Ziziphus:
Scalable Data Management Across Byzantine Edge Severs

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

Abstract

Edge computing while bringing computation and data closer to users in order to improve response time, distributes edge servers in a wide area network resulting in increased communication latency between the servers. Synchronizing globally distributed edge servers, especially in the presence of Byzantine servers, becomes very costly due to the high communication complexity of Byzantine fault-tolerant consensus protocols. While geo-replicated systems try to confine the malicious behavior of nodes within local datacenters, replicating the entire data across all datacenters requires global synchronization for every single user transaction resulting in high latency. In this paper, we propose *Ziziphus*, a geo-distributed system for mobile users 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., migration of clients from one zone to another. The two-level architecture of Ziziphus, on one hand, confines the malicious behavior of nodes within zones requiring a much cheaper protocol at the top level for global synchronization and, on the other hand, processes most user transactions within zones by edge servers closer to users 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. The experimental results reveal the efficiency of Ziziphus in terms of both performance and scalability (i.e., an increasing number of nodes and zones) especially for workloads requiring a low percentage of global synchronization.

1 Introduction

Edge computing shifts computation and data closer to users [50]. While this computational paradigm improves response time and saves bandwidth, the edge servers may be distributed in the wide area network far away from each other resulting in high communication latency among servers. Edge servers need to communicate with each other in order to establish agreement on the order of user transactions using fault-tolerant protocols.

While large-scale data management systems, such as Google's Spanner [20], Amazon's Dynamo [24], and Facebook's Tao [15], rely on crash fault-tolerant (CFT) protocols, e.g., Paxos [39], establishing consensus among edge servers requires Byzantine fault-tolerant (BFT) protocols, e.g., PBFT [17] due to the non-trustworthiness of edge infrastructures. Byzantine fault-tolerant protocols have also been used in recent large-scale data management systems– permissioned blockchains [37][18][10].

Byzantine fault-tolerant protocols, however, suffer from their high complexity, especially in wide-area networks. For instance, PBFT [17], the most known BFT protocol, requires $3f + 1$ servers (where f is the number of simultaneous malicious servers), three communication phases, and quadratic message complexity in terms of the number of servers, which make it impractical to establish 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.

Steward [4] is designed for large-scale fully replicated systems over multiple wide area sites, where each site consists of several servers and data is fully replicated across all servers. Steward uses a hierarchical approach where the maliciousness of Byzantine servers is confined within a site and a crash fault-tolerant protocol is used to establish global consensus among sites. Thus, every single user transaction needs to be globally synchronized to ensure that all copies of data remain mutually consistent with each other. Blockplane [46] uses a similar model and runs global synchronization for every user transaction in order to tolerate the failure of sites, i.e., datcenters, in case of datacenter-scale outages. The full data replication technique has also been used by the permissioned blockchain system, Resilientdb [31], where at every round, each site locally establishes consensus on a single transaction and multicasts the locally-replicated transaction to other sites. All sites then, execute all transactions of that round in a predetermined order.

Sharded distributed systems, on the other hand, shard data and replicate a shard of data on each site. A transaction that accesses a single shard is processed by the nodes of the specific site, while a global cross-shard transaction is executed on *all* shards. Since nodes may be malicious, sites need to use BFT protocols to establish consensus on their local transactions. In sharded distributed systems, in contrast to Geo-replicated systems, there is no need to run global consensus for every single transaction. However, global synchronization in such systems still uses a BFT protocol resulting in high latency. Caper [5], a permissioned blockchain system, uses this approach to process local and global transactions among a set of enterprises (i.e., sites that maintain shards of data).

Ziziphus combines the best of both worlds and targets large-scale geo-distributed systems serving mobile clients. 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. Each zone replicates the data of its nearby clients on its nodes and processes local transactions initiated by those clients independent of other zones. Thus, the global synchronization requirements in Ziziphus are considerably reduced in comparison to geo-replicated systems. Moreover, at the global level, Ziziphus not only tolerate f failures within each zone but also tolerates $\lfloor \frac{Z-1}{2} \rfloor$ entire zone failures out of Z zones (which

might fail due to natural disasters). Ziziphus further is able to tolerate zone failures for local transactions using *lazy synchronization* where local updates are propagated to other zones only when global synchronization is needed.

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 to reduce the load on the entire network. Ziziphus maintains global system meta-data including the number of clients of each zone, the history of client migrations, etc. on all nodes of every zone. Global synchronization is needed only 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 global consensus protocol among all zones where, in contrast to sharded distributed systems, the protocol requires linear communication and needs 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.

As the system scales, the number of zones might increase to hundreds or even thousands of zones over wide area networks. Running global synchronization 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. Zones within a zone cluster maintain the same (regional) system meta-data (instead of global meta-data) which is different from the system meta-data maintained by other zone clusters. Ziziphus presents a *cross-cluster data synchronization* protocol to deal with migration cases where source and destinations 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 from one zone to another;
- A data synchronization protocol to globally synchronize zones and a data migration protocol to migrate the client data from a source to a destination zone when migration occurs; 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. A cross-cluster data synchronization protocol is presented to address migration cases where source and destinations zones are in two different zone clusters.

The rest of this paper is organized as follows. The Ziziphus model is introduced in Section 2. Section 3 presents transaction processing in Ziziphus. Scalability of Ziziphus is discussed in Section 4. Section 5 presents a performance evaluation of Ziziphus. Section 6 discusses related work, and Section 7 concludes the paper.

2 System Model

Ziziphus is an asynchronous geo-distributed system designed to process transactions initiated by mobile edge clients on their own data. In Ziziphus, nodes are partitioned into fault-tolerant zones where each zone processes transactions initiated by nearby clients. Clients are expected to initiate transactions on their own local data

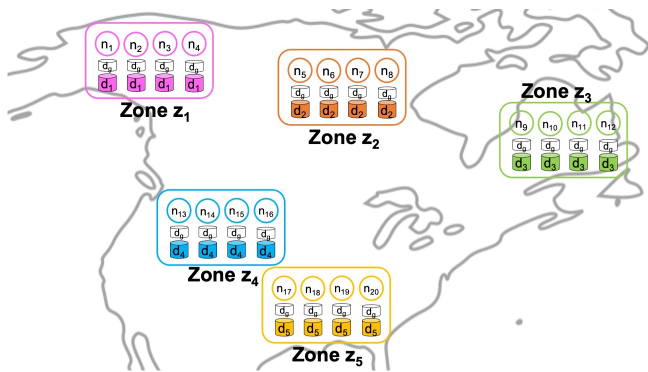


Figure 1: Ziziphus network consisting of five zones

while residing in a zone and migrating to another zone after some period of time. In this section, we present the network infrastructure of Ziziphus followed by its data and transaction model.

2.1 Network Infrastructure

The underlying infrastructure consists of a set of nodes in a large-scale asynchronous distributed system. Nodes follow the Byzantine failure model, i.e., faulty nodes may exhibit arbitrary, potentially malicious behavior. 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 [13]. 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. Figure 1 presents a Ziziphus network with five different zones z_1 to z_5 where each zone hosts four nodes, i.e., $3f + 1$ nodes where $f = 1$.

Nodes are connected by point-to-point bi-directional communication channels. Network channels are pairwise authenticated, which guarantees that a malicious node cannot forge a message from a correct node. Furthermore, messages contain public-key signatures and message digests [17]. A *message digest* is a numeric representation of the contents of a message produced by collision-resistant hash functions. We denote a message m signed by node r as $\langle m \rangle_{\sigma_r}$ and the digest of a message m by $D(m)$. Message digests are used to detect changes and alterations to any part of the message. All nodes have access to the public keys of all other nodes and are able to verify their signatures.

2.2 Data and Transactions

Each zone processes the transactions of its nearby clients and maintains their data. 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 the number of clients of each zone, history of client migration, etc. Global system meta-data is 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 to reduce the load on the entire network. Global system meta-data is globally replicated on every node of every zone.

In Figure 1, d_i represents the local data of zone z_i that is replicated on all nodes of zone z_i. For example, d_1 is replicated on nodes n_1,

n_2, n_3 and n_4. In contrast, global system meta-data d_g is replicated on every node of all 5 zones, i.e., n_1 to n_{20}.

Ziziphus supports two types of *local* and *global* transactions. Local transactions 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. The global system meta-data might also need to be updated. In particular, clients of an edge network are mobile and might migrate from a *source* zone to a *destination* zone. When a client migrates, the client initiates a *global transaction*. A global transaction resulting from a client migration consists of two atomic sub-transactions. During the first sub-transaction, all zones establish agreement on the global transaction, i.e., client migration, to verify that network-wide policies are not violated and the global transaction is executed on the global system meta-data.

The second sub-transaction is initiated by migrating the client data from the source zone to the destination zone enabling the destination zone to process client transactions. In the second sub-transaction, only the source and the destination zones are involved. It should be noted that when a client migrates from a source to a destination zone and starts initiating transactions in the destination zone, its data in the source zone become out-of-date and the destination zone now maintains its most up-to-date data. In fact, while migrations result in maintaining the data of the same client on multiple zones, only one zone has its most up-to-date data at any moment.

When the global system meta-data needs to be updated without any user migration, e.g., new clients are added to a zone, the global transaction does not include the second sub-transaction, i.e., all zones establish agreement on the global transaction to ensure that network-wide policies are not violated and then the transaction is executed on the global system meta-data. In this paper, we focus on the client migration case as the more complex case.

3 Transaction Processing in Ziziphus

Processing transactions requires establishing consensus on a unique ordering of incoming requests. To establish consensus, asynchronous fault-tolerant protocols can be used. Fault-tolerant protocols need to satisfy (1) *safety*: all correct nodes receive the same requests in the same order, and (2) *liveness*: all correct client requests are eventually ordered. Fischer et al. [27] show that in an asynchronous system, where nodes can fail, consensus has no solution that is both safe and live. Based on that impossibility result, in most fault-tolerant protocols, safety is satisfied without any synchrony assumption, however, a synchrony assumption is considered to ensure liveness.

3.1 Local Transactions

Clients initiate local transactions on their data that is replicated on nodes of the nearby zone. A local transaction is processed by nodes of a zone without any communication with other zones. This is in contrast to the multi-cluster globally distributed systems, Steward [4] and Blockplane [46], where for every single transaction, communication across all zones is needed. Since nodes may fail in a Byzantine manner, all local transactions in Ziziphus are processed using the known Byzantine fault-tolerant protocol PBFT [35]. In PBFT, nodes move through a succession of configurations called *views* [25][26]. In a view, one node is *the primary* and the others are *backups* where the primary node initiates consensus among nodes. PBFT consists of *agreement* and *view change* routines where the agreement routine orders requests for execution by the nodes, and the view change routine coordinates the election of a new primary (within the zone) when the current primary is faulty.

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 within the zone. Once a node receives a valid pre-prepare message from the primary, it broadcasts a prepare message to all nodes within the zone. Upon collecting $2f$ valid matching prepare messages (including its own message) that also match the pre-prepare message received from the primary, each node broadcasts a commit message. The pre-prepare and prepare phases of the protocol guarantee that non-faulty nodes agree on a total order for the requests within a view. Once a node receives $2f + 1$ valid matching commit messages (including its own message), it commits the request, executes the request (if it has executed all requests with lower sequence number) and sends the execution results back to the client. Finally, the client waits for $f + 1$ valid matching responses from different nodes to make sure at least one non-faulty node has executed its request. PBFT also has a view change routine that provides liveness by allowing the system to make progress when the primary fails.

While Ziziphus processes local transactions using PBFT in its current design, the local consensus protocol of Ziziphus is pluggable and any BFT protocols can be used to process local transactions.

3.2 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 zone to a destination zone, hence, we describe global transactions for this case. In the client migration use-case, as discussed earlier, the global transaction has two atomic sub-transactions where the first sub-transaction updates the global system meta-data of all zones, and the second sub-transaction copies the actual client data from the source to the destination zone. In this section, we present the *data synchronization* protocol to update global system meta-data, i.e., first sub-transaction, and the *data migration* protocol to copy the client data from the source to the destination zone, i.e, second sub-transaction.

3.2.1 *Data Synchronization Protocol*

In the data synchronization protocol, agreement from a majority of zones is needed to establish consensus on the order of a global transaction among all global transactions. While local transactions are processed using PBFT that has a quadratic communication complexity and requires at least two-thirds of non-faulty nodes to participate in each communication phase, the two-level architecture of Ziziphus confines the effect of all malicious behavior of Byzantine nodes within zones. As a result, in the *data synchronization* protocol, neither a quadratic communication complexity nor a minimum of two-thirds non-faulty participants is required.

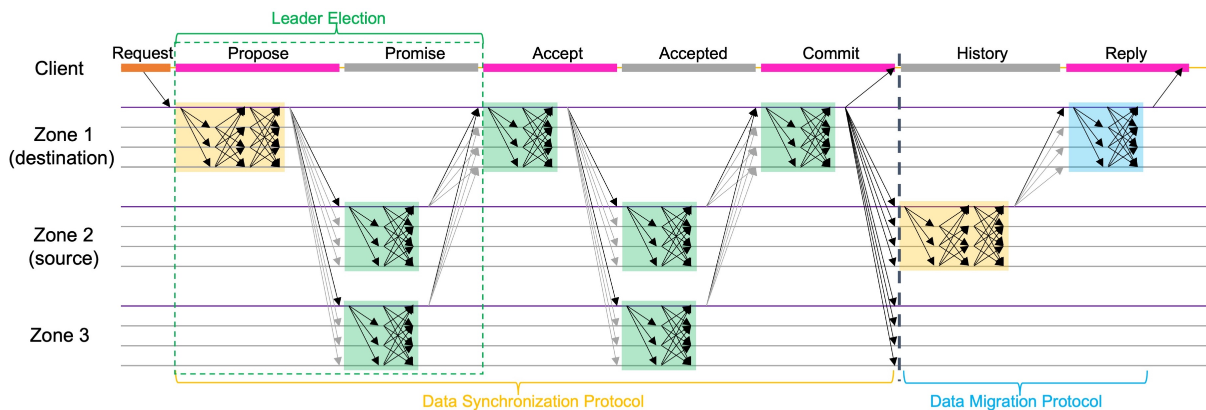


Figure 2: Processing global transactions consisting of data synchronization protocol and data migration protocol

The data synchronization protocol is a two-level protocol where at the top level only the primary nodes of all zones communicate with each other to globally agree on the order of the global transaction. When a primary node communicates with the primary nodes of other zones, its messages need to be endorsed by the nodes in its zone. As a result, 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 [51][16].

As shown in Figure 2 the data synchronization protocol at the top level, similar to Paxos, proceeds through propose, promise, accept, accepted, and commit phases where the goal of the propose and promise phases is to elect the primary node (i.e., leader), 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. At the bottom level and within each zone, nodes communicate with each other to *endorse* the message that will be sent by the primary node at the top level.

When a 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. Upon receiving a migration request message, as shown in Figure 2, a 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 order of 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 2, 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 node of the zone, nodes multicast commit messages to each other. The primary node of each zone then collects $2f+1$ messages from different nodes in its zone to construct a certificate that is used in the top-level communication with the primary node of other zones.

Algorithm 1 presents the normal case operation of the data synchronization protocol to process the first sub-transaction of a global transaction. Although not explicitly mentioned, every sent and received message is logged by the nodes. As indicated in lines 1-6 of the algorithm, 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 where views are numbered consecutively, and $\pi(z)$ is the primary node 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. As shown in lines 7-9, upon receiving a valid signed migration request $m = \langle \text{MIG-REQUEST}, op, ts_c, c \rangle_{\sigma_c}$ from an authorized client c (with timestamp t_c) to execute a transaction op on global system meta-data, 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. Timestamp ts_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. A client sends a request with timestamp $i+1$ only after it receives a reply for a request with timestamp i. op is a simple operation that updates global system meta-data based on the pre-defined policies once executed, e.g., updates the number of clients in the source and the destination zones.

The primary then multicasts $\langle \text{PRE-PREPARE}, v(z_i), \langle n, z_i \rangle, d, m \rangle_{\sigma_{\pi(z_i)}}$ message to *all* nodes of its zone z_i where $v(z_i)$ is the view of node $\pi(z_i)$ in zone z_i, m is the client's migration request message and $d = D(m)$ is digest of m. Upon receiving a pre-prepare message, as indicated in lines 10-12, 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

Algorithm 1 Endorsement Protocol

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

■ Endorsement in the initiator zone (PROPOSE phase) ■

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

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

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

■ Endorsement in a follower zone z_f (PROMISE phase) ■

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

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

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

■ Endorsement in the initiator zone (ACCEPT phase) ■

▷ $r = \pi(z_i)$:
26: **upon receiving** valid q=$\langle \text{PROMISE}, v(z_f), \langle n, z_i\rangle, \langle l, z_l\rangle, C_f, d\rangle_{\sigma_{\pi(z_f)}}$ from Q_M
27: **multicast** $\langle \text{PRE-PREPARE}, v(z_i), \langle n, z_i\rangle, \langle l, z_l\rangle, q_1, q_2,.., d_{q_1}, d_{q_2},.., d\rangle_{\sigma_{\pi(z_i)}}$ to z_i

▷ $r \in z_i$:
28: **upon receiving** valid $\langle \text{PRE-PREPARE}, v(z_i), \langle n, z_i\rangle, \langle l, z_l\rangle, q_1, q_2,.., d_{q_1}, d_{q_2},.., d\rangle_{\sigma_{\pi(z_i)}}$
29: **multicast** $\langle \text{LOCAL-ACCEPT}, v(z_i), \langle n, z_i\rangle, \langle l, z_l\rangle, d, r\rangle_{\sigma_r}$ to z_i

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

■ Endorsement in a follower zone z_f (ACCEPTED phase) ■

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

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

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

■ Endorsement in the initiator zone (COMMIT phase) ■

▷ $r = \pi(z_i)$:
39: **upon receiving** valid a=$\langle \text{ACCEPTED}, v(z_f), \langle n, z_i\rangle, \langle l, z_l\rangle, C_f, d\rangle_{\sigma_{\pi(z_f)}}$ from Q_M
40: **multicast** $\langle \text{PRE-PREPARE}, v(z_i), \langle n, z_i\rangle, \langle l, z_l\rangle, a_1, a_2, .., d_{a_1}, d_{a_2}, .., d\rangle_{\sigma_{\pi(z_i)}}$ to z_i

▷ $r \in z_i$:
41: **upon receiving** valid $\langle \text{PRE-PREPARE}, v(z_i), \langle n, z_i\rangle, \langle l, z_l\rangle, a_1, a_2, .., d_{a_1}, d_{a_2}, .., d\rangle_{\sigma_{\pi(z_i)}}$
42: **multicast** $\langle \text{LOCAL-COMMIT}, v(z_i), \langle n, z_i\rangle, \langle l, z_l\rangle, d, r\rangle_{\sigma_r}$ to z_i

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

■ Updating Global Meta-data (EXECUTION phase) ■

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

within a predefined small range to prevent a malicious primary from exhausting the space of sequence numbers by choosing a very large value [17]. Node r then multicasts a prepare message $\langle \text{PREPARE}, v(z_i), \langle n, z_i\rangle, d, r\rangle_{\sigma_r}$ to all nodes of the initiator zone z_i. When a node r logs matching prepare messages from a quorum of $2f$ different nodes that match the pre-prepare message received from the primary, as shown in lines 13-14, 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. Upon receiving a quorum of $2f + 1$ local-propose messages (lines 15-16), the primary aggregates these messages to construct a certificate C. The primary then multicasts a propose message $\langle \text{PROPOSE}, v(z_i), \langle n, z_i\rangle, C, d, m\rangle_{\sigma_{\pi(z_i)}}$ 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 17-20, it first checks the request, the message and the certificate 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 a pre-prepare message $\langle \text{PRE-PREPARE}, v(z_f), \langle n, z_i\rangle, \langle l, z_l\rangle, p, d_p, d\rangle_{\sigma_{\pi(z_f)}}$ to the nodes of its zone z_f where 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 (and either committed or not) by the follower zone z_f. Ballot number $\langle l, z_l\rangle$ is used to determine the order of executing 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 21-23), each node r in z_f checks the request, the message, the certificate 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 $\langle \text{LOCAL-PROMISE}, v(z_f), \langle n, z_i, \rangle, \langle l, z_l\rangle, d, r\rangle_{\sigma_r}$

to all nodes of zone z_f. Upon receiving $2f + 1$ valid matching local-promise messages from different nodes (lines 24-25), the primary node of each follower zone z_f aggregates these messages to construct a certificate C_f and then multicasts a promise message $\langle \text{PROMISE}, v(z_f), \langle n, z_i \rangle, \langle l, z_l \rangle, C_f, d \rangle_{\sigma_{\pi(z_f)}}$ to the nodes of the initiator zone z_i.

Accept phase. The global primary $\pi(z_i)$ waits for promise messages from one-half of zones to ensure that the majority of zones (including itself) agree with the proposed request. This follows the safety argument of Paxos [39] where to become the leader promise messages from a majority of acceptors is needed. In Ziziphus, since each message at the top-level needs to be endorsed with a quorum of $2f + 1$ signatures, nodes can not behave maliciously and the failure model of nodes is reduced to crash failure model. As a result, the safety condition of crash fault-tolerant protocols like Paxos is sufficient to guarantee the safety of the data synchronization protocol.

Upon receiving sufficient valid promise messages from different follower zones (lines 26-27), the global primary multicasts a pre-prepare message $\langle \text{PRE-PREPARE}, v(z_i), \langle n, z_i \rangle, \langle l, z_l \rangle, q_1, q_2, .., d_{q_1}, d_{q_2}, .., d \rangle_{\sigma_{\pi(z_i)}}$ 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 28-29). Upon receiving a quorum of $2f + 1$ local-accept messages (lines 30-31), the global primary aggregates these messages, constructs a certificate C, and multicasts an accept message $\langle \text{ACCEPT}, v(z_i), \langle n, z_i \rangle, \langle l, z_l \rangle, C, d \rangle_{\sigma_{\pi(z_i)}}$ to all nodes of every zone.

Accepted phase. Upon receiving a valid accept message, the primary node of a follower zone z_f, as shown in lines 32-34, checks the message and the certificate 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 $\langle \text{PRE-PREPARE}, v(z_f), \langle n, z_i \rangle, \langle l, z_l \rangle, a, d_a, d \rangle_{\sigma_{\pi(z_f)}}$ to all nodes of its zone, z_f, where a is the received accept message and d_a is its digest. Nodes of z_f validate the received pre-prepare message and multicast a local-accepted to the primary node of z_f (lines 35-36). The primary of z_f waits for a quorum of $2f + 1$ local-accepted messages from different nodes (lines 37-38), constructs a certificate C_f, and multicasts $\langle \text{ACCEPTED}, v(z_f), \langle n, z_i \rangle, \langle l, z_l \rangle, C_f, d \rangle_{\sigma_{\pi(z_f)}}$ 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 39-40) to multicast a $\langle \text{PRE-PREPARE}, v(z_i), \langle n, z_i \rangle,, \langle l, z_l \rangle, a_1, a_2, .., d_{a_1}, d_{a_2}, .., d \rangle_{\sigma_{\pi(z_i)}}$ message to all the nodes of zone z_i where 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 41-42). Upon receiving a quorum of $2f + 1$ local-commit messages (lines 43-44), the global primary aggregates these messages, constructs a certificate C, and multicasts a commit message $\langle \text{COMMIT}, v(z_i), \langle n, z_i \rangle, \langle l, z_l \rangle, C, d \rangle_{\sigma_{\pi(z_i)}}$ to all nodes of every zone.

Execution phase. Once a node in any zones receives a valid commit message from the global primary, 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 client's migrations in some period.

Nodes of the initiator zone also send a reply message including the execution results to the client informing the client that the first sub-transaction has been committed. The client waits for $f + 1$ valid matching responses from different nodes within the initiator zone to ensure that at least one correct node executed its request. If the client does not receive reply messages soon enough, it multicasts the request to all nodes within the initiator zone. If the request has already been processed, the nodes simply re-send the reply message to the client (nodes remember the last reply message they sent to each client). Otherwise, if the node is not the primary, it relays the request to the primary. If the primary does not multicast the request to the nodes, it will eventually be suspected to be faulty by nodes to cause a primary failure handling routine.

3.2.2 *Data Synchronization Protocol with Stable Primary Node*

In the proposed data synchronization protocol and in order to process a global transaction, the primary node of the destination zone becomes the global primary and initiates consensus on the order of the transaction among all zones. In this way, for every global transaction, a new node becomes the primary (leader) and, similar to Paxos, the propose and promise phases of the data synchronization protocol are needed for leader election.

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 and the primary of the initiator zone after establishing consensus on the request in its zone, multicasts an accept message to all other zones. If the primary fails, another node from the same zone becomes the primary (Section 3.3) and processes global transactions.

3.2.3 *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. The second sub-transaction is initiated by the primary node of the source zone, i.e., the zone that the client has migrated from. The primary node needs to collect the local data records, i.e., *history*, of the client, establish consensus on the client history within its zone to construct a certificate including $2f + 1$ signatures and multicast the history to the destination zone. The destination zone validates the message and appends the history to its database. At this point, the destination zone can process the incoming requests of the client.

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

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-HISTORY, v(z_s), ⟨n, z_i⟩, d_c, d, r⟩_σr to z_s
    ▷ r = π(z_s):
16: upon receiving matching ⟨LOCAL-HISTORY, v(z_s), ⟨n, z_i⟩, d_c, d, r⟩_σr from Q_zs
17:   multicast ⟨HISTORY, 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=⟨HISTORY, 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), ts_c⟩_σr to client c
```

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 client history $R(c)$. The client history includes all transaction records of client c. The node then initiates consensus on $R(c)$ by multicasting a pre-prepare message $\langle \text{PRE-PREPARE}, v(z_s), \langle n, z_i \rangle, R(c), d_c, d \rangle_{\sigma_{\pi(z_s)}}$ to the nodes of zone z_s where d_c is the digest of $R(c)$ and d is the digest of the client request. Upon receiving a valid pre-prepare message including a client record $R(c)$ from the primary node (lines 12-13), each node r of the zone z_s multicasts a prepare message $\langle \text{PREPARE}, v(z_s), \langle n, z_i \rangle, d_c, d, r \rangle_{\sigma_r}$ to all nodes in zone z_s. Each node waits for a quorum of $2f + 1$ valid prepare messages and multicasts a local-history message $\langle \text{LOCAL-HISTORY}, v(z_s), \langle n, z_i \rangle, d_c, d, r \rangle_{\sigma_r}$ to all nodes in zone z_s (lines 14-15). The primary node $\pi(z_s)$ collects $2f + 1$ local-history messages from different nodes (lines 16-17), constructs a certificate C, and multicasts $\langle \text{HISTORY}, v(z_s), \langle n, z_i \rangle, C, R(c), d_c, d \rangle_{\sigma_{\pi(z_s)}}$ message to the nodes of the destination zone z_d.

Record Appending. Upon receiving a valid history message, the primary node of the destination zone z_d, as shown in lines 18-19, checks the message and certificate C to be valid and multicasts pre-prepare message $\langle \text{PRE-PREPARE}, v(z_d), \langle n, z_i \rangle, h, d_h \rangle_{\sigma_{\pi(z_d)}}$ to nodes of its zone where h is the received history message and d_h is its digest. 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, appends the client record $R(c)$ to its database and sends a reply to the client informing that the migration has been performed successfully. Similar to the first sub-transaction, The client waits for $f + 1$ valid matching responses from different nodes within the destination zone to ensure that at least one correct node executed its request and if it does not receive reply messages, it multicasts the request to all nodes within the destination zone.

3.3 Primary Failure Handling

The primary failure handling routine improves liveness by allowing the system to make progress when a primary node fails. In 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.

For global transactions, however, detecting and handling failures is more difficult. While the data synchronization protocol follows crash-fault tolerant protocols, i.e., it requires linear communication phases and majority quorums, the participants in the global consensus are (Byzantine) primary nodes of different zones. Although a primary node can not behave maliciously since any messages need to be endorsed by $2f + 1$ nodes of a zone, 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.

To handle failures, 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 responses-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 responses-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 responses-query messages. If a node receives responses-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 zones. The data migration protocol handles failure in the same way for history 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 resulting in triggering the primary failure handling routine.

3.4 Fault Tolerance and Availability

Ziziphus processes global transactions with agreement from only a majority of zones. This means that at the global level, Ziziphus is able to tolerate $\lfloor \frac{Z-1}{2} \rfloor$ failures out of Z zones. Other than a malicious primary, a zone might fail due to natural disasters like tornadoes or earthquakes. Note that, using the stable leader technique, if the stable initiator zone fails, Ziziphus can rely on some other zone to initiate all global transactions.

Consider a simplified scenario where faulty nodes are uniformly distributed in three different zones and each zone includes $3f+1$. To process a global transaction, Ziziphus requires agreement from the majority of zones, i.e. two zones, and within each zone, the message needs to be endorsed by $2f + 1$ nodes. As a result, in this scenario, Ziziphus can process a global transaction with $4f + 2$ nodes out of $9f + 3$ nodes. Processing a transaction using PBFT among these nodes in a flat manner assuming $F = 3f$, however, requires $6f + 1$ nodes out of $9f + 1$ nodes.

This clearly demonstrates the advantage of first, clustering the noses into zones in order to confine the maliciousness of Byzantine nodes within their zones and second, being aware of where the faulty nodes are and uniformly distribute nodes in such a way that each zone includes exactly $3f + 1$ nodes.

Ziziphus, however, can tolerate the failure of zones only for global transactions because the results of local transactions are replicated only on the nodes of a single zone and if a zone fails due to natural disasters, no other zone can process the local transactions of the failed zone. This is in contrast to Steward [4] and Blockplane [46] where the failure of zones is tolerated for all transactions by replicating every single transaction on all zones.

Providing the same level of fault tolerance as Steward and Blockplane, however, requires running global synchronization for every single transaction resulting in high communication latency. To address this problem, Ziziphus can use lazy synchronization techniques to provide some degree of fault tolerance for local transactions without running global synchronization for every transaction. Byzantine fault-tolerant protocols, e.g., PBFT, use a checkpointing mechanism to produce the last stable state of data (i.e., a persisted state). Checkpoints are usually generated periodically when a transaction with a sequence number divisible by some constant is executed. In an edge network where the percentage of migration requests is not supposed to be high, zones can generate checkpoints to capture the state of their executed local transactions whenever they receive a migration request. The checkpoint is generated by the primary node of the zone 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 node 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.

3.5 Correctness

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

LEMMA 3.1. *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 [17]. We just need to show that safety is guaranteed for global transactions, i.e., the data synchronization and data migration protocols. To commit a transaction (promise and) accepted messages from a majority of zones is needed. As a result, if two different transactions m and m' have been committed with the same sequence number, 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 the order of m and a quorum of $2f + 1$ nodes, $Q_{m'}$ has agreed with the order of m'. 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 which violates 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.

Furthermore, the validity of messages is guaranteed based on standard cryptographic assumptions about collision-resistant hashes, encryption, and signatures which the adversary cannot subvert. All messages are endorsed by $2f + 1$ nodes and either the request or its digest is included in each message to prevent alterations to any part of the message. In this way, we ensure that if a request is committed, the same request must have been proposed earlier.

LEMMA 3.2. *A request m issued by a correct client will eventually be complete if the majority of zones can still communicate.*

Due to the FLP result [27], 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 3.3, the failure 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, the timer of its request will be expired 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.

4 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 wide area networks. Running consensus among all these

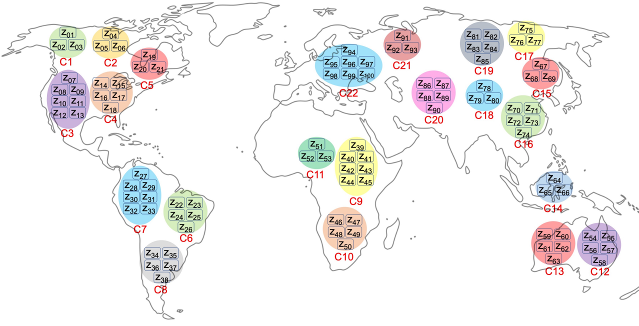


Figure 3: Ziziphus scalability using zone clusters

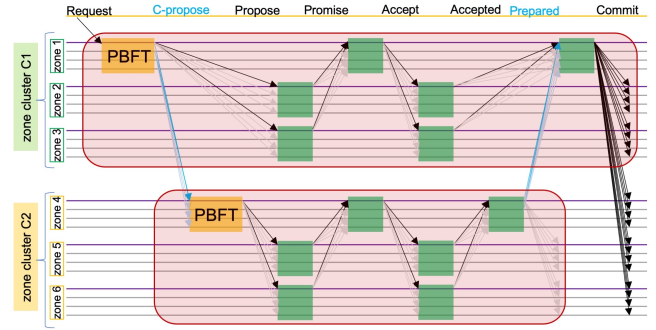


Figure 4: cross-cluster data synchronization protocol

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), however, different zone clusters maintain different system meta-data. This is a reasonable assumption because most policies need to be enforced at the regional level, e.g., country-wide, 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 all around the world.

Figure 3 demonstrates a network with 100 different zones z_1 to z_{100} where zones are clustered into 22 different clusters C_1 to C_{22}. Each cluster consists of several zones, e.g., C_1 includes three zones z_1, z_2, and z_3 while C_3 consists of seven zones z_7 to z_{13}. In this Figure, each cluster C_i maintains its own regional system meta-data which is replicated on every node of all its zones and is different from system meta-data of cluster C_j ($i \neq j$).

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 protocol, as presented in Algorithm 1, to establish consensus among zones within a zone cluster. Using the data migration protocol, as shown in Algorithm 2, the client data is then moved from the source to the destination zone. For example, in Figure 3, 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 requests a migration to a zone in a different zone cluster, e.g., from zone z_1 in C_1 to zone z_4 in C_2, Ziziphus, however, requires agreement from zones of both zone clusters C_1 and C_2 to process the request. To process such global transactions, inspired by the traditional coordinator-based sharding techniques used in distributed databases, Ziziphus proposes a cross-cluster data synchronization protocol where zones within two different zone clusters, i.e., the source cluster and the destination cluster, establish agreement on the order of the global transaction.

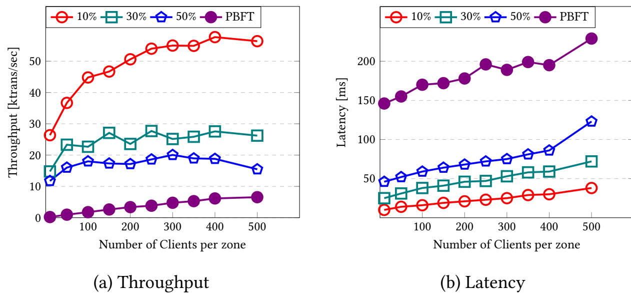
Figure 4 presents the cross-cluster data synchronization protocol between two zone clusters C_1 and C_2 where z_1 is the destination (initiator) zone and z_4 is the source zone. The destination zone (i.e., z_1) initiates the cross-cluster data synchronization protocol (i.e., plays the coordinator role) 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 $m = \langle \text{MIG-REQUEST}, op, ts_c, c \rangle_{\sigma_c}$ 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 on the order the request among nodes of its zone z_i.

In cross-cluster data synchronization protocol 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, i.e., $v_z \mod r = 0$, is always a proxy node.

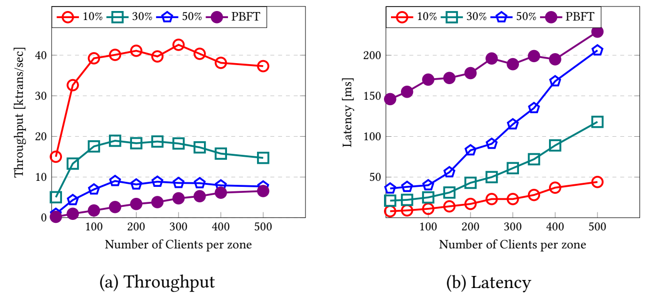
Once local consensus on the order of request in zone z_i is achieved, proxy nodes aggregate $2f+1$ local-propose messages (received in the last phase of local consensus) to construct a certificate C. The proxy nodes then multicasts $\langle \text{CROSS-PROPOSE}, v(z_i), \langle n, z_i \rangle, C, d, m \rangle_{\sigma_{\pi(z_i)}}$ message to all nodes of the source zone, i.e., the zone in the other zone cluster where client has migrated from. 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 (from any nodes), the primary node of the source zone z_j in the source cluster, e.g., z_4 in Figure 4, 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.



(a) Throughput (b) Latency

Figure 5: Performance over three zones



(a) Throughput (b) Latency

Figure 6: Performance over three zones (leader election)

As a result, each cluster requires a separate ordering, e.g., in Figure 4 both destination zone z_1 and source zone 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, however, each proxy node r of the source zone z_j in the source cluster, e.g., z_4, constructs a certificate C_s, and multicasts a $\langle \text{PREPARED}, v(z_j), \langle m, z_j \rangle, C_s, d, r \rangle_{\sigma_r}$ message to all nodes of the destination zone in the destination cluster, e.g., z_1 in Figure 4, 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) a quorum of $2f + 1$ local-commit messages from the nodes of its zone (in response to accepted messages), and (2) a valid a prepared message from the source zone. The primary node then constructs certificate C and multicasts a $\langle \text{COMMIT}, v(z_i), \langle n, z_i \rangle, v(z_j), \langle m, z_j \rangle, C, C_s, d \rangle_{\sigma_{\pi(z_i)}}$ message 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 from the source zone to the destination zone using the data migration protocol (Algorithm 2).

Correctness. The safety and liveness (during the period of synchrony) of each zone is guaranteed by PBFT [17]. The safety and liveness of each zone cluster also follow the correctness arguments discussed in Section 3.5. We now briefly discussed the correctness of communication across zone clusters. 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 3.5, safety is guaranteed.

In cross-cluster data synchronization protocol and to handle failures, nodes of the destination zone multicast responses-query messages (with the same structure as discussed in Section 3.3) 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 responses-query messages to the destination zone. Moreover, in the cross-cluster data synchronization protocol, any communications across clusters are performed by $f + 1$ nodes of each zone (proxy nodes) to prevent a malicious primary from delaying the protocol by not sending messages. The failure handling routine follows similar steps as Section 3.3. Ziziphus guarantees liveness only during periods of synchrony where in *each* zone cluster, a majority of zones can still communicate.

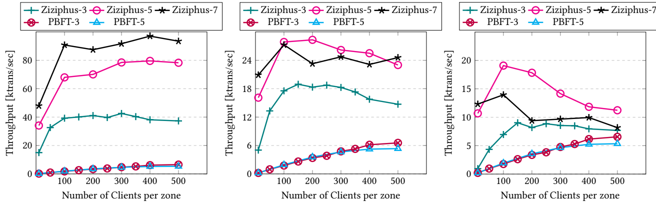
5 Experimental Evaluations

This section evaluates the performance of Ziziphus. We implemented a prototype of Ziziphus using Golang and deployed a simple accounting 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 [17]. 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% of global transactions. The workload with 10% global transaction (and 90% local transaction) is the typical setting in partitioned databases [53]. 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 further compare Ziziphus with a flat implementation of PBFT where for every transaction, PBFT runs among all nodes. The network size of Ziziphus and PBFT, however, is different, i.e., Ziziphus requires $Z * (3f + 1)$ nodes where Z is the number of zones while PBFT requires $3 * Zf + 1$ nodes where $Z * f$ is the total number of faulty nodes. The distribution of PBFT nodes in zones will be explained in each experiment in detail. In all experiments, we consider $f = 1$, i.e., each zone includes four nodes. While we have not compared Ziziphus with Steward, Steward can be seen as Ziziphus with 100% global transactions, i.e., every single transaction requires global synchronization across all zones.

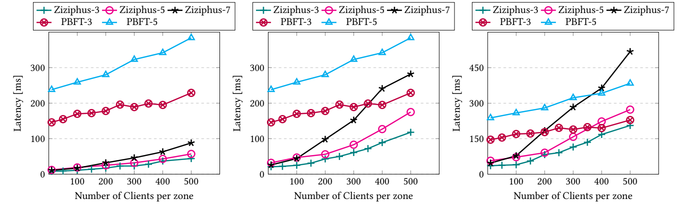
We have not run any experiments to evaluate the cross-cluster data synchronization protocol for two reasons. First, as shown in Figure 4, only cross-propose and prepared messages require cross-cluster communication. Hence, zones can process client transactions simultaneously and independently of each other and Ziziphus clearly scales linearly with increasing the number of zone clusters. Second, evaluating cross-cluster data synchronization protocol requires hundreds of nodes to be run concurrently, e.g., to run the network presented in Figure 3, 400 VM instances (other than client VMs) would be needed requiring significant resources.

In each experiment, we vary the number of requests sent by all the clients per second and measure the end-to-end throughput (x axis) and latency (y axis) of the system. We increase the number



(a) 10% global transactions (b) 30% global transactions (c) 50% global transactions

Figure 7: Throughput with increasing the number of zones



(a) 10% global transactions (b) 30% global transactions (c) 50% global transactions

Figure 8: Latency with increasing the number of zones

of concurrent clients per zone from 10 to 500 clients. Each client waits for a reply before sending a subsequent request.
The experiments were conducted on the Amazon EC2 platform where zones are placed in geographically far apart regions (in each experiment, we explain the settings in detail). All instances are size c4.large (3.75GB memory) running Amazon Linux 2 AMI.

5.1 Ziziphus Performance Over Three Zones

In the first set of experiments, we measure the performance of Ziziphus in a setting with three zones distributed over three AWS regions, i.e., California (*CA*), Ohio (*OH*), and Quebec (*QC*) where $CA \rightleftharpoons OH$: 52 ms, $CA \rightleftharpoons QC$: 80 ms, and $OH \rightleftharpoons QC$: 46 ms.
In the Ziziphus deployment, each zone consists 4 nodes, i.e., $3f+1$ where $f = 1$. BBFT, however, requires 10 nodes in total, i.e., $3{*}3f{+}1$. As a result, PBFT runs on 4 nodes in *CA* and 3 nodes in each of *QC* and *OH* datacenters. 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.
In this experiment, and for the data synchronization protocol, we use the stable leader technique, as discussed in Section 3.2.2, where a stable primary node initiates all instances of the data synchronization protocol, i.e., there is no need to perform the leader election (propose and promise) phases.
As shown in Figure 5, in the workload with 10% global transactions and in the presence of 400 concurrent clients, Ziziphus is able to process more than 57 k transactions per second with 30 ms latency before the end-to-end throughput is saturated. Increasing the number of concurrent clients to 500, however, reduces the throughput of Ziziphus and increases its performance. As can be seen in all three workloads, i.e. 10%, 30%, and 50% global transactions, Ziziphus demonstrates better performance than PBFT. This is expected because first, Ziziphus processes local transactions in parallel and second, the global transactions are processed using a cheaper protocol than PBFT.

5.2 Performance with Leader Election Phase

We then repeat the first set of experiments with the same settings but this time the leader election (propose and promise) phase is added to the data synchronization protocol. Adding this phase increases the latency of global transactions processing and reduces the overall performance. As shown in Figure 5, in the workload with 10% global transactions and in the presence of 300 concurrent clients (peak performance), Ziziphus is able to process 42.5 k transactions per second with 23 ms which demonstrates 22% lower throughput and 8% higher latency in comparison to the same experiment with a stable leader.

Increasing the percentage of global transactions to 50% results in a larger gap between two experiments; one with a stable leader and one with leader election. In the workload with 50% global transactions and in the presence of 400 concurrent clients, Ziziphus is able to process 8 k transactions per second with 168 ms which demonstrates 57% lower throughput and 95% higher latency in comparison to the same experiment with a stable leader. With 50% global transactions, Ziziphus demonstrates its best performance with 150 clients where it is able to process 9 k transactions per second with 56 ms latency (only 14% higher latency in comparison to the same experiment with a stable leader). Adding the leader election phase does not affect PBFT.

5.3 Scalability with number of Zones

In the last set of experiments, we measure the scalability of Ziziphus by increasing the number of zones to 5 and 7. The setting with 3 zones is the same as before where zones are placed in California (*CA*), Ohio (*OH*), and Quebec (*QC*) datacenters. In the 5-zone settings, zones are placed in California (*CA*), Sydney (*SYD*), and Paris (*PAR*), London (*LDN*), Tokyo (*TY*) datacenters, and with 7-zone settings, we distributed zones into *CA*, *OH*, *QC*, *SYD*, *PAR*, *LDN*, and *TY*. The average Round-Trip Time (RTT) between every pair of Amazon datacenters can be found at https://www.cloudping.co/grid.
With 10% global transactions (the typical setting in partitioned databases [53]), as shown in Figure 7(a), increasing the number of zones improves the overall throughput of Ziziphus. In this workload and with 7 zones and 400 concurrent clients in each zone, Ziziphus processes 97 k transactions per second with 63 ms latency (as shown in Figure 8). This clearly demonstrates the scalability of Ziziphus in comparison to PBFT; with 5 zones and 400 concurrent clients in each zone, Ziziphus processes 79.5 k transactions per second with 43 ms latency while PBFT processes only 5.2 k transactions per second (only 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.
Increasing the percentage of global transactions to 30% and 50% reduces the overall throughput of Ziziphus (as shown in Figure 7(b) and (c)) and increases its latency (as shown in Figure 8(b) and (c)) in different settings with 3, 5, or 7 zones as expected.
Interestingly, while with 10% global transactions (Figure 7(a)), Ziziphus demonstrates its highest throughput in the setting with 7 zones, with 50% global transactions (Figure 7(c)), the setting with 5 zones shows the best throughput. This demonstrates a trade-off between the larger number of parallel instances of local consensus in

different zones (i.e., 7 zones) and the smaller number of participants in the global transactions (i.e., 5 zones).

6 Related Work

State Machine Replication (SMR) is a technique for implementing a fault-tolerant service by replicating servers [38]. Several approaches [49][39][47] generalize SMR to support crash failures among which Paxos [39] 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. Many protocols are proposed to either reduce the number of phases, e.g., Multi-Paxos which assumes the leader is relatively stable, or Fast Paxos [40] and Brasileiro et al. [14] which add f more replicas, or reduce the number of replicas, e.g., Cheap Paxos [41] which tolerates f failures with f+1 active and f passive processors. Flexible Paxos [33] (optimized for WAN in WPaxos [2]) shows that replication quorums can be arbitrarily if they all still intersect with a leader election quorum. DPaxos [45] 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 [33] 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.

Byzantine fault tolerance refers to servers that behave arbitrarily after the seminal work by Lamport, et al. [42]. Practical Byzantine fault tolerance protocol (PBFT) [17] is one of the first and probably the most instructive state machine replication protocol to deal with Byzantine failures. Although practical, the cost of implementing PBFT is quite high, requiring at least $3f + 1$ replicas, 3 communication phases, and a quadratic number of messages in terms of the number of replicas. Thus, numerous approaches have been proposed to explore a spectrum of trade-offs between the number of phases/messages (latency), number of processors, the activity level of participants (replicas and clients), and types of failures.

FaB [44] and Bosco [52] reduce the communication phases by adding more nodes. Speculative protocols, e.g., Zyzzyva [36], HQ [22], and Q/U [1], also reduce the communication by executing requests without running any agreement between nodes and optimistically rely on clients to detect inconsistencies between nodes. To reduce the number of nodes, some approaches rely on a trusted component (a counter in A2M-PBFT-EA [19] MinBFT [56] and EBAWA [55], a hypervisor [48], or a whole operating-system instance [21]) that prevents a faulty node from sending conflicting (i.e., asymmetric) messages to different nodes without being detected. SBFT [29] and Hotstuff [57] attain linear communication overhead by increasing the number of communication phases and using advanced encryption techniques, e.g., signature aggregation [12]. MultiBFT [30] uses multiple parallel primary nodes to parallelize transaction processing. Flexible BFT [43] reduces the size of PBFT quorums for alive-but corrupt failures. Finally, SeeMoRe [9] as an asynchronous hybrid protocol takes advantage of being aware of where the crash or malicious faults may occur and either reduces the number of communication phases and message exchanges or decreases the number of required nodes. Ziziphus is different from all these protocols mainly in its two-level architecture where the maliciousness of nodes is confined within the zones.

Partitioning the data into multiple shards that are maintained by different subsets of nodes is a proven approach to enhance the scalability of databases [20]. Data sharding techniques are commonly used in globally distributed databases such as H-store [34], Calvin [54], Spanner [20], Scatter [28], Google's Megastore [11], Amazon's Dynamo [24], Facebook's Tao [15], and E-store [53]. In such systems, however, nodes are assumed to be crash-only and a coordinator-based approach is used to process cross-shard transactions where a single node plays the coordinator role.

Clustering Byzantine nodes into local fault-tolerant clusters to improve scalability has been addressed in permissioned blockchain systems, e.g., ResilientDB [31], Blockplane [46], AHL [23], Chainspace [3], Saguaro [8], SharPer [7][6] and Cerberus [32].

In ResilientDB [31], the entire ledger is replicated on every node of all clusters and, at every round, each cluster locally establishes BFT consensus on a single transaction and multicasts the locally-replicated transaction to other clusters. All clusters then, execute all transactions of that round in a predetermined order. The global synchronization required at every round results in high latency. The full replication strategy has also been used in Blockplane [46] where similar to Steward [4], each cluster locally establishes BFT consensus on a single transaction and at the top level, a CFT consensus protocol is used to globally synchronize all clusters.

Sharded-ledger approaches, on the other hand, shard the ledger and partially replicate it on each cluster. Each cluster then establishes BFT consensus on its local transactions and either a coordinator-based approach, e.g., AHL [23] and Saguaro [8], or a flattened approach, e.g., SharPer [7] and Cerberus [32], is used to process cross-shard transactions. Both coordinator-based and flattened approaches, however, follow Byzantine fault-tolerant protocols.

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.

7 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. In this paper, we present Ziziphus, a geo-distributed system that partitions Byzantine edge servers into fault-tolerant zones where each zone processes transactions initiated by nearby clients locally. When a client migrates from one zone to another, Ziziphus runs a two-level global synchronization among zones where only the primary node of each zone participates in the top-level protocol and agreement from a majority of zones is sufficient to commit the global transaction, i.e., migration. Ziziphus also defines zone clusters to improve the scalability of the system when the number of zones increases. Using zone clusters, instead of running global synchronization among all zones, only zones of a single cluster are synchronized. Based on our experiments, in workloads with a low percentage of global transactions (typical settings), Ziziphus achieves significantly better performance in comparison to PBFT (850% higher throughput with only 0.15% latency). Similarly, the performance of Ziziphus improves semi-linearly with increasing the number of zones in workloads with a low percentage of global transactions; with 7 zones, Ziziphus processes more than 97000 transactions per second.

References

[1] Michael Abd-El-Malek, Gregory R Ganger, Garth R Goodson, Michael K Reiter, and Jay J Wylie. 2005. Fault-scalable Byzantine fault-tolerant services. *Operating Systems Review (OSR)* 39, 5 (2005), 59–74.

[2] Ailidani Ailijiang, Aleksey Charapko, Murat Demirbas, and Tevfik Kosar. 2019. WPaxos: Wide area network flexible consensus. *IEEE Transactions on Parallel and Distributed Systems* 31, 1 (2019), 211–223.

[3] Mustafa Al-Bassam, Alberto Sonnino, Shehar Bano, Dave Hrycyszyn, and George Danezis. 2018. Chainspace: A sharded smart contracts platform. In *Network and Distributed System Security Symposium (NDSS)*.

[4] Yair Amir, Claudiu Danilov, Danny Dolev, Jonathan Kirsch, John Lane, Cristina Nita-Rotaru, Josh Olsen, and David Zage. 2008. Steward: Scaling Byzantine fault-tolerant replication to wide area networks. *IEEE Transactions on Dependable and Secure Computing* 7, 1 (2008), 80–93.

[5] Mohammad Javad Amiri, Divyakant Agrawal, and Amr El Abbadi. 2019. CAPER: a cross-application permissioned blockchain. *Proc. of the VLDB Endowment* 12, 11 (2019), 1385–1398.

[6] Mohammad Javad Amiri, Divyakant Agrawal, and Amr El Abbadi. 2019. On Sharding Permissioned Blockchains. In *Int. Conf. on Blockchain*. IEEE, 282–285.

[7] Mohammad Javad Amiri, Divyakant Agrawal, and Amr El Abbadi. 2021. SharPer: Sharding Permissioned Blockchains Over Network Clusters. In *SIGMOD Int. Conf. on Management of Data*. ACM, 76–88.

[8] Mohammad Javad Amiri, Ziliang Lai, Liana Patel, Boon Thau Loo, Eric Loo, and Wenchao Zhou. 2021. Saguaro: Efficient Processing of Transactions in Wide Area Networks using a Hierarchical Permissioned Blockchain. *arXiv preprint arXiv:2101.08819* (2021).

[9] Mohammad Javad Amiri, Sujaya Maiyya, Divyakant Agrawal, and Amr El Abbadi. 2020. Seemore: A fault-tolerant protocol for hybrid cloud environments. In *36th Int. Conf. on Data Engineering (ICDE)*. IEEE, 1345–1356.

[10] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, et al. 2018. Hyperledger Fabric: a distributed operating system for permissioned blockchains. In *European Conf. on Computer Systems (EuroSys)*. ACM, 30.

[11] Jason Baker, Chris Bond, James C Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. 2011. Megastore: Providing scalable, highly available storage for interactive services. In *Conf. on Innovative Data Systems Research (CIDR)*.

[12] Dan Boneh, Ben Lynn, and Hovav Shacham. 2004. Short signatures from the Weil pairing. *Journal of cryptology* 17, 4 (2004), 297–319.

[13] Gabriel Bracha and Sam Toueg. 1985. Asynchronous consensus and broadcast protocols. *Journal of the ACM (JACM)* 32, 4 (1985), 824–840.

[14] F. Brasileiro, F. Greve, A. Mostéfaoui, and M. Raynal. 2001. Consensus in one communication step. In *Int. Conf. on Parallel Computing Technologies (PaCT)*. Springer, 42–50.

[15] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, et al. 2013. TAO: Facebook's Distributed Data Store for the Social Graph. In *Annual Technical Conf. (ATC)*. USENIX Association, 49–60.

[16] Christian Cachin, Klaus Kursawe, and Victor Shoup. 2005. Random oracles in Constantinople: Practical asynchronous Byzantine agreement using cryptography. *Journal of Cryptology* 18, 3 (2005), 219–246.

[17] Miguel Castro, Barbara Liskov, et al. 1999. Practical Byzantine fault tolerance. In *Symposium on Operating systems design and implementation (OSDI)*, Vol. 99. USENIX Association, 173–186.

[18] JP Morgan Chase. 2016. Quorum white paper.

[19] Byung-Gon Chun, Petros Maniatis, Scott Shenker, and John Kubiatowicz. 2007. Attested append-only memory: Making adversaries stick to their word. In *Operating Systems Review (OSR)*, Vol. 41-6. ACM SIGOPS, 189–204.

[20] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, et al. 2013. Spanner: Google's globally distributed database. *Transactions on Computer Systems (TOCS)* 31, 3 (2013), 8.

[21] Miguel Correia, Nuno Ferreira Neves, and Paulo Verissimo. 2004. How to tolerate half less one Byzantine nodes in practical distributed systems. In *Int. Symposium on Reliable Distributed Systems (SRDS)*. IEEE, 174–183.

[22] James Cowling, Daniel Myers, Barbara Liskov, Rodrigo Rodrigues, and Liuba Shrira. 2006. HQ replication: A hybrid quorum protocol for Byzantine fault tolerance. In *Symposium on Operating systems design and implementation (OSDI)*. USENIX Association, 177–190.

[23] Hung Dang, Tien Tuan Anh Dinh, Dumitrel Loghin, Ee-Chien Chang, Qian Lin, and Beng Chin Ooi. 2019. Towards Scaling Blockchain Systems via Sharding. In *SIGMOD Int. Conf. on Management of Data*. ACM.

[24] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: amazon's highly available key-value store. In *Operating Systems Review (OSR)*, Vol. 41. ACM SIGOPS, 205–220.

[25] Amr El Abbadi, Dale Skeen, and Flaviu Cristian. 1985. An efficient, fault-tolerant protocol for replicated data management. In *SIGACT-SIGMOD symposium on Principles of database systems*. ACM, 215–229.

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

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

[28] Lisa Glendenning, Ivan Beschastnikh, Arvind Krishnamurthy, and Thomas Anderson. 2011. Scalable consistency in Scatter. In *Symposium on Operating Systems Principles (SOSP)*. ACM, 15–28.

[29] Guy Golan Gueta, Ittai Abraham, Shelly Grossman, Dahlia Malkhi, Benny Pinkas, Michael K Reiter, Dragos-Adrian Seredinschi, Orr Tamir, and Alin Tomescu. 2019. SBFT: a Scalable Decentralized Trust Infrastructure for Blockchains. In *Int. Conf. on Dependable Systems and Networks (DSN)*. IEEE/IFIP, 568–580.

[30] Suyash Gupta, Jelle Hellings, and Mohammad Sadoghi. 2021. RCC: Resilient Concurrent Consensus for High-Throughput Secure Transaction Processing. In *Int. Conf. on Data Engineering (ICDE)*. IEEE.

[31] Suyash Gupta, Sajjad Rahnama, Jelle Hellings, and Mohammad Sadoghi. 2020. ResilientDB: Global Scale Resilient Blockchain Fabric. *Proceedings of the VLDB Endowment* 13, 6 (2020), 868–883.

[32] Jelle Hellings, Daniel P Hughes, Joshua Primero, and Mohammad Sadoghi. 2020. Cerberus: Minimalistic Multi-shard Byzantine-resilient Transaction Processing. *arXiv preprint arXiv:2008.04450* (2020).

[33] Heidi Howard, Dahlia Malkhi, and Alexander Spiegelman. 2017. Flexible Paxos: Quorum Intersection Revisited. In *20th International Conference on Principles of Distributed Systems*.

[34] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan PC Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, et al. 2008. H-store: a high-performance, distributed main memory transaction processing system. *Proc. of the VLDB Endowment* 1, 2 (2008), 1496–1499.

[35] Rüdiger Kapitza, Johannes Behl, Christian Cachin, Tobias Distler, Simon Kuhnle, Seyed Vahid Mohammadi, Wolfgang Schröder-Preikschat, and Klaus Stengel. 2012. CheapBFT: resource-efficient byzantine fault tolerance. In *European Conf. on Computer Systems (EuroSys)*. ACM, 295–308.

[36] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. 2007. Zyzzyva: speculative byzantine fault tolerance. *Operating Systems Review (OSR)* 41, 6 (2007), 45–58.

[37] Jae Kwon. 2014. Tendermint: Consensus without mining. *Draft v. 0.6, fall* (2014).

[38] Leslie Lamport. 1978. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM* 21, 7 (1978), 558–565.

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

[40] Leslie Lamport. 2006. Fast paxos. *Distributed Computing* 19, 2 (2006), 79–103.

[41] Leslie Lamport and Mike Massa. 2004. Cheap paxos. In *Int. Conf. on Dependable Systems and Networks (DSN)*. IEEE, 307–314.

[42] Leslie Lamport, Robert Shostak, and Marshall Pease. 1982. The Byzantine generals problem. *Transactions on Programming Languages and Systems (TOPLAS)* 4, 3 (1982), 382–401.

[43] Dahlia Malkhi, Kartik Nayak, and Ling Ren. 2019. Flexible byzantine fault tolerance. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 1041–1053.

[44] J-P Martin and Lorenzo Alvisi. 2006. Fast byzantine consensus. *Transactions on Dependable and Secure Computing* 3, 3 (2006), 202–215.

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

[46] Faisal Nawab and Mohammad Sadoghi. 2019. Blockplane: A global-scale byzantizing middleware. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 124–135.

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

[48] Hans P Reiser and Rudiger Kapitza. 2007. Hypervisor-based efficient proactive recovery. In *Int. Symposium on Reliable Distributed Systems (SRDS)*. IEEE, 83–92.

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

[50] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. 2016. Edge computing: Vision and challenges. *IEEE internet of things journal* 3, 5 (2016), 637–646.

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

[52] Yee Jiun Song and Robbert van Renesse. 2008. Bosco: One-step byzantine asynchronous consensus. In *Int. Symposium on Distributed Computing (DISC)*. Springer, 438–450.

[53] Rebecca Taft, Essam Mansour, Marco Serafini, Jennie Duggan, Aaron J Elmore, Ashraf Aboulnaga, Andrew Pavlo, and Michael Stonebraker. 2014. E-store: Fine-grained elastic partitioning for distributed transaction processing systems. *Proc. of the VLDB Endowment* 8, 3 (2014), 245–256.

[54] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J Abadi. 2012. Calvin: fast distributed transactions for partitioned

database systems. In *SIGMOD Int. Conf. on Management of Data*. ACM, 1–12.

[55] Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, and Lau Cheuk Lung. 2010. EBAWA: Efficient Byzantine agreement for wide-area networks. In *Int. Symposium on High Assurance Systems Engineering (HASE)*. IEEE, 10–19.

[56] Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, Lau Cheuk Lung, and Paulo Verissimo. 2013. Efficient byzantine fault-tolerance. *Transactions on Computers* 62, 1 (2013), 16–30.

[57] Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. 2019. HotStuff: BFT consensus with linearity and responsiveness. In *Symposium on Principles of Distributed Computing (PODC)*. ACM, 347–356.