clients located in hundreds or even thousands of zones over a wide area. In such a setting, different areas (e.g., states, countries) might even have their own policies and regulations, e.g., GDPR, to be enforced.

III. System Model

Ziziphus is designed for edge computing applications where performance in terms of throughput and latency is paramount, data accesses have an affinity towards locality and the probability of failure of an entire zone is insignificant. To address these requirements, Ziziphus makes two main design decisions: (1) clustering nodes into fault-tolerant *zones*, and (2) replicating local transactions of edge devices only on nodes of their nearby zone. These two design decisions demonstrate two trade-offs. First, similar to most clustered distributed systems, Ziziphus is more prone to denial-of-service attacks in comparison to a flat system with the same number of nodes because it is easier for an attacker to target a single zone (i.e., cluster, shard) instead of the entire network. This represents a trade-off between scalability and security. Second, the availability of Ziziphus is reduced if an entire zone fails, e.g., due to natural disasters. This design decision demonstrates a trade-off between performance and availability.

A. Network Infrastructure

The underlying infrastructure consists of a set of nodes in a large-scale asynchronous distributed system. Nodes follow the Byzantine failure model where faulty nodes may exhibit arbitrary, potentially malicious behavior. We assume a strong computationally-bounded adversary that can coordinate malicious nodes and delay communication to compromise the service. However, the adversary cannot subvert cryptographic

assumptions. Ziziphus clusters nodes into fault-tolerant zones where each zone consists of $3f+1$ Byzantine nodes to guarantee safety in the presence of f malicious nodes [10]. Each zone has a designated *primary* node that initiates local consensus among the nodes of the zone and participates in the processing of global transactions with other zones. Nodes within a zone are ideally located geographically close to each other and have low communication latency amongst themselves.

B. Data and Transactions

Each zone maintains the data of its nearby clients. Client data in each zone is *local* and only replicated on the nodes of the zone to provide fault tolerance. In addition to local data, all zones maintain *global system meta-data*. The global system meta-data contains data that are needed to enforce network-wide policies, e.g., a zone cannot host more than 10000 clients or a client can migrate at most 10 times a year. The meta-data is small in size and includes only the required information to enforce such policies, e.g., the number of clients per zone and the number of migrations per client. Global system meta-data is globally replicated on every node of every zone.

Ziziphus supports *local* and *global* transactions. Local transactions, which are assumed to have locality affinity, are initiated by the clients of a zone on their local data in the zone. Nodes of a zone process local transactions independently of other zones and update the local data accordingly. On the other hand, the global system meta-data needs to be updated as a result of a *global transaction*. Clients of an edge network might be mobile and migrate from a *source* zone to a *destination* zone. When a client migrates, it initiates a global transaction. Since client data in each zone is local and stored only on nodes of the zone, Ziziphus does not need to deal with cross-zone transactions (in contrast to sharded distributed systems where a transaction might access multiple shards). However, the zonal abstraction presented by Ziziphus can be easily extended to support cross-zone transactions.

IV. Transaction Processing in Ziziphus

Ziziphus relies on the State Machine Replication (SMR) algorithm [26][32] used for building a fault-tolerant service. An SMR-based BFT protocol commits transactions as a linearizable log to provide a consistent view of the log equivalent to a single non-faulty node, while providing *safety*, i.e., all non-faulty nodes commit the same requests in the same order, and *liveness*, i.e., all non-faulty client requests are eventually ordered. Ziziphus guarantees safety in an asynchronous network; however, it assumes a synchrony assumption to ensure liveness (FLP impossibility result [19]).

A. Local Transactions

Ziziphus targets edge computing applications where clients' transactions are processed by nearby edge servers. The client data are replicated on nodes of the nearby zone where nodes process transactions without any communication with other zones. Since nodes may fail in a Byzantine manner, all local transactions in Ziziphus need to be processed using a BFT protocol. The local consensus protocol is pluggable and any BFT protocol can be used to process local transactions. In its current design, Ziziphus processes local transactions using PBFT [24]. In PBFT, nodes move through a succession of configurations called *views* [17][18]. In a view, one node is *the primary* and the others are *backups* where the primary initiates consensus among nodes.

During a normal case execution of PBFT, a client sends a local request to the primary node of the nearby zone. Upon receiving a valid request from an authorized client, the primary first ensures that the client's data within the zone is up-to-date. Nodes maintain a $lock$ bit for each client to keep track of its mobility, i.e., if the $lock$ bit is TRUE the client data is up-to-date. The primary then assigns a sequence number to the request and broadcasts a pre-prepare message to all nodes of the zone. Nodes then go trough prepare and commit phases to commit and execute the request. PBFT also has a view change routine that provides liveness when the primary fails.

B. Global Transactions

Global transactions are needed when the global system meta-data, which is replicated on all zones, needs to be updated. The most common case occurs when a client migrates from a source to a destination zone; hence, we describe global transactions for this case. The global transaction to support client migration consists of two atomic sub-transactions. The first sub-transaction updates the global system meta-data of all zones using the *data synchronization* protocol, and the second sub-transaction copies the actual client data from the source to the destination zone using the *data migration* protocol.

1) Data Synchronization Protocol

In the *data synchronization protocol*, in contrast to local transactions where votes from more than two-thirds of nodes is required, Ziziphus establishes consensus on the order of a global transaction with agreement from only a majority of zones. This is because Ziziphus confines the effect of all malicious behavior of Byzantine nodes within zones. As a result, instead of using PBFT that has a quadratic communication complexity and requires more than two-thirds of nodes to participate in each communication phase, the data synchronization protocol processes transactions with linear communication complexity and agreement from a majority of zones. The data synchronization protocol is a *two-level* protocol where at the top level, only the primary node of each zone participates to globally agree on the order of a global transaction and at the bottom level, all nodes within each zone communicate with each other to *endorse* the message that will be sent by the primary node at the top level.

To prevent a primary node from acting maliciously in communication with the primary nodes of other zones at the top level, the messages sent by a primary node at the top level need to be endorsed by the nodes of its zone. Each primary node constructs a *certificate* proving that a quorum of $2f+1$ different nodes within its zone agree on the message it sends. A certificate for message consists of a collection of $2f+1$ (identical) messages m signed by different nodes within the same zone. Ziziphus can also use a threshold signature scheme to represent $2f+1$ signatures (out of $3f+1$) using a single constant-sized threshold signature [34][12].

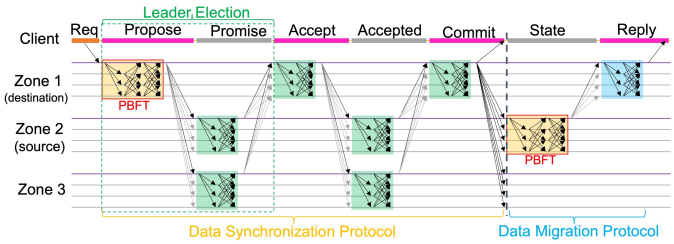

Figure 1. Global transaction processing

Using certificates, the maliciousness of nodes is confined within zones and if a primary node acts maliciously at the top level, its behavior can be easily detected (i.e., its messages will be invalid). As a result, the data synchronization protocol at the top level, As shown in Figure 1, follows the crash fault-tolerant protocol Paxos and proceeds through propose, promise, accept, accepted, and commit phases. The goal of the propose and promise phases is to elect the primary (i.e., leader) node, the goal of the accept and accepted phases is to decide on the request (i.e., value) and the commit phase replicates the request on every node.

The first sub-transaction is initiated by a client. When the client migrates from a source to a destination zone, it sends a migration request message m to the primary node of the destination zone. This primary is referred to as the *global primary*. The destination zone is referred to as the *initiator* zone, and all other zones, the *follower* zones. Note that each message sent by the nodes to a client includes the current view number, allowing the client to track the view and hence the current primary. Nonetheless, if a backup (non-primary) node receives a migration request from a client, it relays it to the global primary. Upon receiving a migration request message, as shown in Figure 1, the local consensus protocol, PBFT, is run in three phases (i.e., pre-prepare, prepare and commit) within the initiator zone to assign a *Ballot number*, establish consensus on the global request and endorse the message.

All other bottom-level communications (green boxes) within either the initiator or follower zones, however, as shown in Figure 1, consist of only the pre-prepare and commit communication phases and the prepare phase of PBFT is skipped. This is because the goal of the prepare phase in PBFT is to ensure that non-faulty nodes agree on the order (i.e., Ballot number) that is assigned by the primary node, i.e., they all received the matching message from the primary. In Ziziphus, however, since the Ballot number is already assigned by the global primary and certified by $2f + 1$ nodes of the initiator zone, there is no need to run the prepare phase of PBFT and upon receiving valid messages from the primary of the zone, nodes multicast commit messages. The primary node of each zone then collects $2f + 1$ messages from nodes of its zone to construct a certificate that is used at the top level communication with the primary node of other zones.

Algorithm 1 presents the normal case operation of the data synchronization protocol. Although not explicitly mentioned, every sent and received message is logged by the nodes. As indicated in lines 1-5, r denotes the id of the node running the algorithm, z_i is the initiator zone, $v(z)$ specifies the view number of the node in zone z, and $\pi(z)$ is the primary of zone z. We use Q_z to denote a quorum of $2f + 1$ different nodes in zone z and Q_M to denote a majority of primary nodes of different zones. Q_z is used for local consensus within a zone and Q_M is used for global consensus where agreement from the majority of zones is needed.

Propose phase. The goal of the propose and promise phases is to elect the primary (i.e., leader) node. Upon receiving a valid signed migration request m from an authorized client c (with timestamp t_c) to execute a transaction o on global system meta-data (lines 6-8), the primary node $\pi(z_i)$ of the initiator zone z_i (the global primary) assigns a global Ballot number $\langle n, z_i \rangle$ to the request where n is the highest global sequence number that $\pi(z_i)$ is aware of it and z_i is the zone id and multicasts a pre-prepare message to *all* nodes of its zone z_i. Timestamp t_c is used to ensure exactly-once semantics for the execution of requests and prevent replay attacks. The timestamps for requests of each client are totally ordered. o is a simple operation that updates meta-data based on the pre-defined policies once executed, e.g., updates the number of clients in the source and the destination zones.

Upon receiving a pre-prepare message, as indicated in lines 9-11, each node r of the initiator zone z_i checks the migration request to be valid and sequence number n to be the highest global sequence number that the node knows and also be within a predefined small range to prevent a malicious primary from exhausting the space of sequence numbers by choosing a very large value [13]. Node r then multicasts a prepare message to all nodes of the initiator zone z_i. Upon receiving a quorum of matching prepare messages from $2f$ different nodes (including itself) that match the pre-prepare message received from the primary (lines 12-13), it multicasts a local-propose message (equal to commit message in PBFT) to all nodes of the initiator zone z_i. This local-propose message is used by the primary in constructing the certificate to prove that a quorum of $2f + 1$ nodes within zone z_i agree with the propose message. The primary aggregates a quorum of $2f+1$ local-propose messages (lines 14-15) to construct a certificate $\mathcal{C}$ and multicasts a propose message to all nodes of every zone.

Promise phase. When the primary node of a follower zone z_f receives a propose message, as shown in lines 16-19, it first checks the request, the message and the certificate $\mathcal{C}$ to be valid and the global sequence number n to be greater than any global sequence number that the node is aware of. Nodes, as mentioned before, maintain a $lock$ bit for each client to keep track of its mobility where $lock$ = TRUE means the client data is up-to-date. If z_f is the source zone, its primary node sets $lock(c)$ to be FALSE. At this point, the source zone does not accept any local requests from client c anymore.

The primary node of the follower zone z_f then multicasts $\langle \text{PRE-PREPARE}, v(z_f), \langle n, z_i \rangle, \langle l, z_l \rangle, p, d_p, d \rangle_{\sigma_{\pi(z_f)}}$ message to the nodes of its zone z_f to initiate consensus on the received propose message. In the pre-prepare message, p is the received propose message, d_p is its digest, and $\langle l, z_l \rangle$ is the Ballot number of the latest migration request that has been accepted

Algorithm 1 Data Synchronization Protocol

1: r := Id of the node running the algorithm, z_i := the initiator zone id
2: $v(z)$:= view number of node r in zone z
3: $\pi(z)$:= the primary node of zone z
4: Q_z := a quorum of $2f+1$ different nodes in zone z
5: Q_M := a (majority) quorum of primary node from different zones

Endorsement in the initiator zone (PROPOSE phase)

▷ $r = \pi(z_i)$:
6: **upon receiving** valid $m = \langle$MIG-REQUEST, $op, ts_c, c\rangle_{\sigma_c}$
7: **assign** *Ballot number* $\langle n, z_i\rangle$ to m
8: **multicast** $\langle$PRE-PREPARE, $v(z_i), \langle n, z_i\rangle, d, m\rangle_{\sigma_{\pi(z_i)}}$ to z_i

▷ $r \in z_i$:
9: **upon receiving** $\langle$PRE-PREPARE, $v(z_i), \langle n, z_i\rangle, d, m\rangle_{\sigma_{\pi(z_i)}}$
10: if message m and Ballot number $\langle n, z_i\rangle$ are valid then
11: **multicast** $\langle$PREPARE, $v(z_i), \langle n, z_i\rangle, d, r\rangle_{\sigma_r}$ to z_i
12: **upon receiving** $\langle$PREPARE, $v(z_i), \langle n, z_i\rangle, d, r\rangle_{\sigma_r}$ from Q_{z_i}
13: **multicast** $\langle$LOCAL-PROPOSE, $v(z_i), \langle n, z_i\rangle, d, r\rangle_{\sigma_r}$ to z_i

▷ $r = \pi(z_i)$:
14: **upon receiving** $\langle$LOCAL-PROPOSE, $v(z_i), \langle n, z_i\rangle, d, r\rangle_{\sigma_r}$ from Q_{z_i}
15: **multicast** $\langle$PROPOSE, $v(z_i), \langle n, z_i\rangle, \mathcal{C}, d, m\rangle_{\sigma_{\pi(z_i)}}$ to every node

Endorsement in a follower zone z_f (PROMISE phase)

▷ $r = \pi(z_f)$:
16: **upon receiving** $p = \langle$PROPOSE, $v(z_i), \langle n, z_i\rangle, \mathcal{C}, d, m\rangle_{\sigma_{\pi(z_i)}}$
17: if n is greater than any received sequence number and $\mathcal{C}$ is valid
18: if z_f is the source zone then $lock(c)$ = FALSE
19: **multicast** $\langle$PRE-PREPARE, $v(z_f), \langle n, z_i\rangle, \langle l, z_l\rangle, p, d_p, d\rangle_{\sigma_{\pi(z_f)}}$ to z_f

▷ $r \in z_f$:
20: **upon receiving** $\langle$PRE-PREPARE, $v(z_f), \langle n, z_i\rangle, \langle l, z_l\rangle, p, d_p, d\rangle_{\sigma_{\pi(z_f)}}$
21: if z_f is the source zone then $lock(c)$ = FALSE
22: **multicast** $\langle$LOCAL-PROMISE, $v(z_f), \langle n, z_i\rangle, \langle l, z_l\rangle, d, r\rangle_{\sigma_r}$ to z_f

▷ $r = \pi(z_f)$:
23: **upon receiving** $\langle$LOCAL-PROMISE, $v(z_f), \langle n, z_i\rangle, \langle l, z_l\rangle, d, r\rangle_{\sigma_r}$ from Q_{z_f}
24: **multicast** $\langle$PROMISE, $v(z_f), \langle n, z_i\rangle, \langle l, z_l\rangle, \mathcal{C}_f, d\rangle_{\sigma_{\pi(z_f)}}$ to z_i

Endorsement in the initiator zone (ACCEPT phase)

▷ $r = \pi(z_i)$:
25: **upon receiving** $q{=}\langle$PROMISE, $v(z_f), \langle n, z_i\rangle, \langle l, z_l\rangle, \mathcal{C}_f, d\rangle_{\sigma_{\pi(z_f)}}$ from Q_M
26: **multicast** $\langle$PRE-PREPARE, $v(z_i), \langle n, z_i\rangle, \langle l, z_l\rangle, q_1,.., d_{q_1},.., d\rangle_{\sigma_{\pi(z_i)}}$ to z_i

▷ $r \in z_i$:
27: **upon receiving** $\langle$PRE-PREPARE, $v(z_i), \langle n, z_i\rangle, \langle l, z_l\rangle, q_1,.., d_{q_1},.., d\rangle_{\sigma_{\pi(z_i)}}$
28: **multicast** $\langle$LOCAL-ACCEPT, $v(z_i), \langle n, z_i\rangle, \langle l, z_l\rangle, d, r\rangle_{\sigma_r}$ to z_i

▷ $r = \pi(z_i)$:
29: **upon receiving** $\langle$LOCAL-ACCEPT, $v(z_i), \langle n, z_i\rangle, \langle l, z_l\rangle, d, r\rangle_{\sigma_r}$ from Q_{z_i}
30: **multicast** $\langle$ACCEPT, $v(z_i), \langle n, z_i\rangle, \langle l, z_l\rangle, \mathcal{C}, d\rangle_{\sigma_{\pi(z_i)}}$ to every node

Endorsement in a follower zone z_f (ACCEPTED phase)

▷ $r = \pi(z_f)$:
31: **upon receiving** $a{=}\langle$ACCEPT, $v(z_i), \langle n, z_i\rangle, \langle l, z_l\rangle, \mathcal{C}, d\rangle_{\sigma_{\pi(z_i)}}$
32: if n is greater than any received sequence number and $\mathcal{C}$ is valid
33: **multicast** $\langle$PRE-PREPARE, $v(z_f), \langle n, z_i\rangle, \langle l, z_l\rangle, a, d_a, d\rangle_{\sigma_{\pi(z_f)}}$ to z_f

▷ $r \in z_f$:
34: **upon receiving** $\langle$PRE-PREPARE, $v(z_f), \langle n, z_i\rangle, \langle l, z_l\rangle, a, d_a, d\rangle_{\sigma_{\pi(z_f)}}$
35: **multicast** $\langle$LOCAL-ACCEPTED, $v(z_f), \langle n, z_i\rangle, \langle l, z_l\rangle, d, r\rangle_{\sigma_r}$ to z_f

▷ $r = \pi(z_f)$:
36: **upon receiving** $\langle$LOCAL-ACCEPTED, $v(z_f), \langle n, z_i\rangle, \langle l, z_l\rangle, d, r\rangle_{\sigma_r}$ from Q_{z_f}
37: **multicast** $\langle$ACCEPTED, $v(z_f), \langle n, z_i\rangle, \langle l, z_l\rangle, \mathcal{C}_f, d\rangle_{\sigma_{\pi(z_f)}}$ to z_i

Endorsement in the initiator zone (COMMIT phase)

▷ $r = \pi(z_i)$:
38: **upon recv.** $a{=}\langle$ACCEPTED, $v(z_f), \langle n, z_i\rangle, \langle l, z_l\rangle, \mathcal{C}_f, d\rangle_{\sigma_{\pi(z_f)}}$ from Q_M
39: **multicast** $\langle$PRE-PREPARE, $v(z_i), \langle n, z_i\rangle, \langle l, z_l\rangle, a_1,.., d_{a_1},.., d\rangle_{\sigma_{\pi(z_i)}}$ to z_i

▷ $r \in z_i$:
40: **upon receiving** $\langle$PRE-PREPARE, $v(z_i), \langle n, z_i\rangle, \langle l, z_l\rangle, a_1,.., d_{a_1},.., d\rangle_{\sigma_{\pi(z_i)}}$
41: **multicast** $\langle$LOCAL-COMMIT, $v(z_i), \langle n, z_i\rangle, \langle l, z_l\rangle, d, r\rangle_{\sigma_r}$ to z_i

▷ $r = \pi(z_i)$:
42: **upon receiving** $\langle$LOCAL-COMMIT, $v(z_i), \langle n, z_i\rangle, \langle l, z_l\rangle, d, r\rangle_{\sigma_r}$ from Q_{z_i}
43: **multicast** $\langle$COMMIT, $v(z_i), \langle n, z_i\rangle, \langle l, z_l\rangle, \mathcal{C}, d\rangle_{\sigma_{\pi(z_i)}}$ to every node

Updating Global Meta-data (EXECUTION phase)

▷ $\forall r$:
44: **upon receiving** $\langle$COMMIT, $v(z_i), \langle n, z_i\rangle, \langle l, z_l\rangle, \mathcal{C}, d\rangle_{\sigma_{\pi(z_i)}}$
45: **execute** client request o on global meta-data

(and either committed or not) by the follower zone z_f. The digest d_p is used to detect changes and alterations to any part of the message. Ballot number $\langle l, z_l\rangle$ determines the execution order of global requests. Note that sending the Ballot Number of the latest accepted migration request irrespective of whether it was committed or not is different from Paxos where follower (i.e., acceptor) nodes send the latest (actual) value that is decided (i.e., accepted) but not yet committed (because the previous leader has failed) to the new leader and the new leader has to propose and commit that value before proposing its own value. The reason is that in Ziziphus, when the primary of a zone fails, another node from the same zone becomes the primary and will continue to process the request, hence, there is no need for the primary node of other zones to recover an accepted value. However, the order of the global requests needs to be preserved, i.e., a request with a lower sequence number must be executed earlier than a request with a higher sequence number. It is also different from PBFT where a single primary node assigns incremental sequence numbers to the requests and nodes execute requests in the same order. Here, since different nodes, i.e., the primary node of different zones, might become the global primary and there might be some gap between the sequence number of consecutive global requests, each request includes the sequence number of its previous global request to provide an ordering for the execution, e.g., if a zone has not received the previous global request, the zone becomes aware of that request by checking the $\langle l, z_l\rangle$ parameter in the current request.

Upon receiving a pre-prepare message from the primary of zone z_f (lines 20-22), each node r in z_f checks the request, the message, the certificate $\mathcal{C}$, and both Ballot numbers to be valid. Similarly, if z_f is the source zone, each node sets $lock(c)$ to be FALSE. The node then multicasts a local-promise message to all nodes of zone z_f. Upon receiving $2f+1$ valid matching local-promise messages from different nodes (lines 23-24), the primary node of each follower zone z_f aggregates these messages to construct a certificate $\mathcal{C}_f$ and then multicasts a promise message to the nodes of the initiator zone z_i.

Accept phase. The accept and accepted phases are used to decide on the request. The global primary $\pi(z_i)$ waits for promise messages from the majority of zones (including itself). This is because each valid message at the top level has been certified by a quorum of $2f+1$ signatures. Hence, any malicious behavior of a Byzantine node (e.g., sending invalid messages) can be easily detected by nodes without communicating with each other. As a result, the safety condition of CFT protocols, e.g., Paxos [27] is sufficient to guarantee the safety of the data synchronization protocol. This is in contrast to PBFT where to detect the malicious behavior of a Byzantine node (e.g., sending an inconsistent message to different replicas) communication between nodes (prepare messages) is needed.

Upon receiving sufficient promise messages from different follower zones (lines 25-26), the global primary multicasts $\langle$PRE-PREPARE, $v(z_i), \langle n, z_i\rangle, \langle l, z_l\rangle, q_1, q_2,.., d_{q_1}, d_{q_2},.., d\rangle_{\sigma_{\pi(z_i)}}$ to all nodes of its zone z_i where l is the greatest (previous) global sequence number that either $\pi(z_i)$ is aware of or is received in promise messages and each q_j is the promise message received from zone z_j and d_{q_j} is its digest. Nodes of z_i validate the received pre-prepare message and multicast a local-accept to the primary node of z_i (lines 27-28). Upon receiving a quorum of $2f+1$ local-accept messages (lines 29-30), the global primary aggregates these messages,

constructs a certificate $\mathcal{C}$, and multicasts an accept message to all nodes of every zone.

Accepted phase. Upon receiving a valid accept message (lines 31-33), the primary node of a follower zone z_f checks the message and the certificate $\mathcal{C}$ to be valid and the global sequence number n to be greater than any sequence number that the node is aware of and ensures that it has not accepted any global sequence number greater than l. The primary node then multicasts pre-prepare message to all nodes of its zone, z_f. Nodes of z_f validate the received pre-prepare message and multicast a local-accepted to the primary node of z_f (lines 34-35). The primary of z_f waits for a quorum of $2f+1$ local-accepted messages from different nodes (lines 36-37), constructs a certificate $\mathcal{C}_f$, and multicasts an accepted message to the nodes of the initiator zone z_i.

Commit phase. The global primary waits for accepted messages from the primary nodes of a majority of zones (lines 38-39) and then multicasts a pre-prepare message to all the nodes of zone z_i to establish consensus on the received accepted message and construct a certificate. In the pre-prepare message, each a_j is the accepted message received from zone z_j and d_{a_j} is its digest. Nodes of z_i validate the received pre-prepare message and multicast a local-commit to the primary node of z_i (lines 40-41). Upon receiving a quorum of $2f+1$ local-commit messages (lines 42-43), the global primary aggregates these messages, constructs a certificate $\mathcal{C}$, and multicasts a commit message to all nodes of every zone.

Execution phase. Once a node in any zones receives a valid commit message from the global primary, the node considers the global transaction as a committed transaction. if the node has executed the previous global transaction with Ballot number $\langle l, z_l \rangle$, the node executes the client request on the global system meta-data. This ensures that all nodes execute requests in the same order as required to ensure safety. Depending on the predefined network-wide policies, executing the request might result in updating the number of clients in the source and the destination zone and incrementing the number of clients' migrations in some period.

Nodes of the initiator zone also send a reply including the execution results to the client to inform that the first sub-transaction has been committed. The client waits for $f+1$ matching responses from different nodes of the initiator zone to ensure that at least one correct node executed its request.

Ziziphus can benefit from the stable leader technique used in multi-Paxos to process global transactions more efficiently. Using the stable leader technique, one of the zones initiates all global transactions and clients irrespective of the source and the destination zones send their migration request messages to the stable initiator zone. In this manner, there is no need for the propose and promise (leader election) phases in the data synchronization protocol.

2) Data Migration Protocol

The first sub-transaction of the global transaction establishes agreement among all zones on the client migration and updates the global system meta-data. In the second sub-transaction, the client data is migrated from the source to the destination zone.

Algorithm 2 Data Migration Protocol

```
     init():
1:     r := Id of the node running the algorithm
2:     z_i := the initiator zone id
3:     z_s := the source zone id
4:     z_d := the destination zone id
5:     v(z) := view number of node r in zone z
6:     π(z) := the primary node of zone z
7:     Q_z := a quorum of 2f + 1 different nodes in zone z
8:     R(c) := records of client c
       Record generation in the source zone z_s
       ▷ r = π(z_s):
9:   upon receiving valid m = ⟨COMMIT, v(z_i), ⟨n, z_i⟩, C, d⟩_σπ(z_i)
10:    extract R(c) from database
11:    multicast ⟨PRE-PREPARE, v(z_s), ⟨n, z_i⟩, R(c), d_c, d⟩_σπ(z_s) to z_s
       ▷ r ∈ z_s:
12:  upon receiving valid ⟨PRE-PREPARE, v(z_s), ⟨n, z_i⟩, R(c), d_c, d⟩_σπ(z_s)
13:    multicast ⟨PREPARE, v(z_s), ⟨n, z_i⟩, d_c, d, r⟩_σr to z_s
14:  upon receiving matching ⟨PREPARE, v(z_s), ⟨n, z_i⟩, d_c, d, r⟩_σr from Q_zs
15:    multicast ⟨LOCAL-STATE, v(z_s), ⟨n, z_i⟩, d_c, d, r⟩_σr to z_s
       ▷ r = π(z_s):
16:  upon receiving matching ⟨LOCAL-STATE, v(z_s), ⟨n, z_i⟩, d_c, d, r⟩_σr from Q_zs
17:    multicast ⟨STATE, v(z_s), ⟨n, z_i⟩, C, R(c), d_c, d⟩_σπ(z_s) to z_d
       Record appending in the destination zone z_d
       ▷ r = π(z_d):
18:  upon receiving valid h=⟨STATE, v(z_s), ⟨n, z_i⟩, C, R(c), d_c, d⟩_σπ(z_s)
19:    multicast ⟨PRE-PREPARE, v(z_d), ⟨n, z_i⟩, h, d_h⟩_σπ(z_d) to z_d
       ▷ r ∈ z_d:
20:  upon receiving valid ⟨PRE-PREPARE, v(z_d), ⟨n, z_i⟩, h, d_h⟩_σπ(z_d)
21:    multicast ⟨LOCAL-COMMIT, v(z_d), ⟨n, z_i⟩, d, d_h, r⟩_σr to z_d
22:  upon receiving ⟨LOCAL-COMMIT, v(z_d), ⟨n, z_i⟩, d, d_h, r⟩_σr from Q_zd
23:    lock(c) = TRUE
24:    append R(c) to the database
25:    send ⟨REPLY, v(z_d), t_c⟩_σr to client c
```

Since the actual client data might be large, only the client data state consisting of the information that is needed to process its transactions, e.g., the account balance of the client in a micropayment application, needs to be moved.

The second sub-transaction is initiated by the primary node of the source zone, i.e., the zone that the client has migrated from, since the source zone maintains the fresh date. The primary needs to generate the state of the client, establish consensus on the client state within its zone to construct a certificate including $2f+1$ signatures and multicast the state to the destination zone. The destination zone validates the message and appends the state to its database. At this point, the destination zone can process the client requests.

Algorithm 2 presents the normal case operation of the data migration protocol to process the second sub-transaction of a global transaction. Although not explicitly mentioned, every sent and received message is logged by the nodes. As indicated in lines 1-8 of the algorithm, r denotes the node id and z_i, z_s and z_d are the initiator, the source and the destination zones respectively. Note that in the common case, the destination zone is the same as the initiator zone ($z_d = z_i$). Using the stable leader technique, however, the destination zone might be different from the initiator zone. $v(z)$ specifies the view number of node r in zone z, $\pi(z)$ is the primary node of zone z, Q_z is a quorum of $2f+1$ different nodes in zone z and $R(c)$ refers to the data records of client c in the source zone.

Record Generation. When the primary node of zone z_s has committed and executed a migration request received from a client c that has migrated from z_s to z_d, as shown in lines 9-11, the primary node $\pi(z_s)$ first generates the application-dependent client data state $R(c)$. The node then initiates local

consensus, i.e., PBFT, on $R(c)$ by multicasting a pre-prepare message to the nodes of zone z_s. Upon receiving a valid pre-prepare message including a client state $R(c)$ from the primary node (lines 12-13), each node r of the zone z_s multicasts a prepare message to all nodes in zone z_s. Each node waits for a quorum of $2f + 1$ valid prepare messages and multicasts a local-state message to all nodes in zone z_s (lines 14-15). The primary $\pi(z_s)$ collects $2f + 1$ local-state messages from different nodes (lines 16-17), constructs a certificate $\mathcal{C}$, and multicasts a state message to the destination zone z_d.

Record Appending. Upon receiving a valid state message, the primary node of the destination zone z_d, as shown in lines 18-19, checks the message and certificate $\mathcal{C}$ to be valid and multicasts pre-prepare message to nodes of its zone. Nodes of z_d validate the received pre-prepare message and multicast a local-commit to all nodes within zone z_d (lines 20-21). Upon receiving a quorum of $2f + 1$ local-commit messages from different nodes (lines 22-25), each node in z_d sets $lock(c)$ to be TRUE to show that the client data is up-to-date, appends the client state $R(c)$ to its database and sends a reply to the client informing that the migration has been performed successfully.

3) Cross-Zone Transactions

The goal of Ziziphus is to support edge computing applications where transactions have location affinity and only update data in a *single* zone. However, the zonal abstraction of Ziziphus can be used to support *cross-zone transactions*, where a transaction accesses different data located in *different* zones. This will require a few minor modification in the data synchronization protocol. First, in processing cross-zone transactions, we consider the initiator zone as the primary. As a result, there is no need for the primary election phase. Second, in communication across zones, messages are sent only to the involved zones (not all zones), and finally, since zones maintain different data, each zone needs to run consensus on the order of transactions. This contrasts with global synchronization where all zones maintain the same global system metadata. Hence, the primary zone proposes a transaction order, and all other zones only validate the order (without the need to rerun consensus).

V. Ziziphus Analysis

In this section, we discuss the primary failure handling routine of Ziziphus and analyze the fault tolerance, availability, and correctness of Ziziphus.

A. Primary Failure Handling

The timeout mechanism prevents the protocol from blocking and waiting forever. Nodes use different timers for local and global transactions because processing transactions across zones usually takes more time. For local transactions and upon failure of the primary node of a zone, the view change routine of PBFT is triggered by timeouts to replace the faulty primary.

However, for global transactions, detecting and handling failures is more difficult. While the data synchronization protocol follows CFT protocols, i.e., it requires linear communication phases and majority quorums, the participants in the global consensus are (Byzantine) primary nodes of different zones. Since any messages need to be endorsed by $2f + 1$ nodes of a zone, the malicious behavior of a Byzantine primary node can be easily detected by other nodes without requiring any communication. A malicious primary node can choose not to send any endorsed messages or only send messages to a subset of the nodes. Hence, in the worst-case scenario, all nodes participating in the global consensus might be malicious and not send any messages. The failure handling of Ziziphus needs to handle all such situations.

If the follower zone z_f has gone through the accepted phase for a request and node r of a follower zone does not receive a commit message from the global primary (i.e., the primary node of the initiator zone z_i) and its timer expires, node r multicasts a $\langle \text{RESPONSE-QUERY}, v(z_f), \langle n, z_i \rangle, d, r \rangle_{\sigma_r}$ message to all nodes of the initiator zone including the request digest d. Similarly, nodes of the initiator zone multicast response-query messages to the nodes of a follower zone if they do not receive accepted messages for their migration request.

In all such cases, if the message has already been processed, the nodes simply re-send the corresponding response and log the response-query messages to detect denial-of-service attacks initiated by malicious nodes. If the node (of a follower zone) has accepted another migration request with a higher ballot number in between, the node simply ignores the response-query messages. If a node receives response-query message from $2f+1$ nodes of another zone (without receiving any other migration request with a higher ballot number in between), it suspects that the primary node of its zone might be faulty triggering the execution of the failure handling routine. Moreover, since all messages from a primary of a zone (either initiator or follower) are multicast to every node of the other zone(s), if the primary of the receiver zone does not initiate consensus on the message among the nodes of its zone (even after the message is relayed by nodes of its zone), it will eventually be suspected to be faulty by the nodes of its zone.

The data migration protocol handles failure in the same way for state messages. Finally, if a client does not receive a reply soon enough, it multicasts the request to all nodes of the destination zone. If the request has already been processed, the nodes simply send the execution result back to the client. Otherwise, if the node is not the primary, it relays the request to the primary. If the nodes do not receive pre-prepare messages, the primary will be suspected to be faulty, triggering the primary failure handling routine.

B. Fault Tolerance and Availability

Ziziphus, as discussed in Section III, clusters nodes into fault-tolerant zones, and replicates local transactions of edge devices only on nodes of their nearby zone. These two decisions demonstrate two trade-offs; one between performance and security, and the other, between performance and availability. In this section, we analyze these two trade-offs.

Proposition 5.1: Ziziphus tolerates $\lfloor \frac{Z-1}{2} \rfloor$ failures out of Z zones for global transactions.

Ziziphus processes global transactions with agreement from only a majority of zones. This means that at the global level, Ziziphus tolerates $\lfloor \frac{Z-1}{2} \rfloor$ failures out of Z zones for global

transactions. Other than a malicious primary, a zone might fail due to natural disasters like tornadoes or earthquakes.

This clearly demonstrates the advantage of clustering the nodes into zones in order to confine the maliciousness of Byzantine nodes within their zones.

Proposition 5.2: Ziziphus guarantees the availability of a zone data if at least $2f+1$ nodes of the zone are non-faulty.

Since each zone includes $3f+1$ nodes, safety is guaranteed even if f nodes within a zone are compromised [10].

Proposition 5.3: A clustered fault-tolerant system, e.g., Ziziphus, provides weaker security guarantees compared to a flat fault-tolerant system with the same number of nodes.

Ziziphus is more prone to denial-of-service attacks because it is easier for an attacker to compromise f nodes within a single zone instead of $Z * f$ nodes in the entire network where Z is the number of zones. This represents a trade-off between performance and security. Ziziphus assumes pre-determined fault-tolerant zones where each zone includes less than one-third faulty nodes. To avoid security attacks, nodes can be randomly assigned to zones in order to uniformly distribute faulty nodes. While this solution provides higher security guarantees, it only guarantees *probabilistic* safety as a zone might include more than one-third faulty nodes (unlike Ziziphus which guarantees *deterministic* safety). To achieve high probability, e.g., $1-2^{-20}$, however, the clusters need to be large-sized, e.g., 80 nodes in AHL [15], resulting in increased latencies. Moreover, to prevent security attacks, clusters need to be reconfigured periodically, e.g., OmniLedger [25].

Proposition 5.4: Given a Ziziphus deployment; if an entire zone fails, the zone data becomes unavailable.

In Ziziphus local data are only updated in nodes of a single zone. As a result, if an entire zone fails, other zones are not able to process the local transactions of the failed zone. Hence, the zone data becomes unavailable during the zone failure, without impacting safety, since no local transactions are executed. This is in contrast to Steward [2] and Blockplane [30] where the failure of zones is tolerated for all transactions by replicating transactions on all zones.

To provide availability despite zone failure, Ziziphus can replicate local transactions on multiple zones where for every local transaction that requires zonal fault tolerance, consensus among all the zones that maintain the data is needed. This approach is similar to the cross-zone transaction processing, discussed in Section IV.B(3). However, in this case, different zones maintain the same data rather than different data. The consequence of this design choice is that while availability is increased, every transaction will incur latencies that are at geo-scale (i.e., 100s of milliseconds versus 10s of milliseconds or less), demonstrating a trade-off between performance and availability.

To partially address this problem, Ziziphus can use lazy synchronization techniques to provide a *weaker* degree of fault tolerance for local transactions without running global synchronization for every transaction.

BFT protocols, e.g., PBFT, use a checkpointing mechanism to produce the last stable state of data (i.e., a persisted state). Checkpoints are generated periodically when a transaction with a sequence number divisible by some constant is executed. The checkpoint is generated by the primary and multicast to all nodes (as part of the pre-prepare messages).

Each checkpoint needs to be signed by a quorum of $2f+1$ nodes within the zone (as part of the local-accepted messages) and the primary of the zone includes that in the accepted message sent to the global primary node. The global primary then puts all received stable checkpoints in its commit message and multicasts it to all zones. Each zone then replicates the latest stable state of every zone consisting of all executed local transactions on all its nodes. In this way, if an entire zone fails, transactions that are executed before its last stable checkpoint have been replicated on all other zones.

Choosing the right checkpointing period is challenging and depends on the characteristics of transactions and the workload. In an edge network where the percentage of migration requests is not supposed to be high, zones can generate checkpoints whenever they receive a migration request. However, generating checkpoints for every migration request reduces performance in workloads with a high percentage of global transactions. Similarly, for mission-critical applications, the system might decide to generate a checkpoint whenever a zone receives a migration request even if the workload includes a high percentage of global transactions.

C. Correctness

Consensus protocols have to satisfy *safety* and *liveness*. We briefly analyze the safety and liveness properties of Ziziphus.

Lemma 5.5: If node r commits transaction m with sequence number n, no other non-faulty node commits request m' ($m \neq m'$) with the same sequence number n.

Proof: PBFT guarantees safety [13]. We just need to show that safety is guaranteed for the data synchronization and data migration protocols. To commit transaction m (promise and) accepted messages from a majority of zones are needed. As a result, if two different transactions m and m' have been committed with sequence numbers n and n' and $n = n'$, at least the primary node of one zone sends valid accepted messages for both transactions.

To send a valid accepted message the primary node needs to collect a quorum $2f+1$ matching votes from different nodes of its zone to construct certificates. As a result, to send accepted messages for both m and m', a quorum of $2f+1$ nodes, Q_m has agreed with n and a quorum of $2f+1$ nodes, $Q_{m'}$ has agreed with n'. since Q_m and $Q_{m'}$ intersect on at least one non-faulty node, the non-faulty node must have agreed with both sequence numbers, violating the definition of non-faulty nodes. Hence, if $m \neq m'$ then $n \neq n'$ (where n' is the sequence number of m') and safety is guaranteed.

Lemma 5.6: A request m issued by a correct client will be complete if the majority of zones can still communicate.

Proof: Due to the FLP result [19], Ziziphus guarantees liveness *only* during periods of synchrony where a majority of zones can still communicate. Ziziphus addresses liveness in primary failure and collision situations. In case of primary failure within a zone, as discussed in Section V-A, the failure

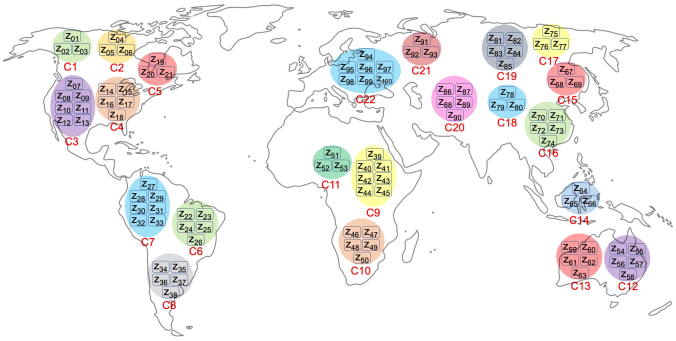

Figure 2. Ziziphus scalability using zone clusters

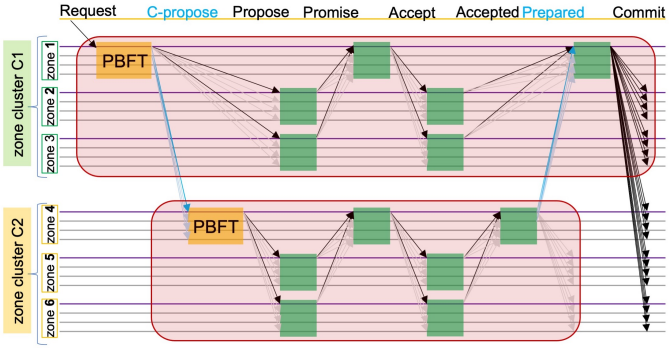

Figure 3. cross-cluster data synchronization protocol

of the primary is detected, and using the view change routine the primary node is replaced. A collision situation happens when the primary nodes of different zones try to initiate global transactions (i.e., data synchronization protocol) in parallel. In this case, when a primary node can not collect a majority quorum, its request's timer will expire, and the primary node needs to re-propose the request. To reduce the probability of consecutive collisions, Ziziphus, similar to Paxos, randomizes the waiting time for the nodes that want to re-propose requests.

VI. Ziziphus Scalability

Processing global transactions in Ziziphus requires establishing consensus among all zones. However, as the system scales, the number of zones might increase to hundreds or even thousands of zones over the wide area network. Running consensus among all these zones for every single global transaction results in low throughput and high latency.

To address this problem, Ziziphus defines *zone clusters* where each zone cluster consists of a set of zones in a region, e.g., country. Zones within a zone cluster maintain the same (regional) system meta-data (instead of global meta-data). Different zone clusters, however, maintain different system meta-data. This is a reasonable assumption because most policies need to be enforced at the regional level, e.g., GDPR, and if zones are spread all around the world, zones in Europe and zones in North America, for instance, do not necessarily follow the same set of policies. As a result, there is no need to maintain global system meta-data by all zones.

Figure 2 demonstrates a network with 100 different zones z_1 to z_{100} where zones are clustered into different zone clusters C_1 to C_{22}. Each cluster consists of several zones, e.g., C_1 has 3 zones z_1, to z_3 while C_3 consists of 7 zones z_7 to z_{13}.

When a client migrates from a source to a destination zone where both source and destination zones are within the same zone cluster, Ziziphus uses the data synchronization and data migration protocols (Algorithms 1 and 2) to process the request. For example, in Figure 2, if a client migrates from zone z_1 to z_2, the data synchronization protocol is run within zone cluster C_1 among only zones z_1, z_2, and z_3 independent of other zone clusters in the network.

If a client migrates to a zone in a different zone cluster, e.g., from z_1 in C_1 to z_4 in C_2, Ziziphus requires agreement from zones of both zone clusters C_1 and C_2. In this case, the zonal abstraction of Ziziphus can be applied on top of two clusters to present a cross-cluster data synchronization protocol. The protocol establishes agreement on the order of the global transaction among the source and the destination cluster. This is different from the data synchronization protocol used within a cluster where all zones need to participate.

Figure 3 presents the cross-cluster data synchronization protocol between two zone clusters C_1 and C_2 where z_1 is the destination (initiator) zone and z_4 is the source zone. The destination zone (i.e., z_1) initiates the cross-cluster protocol (i.e., plays the coordinator role) and also initiates consensus within the destination cluster (i.e., C_1) and the source zone (i.e., z_4) initiates consensus within the source cluster (i.e., C_2).

Upon receiving a migration request m from client c, the (global) primary $\pi(z_i)$ of the initiator zone z_i, similar to the data synchronization protocol, validates the request, assigns a ballot number $\langle n, z_i \rangle$ to the request, and initiates local consensus among nodes of its zone z_i using PBFT.

During cross-cluster data synchronization and to communicate across zone clusters, in contrast to cross-zone communications, Ziziphus does not rely only on the primary nodes. This is because zone clusters process each transaction independently of each other and communicate with each other only in the first and the last phases (i.e., cross-propose and prepared messages). As a result, a malicious global primary might not multicast the cross-propose message to the source zone cluster resulting in high latency (since other zones within the destination cluster cannot detect the malicious behavior of the primary until the last step when they do not receive any prepared messages from the source cluster).

To resolve this issue, we rely on a group of $f+1$ nodes within the destination zone, called *proxy* nodes, to communicate with the source zone. We require $f+1$ nodes because at most f nodes might be malicious. A node r in the zone z is a *proxy* in view v_z if $(v_z \mod r) \in [0, ..., f]$. Note that the primary (the node where $v_z \mod r = 0$) is always a proxy.

Once local consensus on the request order in zone z_i is achieved, proxy nodes aggregate $2f+1$ local-propose messages (received in the last phase of local consensus) to construct a certificate $\mathcal{C}$. The proxy nodes then multicasts $\langle \text{CROSS-PROPOSE}, v(z_i), \langle n, z_i \rangle, \mathcal{C}, d, m \rangle_{\sigma_{\pi(z_i)}}$ message to all nodes of the source zone. The primary node also multicasts a propose message (with the same structure) to all nodes of every zone within its (destination) cluster.

Upon receiving a valid local-propose message, the primary node of the source zone z_j in the source cluster, e.g., z_4 in Figure 3, establishes consensus on the order of the request in the source cluster. In the cross-cluster data synchronization protocol, in contrast to the data synchronization protocol where follower zones only validate the order (Ballot number) proposed by the initiator zone, the source zone also needs to assign a separate Ballot number and establishes consensus on the order of the request. This is because, in the data synchronization protocol, all zones execute global transactions on the same global system meta-data whereas, in the cross-cluster data synchronization protocol, each cluster executes global transactions on its own regional system meta-data. As a result, each cluster requires it own ordering, e.g., in Figure 3 both destination and source zones z_1 and z_4 run PBFT.

Both source and destination clusters then follow the next steps, i.e., promise, accept, and accepted in the same way as data synchronization protocol. Once accepted phase is done, each proxy node r of the source zone z_j in the source cluster, e.g., z_4, constructs a certificate $\mathcal{C}_s$, and multicasts a $\langle \text{PREPARED}, v(z_j), \langle m, z_j \rangle, \mathcal{C}_s, d, r \rangle_{\sigma_r}$ message to all nodes of the destination zone in the destination cluster, e.g., z_1 in Figure 3, to inform them that the message has been prepared in the source cluster with Ballot number $\langle m, z_j \rangle$.

The primary node of the destination zone waits for (1) $2f+1$ local-commit messages from the nodes of its zone (in response to accepted messages), and (2) a prepared message from the source zone. The primary then constructs certificate $\mathcal{C}$ and multicasts commit message $\langle \text{COMMIT}, v(z_i), \langle n, z_i \rangle, v(z_j), \langle m, z_j \rangle, \mathcal{C}, \mathcal{C}_s, d \rangle_{\sigma_{\pi(z_i)}}$ to all nodes of every zone in the source and the destination cluster.

Upon receiving a valid commit message from the initiator primary, each node executes the request on the regional system meta-data. Finally, the client data is migrated to the destination zone using the data migration protocol (Algorithm 2).

Correctness. The safety and liveness of each zone is guaranteed by PBFT [13]. The safety and liveness of each zone cluster also follow the correctness arguments discussed in Section V-C. We now briefly discussed the correctness of communication across zone clusters.

Lemma 6.1: If node r commits cross-cluster transaction m, no other non-faulty node commits cross-cluster transaction m' ($m \neq m'$) with the same sequence number.

Proof: In cross-propose, prepared, and commit phases the message sent by the proxy nodes of the destination zone, the proxy nodes of the source zone or the primary of the destination zone includes a certificate consisting of $2f + 1$ signatures proving the validity of the message. As a result, since any two quorums of nodes within a zone intersect on at least one non-faulty node, with the same argument as Section V-C, safety is guaranteed.

Lemma 6.2: A cross-cluster request m issued by a correct client will eventually be complete if the majority of zones within each cluster and across clusters can still communicate.

Proof: In cross-cluster data synchronization protocol and to handle failures, nodes of the destination zone multicast response-query messages (with the same structure as discussed in Section V-A) to the source zone if they do not receive prepared messages.

Similarly, if commit messages are not received and the timer is expired, nodes of the source zone multicast response-query messages to the destination zone. Moreover, in the cross-cluster data synchronization protocol, any communications across clusters are performed by $f + 1$ (proxy) nodes of each zone to prevent a malicious primary from delaying the protocol by not sending messages. The failure handling routine follows similar steps as Section V-A.

VII. Experimental Evaluations

The goal of our evaluations is to measure the impact of (1) global transactions, (2) leader election, (3) number of zones, (4) zone size, (5) node failure, and (6) zone clusters in various scenarios on the performance of Ziziphus.

We implemented a prototype of Ziziphus using Golang and deployed a simple banking application on top of it where the client data is stored in a key-value store replicated on the nodes in each zone. Each client initiates local transactions to transfer money from its account to another client's account within the same zone. Local transactions are processed using PBFT [13]. If a client migrates to another zone, it initiates a migration request resulting in running the data synchronization protocol among all zones and the data migration protocol between the source and the destination zones.

In each set of experiments, we consider three different workloads with 10%, 30%, and 50% global transactions (i.e., client migrations). The workload with 10% global transaction (and 90% local transaction) is the typical setting in partitioned databases [35]. We consider 50% as the maximum percentage of global transactions because Ziziphus is designed to support edge networks where accesses to data have an affinity towards locality. We simulate the migration of clients (as implementing clients' physical migration across zones is not feasible). We run an instance of each client within all zones and when the client migrates from a source to a destination zone, it can initiate transaction within the destination zone.

We further compare Ziziphus with (1) a flat implementation of PBFT where for every transaction, PBFT runs among all nodes, (2) two-level PBFT (since PBFT is used in Ziziphus) where PBFT is used for both local and global transactions, and (3) Steward [2] that is similar to Ziziphus with 100% global transactions (i.e., every single transaction requires global synchronization across all zones). Note that while Ziziphus requires $Z * (3f + 1)$ nodes where Z is the number of zones, PBFT requires $3 * Zf + 1$ nodes (i.e., $Z - 1$ fewer nodes) where $Z * f$ is the total number of faulty nodes. Moreover, while Ziziphus and Steward require $2F + 1$ zones where F is the number of tolerated zone failures, two-level PBFT needs to be run among $3F + 1$ zones. This is because Ziziphus and Steward use a CFT protocol, i.e., Paxos, to process global transactions while two-level PBFT relies on PBFT at the global level that requires $3F + 1$ participants. On the other hand, Steward tolerates zone failure while Ziziphus and two-level PBFT are not.

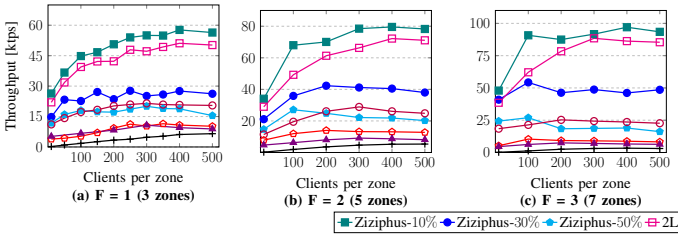

Figure 4. Throughput with increasing the number of zones

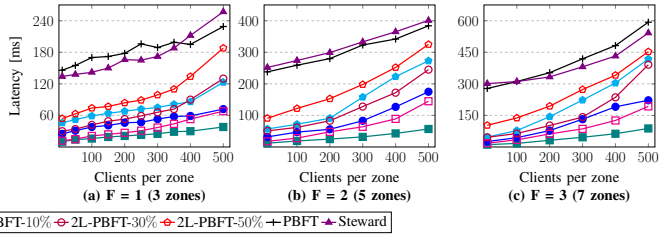

Figure 5. Latency with increasing the number of zones

In each experiment, we increase the number of clients per zone from 10 to 500 and measure the end-to-end throughput (x axis) and latency (y axis) of the system. The experiments were conducted on the Amazon EC2 platform. All instances are size c4.large (3.75GB memory) running Amazon Linux 2 AMI. Clients execute in a closed loop. The reported results are the average of five runs.

A. Performance with Multiple Zones

In the first set of experiments, we measure the performance of Ziziphus in three settings with different number of zones distributed over AWS regions: (1) three zones ($2F+1$ where $F=1$) placed in California (*CA*), Ohio (*OH*), and Quebec (*QC*), (2) five zones ($2F+1$ where $F=2$) placed in California (*CA*), Sydney (*SYD*), Paris (*PAR*), London (*LDN*), and Tokyo (*TY*), and (3) seven zones ($2F+1$ where $F=3$) placed in *CA*, *OH*, *QC*, *SYD*, *PAR*, *LDN*, and *TY* [1].

In the Ziziphus deployment, each zone consists of 4 nodes, i.e., $3f+1$ where $f=1$. In general, Ziziphus requires $Z*(3f+1)$ where Z is the number of zones while to tolerate the same number of failures, PBFT requires $Z-1$ fewer number of nodes. As a result, in each experiment, PBFT runs on 4 nodes in *CA* and 3 nodes in other data centers. Two-level PBFT require $3F+1$ zones to run global synchronization, i.e., 4, 7 and 10 zones in the above settings. To have a fair comparison, instead of adding real zones, we add additional nodes to the *CA* data center that participate the in global synchronization of the two-level PBFT protocol as zone leaders. However, they do not process any local transactions. For example, in the first setting, there are three zones in *CA*, *OH*, and *QC* and an additional single node that participates in global synchronization.

In this experiment, and for the data synchronization protocol of Ziziphus, we use the stable leader technique, where a stable primary node initiates all instances of the data synchronization protocol.

Figure 4 and Figure 5 demonstrate the results. As shown in Figure 4(a), with 3 zones and in the workload with 10% global transactions, Ziziphus is able to process more than 57 ktps with 30 ms latency when 400 clients send their requests concurrently. Two-level PBFT in the workload with 10% global transactions and with 400 concurrent clients, processes 51 ktps with 53 ms latency (i.e., 11% lower throughput and 76% higher latency). Two-level PBFT uses PBFT to process global transactions which is more costly than the global synchronization protocol of Ziziphus. In this workload, Steward processes 10 ktps with 212 ms latency because Steward runs the global synchronization protocol for every transaction (i.e., similar to Ziziphus with 100% global transactions). As can be seen, in all three workloads (i.e. 10%, 30%, and 50% global transactions), Ziziphus demonstrates better performance compared to PBFT because first, Ziziphus processes local transactions in parallel and second, the global transactions are processed using a cheaper protocol than PBFT.

Increasing the number of zones to 5 and 7, as shown in Figure 4(b) and (c), improves the overall throughput of Ziziphus. With 10% global transactions (the typical setting in partitioned databases [35]), and with 7 zones and 400 concurrent clients in each zone, Ziziphus processes 97 ktps with 63 ms latency (as shown in Figure 5). This clearly demonstrates the scalability of Ziziphus compared to PBFT; with 5 zones and 400 concurrent clients in each zone, Ziziphus processes 79.5 ktps with 43 ms latency while PBFT processes only 5.2 ktps (6.5% throughput of Ziziphus) with 342 ms latency (795% latency of Ziziphus) in the same setting. Ziziphus achieves this significant performance by processing local transactions of different zones in parallel and by using a cheap protocol to achieve global consensus among zones.

B. Performance with Node Failure

We repeat the first set of experiments under a single backup failure in each zone. In each setting and for each protocol, we report the throughput and latency only in the scenario (i.e., number of concurrent clients) where end-to-end throughput is saturated (peak performance). As shown in Figure 6, under a backup failure in each zone, Ziziphus with 10% global transactions, attains higher throughput and incurs lower latency than all other protocols in all networks. Faulty backups reduce the performance of flat PBFT more than other protocols. This is because without faulty backups, PBFT is able to construct its quorums with a subset of zones, e.g., with 3 zones, PBFT quorums require 7 out of 10 nodes that can be constructed using nodes of *CA* and *OH* data centers. However, with faulty backups, constructing quorums require the participation of all zones resulting in higher latency.

C. Fault-Tolerance Scalability

In the next set of experiments, we measure the performance of different protocols by increasing the number of tolerated faulty nodes within each zone, f, from 1 to 5, i.e., increasing the zone size from 4 to 16. The network includes 3 different zones, placed in *CA*, *OH*, and *QC* data centers, resulting in 12 to 48 nodes in total (for two-level PBFT, as before, we consider the fourth zone with a single node to participate in global synchronization as a backup). Note that in the PBFT protocol, the network size is between 10 and 46.

[1]The average Round-Trip Time (RTT) between every pair of Amazon data centers can be found at https://www.cloudping.co/grid

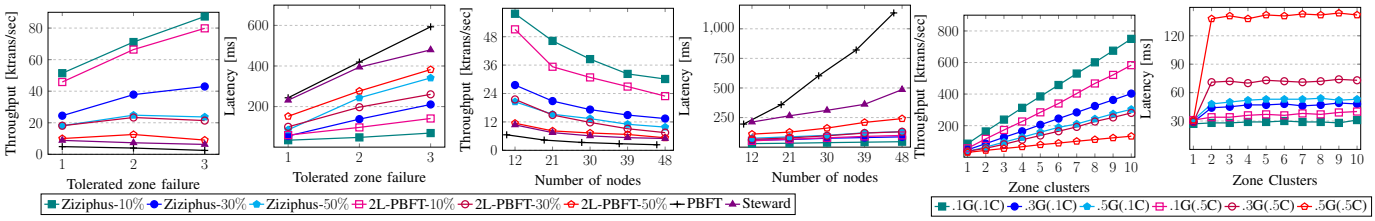

Figure 6. Different number of zones (with failure)

Figure 7. Different number of nodes per zone

Figure 8. Different number of zone clusters

As can be seen in Figure 7, increasing the number of nodes, reduces the overall throughput and increases the end-to-end latency of all protocols. This is expected because all protocols use PBFT to process local transactions and with larger zones, the performance of PBFT is reduced due to its high communication complexity. Increasing the zone size, however, has the lowest impact on the latency of Ziziphus; while Ziziphus demonstrates only 53% higher latency in workload with 10% global transaction by increasing the zone size from 4 to 16, the latency of flat PBFT increases 480%. This is because in the flat PBFT protocol, all nodes of all zones communicate with each other while the zone size does not affect the number of nodes participating in the global synchronization of Ziziphus.

D. Scalability using Zone Clusters

We next present the scalability of Ziziphus in settings with multiple zone clusters (presented in Section VI). We consider different scenarios with 1 to 10 zone clusters where each cluster includes 3 zones and each zone contains 4 nodes ($f = 1$), i.e., 120 nodes in total. Nodes of each cluster are within the same data center where zone clusters are placed in *CA*, *SYD*, *PAR*, *LDN* and *TY* data centers (at most 2 clusters in each). For each of the three different workloads (i.e., 10%, 30%, and 50% global transactions), we consider 10% or 50% **of** global transactions as cross-cluster transactions. This results in 6 different workloads, e.g., 10% Global, 50% Cross-cluster (.1G(.5C)). Since clustering has not been used in Steward and two-level PBFT, we only report results for Ziziphus.

As shown in Figure 8, Ziziphus demonstrates higher throughput by increasing the number of zone clusters. This is expected because all local and most global transactions are performed within zone clusters and only cross-cluster transactions (i.e., a small percentage of global transactions) require communication across only two zone clusters. Similarly, increasing the number of zone clusters does not affect latency (the latency increases from one to two clusters because there is no cross-cluster transaction in the first case). Ziziphus demonstrates its best performance in workload with 10% global transactions where 10% of global transactions are cross-cluster transactions (.1G(.1C)) and is able to process 749 ktps in 31 ms with 10 zone clusters.

VIII. Related Work

State Machine Replication (SMR) is a technique for implementing a fault-tolerant service by replicating servers [26]. Several approaches [32][27][31] generalize SMR to support crash failures among which Paxos [27] is the most well-known. Paxos guarantees safety in an asynchronous network using $2f+1$ processors despite the simultaneous crash failure of any f processors. DPaxos [29] is a variation of Paxos that is, similar to Ziziphus, designed for edge networks. DPaxos partitions nodes into different crash-only zones and utilizes Flexible Paxos [22] to make replication quorums small. DPaxos further allows the leader election quorum to start small and then grow to only intersect with replication quorums that are being used by other leaders. Ziziphus, in contrast to DPaxos, supports the Byzantine failure of nodes. While in both systems, global transactions are processed using a Paxos-like protocol, Ziziphus confines the maliciousness of Byzantine nodes within zones. Moreover, using zone clusters, Ziziphus can be scaled to thousands of zones over wide area networks.

Partitioning Byzantine nodes into local fault-tolerant clusters to improve scalability has been addressed in permissioned blockchain systems using either fully replicated ledgers, e.g., ResilientDB [20] and Blockplane [30], or Sharded-ledger approaches, e.g., AHL [15], Separ [6], Chainspace [1], Saguaro [7] and SharPer [4][5]. Ziziphus, in contrast to fully replicated ledgers, does not require global synchronization for every transaction and, in contrast to sharded-ledger approaches, processes global transactions using a crash fault-tolerant protocol.

IX. Conclusion

Processing client transactions by edge servers is challenging due to the non-trustworthiness of edge infrastructures and their communication latency over wide area networks. This paper presents Ziziphus, a geo-distributed system that partitions Byzantine edge servers into fault-tolerant zones where each zone processes transactions initiated by nearby clients locally. Ziziphus provides a zonal abstraction to confine maliciousness of Byzantine servers within each zone. Based on our experiments, in workloads with a low percentage of global transactions (typical settings), Ziziphus achieves significantly better performance compared to flat PBFT, two-level PBFT and Steward. Similarly, the performance of Ziziphus improves semi-linearly with increasing the number of zones or zone clusters.

Acknowledgements

We thank the anonymous reviewers for their insightful feedback and suggestions. This work is funded by NSF grants CNS-1703560, and CNS-2104882. Sujaya Maiyya was partially funded by IBM PhD Fellowship.

REFERENCES

[1] M. Al-Bassam, A. Sonnino, S. Bano, D. Hrycyszyn, and G. Danezis. Chainspace: A sharded smart contracts platform. In *Network and Distributed System Security Symposium (NDSS)*, 2018.

[2] Y. Amir, C. Danilov, D. Dolev, J. Kirsch, J. Lane, C. Nita-Rotaru, J. Olsen, and D. Zage. Steward: Scaling byzantine fault-tolerant replication to wide area networks. *IEEE Transactions on Dependable and Secure Computing*, 7(1):80–93, 2008.

[3] M. J. Amiri, D. Agrawal, and A. El Abbadi. Caper: a cross-application permissioned blockchain. *Proc. of the VLDB Endowment*, 12(11):1385–1398, 2019.

[4] M. J. Amiri, D. Agrawal, and A. El Abbadi. On sharding permissioned blockchains. In *Int. Conf. on Blockchain*, pages 282–285. IEEE, 2019.

[5] M. J. Amiri, D. Agrawal, and A. El Abbadi. Sharper: Sharding permissioned blockchains over network clusters. In *SIGMOD Int. Conf. on Management of Data*, pages 76–88. ACM, 2021.

[6] M. J. Amiri, J. Duguépéroux, T. Allard, D. Agrawal, and A. El Abbadi. Separ: Towards regulating future of work multi-platform crowdworking environments with privacy guarantees. In *Proceedings of The Web Conf. (WWW)*, pages 1891–1903, 2021.

[7] M. J. Amiri, Z. Lai, L. Patel, B. T. Loo, E. Lo, and W. Zhou. Saguaro: An edge computing-enabled hierarchical permissioned blockchain. In *Int. Conf. on Data Engineering (ICDE)*. IEEE, 2023.

[8] M. J. Amiri, B. T. Loo, D. Agrawal, and A. El Abbadi. Qanaat: A scalable multi-enterprise permissioned blockchain system with confidentiality guarantees. *Proc. of the VLDB Endowment*, 15(11):2839–2852, 2022.

[9] V. Arora, M. J. Amiri, D. Agrawal, and A. El Abbadi. M-db: A continuous data processing and monitoring framework for iot applications. In *Int. Conf. on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, pages 1096–1105. IEEE, 2019.

[10] G. Bracha and S. Toueg. Asynchronous consensus and broadcast protocols. *Journal of the ACM (JACM)*, 32(4):824–840, 1985.

[11] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. Li, et al. Tao: Facebook's distributed data store for the social graph. In *Annual Technical Conf. (ATC)*, pages 49–60. USENIX Association, 2013.

[12] C. Cachin, K. Kursawe, and V. Shoup. Random oracles in constantinople: Practical asynchronous byzantine agreement using cryptography. *Journal of Cryptology*, 18(3):219–246, 2005.

[13] M. Castro, B. Liskov, et al. Practical byzantine fault tolerance. In *Symposium on Operating systems design and implementation (OSDI)*, volume 99, pages 173–186. USENIX Association, 1999.

[14] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, et al. Spanner: Google's globally distributed database. *Transactions on Computer Systems (TOCS)*, 31(3):8, 2013.

[15] H. Dang, T. T. A. Dinh, D. Loghin, E.-C. Chang, Q. Lin, and B. C. Ooi. Towards scaling blockchain systems via sharding. In *SIGMOD Int. Conf. on Management of Data*. ACM, 2019.

[16] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. In *Operating Systems Review (OSR)*, volume 41, pages 205–220. ACM SIGOPS, 2007.

[17] A. El Abbadi, D. Skeen, and F. Cristian. An efficient, fault-tolerant protocol for replicated data management. In *SIGACT-SIGMOD symposium on Principles of database systems*, pages 215–229. ACM, 1985.

[18] A. El Abbadi and S. Toueg. Availability in partitioned replicated databases. In *SIGACT-SIGMOD symposium on Principles of database systems*, pages 240–251. ACM, 1985.

[19] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382, 1985.

[20] S. Gupta, S. Rahnama, J. Hellings, and M. Sadoghi. Resilientdb: Global scale resilient blockchain fabric. *Proceedings of the VLDB Endowment*, 13(6):868–883, 2020.

[21] N. Hassan, S. Gillani, E. Ahmed, I. Yaqoob, and M. Imran. The role of edge computing in internet of things. *IEEE communications magazine*, 56(11):110–115, 2018.

[22] H. Howard, D. Malkhi, and A. Spiegelman. Flexible paxos: Quorum intersection revisited. In *20th International Conference on Principles of Distributed Systems*, 2017.

[23] M. Hu, Z. Xie, D. Wu, Y. Zhou, X. Chen, and L. Xiao. Heterogeneous edge offloading with incomplete information: A minority game approach. *IEEE Transactions on Parallel and Distributed Systems*, 31(9):2139–2154, 2020.

[24] R. Kapitza, J. Behl, C. Cachin, T. Distler, S. Kuhnle, S. V. Mohammadi, W. Schröder-Preikschat, and K. Stengel. Cheapbft: resource-efficient byzantine fault tolerance. In *European Conf. on Computer Systems (EuroSys)*, pages 295–308. ACM, 2012.

[25] E. Kokoris-Kogias, P. Jovanovic, L. Gasser, N. Gailly, E. Syta, and B. Ford. Omniledger: A secure, scale-out, decentralized ledger via sharding. In *Symposium on Security and Privacy (SP)*, pages 583–598. IEEE, 2018.

[26] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[27] L. Lamport. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[28] P. Mach and Z. Becvar. Mobile edge computing: A survey on architecture and computation offloading. *IEEE Communications Surveys & Tutorials*, 19(3):1628–1656, 2017.

[29] F. Nawab, D. Agrawal, and A. El Abbadi. Dpaxos: Managing data closer to users for low-latency and mobile applications. In *Proceedings of the 2018 International Conference on Management of Data*, pages 1221–1236. ACM, 2018.

[30] F. Nawab and M. Sadoghi. Blockplane: A global-scale byzantizing middleware. In *2019 IEEE 35th Int. Conf. on Data Engineering (ICDE)*, pages 124–135. IEEE, 2019.

[31] D. Ongaro and J. K. Ousterhout. In search of an understandable consensus algorithm. In *Annual Technical Conf. (ATC)*, pages 305–319. USENIX Association, 2014.

[32] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *Computing Surveys (CSUR)*, 22(4):299–319, 1990.

[33] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu. Edge computing: Vision and challenges. *IEEE internet of things journal*, 3(5):637–646, 2016.

[34] V. Shoup. Practical threshold signatures. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 207–220. Springer, 2000.

[35] R. Taft, E. Mansour, M. Serafini, J. Duggan, A. J. Elmore, A. Aboulnaga, A. Pavlo, and M. Stonebraker. E-store: Fine-grained elastic partitioning for distributed transaction processing systems. *Proc. of the VLDB Endowment*, 8(3):245–256, 2014.

[36] M. Zhang, C. Krintz, and R. Wolski. Sparta: A heat-budget-based scheduling framework on iot edge systems. In *International Conference on Edge Computing*, 2021.