AVA: Fault-tolerant Reconfigurable Geo-Replication on Heterogeneous Clusters

Tejas Mane*, Xiao Li*, Mohammad Sadoghi†, Mohsen Lesani‡
*University of California, Riverside, †University of California, Davis ‡University of California, Santa Cruz,

***Abstract*—Fault-tolerant replicated database systems consume significantly less energy than the compute-intensive proof-of-work blockchain. Thus, they are promising technologies for the building blocks that assemble global financial infrastructure. To facilitate global scaling, clustered replication protocols are essential in orchestrating nodes into clusters based on proximity. However, the existing approaches often assume a homogeneous and fixed model in which the number of nodes across clusters is the same and fixed, and often limited to a fail-stop fault model. This paper presents heterogeneous and reconfigurable clustered replication for the general environment with arbitrary failures. In particular, we present AVA, a fault-tolerant reconfigurable geo-replication that allows dynamic membership: replicas are allowed to join and leave clusters. We formally state and prove the safety and liveness properties of the protocol. Furthermore, our replication protocol is consensus-agnostic, meaning each cluster can utilize any local replication mechanism. In our comprehensive evaluation, we instantiate our replication with both HotStuff and BFT-SMaRt. Experiments on geo-distributed deployments on Google Cloud demonstrates that members of clusters can be reconfigured without affecting transaction processing, and that heterogeneity of clusters may significantly improve throughput.**

I. Introduction

Blockchains such as Bitcoin [1] and Ethereum [2] maintain a global replicated ledger on untrusted hosts. However, they suffer from a few drawbacks, including high energy consumption, partitions [3], [4], [5], and stake and vote centralization [6]. Byzantine replicated systems such as PBFT [7] and its numerous following variants [8], [9], [10], [11], [12], [13], [14], [15], [16] can maintain consistent replications in the presence of malicious nodes. More interestingly, these techniques avoid energy-intensive proof-of-work hashing. Therefore, they are an appealing technology to serve as the global financial infrastructure. Thus, several projects, such as Hyperledger [11], Solida [17], Tendermint [18], Casper [19], Algorand [20], OmniLedger [21] RapidChain [22], and [23] deployed Byzantine replication protocols to manage blockchains.

However, Byzantine replication protocols need to be improved on two fronts: *scale* and *dynamic* membership. They often require rounds of message-passing between nodes; therefore, they tend not to scale to many or distant nodes. Further, their membership is often fixed. In fact, the resulting blockchains are called permissioned since the nodes are fixed and initially known.

To scale Byzantine replication across the globe, projects such as Steward [24] and ResilientDB [25], [26] and Narwhal [27] try to use global communication judiciously, and decrease global in favor of local communication. They allow neighboring nodes to form clusters. This enables each cluster to order transactions locally while reducing the need for inter-cluster communication to reach an agreement on the global order. Since the communication among members of a cluster is local, clusters can maintain high throughput and low latency. Further, coordination is divided between clusters, and they can order transactions in parallel.

However, existing clustered replication protocols are homogeneous and fixed. The number of nodes is the same across clusters. Further, nodes cannot join or leave clusters. A global financial system needs to be *heterogeneous*: different regions might have different numbers of active nodes. More importantly, decentralization promised *dynamic* membership: active nodes should be able to churn. This is the property that proof-of-work blockchains, such as Bitcoin, observe, allowing any incentivized nodes to join and keep the system running. Reconfiguration has been studied for non-clustered replication [28], [29], [30], [31], [32], [33] but it remains an open problem for clustered replication. Can we have the best of both worlds? *Can we have the energy efficiency, equity, and scalability of clustered Byzantine replication, and the dynamic membership of proof-of-work? Can we have reconfigurable clustered replication?*

Reconfiguring a clustered replication system *without compromising security* is a challenging task. If the reconfigurations are not propagated uniformly to all clusters, correct replicas (*i.e.*, processes) might accept invalid messages or miss valid ones. Inconsistent views of membership may lead to violation of both safety and liveness. Byzantine replicated systems can often tolerate one-third of replicas to be Byzantine. Thus, if a message is received from more than one-third of replicas, at least one correct replica must have sent it; therefore, the message can be trusted. Consider a cluster C_{old} that is not informed of new additions to another cluster C_{new}. The cluster C_{old}'s record of one-third is less than the actual one-third for C_{new}. Therefore, the Byzantine replicas in C_{new} can form a group that is larger than the old one-third, and can make C_{old} accept a invalid message. On the flip side, C_{new} might miss messages from C_{old}. Since C_{old} thinks that C_{new} is smaller, in order to communicate a message, C_{old} might send a message to an insufficient number of replicas in C_{new}. Thus, the Byzantine replicas in C_{new} can censor the message for other replicas in that cluster, hindering liveness. Uniform propagation of reconfigurations is particularly challenging when the leader simultaneously changes.

In this paper, we present AVA, a reconfigurable clustered replication that can tolerate arbitrary faults. It allows replicas to be divided into multiple *heterogeneous* clusters, and further allows *dynamic* membership for clusters: replicas can join and

leave a cluster. This clustered design further reduces the cost of inter-cluster communication by allowing each cluster to independently reach an agreement on its membership and order transactions locally and only propagate the local decisions globally. Reconfigurations are processed efficiently in parallel to transactions. Since the reconfigurations received in a round r take effect for the next round $r + 1$, they do not need to be ordered in round r. Thus, instead of processing them in sequence through the consensus that orders transactions, they are *aggregated* into a set, and processed together. We present the reconfiguration protocol and formally state and prove its safety and liveness.

Reconfiguration of heterogenous clusters introduces nuances that affect ordering and executing transactions. In particular, the inter-cluster communication primitive and the remote leader fault detection mechanism must have up-to-date knowledge of the size of the local and remote clusters in order to ensure safety and liveness.

AVA is a meta-protocol that is *agnostic* to the local replication protocol. We implement AVA for HotStuff [9] in C++, and for BFT-SMaRt [34] in Java. We deployed the resulting systems on geo-distributed clusters in multiple regions of Google Cloud. The experimental results show that heterogeneous geo-distributed deployments significantly improve throughput, can be reconfigured without affecting transaction processing, and can gracefully tolerate Byzantine failures.

In short, this paper makes the following contributions:

- We present AVA, a *reconfigurable clustered replication protocol* that allows replicas to dynamically join and leave clusters safely and efficiently.
- AVA entails *Heterogeneous clustered replication* to support clusters with varying sizes. Thus, AVA includes a novel inter-cluster communication primitive and remote leader replacement mechanism.
- *Formal specification and proof of safety and liveness properties* of AVA including dynamic reconfiguration in Byzantine heterogeneous environments.
- *Implementation and experimental results* that demonstrate that AVA is agnostic to the the local consensus (e.g., HotStuff and BFT-SMaRt). Thorough experiments with the resulting systems (AVA-HOTSTUFF and AVA-BFTSMART) show that AVA supports efficient and fault-tolerant global replication and reconfiguration, and that heterogeneity improves performance.

II. OVERVIEW

In this section, we describe the system and threat model, and illustrate the protocol with diagrams and representative executions.

System and Threat Model. A replicated system consists of a set $\mathcal{P}$ of replicas that are partitioned into clusters $\mathcal{C} = \{C_1, .., C_N\}$. Clients can send requests to any replica to execute operations of two different types: transactions and reconfigurations. The state is replicated at each replica. (In contrast to sharding, the state is not partitioned.) A replica can be correct or Byzantine. A Byzantine replica can fail arbitrarily including but not limited to crash failures, sending

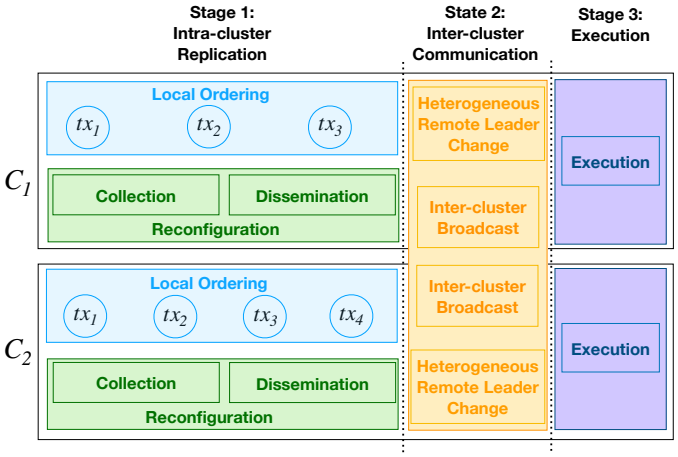
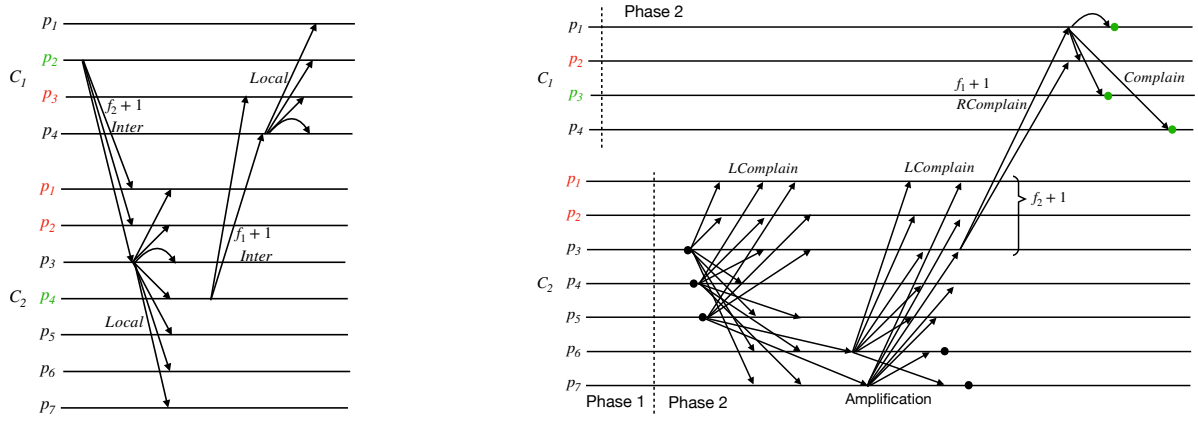


Fig. 1: AVA Reconfigurable Clustered Replication Protocol. Overview of Stages and Sub-protocols

conflicting messages, dropping messages, and impersonating other Byzantine replicas. We assume that at any time in each cluster, at most one-third of replicas can be Byzantine, *i.e.*, at most f out of $3f + 1$ replicas can be Byzantine. (This paper does not consider problems orthogonal to Byzantine fault tolerance such as access control or sybil resistance [35], [36], [37].) We further assume that each replica can be identified by its public key, and that replicas are computationally bound, and cannot subvert standard cryptographic primitives. Thus, replicas can communicate with authenticated links. We consider a partially synchronous network [38]: after an unknown global stabilization time, messages between any pair of correct replicas will be eventually delivered within a bounded delay. Replicas communicate with authenticated perfect links apl, and authenticated best-effort broadcast $abeb$ which simply abstracts apl to send a message to all replicas. Each message m^{σ} delivered from an authenticated link comes with a signature σ of the sender. (It's elided if it's not needed in a context.)

Overview. The protocol proceeds in consecutive rounds r. In order to avoid global communication in favor of local communication, replicas are divided into clusters. Each round has three stages. In the first stage, clusters process requests in parallel; the replicas of each cluster agree on the transactions and their order, and further the set of reconfigurations for that round. In the second stage, clusters communicate these operations with each other. Finally in the third stage, they execute all the operations in the decided orders. Each cluster has a leader that coordinates the replication of both transactions and reconfigurations. Each replica knows the $leader$ of its cluster and its associated timestamp ts. (Leaders of each cluster are elected together with a monotonically increasing timestamp which replicas use to decide which leader is more recent.) A leader might continue to serve for multiple rounds. On the other hand, several leaders might change until a correct leader properly replicates transactions and reconfigurations of the round.

Stages. The AVA protocol has three stages. Fig. 1 shows an overview of the stages and sub-protocols in a round. The replicas are split into clusters. The figure shows two clusters



(a) Inter-cluster Broadcast

(b) Remote Leader Change

Fig. 2: Inter-cluster Communication. $f_1 = 1$, $f_2 = 2$.

C_1 and C_2 that make progress from left to right through the stages.

State 1: Intra-cluster Replication. The first stage is intra-cluster replication where each cluster coordinates replication locally and independently of other clusters. The first stage has two parts that are executed in parallel: local ordering, and reconfiguration. The local ordering protocol orders a batch of transactions uniformly across the replicas of the cluster. The protocol is agnostic to the local ordering sub-protocol; any consensus protocol can be used. The second part, the reconfiguration protocol, collects and uniformly disseminates the reconfiguration requests across the cluster, even if the leader is Byzantine or changes simultaneously.

State 2: Inter-cluster Communication. After the first stage finishes intra-cluster replication, the second stage performs inter-cluster communication: the leader of each cluster broadcasts to other clusters the transactions and reconfigurations that it has locally replicated. Each cluster waits to receive these messages from every other cluster. If a remote leader is Byzantine, it may refrain from sending these messages. Therefore, to ensure progress, if the replicas of a cluster don't receive the message from a remote cluster, they trigger the remote leader change protocol to eventually change the leader of that remote cluster.

Stage 3: Execution. Finally, in the third stage, each replica orders the transactions and reconfigurations that it has received from all clusters by a predefined order for the clusters, executes them in order, and issues responses. This predefined order yields a total-order for operations across replicas. replicas converge to the same state at the end of the round.

A. *Overview of* AVA *Inter-cluster Communication*

Let us consider the inter-cluster communication stage (*i.e.*, stage 2). Fig. 2 shows example executions of this stage for two heterogeneous clusters C_1 and C_2 with 4 and 7 replicas respectively. In Fig. 2a, the leaders of C_1 and C_2 are the green replicas p_2 and p_4 respectively. The Byzantine replicas of C_1 and C_2 are the red replicas $\{p_3\}$ and $\{p_1, p_2\}$ respectively. We note that in both clusters, the number of Byzantine replicas is less than one-third of the size of the cluster: $f < |C|/3$ that is $f_1 = 1 < 4/3$ and $f_2 = 2 < 7/3$.

Inter-cluster Broadcast. Fig. 2a shows an execution of the inter-cluster broadcast protocol. Each cluster has already locally replicated operations (including transactions and reconfiguration requests); each operation is paired with a certificate of consensus which is approval signatures from a quorum of replicas in that cluster. The leader of each cluster sends its operations together with their certificates to other clusters as inter-cluster messages $Inter$. To ensure that at least one correct replica in the remote cluster receives the message, the leader sends the message to $f + 1$ replicas in the remote cluster. In our heterogeneous clusters example, the leader p_2 of C_1 sends the message to $2 + 1 = 3$ replicas in C_2, and the leader p_4 of C_2 sends the message to $1 + 1 = 2$ replicas in C_1. In the remote cluster, the correct replica that receives the $Inter$ message then broadcasts the operations as $Local$ messages to replicas in its own cluster. Thus, if the leaders are correct, all correct replicas eventually receive operations from all clusters.

Clustering reduces the number of rounds and message complexity for global communication. We just considered the inter-cluster broadcast of stage 2 above. Let us compare the complexity of classical (*i.e.*, not clustered) replication such as PBFT with clustered replication. Consider $n_1 = |\mathcal{C}_1|$, $n_2 = |\mathcal{C}_2|$, and the total number $n = n_1 + n_2$ replicas. To process a single transaction, replication requires 2 global rounds with message complexity $\mathcal{O}((n_1 + n_2)^2)$. To process 2 transactions in parallel in C_1 and C_2, clustered replication executes stage 1 with 2 local rounds with message complexity $\mathcal{O}(n_1^2+n_2^2)$, and then stage 2 with 1 global round with message complexity $(f_1+1)+(f_2+1) = \mathcal{O}(n_1+n_2)$, and finally, 1 local round with message complexity $(f_1+1)\times n_1+(f_2+1)\times n_2 = \mathcal{O}(n_1^2 + n_2^2)$. Therefore, global communication is reduced from 2 rounds of complexity $\mathcal{O}((n_1 + n_2)^2)$ to 1 round of complexity $\mathcal{O}(n_1 + n_2)$.

Remote Leader Change. A Byzantine leader may behave properly in the local cluster, but skip sending $Inter$ messages to other clusters. Let us now consider how the replicas of a cluster can instigate the change of the leader of a remote cluster if they don't receive the expected message from it. Fig. 2b shows an execution of the remote leader change protocol. The current leader p_2 of the cluster C_1 is Byzantine, and will be changed to the correct replica p_3. In cluster C_2, the replicas p_3, p_4 and p_5 have not received the operations of C_1, and their timers expire; thus, they broadcast a local complaint $LComplaint$ in C_2 about C_1. The replicas p_6 and p_7 in C_2

have not already complained, but receive $f_2+1 = 3$ complaints from the three replicas above. Since at least 1 out of 3 is from a correct replica, they amplify the complaint by broadcasting an $LComplaint$ message locally. A replica accepts the local complain only when it receive it from $2 \times f_2 + 1 = 5$ replicas. It can be shown that this prevents a coalition of Byzantine replicas from forcing a leader change, and ensures that all local correct replicas eventually deliver the complaint. When the first $f_2 + 1 = 3$ replicas accept the local complaint, they send a remote complaint $RComplaint$. To make sure that the message is sent to the remote cluster, it is sent by $f_2 + 1$ replicas which contain at least one correct replica. In the first three replicas, p_3 is correct and sends the remote complaint. The complaint should reach at least one correct replica in C_1; thus, p_3 sends it to $f_1 + 1 = 2$ replicas in C_1. The replica p_1 in C_1 is correct, and receives the remote complaint. It accepts the complain if it carries $2 \times f_2 + 1$ signatures from C_2. It then broadcasts a $Complaint$ message locally in C_1. When the correct replicas receive the local complaint (at green circles), they move to the next leader p_3. The protocol should deal with complaint replay attacks, and multiple simultaneous change requests, that we will describe in the next section.

B. Overview of AVA Reconfiguration

Let's now consider reconfiguration. Reconfiguration not only allows replicas to join and leave but also supports rebalancing the system to maintain the proximity of replicas in a cluster, and similarity of performance across clusters.

Attacks. The reconfiguration requests should be uniformly propagated across clusters, *i.e.*, the configurations that every pair of correct replicas (possibly from different clusters) execute in a round should be the same. When they are not, the following Byzantine attacks may arise. Consider two clusters C_1 and C_2 with 4 and 7 replicas, and the failure thresholds $f_1 = 1$ and $f_2 = 2$ respectively. Assume that 3 new replicas join C_1 and one of them is Byzantine. The updated C_1 now has 7 replicas, and the failure threshold is $f_1' = 2$. However, assume that the correct replicas in C_2 are unaware of the newly joined replicas in C_1; they keep the stale failure threshold $f_1 = 1$, and will accept any operations with $2 \times f_1 + 1 = 3$ signatures. If C_1 has a Byzantine leader, it can forge a certificate for a set of operations ops_1: it can get a signature from only one correct replica for ops_1. Then, it can also have signatures from itself and the other Byzantine replica, to have a total of 3 signatures. It can then make the replicas in C_2 accept ops_1 with the forged certificate. However, it can lead the correct replicas in C_1 to eventually replicate a different set of operations. Thus, the correct replicas in C_1 and C_2 diverge.

Let us now consider another attack in the same setting. Since the correct leader of C_2 has a stale failure threshold $f_1 = 1$, it sends $f_1 + 1 = 2$ inter-cluster broadcast messages to C_1. The receiver replicas in C_1 can be both Byzantine, and may drop the message. Then, the timers of the correct replicas in C_1 will eventually trigger, and they complain about the leader of C_2. The remote leader change eventually replaces the correct leader in C_2. Unfortunately, the Byzantine replicas in C_1 can repeat changing the leader until a Byzantine replica is in control in C_2.

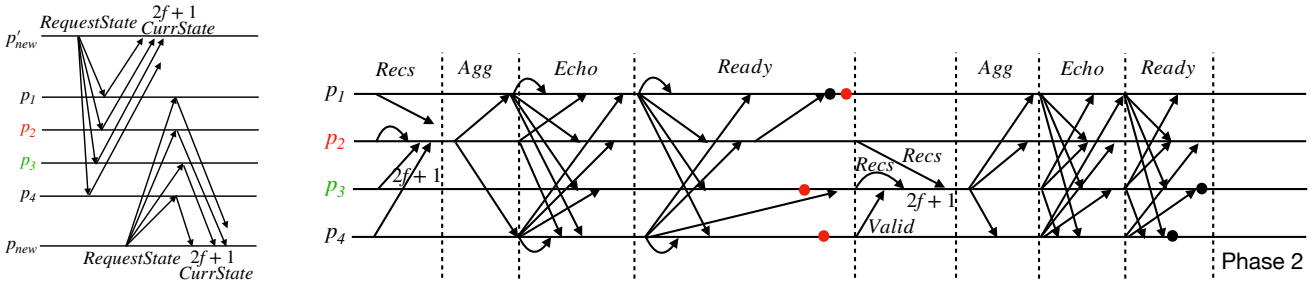
Let's consider the reconfiguration protocol. Replicas can request join and leave reconfigurations in stage 1 (Intra-cluster replication). Clusters communicate only in stage 2 (Inter-cluster communication). Clusters communicate only in stage 2 (Inter-cluster communication). Thus, if the reconfigurations requested in a cluster in stage 1 are processed as they are requested, remote clusters will have an inconsistent view of membership for the local cluster. We explained above that these inconsistencies are unsafe. Therefore, the reconfigurations requested in a round are locally collected and disseminated in stage 1, are remotely communicated in stage 2, and applied in stage 3 to uniformly update membership for the next round. Thus, in each round, they can be collected as a set, and the order that they are processed in is immaterial. Therefore, collecting them can be taken off the critical path that orders transactions. Thus, as Fig. 1 shows, reconfigurations are collected and disseminated as a separate workflow in parallel to transaction processing. Fig. 3 shows example executions for both parts of the reconfiguration protocol, collection and dissemination, which we will describe next.

Collection. In Fig. 3a, two replicas p_{new} and p'_{new} request to join the cluster. Each broadcasts a $RequestJoin$ message. When a correct replica delivers a $RequestJoin$ message, it adds the join request to its set of reconfiguration requests, and responds back by a Ack message. A joining replica periodically keeps sending $RequestJoin$ messages until it receives the Ack message with the same configuration from a quorum of $2 \times f + 1 = 3$ replicas. It stops then as it learns that Byzantine replicas cannot censor the request.

Dissemination. The same set of reconfigurations should be uniformly disseminated to all correct replicas in the cluster. Otherwise, as we discussed in the introduction, an inconsistent view of members can lead to accepting fake, or discarding genuine messages. We describe an execution where the leader is Byzantine and is changed; nonetheless, the same set of reconfigurations are uniformly delivered to all correct replicas.

As Fig. 3b shows, later in the first stage, each correct replica sends the set of reconfiguration requests that it has collected as $Recs$ messages to the leader replica p_2. When the leader p_2 receives messages from a quorum, it aggregates the received sets of reconfigurations, and the accompanying signatures, and then starts disseminating them. Since there is a correct replica in the intersection of every pair of quorums, the leader does not miss the requests. In Fig. 3a and 3b, the quorum $\{p_1, p_2, p_3\}$ that p'_{new} receives the state from, and the quorum $\{p_2, p_3, p_4\}$ that the leader p_2 receives requests from intersect in the correct replica p_3.

The leader broadcasts the aggregation of the reconfiguration requests that it collected. Upon delivery from the leader, a correct replica checks whether the received reconfigurations are valid: they should be accompanied by at least a quorum of signatures for $Recs$ messages. As we saw in the collection part, a requesting replica makes a quorum of replicas store the reconfiguration request. Therefore, the leader cannot drop requested reconfigurations: if the leader drops a request, and hence, any signature from the quorum of replicas that stored it, then the remaining replicas will be smaller than a quorum, and



(a) Collection

(b) Dissemination

Fig. 3: Reconfiguration: Collection and Dissemination

the leader cannot collect a quorum of signatures. In Fig. 3b, although the leader p_2 is Byzantine, it has to send the complete aggregated set. However, it only sends it to a subset of replicas $\{p_1, p_4\}$. The correct replicas p_1 and p_4 that receive a message from the leader echo it. The Byzantine replica p_2 echos to them but not p_3. Thus, p_1 and p_4 receive a quorum of 3 *Echo* messages, and broadcast a *Ready* message. The Byzantine replica p_2 sends a *Ready* message to only p_1. Thus, only p_1 receives a quorum of 3 *Ready* messages, and delivers the reconfigurations (at the black circle).

The correct replicas p_3 and p_4 don't receive enough *Ready* messages, eventually complain about the leader p_2, and change the leader to the correct replica p_3 (at the red circles). To preserve uniformity, the new leader p_3 should retrieve the set of reconfigurations that p_1 previously delivered. We will describe later in § IV, how the leader retrieves that set, and makes the remaining correct replicas p_3 and p_4 eventually deliver the same set (at black circles).

We note that the classical Byzantine reliable broadcast (BRB) and Byzantine consensus would be inadequate for reconfiguration dissemination. Firstly, in contrast to BRB that guarantees termination only when the sender is correct, the reconfigurations are expected to be eventually delivered in each round even if the initial leader is Byzantine. Thus, to ensure termination, the leader might be changed during dissemination. The challenge is to keep uniformity across leaders. Further, in contrast to BRB where a message from one designated sender is broadcast, and in contrast to consensus where a proposal from one replica is decided, this protocol should aggregate and broadcast a collection of reconfigurations from a quorum of replicas.

In this section, we saw an overview of the stages (Fig. 1 and the accompanying description). Next, we first consider the two more important sub-protocols: inter-cluster communication (§ III), and reconfiguration (§ IV). (We detail the the stages in the extended report [39].

III. Inter-cluster Communication

We will now present the inter-cluster broadcast protocol that propagates operations between clusters, and the heterogeneous remote leader change protocol that detects and changes Byzantine leaders for remote clusters.

State. Each replica keeps the set of replicas C_j for each cluster. (We use the index $\boldsymbol{i}$ only for the current cluster, and the index j for clusters in general.) The set $C_{\boldsymbol{i}}$ keeps track of membership within the current cluster $\boldsymbol{i}$, and is used for intra-cluster communication. The sets C_j that keep track of the members of remote clusters j are used for inter-cluster broadcast. Accordingly, a replica has the failure threshold f_j for each cluster C_j as one-third of the size of C_j. Each replica also keeps the current round r. Further, it stores the operations $operations_j$ that it receives from each cluster C_j. Each replica keeps a set of certificates *certs* for its local operations $operations_{\boldsymbol{i}}$. A certificate for an operation contains at least $2 \times f_{\boldsymbol{i}} + 1$ signatures, and is sent to other clusters together with the operation. The protocol uses authenticated perfect links *apl*, and authenticated best-effort broadcast *abeb* (that were described in § II). Each message m^σ delivered from an authenticated link comes with a signature σ of the sender. (We elide the signature when it is not needed in a context.)

Inter-cluster Broadcast. At the end of stage 1, the local ordering stage, the leader calls the function *inter-broadcast* (Alg. 1) to start the second stage. Each cluster broadcasts its locally ordered operations to remote clusters. As Fig. 2a shows, this function sends out the batch of operations *ops* of the local cluster together with their certificates *certs* as *Inter* messages to other clusters (at line 11-14). For each remote cluster j, the *Inter* messages are sent to $f_j + 1$ distinct replicas. Therefore, at least one correct replica at cluster C_j eventually receives the *Inter* message (at line 15). It checks that the certificates are valid: a certificate for an operation from cluster $C_{j'}$ is valid if it contains at least $2 \times f_{j'} + 1$ signatures from the cluster $C_{j'}$. The receiving replica then broadcasts the operations as *Local* messages to other replicas in its own cluster (at line 16). Upon receiving a *Local* message containing operations *ops* from a remote cluster j' with valid certificates (at line 17), the replica maps j' to *ops* in its *operations* map. It also stops a *timer* that watches the leader of cluster j. (We will consider remote leader change in the next paragraph). When operations from all clusters are received, the replica calls the function *execute* to enter stage 3, the ordering and execution stage (at line 21).

Heterogeneous Remote Leader Change. Each replica waits until it receives operations from other clusters. Therefore, if the leader of a cluster is Byzantine, and avoids sending operations to other clusters, it can stall progress. Consider a system where cluster C_j has a Byzantine leader l. For example in Fig. 2b, the leader p_2 of C_1 is Byzantine. It acts as a correct leader internally in C_j for the local ordering stage. The correct replicas in C_j cannot identify l as a Byzantine leader to replace it. However, l does not follow the protocol to send its operations to a remote cluster $C_{j'}$. Thus, replicas of the cluster $C_{j'}$ cannot proceed to the ordering and execution stage.

Algorithm 1: Inter-cluster Broadcast

```
1  vars:
2    C_j : Set[P]                          ▷ Replicas of each cluster C_j
3    i                                     ▷ The number of the current cluster
4    f_j : ℕ                               ▷ Failure threshold for C_j
5    r                                     ▷ The current round
6    operations_j ← ∅                      ▷ Operations from each cluster C_j
7    certs                                 ▷ Certificates for operations_i of C_i
8  uses:
9    apl : AuthenticatedPoint2PointLink
10   abeb : AuthenticatedBestEffortBroadcast in C_i
11 function inter-broadcast(r, ops, certs)
12   foreach C_j, j ≠ i
13     foreach p ∈ P where P ⊆ C_j ∧ |P| = f_j + 1
14       apl request send(p, Inter(r, i, ops, certs))
15 upon apl response deliver(p, Inter(r', j, ops, Σ)) where r' = r
     ∧ Σ is valid (i.e., Σ has at least 2 × f_j + 1 signatures from C_j
     for each op ∈ ops)
16   abeb request broadcast(Local(r, j, ops, Σ))
17 upon abeb response deliver(p, Local(r', j, ops, Σ)) where r' = r
     ∧ Σ is valid
18   operations_j ← ops
19   stop timer_j
20   if |dom(operations)| = N    ▷ N is # of clusters then
21     call execute(operations)
```

Algorithm 2: Heterogeneous Remote Leader Change

```
1  vars:
2    self                                  ▷ The current replica
3    timer_j ← Δ                           ▷ A timer for each cluster C_j
4    cn_j ← rcn_j ← 0   ▷ # of complaints sent to & received from C_j
5    cs_j ← ∅                              ▷ Complaint signatures for each cluster C_j
6    complained_j ← false                  ▷ If complained about each cluster C_j
7  upon timer_j for remote C_j expires
8    abeb request broadcast(LComplaint(j, cn_j, r))
9    complained_j ← true
10 upon abeb response deliver(p, LComplaint(j, c, r')^σ) where
     r' = r ∧ c = cn_j ∧ operations_j = ⊥
11   cs_j ← cs_j ∪ {σ}
12   if |cs_j| ≥ f_i + 1 ∧ ¬complained_j then
13     complained_j ← true
14     abeb request broadcast(LComplaint(j, c, r))
15   if |cs_j| ≥ 2 × f_i + 1 then
16     let S := first f_i + 1 replicas of C_i in
17     if self ∈ S then
18       apl request send(p, RComplaint(cn_j, i, cs_j, r)),
           for each p ∈ S' in a set S' such that S' ⊆ C_j ∧
           |S'| = f_j + 1
19     cn_j ← cn_j + 1
20     cs_j ← ∅;   complained_j ← false;   reset timer_j
21 upon apl response deliver(p, RComplaint(c, j', Σ, r)) where
     r = r' ∧ c = rcn_j' ∧ Σ contains 2 × f_j' + 1 signatures from C_j'
22   abeb request broadcast(Complaint(c, j', Σ))
23 upon abeb response deliver(p, Complaint(c, j', Σ)) where
     c = rcn_j' ∧ Σ contains 2 × f_j' + 1 signatures from C_j'
24   rcn_j' ← rcn_j' + 1
25   if Δ − timer_i > ε then
26     le request next-leader
```

Intuition. Let us briefly describe how the local cluster can trigger leader change in a remote cluster. Each replica keeps a timer $timer_j$ for the leader of each cluster C_j. It resets the timers for all clusters at the beginning of each round. When a local replica does not receive the operations of a remote cluster, and the timer expires, it broadcasts a complaint in its local cluster. When enough local replicas complain, the complaint is eventually accepted locally. A subset of local replicas that accept a complaint send complaints to remote replicas which in turn broadcast it in the remote cluster. Once remote replicas receive the remote complaint, they change the remote leader.

A remote replica accepts a remote complaint only if it comes with a quorum of signatures from the complaining cluster. This prevents any coalition of Byzantine replicas in the complaining cluster to force a remote leader change. However, a Byzantine replica in the remote cluster can keep a valid complaint and its accompanying signatures, and launch a replay attack: it can resend the valid complaint to repeatedly change the leader. To prevent this attack, the complaining cluster maintains a complaint number cn_j for each remote cluster C_j, which is incremented on every remote complaint sent to C_j. A remote replica maintains the number of complaints received $rcn_{j'}$ from each other cluster $C_{j'}$, and only accepts a complaint with the next expected number, and then increments the number. Therefore, a remote replica accepts each remote complaint only once.

Protocol. As Alg. 2 presents, if a local replica finds that the timer $timer_j$ for a remote cluster C_j is expired (at line 7), it broadcasts a local complaint $LComplaint$ message about C_j to replicas in its own local cluster (at line 8). In Fig. 2b, the replicas $\{p_3, p_4, p_5\}$ in C_2 send $LComplaint$ messages. The message includes the current complaint number cn_j. Once a local replica receives a local complaint for a remote cluster C_j with the expected complaint number cn_j, and it has not received operations from that cluster (at line 10), it records the accompanying signature σ in the set of complaint signatures cs_j (at line 11). If the replica receives $f_i + 1$ complaint signatures, since at least one is from a correct replica, the replica amplifies the complaint locally if it has not already complained (at line 12-14). In Fig. 2b, the replicas $\{p_6, p_7\}$ in C_2 amplify the $LComplaint$ message.

Once a replica receives $2 \times f_i + 1$ complaint signatures (at line 15), it accepts the local complaint. Since there is at least one correct replica in the senders, Byzantine replicas cannot force a leader change. Further, since the complaint is received from $2 \times f_i + 1$ replicas, it can be shown that all correct replicas in the local cluster eventually deliver the complaint. The complaint should reach at least one correct replica in the remote cluster C_j. Therefore, the remote complaint message $RComplaint$ should be sent to at least $f_j + 1$ remote replicas. Further, at least one *correct* replica should send these messages. Therefore, at least $f_i + 1$ replicas should send it. The first $f_i + 1$ replicas of the local cluster (by a predefined order) send the complaint (at line 16); we call them the sender set. In Fig. 2b, the sender set is $\{p_1, p_2, p_3\}$. The two replicas p_1 and p_2 are Byzantine but p_3 is correct and sends the message. If the current replica is in the sender set, it sends a remote complaint $RComplaint$ message to a subset of C_j of size $f_j + 1$ (at line 17-18). The remote complaint message includes the complaint number cn_j and the collected signatures cs_j. Finally, the local replica increments the complaint number, and resets the state for the next complaint (at line 19-20).

Algorithm 3: Collection

```
1  request : join, leave
2  response : joined, left
3  vars:
4      recs ← ∅                                   ▷ Set of reconfigurations
5      client-timer ← Δ
6  upon request join
7      abeb request broadcast(RequestJoin(r))
8  upon request leave
9      abeb request broadcast(RequestLeave(r))
10 upon client-timer expires
11     if requested join then
12         abeb request broadcast(RequestJoin(r))
13     else if requested leave then
14         abeb request broadcast(RequestLeave(r))
15     reset client-timer to a longer period
16 upon abeb response deliver(p, RequestJoin^σ(r')) where r = r'
17     recs ← recs ∪ {join(p)^σ}
18     apl request send(p, Ack(C_i, r))
19 upon abeb response deliver(p, RequestLeave^σ(r')) where r = r'
20     recs ← recs ∪ {leave(p)^σ}
21     apl request send(p, Ack(C_i, r))
22 upon apl response deliver(p̄, Ack(C', r')) where
       |{p̄}| ≥ 2 × f_i + 1 where r = r'
23     stop client-timer
```

Algorithm 4: Dissemination

```
24 uses:
25     brd : ByzantineReliableDissemination in C_i
26 function send-recs
       ▷ Called by each replica before the end of stage 1.
27     brd request broadcast(Recs(r, recs))
28 upon brd response deliver(Recs(r', recs), Σ)^Σ' where r' = r ∧
       Σ and Σ' are valid.
29     append Reconfig(∪ recs) to operations_i
30     add Σ, Σ' to certs
31 upon brd response complain(p)
32     call complain(p)
```

Once a replica receives the remote complaint message (at line 21), if the message has the next expected complaint number $rcn_{j'}$, and it carries $2 \times f_{j'} + 1$ signatures from the complaining cluster $C_{j'}$, it broadcasts a *Complaint* message in its own cluster (at line 22). When a replica receives the complaint message from its local cluster (at line 23), it performs similar checks to accept it. It then increments the received complaint number $rcn_{j'}$ for the complaining cluster $C_{j'}$, and unless the leader is recently changed, it requests the local leader election module le to move to the next leader (at line 24-26). (We will consider the local leader election module le in § IX.) If the leader is changed recently (*i.e.*, only a small amount of time ϵ is passed since the $timer_i$ is reset to Δ), the protocol avoids requesting to change the leader again so that the new leader is not disrupted. In particular, this happens when multiple remote clusters complain about the same leader at almost the same time.

IV. Reconfiguration

A replica p can issue a *join* or *leave* request to join or leave. Later, it receives a *joined* or *left* response (when the reconfiguration is executed in stage 3). As we showed in Fig. 1 and briefly described in the overview § II, reconfiguration requests are collected, and then disseminated locally in stage 1. We now consider these two steps.

Collection. As Alg. 3 presents, when a client process (or replica) p receives a *join* request (at line 6), it broadcasts *RequestJoin* messages in the local cluster (at line 7). In Fig. 3a, two replicas p_{new} and p'_{new} request to join. Similarly, when a correct replica p receives a *leave* request, it sends out *RequestLeave* messages. The client uses the *client-timer* to track progress while it waits for a response. If the timer expires (at line 10), it resends the messages, and resets the timer to a larger period. When a correct replica delivers the *RequestJoin* message from p (at line 16), it adds the reconfiguration request $join(p)$ to its set of collected reconfigurations $recs$, and sends back an Ack message (at line 17-18). The steps are similar for the *RequestLeave*. When the requesting replica receives Ack messages with the same cluster members, and round from a quorum (at line 22), it learns that the request cannot be censored by Byzantine replicas; therefore, it stops the timer. In Fig. 3a, the two joining replicas stop the timer when they receive Ack from 3 replicas.

Dissemination. Before completing the first stage, a correct replica calls *send-recs* (Alg. 4 at line 26) that sends a *Recs* message containing the set of reconfiguration requests $recs$ that it has collected to the Byzantine Reliable Dissemination (BRD) module (at line 27).

BRD collects messages and disseminates them. It eventually issues a response with a set of collected reconfigurations $\overline{recs}$ (at line 28). The delivery is accompanied by two certificates. The certificate Σ attests that $\overline{recs}$ are collected from at least a quorum of replicas. In the collection part, a reconfiguration request was stored in at least a quorum of replicas. If Σ is valid, then BRD has collected reconfigurations from at least a quorum of replicas. Since there is a correct replica in the intersection of two quorums, a Byzantine leader cannot censor the reconfiguration request. The certificate Σ' attests that a quorum of replicas voted to deliver the set; therefore, correct replicas will eventually deliver the same set. If the certificates are valid, the receiving replica appends the union of $\overline{recs}$ to $operations_i$, and the certificates to $certs$ (at line 28-30). The BRD module may complain if the leader does not lead delivery in a timely manner (at line 31-32). The complaint is forwarded to the local leader election module le.

Byzantine Reliable Dissemination. In this section, we present the Byzantine Reliable Dissemination (BRD) protocol that we just used. We present it as a general reusable module, that is of independent interest.

Module. BRD accepts a $broadcast(m)$ request from each replica. It then collects and disseminates messages m. It issues a response $deliver(M, \Sigma)^{\Sigma'}$ where M is a set of messages, and Σ and Σ' are two sets of signatures. The certificate Σ attests that M is a set of messages from a quorum of replicas, and the certificate Σ' attests that M is the only delivered set, and every correct replica will eventually deliver it. In our reconfiguration protocol, these certificate are sent to other clusters as a proof of these properties for the dissemination in the current cluster. Further, the component may issue a

Algorithm 5: BRD (1/2)

1 **request** : $broadcast(m)$, $new\text{-}leader(p, ts)$
2 **response** : $deliver(\{\overline{m}\}, \Sigma)$, $complain(p)$
3 **uses**:
4 apl : AuthenticatedPoint2PointLink
5 $abeb$: AuthenticatedBestEffortBroadcast
6 **vars**:
7 $(leader, ts) \leftarrow (p_0, 0)$
8 $my\text{-}m \leftarrow \bot$
9 $echoed, readied, delivered \leftarrow$ false ▷ Tracking reliable delivery
10 $valid, high\text{-}valid \leftarrow \bot$ ▷ Validated set of requests
11 $q, M, \Sigma \leftarrow \emptyset$ ▷ Collected senders, messages, and signatures
12 $timer \leftarrow \Delta$
13 **upon request** $broadcast(m)$
14 $my\text{-}m \leftarrow m$
15 apl **request** $send(leader, \langle m, ts \rangle)$
16 reset $timer$
17 **upon** apl **response** $deliver(p, \langle m, t \rangle^{\sigma})$ where ***self*** $= leader \wedge t = ts$
18 $q \leftarrow q \cup \{p\}$
19 $M \leftarrow M \cup \{m\}$
20 $\Sigma \leftarrow \Sigma \cup \{\sigma\}$
21 **upon** $|q| \geq 2 \times f + 1 \wedge high\text{-}valid = \bot$
22 $abeb$ **request** $broadcast(Agg(M, \Sigma, ts))$
23 **upon** $abeb$ **response** $deliver(p, Agg(M, \Sigma, t))$ where $p = leader \wedge t = ts \wedge \neg echoed \wedge \Sigma$ attests M (*i.e.*, Σ has either at least $2 \times f + 1$ signatures for M, at least $2 \times f + 1$ $Echo(M)$ messages, or $f + 1$ $Ready(M)$ messages)
24 $echoed \leftarrow$ true
25 $abeb$ **request** $broadcast(Echo(M, ts))$
26 **upon** $abeb$ **response** $deliver(\overline{p}, Echo(M, t)^{\overline{\sigma}})$ where $|\{\overline{p}\}| \geq 2 \times f + 1 \wedge t = ts \wedge \neg readied$
27 $readied \leftarrow$ true
28 $abeb$ **request** $broadcast(Ready(M, ts))$
29 $valid \leftarrow \langle M, \overline{\sigma}, ts \rangle$

Algorithm 6: BRD (2/2)

30 **upon** $abeb$ **response** $deliver(\overline{p}, Ready(M, t)^{\overline{\sigma}})$ where $|\{\overline{p}\}| \geq f + 1 \wedge t = ts \wedge \neg readied$
31 $readied \leftarrow$ true
32 $abeb$ **request** $broadcast(Ready(M, ts))$
33 $valid \leftarrow \langle M, \overline{\sigma}, ts \rangle$
34 **upon** $abeb$ **response** $deliver(\overline{p}, Ready(M, t)^{\overline{\sigma}})$ where $|\{\overline{p}\}| \geq 2 \times f + 1 \wedge t = ts \wedge \neg delivered$
35 $delivered \leftarrow$ true
36 **response** $deliver(M, \Sigma)^{\overline{\sigma}}$
37 stop $timer$
38 **upon** $timer$ expires
39 **response** $complain(leader)$
40 **upon request** $new\text{-}leader(p, t)$
41 $(leader, ts) \leftarrow (p, t)$
42 $echoed, readied \leftarrow$ false
43 $valid, high\text{-}valid \leftarrow \bot$
44 $q, M, \Sigma \leftarrow \emptyset$
45 reset $timer$
46 **if** $valid \neq \bot$ **then**
47 apl **request** $send(leader, Valid(valid))$
48 **else**
49 **if** $my\text{-}m \neq \bot$ **then**
50 apl **request** $send(leader, \langle my\text{-}m, ts \rangle)$
51 **upon** apl **response** $deliver(p, Valid(M, \Sigma, t))$ where ***self*** $= leader \wedge \Sigma$ attests M (*i.e.*, Σ has at least $2 \times f + 1$ $Echo(M)$ messages or $f + 1$ $Ready(M)$ messages)
52 **let** $\langle \_, \_, ht \rangle \coloneqq high\text{-}valid$ **in**
53 **if** $t > ht$ **then** $high\text{-}valid \leftarrow \langle M, \Sigma, t \rangle$
54 $q \leftarrow q \cup \{p\}$
55 **upon** $|q| \geq 2 \times f + 1 \wedge high\text{-}valid \neq \bot$
56 **let** $\langle M, \Sigma, \_ \rangle \coloneqq high\text{-}valid$ **in**
57 $abeb$ **request** $broadcast(Agg(M, \Sigma, ts))$

$complain(p)$ event to complain about the current leader p, and accepts a $new\text{-}leader(p, ts)$ request to set a new leader p with a timestamp ts. Leaders are elected with monotonically increasing timestamps. BRD guarantees the following properties. Integrity: A correct replica may only deliver messages from at least a quorum of replicas. No duplication: Every correct replica delivers at most one set of messages. Uniformity: No two correct replicas deliver different set of messages. Termination: If all correct replicas broadcast messages, then every correct replica eventually delivers a set of messages. Totality: If a correct replica delivers a set of messages, then all correct replicas deliver a set of messages. Validity: If a correct replica delivers a set of messages containing m from a correct sender p, then m was broadcast by p.

Protocol. As Alg. 5 presents, when a replica broadcasts a message (at line 13), it stores it and sends it to the leader (at line 14-15). It also resets the timer to watch the leader (at line 16). The leader adds messages and the accompanying signatures that it receives (at line 17) to the set of messages M and signatures Σ (at line 18-20). Once it collects messages from a quorum (at line 21), it broadcasts an aggregation message Agg containing M and Σ (at line 22). Massages carry the timestamp ts of the current $leader$ as well; any message with a stale timestamp is ignored. Upon delivery of the aggregation (at line 23), a correct replica accepts it if M is attested by accompanying signatures Σ. The signatures Σ attest M if they include at least a quorum of signatures for the messages M. The signatures serve as a proof that the $leader$ has genuinely collected messages from at least a quorum. Therefore, the leader cannot drop the reconfiguration request of a replica that has reached out to at least a quorum. For example in Fig. 3a and 3b, the quorum that p'_{new} stored the request at, and the quorum that the leader p_2 receives requests from intersect in the correct replica p_3. Even though the leader p_2 is Byzantine, and sends the aggregated set to only a subset of replicas $\{p_1, p_4\}$, it cannot drop reconfigurations from the aggregated set.

If the accepting replica hasn't sent the $Echo$ message, it records (in the variable $echoed$) that it is sending it, and broadcasts the $Echo$ message (at line 24-25). In Fig. 3b, the correct replicas p_1 and p_4 that receive an attested set of messages from the leader echo it. Upon delivery of an $Echo$ message from a quorum, if the receiving replica has not sent $Ready$ messages (at line 26), it records (in the variable $readied$) that it is sending it, and then broadcasts a $Ready$ message (at line 27-28). In Fig. 3b, replicas p_1 and p_4 receive a quorum of 3 $Echo$ messages, and broadcast $Ready$.

If the leader changes during the broadcast, some correct replica might have delivered the aggregated messages while others may have not. Thus, to preserve the uniformity of delivered messages across replicas, the new leader should retrieve the previously delivered messages, and rebroadcast them. Thus, when a replica accepts a sufficiently echoed set, it stores it together with its accompanying signatures, as $valid$ (at line 29), and later forwards it to a new leader. In Fig. 3b,

p_1 and p_4 record a $valid$ set at the end of the $Echo$ step.

When a replica receives at least $f + 1$ $Ready$ messages (at line 30), at least one of them is correct and has received at least a quorum of $Echo$ messages. Therefore, the replica trusts the $Ready$ message and amplified it: it records that it is sending it, and broadcasts a $Ready$ message (at line 31-32). It also records the received messages M and signatures of the received $Ready$ messages as $valid$ (at line 33), and later forwards it to a new leader.

Finally, when a replica receives a quorum of $Ready$ messages, and it has not delivered the aggregated messages yet (at line 34), it records (in the variable $delivered$) that it is delivering, delivers the aggregated messages M, and stops the timer (at line 35-37). If a replica does not deliver the aggregated messages before the timer times out, it complains about the current leader (at line 38-39). In Fig. 3b, the correct replica p_1 receives a quorum of 3 $Ready$ messages, and delivers the reconfigurations (at the black circle). However, the other correct replicas don't receive enough $Ready$ messages, complain about the leader, and eventually change the leader to the correct replica p_3 (at the red circles).

To preserve uniformity, the new leader should retrieve the set of reconfigurations that have been previously delivered. When a replica is informed of a new leader (at line 40), it records the new leader and timestamp, resets the state and the timer (at line 41-45), and then sends a message to the new leader to inform him about the current state of dissemination. If a $valid$ set of messages is recorded during the execution with the previous leaders, the replica sends it to the new leader (at line 47). Otherwise, it sends the message that it originally broadcast (line 13-15) to the current leader (at line 50). In Fig. 3b, the two replicas p_2 and p_3 send to the new leader the set of reconfigurations that they had collected and sent to the previous leader. However, p_4 has a $valid$ set of reconfigurations and sends them to the leader.

Let l be the latest leader with the timestamp ts that has guided the system to delivery of a set M at a correct replica. Consider the next leader l' with the timestamp ts'. To preserve uniformity, l' should adopt M. In order to find M, l' waits to receive messages from a quorum of replicas, and then picks the $valid$ set with the largest timestamp. Let us explain why. The set M was delivered only after a quorum of $Ready$ messages was received. At least $f + 1$ of the senders are correct. A correct replica sends a $Ready$ message only after receiving $2 \times f + 1$ $Echo$ messages, or $f + 1$ $Ready$ messages. In both of those cases, the receiving replica stores M with ts as $valid$. Thus, at least $f + 1$ correct replicas P have stored M with ts as $valid$. Therefore, if l' receives messages from a quorum $(2 \times f + 1)$ of replicas, and retrieves any $valid$ sets, then M with the largest timestamp ts is retrieved from at least one replica in P. The leader l' adopts and broadcasts M. Even if it does not lead to any new delivery of M, any $valid$ set that is stored under his leadership will have the same set M with now the larger timestamp ts'.

When the leader receives a $valid$ set (at line 51), it checks that the accompanying signatures attest its validity: there are at least $2 \times f + 1$ signatures of $Echo$ messages, or $f + 1$ signatures of $Ready$ messages. The leader keeps the $valid$ set with the highest timestamp as $high\text{-}valid$ (at line 52-53). Finally, when the leader has collected messages from a quorum, if it has received a $valid$ set (at line 55), it broadcasts $high\text{-}valid$ (at line 57). Otherwise, similar to the first leader (at line 21), it broadcasts the aggregated messages. In Fig. 3b, the new correct leader p_3 waits for 3 messages, adopts the $valid$ set that p_4 sends, goes through the $Echo$ and $Ready$ steps, and makes the remaining correct replicas p_3 and p_4 deliver the same set (at black circles).

V. Correctness

We now state the correctness properties of the sub-protocols and then the end-to-end protocol. The proofs are available in the extended report [39].

Remote Leader Change.

Lemma 1 (Eventual Succession)**.** *Let ops be the locally replicated operations of a cluster C in a round. Either ops are eventually delivered to all correct processes of every other cluster in that round, or correct processes in C eventually adopt a new leader.*

Lemma 2 (Eventual Agreement)**.** *All correct processes in the same cluster eventually adopt the same leader.*

Lemma 3 (Overthrow Resistance)**.** *A correct process does not adopt a new leader unless at least one correct process complains about the previous leader.*

Inter-cluster Broadcast.

Lemma 4 (Termination)**.** *In every round, every correct process eventually receives operations from each other cluster.*

Lemma 5 (Agreement)**.** *In every round, the operations that every pair of correct processes receive from a cluster are the same.*

Byzantine Reliable Dissemination.

Lemma 6 (Integrity)**.** *Every delivered set contains at least a quorum of messages from distinct processes.*

Lemma 7 (Termination)**.** *If all correct processes broadcast messages then every correct process eventually delivers a set of messages.*

Lemma 8 (Uniformity)**.** *No correct pair of processes deliver different sets of messages.*

Lemma 9 (No duplication)**.** *Every correct process delivers at most one set of messages.*

Lemma 10 (Validity)**.** *If a correct process delivers a set of messages containing m from a correct sender p, then m was broadcast by p.*

Reconfiguration Properties.

Lemma 11 (Completeness)**.** *If a correct process p requests to join (or leave) cluster i, then every correct process will eventually have a configuration C such that $p \in C$ (or $p \notin C$).*

Lemma 12 (Accuracy)**.** *Consider a correct process p that has a configuration C in a round, and then another configuration*

ms	US	EU	Asia
US	0	148	214
EU	148	0	134
Asia	214	134	0

Table I: Inter-region round-trip latency for three regions: US (us-west1-b), EU (europe-west3-c), Asia (asia-south1-c).

C' in a later round. If a correct process $p \in C'_i \setminus C_i$, then p requested to join the cluster i. Similarly, if a correct process $p \in C_i \setminus C'_i$, then p requested to leave the cluster i.

Lemma 13 (Uniformity)**.** *In every round, the configurations that every pair of correct processes execute are the same.*

Reconfigurable Clustered Replication.

Theorem 1 (Validity)**.** *Every operation that a correct process requests is eventually executed by a correct process.*

Theorem 2 (Agreement)**.** *If a correct process executes an operation in a round then every correct process executes that operation in the same round.*

Theorem 3 (Total-order)**.** *For every pair of operations o and o', if a correct process executes only o, or executes o before o', then every correct process executes o' only after o.*

VI. Experimental Results

Implementation. The clustered replication protocol is parametric with respect to the local replication protocol. We instantiated it for both HotStuff [9] and BFTSmart [34] as the local replication protocol to implement replicated systems that we call AVA. We refer to the two as Ava-HotStuff (A.H) and Ava-BftSmart (A.B). We have released all the code and workloads as open source software (https://icdeava.github.io).

Questions. We perform experiments to answer the following questions: (E0-E2): How does clustered replication impact performance? (E3) What is the impact of introducing heterogeneity in the clusters on the performance of clustered replication? (E4) What is the impact of failures on performance? We are especially interested in leader failures. (E5) What is the impact of reconfiguration requests on performance?

Platform. We used Google cloud compute to deploy instances acting as servers and clients in our system. Each instance runs Ubuntu Server 22.04 LTS, and has a uni-core processor with 16GB of main memory. We deploy our framework globally on nodes across 3 Google compute regions, namely US (us-west1-b), Asia (asia-south1-c) and Europe (europe-west3-b). The inter-region network latency is presented in table I. For both Ava-HotStuff and Ava-BftSmart, we choose the YCSB benchmark with a 85% read and 15% write ratio. We deployed one client per cluster with multiple threads that issued its requests with the Zipfian distribution one after the other without any delays. We batched transactions (to batches of size 100) in each round. We issued operations of size 1KB. All experiments were run for 3 minutes and the results were taken from the last minute.

E0. Multi-cluster Single-region. We investigate the impact of multi-cluster deployment in one Google region on throughput and latency. We keep the total number of nodes constant (96), and divide them to different number of clusters (2, 3, 4, 6, 8, 10, 12). Fig. 4 reports the effect of the number of clusters on throughput and latency. (In the throughput plot, the left y-axis is for Ava-BftSmart and the right y-axis is for Ava-HotStuff). *Assessment.* We observe that as the number of clusters increases, the throughput of both Ava-HotStuff and Ava-BftSmart increases. Ava-HotStuff exhibits higher throughput than Ava-BftSmart. We observe that as the number of clusters increase, the latency decreases for both Ava-HotStuff and Ava-BftSmart. Ava-BftSmart exhibits lower latency than Ava-HotStuff. As the number of clusters increase, each cluster has fewer nodes, and local replication is more efficient, and further, clusters execute the divided workload in parallel. Thus, the throughput and latency of local replication is improved that in turn improves the end-to-end throughput and latency. The two AVA implementations outperform non-clustered replication for both throughput and latency.

E1. Multi-cluster Multi-region. We study the impact of deploying clusters in multiple Google regions on the throughput and latency in Fig. 4. We equally split 96 nodes into different number of clusters (2, 3, 4, 6, 8, 12), and host them on 3 regions. A cluster is completely hosted on a single region. For example, for the 4 clusters setup, we divide the 96 nodes into 4 clusters of 24 nodes where the first region hosts two clusters, and the second and third regions host one cluster each. *Assessment.* Similar to the previous experiment, as the number of clusters increase, the throughput increases and the latency decreases for both systems. Similarly, since the number of nodes per cluster decreases, and the workload is divided between clusters, the throughput and latency are improved. However, overall performance is lower than the previous experiment since inter-cluster communication across regions is slower than within one region. We observe that with multiple regions, Ava-HotStuff and Ava-BftSmart clustered replication still outperform non-clustered replication for both throughput and latency.

E2. Latency Breakdown. In Fig. 5a, we show the latency breakdown for processing transactions. We report the average latency for read and write transactions. Read transaction have lower latency than writer transactions since the former can be immediately processed but the latter go through three stages. We experiment with 3 clusters each containing 4 nodes in three setups where clusters span one (Asia), two (EU and Asia), and three (EU, Asia, US) regions. With one region, the bottleneck is the local ordering as it involves 4 rounds of messages. On the other hand, the inter-cluster broadcast that involves one round of messages is relatively a smaller part. With two regions, the latency is dominated by the inter-cluster broadcast when messages have to travel across regions. With three regions, the inter-cluster broadcast is still the dominating part, and is further increased. As shown in Table I, the round-trip time for EU and Asia is about 134, but when US is added, it is about 214. This experiment shows that it is crucial to minimize cross-region messaging as our clustered protocol does.

E3. Heterogeneity in Clusters. We investigate the impact of heterogeneity on throughput and latency for Ava-

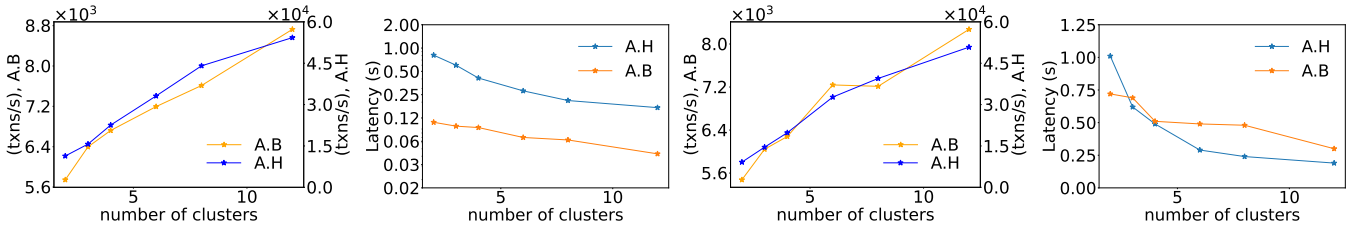


Fig. 4: Throughput and latency as a function of number of clusters (with 96 nodes) E0. in the same region (two left plots) and E1. across regions (two right plots). (In the throughput plot, the left y-axis is for AVA-BFTSMART and the right y-axis is for AVA-HOTSTUFF).

HOTSTUFF in Fig. 5b and 5c, and for AVA-BFTSMART in 5d and 5e. Consider 9 nodes in Asia (ap-south-1) and 5 nodes in EU (eu-central-1) regions. We consider a scale factor s of these numbers varying from 1 to 5. For example, with scale factor 2, we have $2 \times 9 = 18$ nodes in Asia, and $2 \times 5 = 10$ nodes in EU. For each scale, we consider 3 setups: (1) Equal sized clusters. C_1: 7 in Asia. C_2: 2 in Asia and 5 in EU. (2) Partition based on region. C_1: 9 in Asia. C_2: 5 in EU. (3) Partition based on region, and within region. C_1: 5 in Asia. C_2: 4 in Asia. C_3: 5 in EU. In contrast to previous works, AVA supports the heterogeneous setups 2 and 3. *Assessment.* For both AVA-HOTSTUFF and AVA-BFTSMART, setup 2 exhibits higher throughput and lower latency than setup 1 especially at higher scales. The setup 2 exploits heterogeneity to host all members of each cluster in the same region. Therefore, it deceases the the cost of local replication. Similarly, setup 3 exhibits higher throughput and lower latency than setup 2 at higher scales. The setup 3 splits a cluster into two smaller clusters in the same region. Therefore, it further deceases the the cost of local replication. Further, the general trend is that throughput and latency are better at lower scale factors, since they have lower cost of local replication.

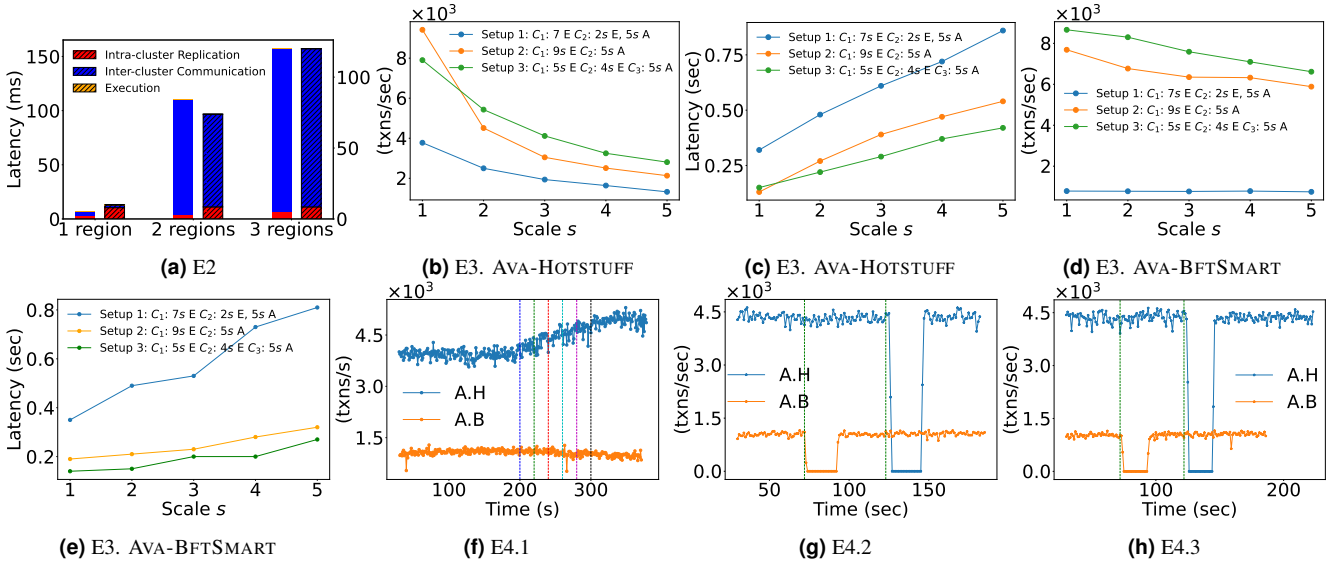
E4. Failures. We investigate the impact of failures on the performance. We measure the performance for 2 clusters with 10 nodes per cluster ($f_1 = f_2 = 3$). We consider three failure scenarios: *(1) Up to f non-leader failures.* In Fig. 5f, we test the resiliency of both AVA systems by failing up to f non-leader nodes in each cluster. The vertical lines show the failure time. The system tolerates the failures, and remains functional. As a side effect, after the recovery, the throughput can slightly increase since local replication is more efficient with fewer number of nodes. *(2) Leader Failure.* In Fig. 5g, we fail the leader of a cluster. After a short window, the leader is properly changed, and the throughput is recovered to the same level. The timeout for leader change can be adjusted according to the local network latency. This experiment set it to 20 second; thus, the window to complete the leader change is slightly more than 20 seconds. *(3) Byzantine Leader and Remote Leader Change.* We inject Byzantine behavior into leaders to trigger remote leader change. We make the leader replica complete the first stage within its cluster as a correct leader, but avoid sending inter-cluster broadcast messages. As we can see in Fig. 5h, after a short period, the leader is properly changed, and the throughput comes back up. The 20 seconds period is the adjustable timeout for multi-cluster message-passing.

E5. Reconfiguration Requests. We investigate the impact of reconfiguration on the performance in two experiments. *(1)* In a system of two clusters with 7 nodes each, we issue three join and three leave requests to each cluster at the vertical lines. The requesting nodes can properly join and leave the clusters. Fig. 6a presents the throughput of the whole system during the reconfigurations. We observe that the throughput slightly decreases as nodes have to communicate with the requesting node in addition to processing transactions. Further, for joins, the throughput has a slightly decreasing trend since the local ordering stage is less efficient in larger clusters. However, for leaves, the throughput stays steady. *(2)* As we described in § II, the protocol takes reconfigurations off the critical path that orders transactions, and processes them in a parallel workflow. In this experiment, we compare the parallel workflows with a single workflow that processes reconfigurations in the same sequence as transactions. We setup connections between replicas of two clusters with 10 and 8 nodes, respectively, and 3 clients: one client for each cluster that issues write-only transactions, one dedicated client that issues join and leave requests. A node is repeatedly made to join and leave the system. Fig. 6b shows the throughput of both AVA-HOTSTUFF and AVA-BFTSMART and their single workflow versions. The configuration starts for BftSmart and Hotstuff at 60 sec and 115 sec respectively. We find that the parallel workflows outperform the single workflow in both systems. In a single workflow, the reconfigurations take slots from transactions and are processed in sequence. In contrast, AVA processes them in parallel with transactions, and collects them as a set rather than ordering them individually.

VII. RELATED WORK

Classical Replication. Since PBFT [7], the first practical Byzantine replication protocol, the followup works [40], [9], [41], [42], [43], [44], [45], [46], [47], [48] improve different aspects of Byzantine consensus not only for partially synchronous but also asynchronous networks. However, these protocols can only work with fixed membership: the set of participants is fixed and known to all participants at the outset. In contrast, our paper proposes a clustered replication protocol whose members can be reconfigured at runtime.

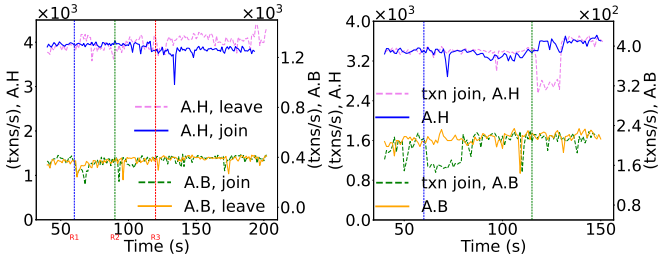
Clustered Replication. Clustered replication systems divide replicas into small clusters, and perform consensus within local clusters in parallel. Compared to non-clustered replication, clustered replication has fewer number of replicas in each cluster; therefore, it exhibits improved performance and scalability. Steward [24] implements a replication protocol where replicas are partitioned into multiple sites. A leader site is responsible for driving an inter-site coordination protocol similar to Paxos [49], which may become the bottleneck. The



(a) E2 **(b)** E3. AVA-HOTSTUFF **(c)** E3. AVA-HOTSTUFF **(d)** E3. AVA-BFTSMART

(e) E3. AVA-BFTSMART **(f)** E4.1 **(g)** E4.2 **(h)** E4.3

Fig. 5: E2. Latency breakdown for AVA-BFTSMART (left) and AVA-HOTSTUFF (right and shaded) in (a). 1 region: Asia, 2 regions: EU and Asia, and 3 regions: EU, Asia, US. E3. Impact of heterogeneity on throughput and latency for AVA-HOTSTUFF in (b) and (c) and for AVA-BFTSMART in (d) and (e). E4. 1. The Impact of multiple non-leader failure on throughput in (f). 2. The Impact of leader failure on throughput in (g). 3. The impact of remote leader change on throughput in (h).



(a) E5.1 **(b)** E5.2

Fig. 6: E5. 1. The impact of multiple reconfigurations on throughput in (a). 2. The impact of the parallel workflows on on throughput in (b). (The left y-axis is for AVA-HOTSTUFF and the right y-axis is for AVA-BFTSMART.)

inspiring work GeoBFT [25] alleviates the need for a leader site, and enables higher throughput by letting clusters process their own transactions, and then propagate them. However, it does not support reconfiguration. Our clustered replication protocol supports heterogeneity and reconfiguration across clusters which allows more flexible and efficient setups.

Another line of work is sharding-based consensus [50], [51], [52]. Elastico [53] presents a sharding-based consensus protocol for permissionless blockchains. OmniLedger [21] and RapidChain [22] support reconfiguration for sharding-based consensus. OmniLedger and RapidChain are linearly scalable; but, they suffer from replay attacks in cross-shard commit protocols [54]. In contrast, our protocol provides full replication and avoids complications of cross-shard synchronization.

Group and Open Membership. A group membership service maintains the set of active replicas by installing new views. Since accurate membership is as strong as consensus [55], [56], classical [57], [31], [32], [33], [58], [59], [60], [61] group membership and reconfiguration protocols use consensus to reach an agreement on membership and adjust quorums accordingly.

SmartMerge [29] provides replication, and uses a commutative, associative and idempotent merge function on reconfiguration requests to avoid consensus. It ensures that all replicas eventually perform the merge of all the reconfiguration requests. Dyno [28] provides replication and group membership in the primary partition model. Similar to [62], it uses an instance of consensus to order reconfiguration requests. In contrast to SmartMerge and Dyno, AVA presents reconfiguration for clustered replication systems without relying on a single instance of consensus to safely apply reconfiguration request, a bottleneck that is further amplified in a wide area network. Furthermore, SmartMerge reliance on eventual consistency is insufficient.

Solida [17], Hybird Consensus [63], Tendermint [18], [64], Casper [19], Algorand [20], RapidChain [22], and OmniLedeger [21] blockchains combine permissionless and permissioned (Byzantine) consensus to provide both efficiency and dynamic membership [65]. They use permissionless consensus, computational puzzles, or verifiable random functions to dynamically choose validators for permissioned consensus. Related works further provide reconfiguration for crash fault-tolerant consensus protocols [62], [66], and reconfiguration protocols for random beacons [67], [68]. In contrast, we focus a reconfiguration protocol fault-tolerant clustered setting.

VIII. CONCLUSION

We presented heterogeneous and reconfigurable clustered replication. It presented a protocol that adapts to different cluster sizes, and allows replicas to join and leave clusters efficiently. Further, it stated and proved the safety and liveness properties of the protocol. It implemented the protocol, built two clustered replicated systems, and empirically showed that they can be efficiently reconfigured, and their heterogeneity significantly improves performance.

REFERENCES

[1] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," *White paper*, 2008.

[2] G. Wood *et al.*, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum project yellow paper*, vol. 151, no. 2014, pp. 1–32, 2014.

[3] M. Tran, I. Choi, G. J. Moon, A. V. Vu, and M. S. Kang, "A stealthier partitioning attack against bitcoin peer-to-peer network," in *2020 IEEE symposium on security and privacy (SP)*. IEEE, 2020, pp. 894–909.

[4] A. Gervais, G. O. Karame, K. Wüst, V. Glykantzis, H. Ritzdorf, and S. Capkun, "On the security and performance of proof of work blockchains," in *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, 2016, pp. 3–16.

[5] M. Saad and D. Mohaisen, "Three birds with one stone: Efficient partitioning attacks on interdependent cryptocurrency networks," in *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2023, pp. 111–125.

[6] A. Wahrstätter, J. Ernstberger, A. Yaish, L. Zhou, K. Qin, T. Tsuchiya, S. Steinhorst, D. Svetinovic, N. Christin, M. Barczentewicz *et al.*, "Blockchain censorship," in *Proceedings of the ACM on Web Conference 2024*, 2024, pp. 1632–1643.

[7] M. Castro, B. Liskov *et al.*, "Practical byzantine fault tolerance," in *OSDI*, vol. 99, no. 1999, 1999, pp. 173–186.

[8] A. Miller, Y. Xia, K. Croman, E. Shi, and D. Song, "The honey badger of bft protocols," in *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, 2016, pp. 31–42.

[9] M. Yin, D. Malkhi, M. K. Reiter, G. G. Gueta, and I. Abraham, "Hotstuff: Bft consensus with linearity and responsiveness," in *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, 2019, pp. 347–356.

[10] A. Spiegelman, N. Giridharan, A. Sonnino, and L. Kokoris-Kogias, "Bullshark: Dag bft protocols made practical," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 2705–2718.

[11] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich *et al.*, "Hyperledger fabric: a distributed operating system for permissioned blockchains," in *Proceedings of the thirteenth EuroSys conference*, 2018, pp. 1–15.

[12] C. Stathakopoulou, T. David, and M. Vukolic, "Mir-bft: High-throughput bft for blockchains," *arXiv preprint arXiv:1906.05552*, p. 92, 2019.

[13] M. J. Amiri, D. Agrawal, and A. E. Abbadi, "Caper: a cross-application permissioned blockchain," *Proceedings of the VLDB Endowment*, vol. 12, no. 11, pp. 1385–1398, 2019.

[14] S. Gupta, J. Hellings, S. Rahnama, and M. Sadoghi, "Proof-of-execution: Reaching consensus through fault-tolerant speculation," *arXiv preprint arXiv:1911.00838*, 2019.

[15] P. Ruan, T. T. A. Dinh, D. Loghin, M. Zhang, G. Chen, Q. Lin, and B. C. Ooi, "Blockchains vs. distributed databases: Dichotomy and fusion," in *Proceedings of the 2021 International Conference on Management of Data*, 2021, pp. 1504–1517.

[16] P. Ruan, T. T. A. Dinh, Q. Lin, M. Zhang, G. Chen, and B. C. Ooi, "Lineagechain: a fine-grained, secure and efficient data provenance system for blockchains," *The VLDB Journal*, vol. 30, pp. 3–24, 2021.

[17] I. Abraham, D. Malkhi, K. Nayak, L. Ren, and A. Spiegelman, "Solida: A blockchain protocol based on reconfigurable byzantine consensus," *arXiv preprint arXiv:1612.02916*, 2016.

[18] E. Buchman, "Tendermint: Byzantine fault tolerance in the age of blockchains," Ph.D. dissertation, University of Guelph, 2016.

[19] V. Buterin and V. Griffith, "Casper the friendly finality gadget," *arXiv preprint arXiv:1710.09437*, 2017.

[20] Y. Gilad, R. Hemo, S. Micali, G. Vlachos, and N. Zeldovich, "Algorand: Scaling byzantine agreements for cryptocurrencies," in *Proceedings of the 26th symposium on operating systems principles*, 2017, pp. 51–68.

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

[22] M. Zamani, M. Movahedi, and M. Raykova, "Rapidchain: A fast blockchain protocol via full sharding." *IACR Cryptol. ePrint Arch.*, vol. 2018, p. 460, 2018.

[23] C. Cachin, G. Losa, and L. Zanolini, "Quorum systems in permissionless networks," in *26th International Conference on Principles of Distributed Systems*, 2023.

[24] Y. Amir, C. Danilov, D. Dolev, J. Kirsch, J. Lane, C. Nita-Rotaru, J. Olsen, and D. Zage, "Steward: Scaling byzantine fault-tolerant replication to wide area networks," *IEEE Transactions on Dependable and Secure Computing*, vol. 7, no. 1, pp. 80–93, 2008.

[25] S. Gupta, S. Rahnama, J. Hellings, and M. Sadoghi, "Resilientdb: Global scale resilient blockchain fabric," *Proceedings of the VLDB Endowment*, vol. 13, no. 6, 2020.

[26] S. Gupta, J. Hellings, and M. Sadoghi, "Rcc: Resilient concurrent consensus for high-throughput secure transaction processing," in *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 2021, pp. 1392–1403.

[27] G. Danezis, L. Kokoris-Kogias, A. Sonnino, and A. Spiegelman, "Narwhal and tusk: a dag-based mempool and efficient bft consensus," in *Proceedings of the Seventeenth European Conference on Computer Systems*, 2022, pp. 34–50.

[28] S. Duan and H. Zhang, "Foundations of dynamic bft," in *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 2022, pp. 1546–1546.

[29] L. Jehl, R. Vitenberg, and H. Meling, "Smartmerge: A new approach to reconfiguration for atomic storage," in *International Symposium on Distributed Computing*. Springer, 2015, pp. 154–169.

[30] S. Duan, H. Meling, S. Peisert, and H. Zhang, "Bchain: Byzantine replication with high throughput and embedded reconfiguration," in *Principles of Distributed Systems: 18th International Conference, OPODIS 2014, Cortina d'Ampezzo, Italy, December 16-19, 2014. Proceedings 18*. Springer, 2014, pp. 91–106.

[31] L. Lamport, D. Malkhi, and L. Zhou, "Reconfiguring a state machine," *ACM SIGACT News*, vol. 41, no. 1, pp. 63–73, 2010.

[32] L. LAMPORT, "The part-time parliament," *ACM Transactions on Computer Systems*, vol. 16, no. 2, pp. 133–169, 1998.

[33] R. Guerraoui, N. Knežević, V. Quéma, and M. Vukolić, "The next 700 bft protocols," in *Proceedings of the 5th European conference on Computer systems*, 2010, pp. 363–376.

[34] A. Bessani, J. Sousa, and E. E. Alchieri, "State machine replication for the masses with bft-smart," in *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE, 2014, pp. 355–362.

[35] T. Rajabi, A. A. Khalil, M. H. Manshaei, M. A. Rahman, M. Dakhilalian, M. Ngouen, M. Jadliwala, and A. S. Uluagac, "Feasibility analysis for sybil attacks in shard-based permissionless blockchains," *Distributed Ledger Technologies: Research and Practice*, vol. 2, no. 4, pp. 1–21, 2023.

[36] N. Tran, J. Li, L. Subramanian, and S. S. Chow, "Optimal sybil-resilient node admission control," in *2011 Proceedings IEEE INFOCOM*. IEEE, 2011, pp. 3218–3226.

[37] X. Zhang, H. Zheng, X. Li, S. Du, and H. Zhu, "You are where you have been: Sybil detection via geo-location analysis in osns," in *2014 IEEE Global Communications Conference*. IEEE, 2014, pp. 698–703.

[38] C. Dwork, N. Lynch, and L. Stockmeyer, "Consensus in the presence of partial synchrony," *Journal of the ACM (JACM)*, vol. 35, no. 2, pp. 288–323, 1988.

[39] T. Mane, X. Li, M. Sadoghi, and M. Lesani, "AVA: reconfigurable fault-tolerant geo-replication on heterogeneous clusters," *arXiv preprint, https://icdeava.github.io*, 2024.

[40] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong, "Zyzzyva: speculative byzantine fault tolerance," in *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, 2007, pp. 45–58.

[41] I. Abraham, K. Nayak, and N. Shrestha, "Optimal good-case latency for rotating leader synchronous bft," *Cryptology ePrint Archive*, 2021.

[42] Z. Xiang, D. Malkhi, K. Nayak, and L. Ren, "Strengthened fault tolerance in byzantine fault tolerant replication," in *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2021, pp. 205–215.

[43] G. G. Gueta, I. Abraham, S. Grossman, D. Malkhi, B. Pinkas, M. Reiter, D.-A. Seredinschi, O. Tamir, and A. Tomescu, "Sbft: A scalable and decentralized trust infrastructure," in *2019 49th Annual IEEE/IFIP international conference on dependable systems and networks (DSN)*. IEEE, 2019, pp. 568–580.

[44] S. Duan, M. K. Reiter, and H. Zhang, "Beat: Asynchronous bft made practical," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 2028–2041.

[45] H. Zhang and S. Duan, "Pace: Fully parallelizable bft from reproposable byzantine agreement," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 3151–3164.

[46] X. Wang, H. Wang, H. Zhang, and S. Duan, "Pando: Extremely scalable bft based on committee sampling," *Cryptology ePrint Archive*, 2024.

[47] B. Guo, Z. Lu, Q. Tang, J. Xu, and Z. Zhang, "Dumbo: Faster asynchronous bft protocols," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 803–818.

[48] B. Guo, Y. Lu, Z. Lu, Q. Tang, J. Xu, and Z. Zhang, "Speeding dumbo: Pushing asynchronous bft closer to practice," *Cryptology ePrint Archive*, 2022.

[49] L. Lamport, "Paxos made simple," *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*, pp. 51–58, 2001.

[50] M. J. Amiri, D. Agrawal, and A. El Abbadi, "Sharper: Sharding permissioned blockchains over network clusters," in *Proceedings of the 2021 international conference on management of data*, 2021, pp. 76–88.

[51] J. Hellings and M. Sadoghi, "Byshard: Sharding in a byzantine environment," *Proceedings of the VLDB Endowment*, vol. 14, no. 11, pp. 2230–2243, 2021.

[52] H. Dang, T. T. A. Dinh, D. Loghin, E.-C. Chang, Q. Lin, and B. C. Ooi, "Towards scaling blockchain systems via sharding," in *Proceedings of the 2019 international conference on management of data*, 2019, pp. 123–140.

[53] L. Luu, V. Narayanan, C. Zheng, K. Baweja, S. Gilbert, and P. Saxena, "A secure sharding protocol for open blockchains," in *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, 2016, pp. 17–30.

[54] A. Sonnino, S. Bano, M. Al-Bassam, and G. Danezis, "Replay attacks and defenses against cross-shard consensus in sharded distributed ledgers," in *2020 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2020, pp. 294–308.

[55] T. D. Chandra, V. Hadzilacos, S. Toueg, and B. Charron-Bost, "On the impossibility of group membership," in *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, 1996, pp. 322–330.

[56] G. V. Chockler, I. Keidar, and R. Vitenberg, "Group communication specifications: a comprehensive study," *ACM Computing Surveys (CSUR)*, vol. 33, no. 4, pp. 427–469, 2001.

[57] M. K. Reiter, "A secure group membership protocol," *IEEE Transactions on Software Engineering*, vol. 22, no. 1, pp. 31–42, 1996.

[58] R. Rodrigues, B. Liskov, K. Chen, M. Liskov, and D. Schultz, "Automatic reconfiguration for large-scale reliable storage systems," *IEEE Transactions on Dependable and Secure Computing*, vol. 9, no. 2, pp. 145–158, 2010.

[59] A. Bessani, E. Alchieri, J. Sousa, A. Oliveira, and F. Pedone, "From byzantine replication to blockchain: Consensus is only the beginning," in *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2020, pp. 424–436.

[60] M. Garcia, A. Bessani, and N. Neves, "Lazarus: Automatic management of diversity in bft systems," in *Proceedings of the 20th International Middleware Conference*, 2019, pp. 241–254.

[61] R. Van Renesse, C. Ho, and N. Schiper, "Byzantine chain replication," in *Principles of Distributed Systems: 16th International Conference, OPODIS 2012, Rome, Italy, December 18-20, 2012. Proceedings 16*. Springer, 2012, pp. 345–359.

[62] J. R. Lorch, A. Adya, W. J. Bolosky, R. Chaiken, J. R. Douceur, and J. Howell, "The smart way to migrate replicated stateful services," in *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, 2006, pp. 103–115.

[63] R. Pass and E. Shi, "Hybrid consensus: Efficient consensus in the permissionless model," *Cryptology ePrint Archive*, 2016.

[64] Y. Amoussou-Guenou, A. Del Pozzo, M. Potop-Butucaru, and S. Tucci-Piergiovanni, "Correctness of tendermint-core blockchains," in *22nd International Conference on Principles of Distributed Systems (OPODIS 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

[65] S. Bano, A. Sonnino, M. Al-Bassam, S. Azouvi, P. McCorry, S. Meiklejohn, and G. Danezis, "Sok: Consensus in the age of blockchains," in *Proceedings of the 1st ACM Conference on Advances in Financial Technologies*, 2019, pp. 183–198.

[66] A. Shraer, B. Reed, D. Malkhi, and F. P. Junqueira, "Dynamic {Reconfiguration} of {Primary/Backup} clusters," in *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, 2012, pp. 425–437.

[67] A. Bhat, N. Shrestha, Z. Luo, A. Kate, and K. Nayak, "Randpiper–reconfiguration-friendly random beacons with quadratic communication," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 3502–3524.

[68] A. Bhat, N. Shrestha, A. Kate, and K. Nayak, "Optrand: Optimistically responsive reconfigurable distributed randomness." in *NDSS*, 2023.

APPENDIX

IX. PROTOCOL STAGES

In the overview § II and Fig. 1, we explained the structure and the three stages of the protocol. We presented two sub-protocols in § III and § IV. In this section, we elaborate the other sub-protocols.

Stage 1: Stage 1 has two parallel parts. We saw the reconfiguration part in § IV. We now consider local ordering and leader change.

Local ordering. We consider local replication for transactions.

In order to process a transaction t, clients can issue a request $process(t)$ at any process of any cluster (at line 15), and will later receive a $return(t, v)$ response. Each cluster uses a total-order broadcast instance tob to propagate transactions to its processes in a uniform order. In addition to $broadcast$ requests and $deliver$ responses, the total-order broadcast abstraction can accept $new\text{-}leader(p, ts)$ requests to install the new leader p with the timestamp ts, and can issue $complain(p)$ responses to complain about the leader p. The protocol is parametric for the total-order broadcast. The total-order broadcast abstracts the classical non-clustered Byzantine replication protocols. If a complaint is received from tob, (at line 25), it is forwarded to the leader election module le (in Alg. 8).

Upon receiving a $process(t)$ request (at line 15), the process uses the tob to broadcast the transaction in its own cluster (at line 16). Each process stores the operations $operations_j$ that it receives from each cluster C_j. Upon delivery of a transaction t from the tob (at line 17), the process appends t to $operations_{\boldsymbol{i}}$ received from this cluster $C_{\boldsymbol{i}}$ (at line 18). Each process keeps the number $\boldsymbol{i}$ of the current cluster $C_{\boldsymbol{i}}$ that it is a member of. (We use the index $\boldsymbol{i}$ only for the current cluster, and the index j for other clusters.) The tob delivers a transaction t with a commit certificate σ that is the set of signatures of the quorum that committed t. Each process keeps the set of certificates $certs$ for the transactions committed in its local cluster (at line 19). In the next stage, the leader sends the transactions together with their certificates to other clusters. The certificates prevent Byzantine leaders from sending forged transactions.

In parallel to receiving $process(t)$ requests and ordering transactions t, processes can receive and propagate $join$ and $leave$ reconfiguration requests. We will describe the reconfiguration protocol in the next subsection. In each round, the processes of each cluster should agree on the reconfigurations before the end of the intra-cluster replication stage (phase 1). The reconfigurations are then propagated to other clusters in the inter-cluster broadcast stage (phase 2). In stage 1, a process collects the set of reconfiguration requests $recs$. It then calls the function $send\text{-}recs$ (at line 21) to send the set of reconfigurations it has collected to the leader who aggregates and uniformly replicates them. In § IV, we presented the Byzantine Reliable Dissemination component that collects and sends reconfigurations to the leader. A process calls this function towards the end of stage 1, *i.e.*, when a large fraction α of the transaction batch is already ordered (at line 20). This leaves ample time in the beginning of stage 1 to accept reconfiguration requests, and also leaves enough time at the end of stage 1 to reach agreement for the reconfigurations. Finally, at the end of stage 1, when $operations_{\boldsymbol{i}}$ contains both the batch of transactions and the reconfigurations (line 22), if the current process (denoted as ***self***) is the leader (line 23), it calls the function $inter\text{-}broadcast$ (at line 24) to start the inter-cluster broadcast stage (phase 2).

Algorithm 7: Local Ordering

```
1  request : process(t)
2  response : return(t, v)
3  uses:
4     tob : TotalOrderBroadcast
5        request : broadcast(t), new-leader(p, ts)
6        response : deliver(p, t), complain(p)
7  vars:
8     r                                          ▷ The current round
9     i                                ▷ The number of the current cluster
10    self                                       ▷ The current process
11    leader : P ← p_0^i                ▷ The leader of current cluster C_i
12    ts ← 0                                     ▷ Timestamp for leader
13    operations_j ← ∅                  ▷ Operations from each cluster C_j
14    certs                          ▷ Certificates for operations_i of C_i
15 upon request process(t)
16    tob request broadcast(t)
17 upon tob response deliver(p, t^σ)
18    append Trans(p, t) to operations_i
19    add σ to certs
20    if |operations_i| = batch-size × α then
21       call send-recs()
22    else if |operations_i| = batch-size + 1 then
                ▷ batch-size transactions + 1 reconfiguration set
23       if self = leader then
24          inter-broadcast(r, operations_i, certs)
25 upon tob response complain(p)
26    call complain(p)
```

Leader Change. A leader orchestrates both the ordering of transactions in the total-order broadcast, and the delivery of the reconfigurations. However, a leader may be Byzantine, and may not properly lead the cluster. Therefore, as presented in Alg. 8, the protocol monitors and changes leaders. As we described, the total-order broadcast tob (Alg. 7 at line 26) and the Byzantine reliable dissemination brd (Alg. 3 at line 32) complain when the delivery of transactions or reconfigurations is not timely. The complains are sent to the leader election module le (at line 7-8).

The protocol uses the classical leader election module le. The implementation of this module is presented in Alg. 9. Once a quorum of processes send complain requests to le, it eventually issues a response $new\text{-}leader(p, ts)$ at all correct processes to elect a new leader p with the timestamp ts. Further, if the current process sends a $next\text{-}leader$ request to the module, it issues a response $new\text{-}leader$ at the current process. This module guarantees that the leader for each timestamp is uniform across processes, the timestamps are monotonically increasing, and eventually a correct leader is elected.

When a process receives a $new\text{-}leader(p, ts)$ response (at line 9), it records the new leader and timestamp (at line 10), and forwards the new leader event to the total-order broadcast

tob and Byzantine reliable dissemination *brd* modules as well (at line 11-12). Further, the previous leader might have failed to communicate the operations of the previous round to other clusters. As we will describe next, clusters wait for the operations of each other in each round; therefore, a remote cluster can fall behind by at most one round. Thus, the new leader sends operations of the previous in addition to the current round (at line 14-18).

Algorithm 8: Leader Change

```
1  uses:
2    le : LeaderElection
3      request : complain(p), next-leader
4      response : new-leader(p, ts)
5  vars:
6    p-ops, p-certs        ▷ ops and certs of the previous round
7  function complain(p)
8    le request complain(p)
9  upon le response new-leader(p, ts')
10   ⟨leader, ts⟩ ← ⟨p, ts'⟩
11   tob request new-leader(leader, ts)
12   brd request new-leader(leader, ts)
13   reset timer_i
14   if leader = self then
15     if |operations_i| = batch-size then
16       call inter-broadcast(r, operations_i, certs)
17     if r > 1 then
18       call inter-broadcast(r − 1, p-ops, p-certs)
```

Stage 2: We already considered this stage in § III.

Stage 3: Execution. At the end of the inter-cluster communication stage, a process receives the batches of operations from each other cluster. It then calls the *execute* function (Alg. 1 at line 21) that performs the last stage: execution (at Alg. 10). Processes uniformly *order* the batches of operations: first, they process the transactions, and then the reconfigurations, and further, use a predefined order of clusters to order transactions (at line 4). Then, they process each operation: they apply each transaction and reconfiguration (at line 6-13). If a transaction has been issued by the current process, a *return* response is issued (at line 9). Finally, in order to prepare for the next round, the timers and variables are reset and the round number is incremented (at line 15-20).

Application of Reconfigurations. The function *reconfigure* is called for each set of reconfigurations rc from a cluster j (at line 21). First, the process adds joining processes, and removes leaving processes from the set of processes C_j of cluster j (at line 25 and 27). Then the function *kickstart* is called on the reconfigurations of the local cluster irc (at line 14). The function *kickstart* (at line 21) processes all the joins before the leave reconfigurations. We keep this specific order since leaving processes may still need to send additional messages for the new processes. If they leave first, then the new processes will not be able to collect enough states to start the execution. If the leave is for the current process, it issues a *left* response (at line 35). To kick-start a new process p, the members of its local cluster send a *CurrState* message to p (at line 33). The message contains the local *state*, the current round number r, and the cluster members C.

Algorithm 9: Leader Election

```
1  Implements: Leader Election
2    request : complain(p)
3    response : new-leader(p, ts)
4    request : next-leader
5  uses:
6    abeb : AuthenticatedBestEffortBroadcast
7  vars:
8    ts ← 1
9    C ← ∅                          ▷ Set of complaining processes
10   c ← false                                  ▷ Complained
11 upon request complain(p)
12   if ¬c then
13     call send-complain()
14 function send-complain()
15   c ← true
16   abeb request broadcast(Complaint(ts))
17 upon abeb response deliver(p, Complaint(ts')) where
     ts = ts'
18   C ← C ∪ {p}
19   if |C| ≥ f + 1 ∧ ¬c then
20     call send-complain()
21   if |C| ≥ 2 × f(i) + 1 then
22     call change()
23 function change()
24   ts ← ts + 1
25   C ← ∅
26   c ← false
27   response new-leader(p_{ts mod N}, ts)
                  ▷ Choose leaders in a round robin order.
                              ▷ N is the number of processes.
28 upon request next-leader
29   call change()
```

Further, the process resets its *echoed*, *readied*, *delivered*, and *valid* variables. When a correct process receives *CurrState* messages with the same state s', cluster members C', and round r' from a quorum (at line 39), the process sets its *state*, cluster C, and round r to the received values. It then issues a *joined* response (at line 41). After an addition or a removal, the process further updates the failure threshold f_j for the cluster j to less than one-third of the new cluster size (at line 28).

X. Proofs

A. Remote Leader Change

Lemma 1 (Eventual Succession). *Let ops be the locally replicated operations of a cluster C in a round. Either ops are eventually delivered to all correct processes of every other cluster in that round, or correct processes in C eventually adopt a new leader.*

Proof. Let C_2 be any other cluster in the system except C. There are two cases regarding the delivery of m in cluster C_2.

In the first case, at least one correct process p in C_2 delivers m. Then, it uses rb to broadcasts m to all members of the local cluster at Alg. 1, line 16. By validity of reliable broadcast, all the correct processes in C_2 deliver m.

In the second case, none of the correct processes in C_2 delivers m. We prove that processes in C_2 will invoke a remote leader change for C and finally correct processes in C adopt a new leader. If none of the correct processes of

Algorithm 10: Stage 3: Execution

```
1  vars:
2      state                                              ▷ Process state
3  function execute(operations)
4      foreach operations_j ∈ order(operations)
5          foreach o ∈ operations_j
6              match o
7                  case Trans(p, t) ⇒
8                      ⟨state, v⟩ ← t(state)
9                      if p = self then response return(t, v)
10                 case Reconfig(rc) ⇒
11                     call reconfigure(j, rc)
12                     if j = i then
13                         irc ← rc
14     call kickstart(irc)
15     p-ops ← operations_j;   p-certs ← certs
16     foreach cluster C_j
17         reset timer_j
18         operations_j ← ∅;   certs ← ∅
19         cn_j ← rcn_j ← 0
20     r ← r + 1
21 function reconfigure(j, rc)
       ▷ Function reconfigure is called in Stage 3.
22     foreach o ∈ rc
23         match o
24             case join(p) ⇒
25                 C_j ← C_j ∪ {p}
26             case leave(p) ⇒
27                 C_j ← C_j \ {p}
28         f_j = ⌊(|C_j| − 1)/3⌋
29 function kickstart(rc)
30     foreach o ∈ rc      ▷ First joins and then leaves.
31         match o
32             case join(p) ⇒
33                 apl request send(p, CurrState(state, C, r))
34             case leave(p) ⇒
35                 if p = self then response left
36     recs ← recs \ {rc}
37     echoed ← readied ← delivered ← false
38     valid ← ⊥
39 upon apl response deliver(p̄, CurrState(s′, C′, r′)) where
       |{p̄}| ≥ 2 × f_i + 1
40     state ← s′;   r ← r′;   C ← C′
41     response joined
```

C_2 delivers m, then their timers will eventually be triggered at Alg. 2, line 7 and all of the correct processes broadcast $LComplaint$ at line 8. Thus, the signatures of all of them are stored in cs_1 variable at line 11. Since there are at least $2 \times f_2 + 1$ correct processes in cluster C_2, all the correct processes eventually receive enough $LComplaint$ messages, and cs_1 will be large enough. Thus, $f_2 + 1$ processes in C_2 send $RComplaint$ messages, and each send it to $f+1$ distinct processes in C at line 18. Thus, at least one correct process in C eventually delivers the $RComplaint$ message at line 21 and verifies the validity of the accompanying signatures Σ. Then, it broadcasts the $Complaint$ message locally at line 22. By validity of $abeb$, all the correct processes in C deliver the complain at line 23, and request the leader election module to move to the next leader at line 26. Thus, the leader election module will eventually choose a new leader. Thus, all the correct processes in C will eventually adopt a new leader at Alg. 8 line 9-10.

□

Lemma 14 (Local Complaint Synchronization)**.** *If a correct process in cluster C_i installs $cn_j = k$, then all the correct processes in C_i eventually install $cn_j = k$.*

Proof. We prove this lemma by induction.

For $cn_j = 0$, all the correct processes assign cn_j to be the same value 0 at initialization.

The induction hypothesis is that if a correct process installs $cn_j = k$, then all the correct processes eventually install $cn_j = k$.

We prove that if a correct process in cluster C_i installs $cn_j = k + 1$, then all the correct processes in C_i eventually install $cn_j = k + 1$

A correct process p increments cn_j to $k+1$ at line 19 after verifying $2f_i + 1$ $LComplaint$ messages has been delivered for the same $cn_j = k$ at line 15. Thus at least $f_i + 1$ correct processes have broadcast $LComplaint$ messages for $cn_j = k$. By the validity of $abeb$, all the correct processes eventually delivers at least $f_i + 1$ consistent $LComplaint$ messages at line 12 and verify the complaint counter: by induction hypothesis and p installed $cn_j = k$, all the correct processes eventually install $cn_j = k$. Then correct processes amplify the complain by broadcasting $LComplaint$ messages for $cn_j = k$ at line 14. There are at least $2f_i + 1$ correct processes in cluster C_i. By the validity of $abeb$, eventually at least $2f_i+1$ $LComplaint$ messages are delivered to all correct processes at line 15 and they increment cn_j to $k+1$ at line 19. Therefore, all the correct processes install $cn_j = k + 1$.

We conclude the induction proof: for $k \geq 0$, if a correct process install $cn_j = k$, then all the correct processes install $cn_j = k$. □

Lemma 15 (Remote Complaint Synchronization)**.** *If a correct process in cluster C_i installs $rcn_j = k$, then all the correct processes in C_i eventually install $rcn_j = k$.*

Proof. We prove this lemma by induction.

For $rcn_j = 0$, all the correct processes assign rcn_j to be the same value 0 at initialization.

The induction hypothesis is that if a correct process in cluster C_i installs $rcn_j = k$, then all the correct processes eventually install $rcn_j = k$.

We prove that if a correct process p in cluster C_i installs $rcn_j = k + 1$, then all the correct processes in C_i eventually install $rcn_j = k + 1$

A correct process p increments rcn_j to $k + 1$ at line 24 after verifying $2f_j + 1$ $LComplaint$ messages was in Σ for the same $rcn_j = k$ at line 15. Thus by Lemma 14 and Σ verifies that a correct process in C_j installed $cn_j = k + 1$, all the correct processes in C_j eventually install $cn_j = k + 1$ at line 19. There are at most f_j Byzantine processes in cluster C_j and $\mathcal{S}$ contains $f_j + 1$ processes, therefore at least one correct process in $\mathcal{S}$ sends $RComplaint(k, j, \Sigma, r)$ messages to $f_i + 1$ processes in C_i. By the validity of apl, at least

one correct process in C_i receives the $RComplaint$ message at line 21 and broadcasts $Complaint(k, j, \Sigma)$ message at line 22. By the validity of $abeb$, all the correct processes in C_i eventually delivers $Complaint$ messages at line 23 and verify the complaint counter: by induction hypothesis and p installed $rcn_j = k$, all the correct processes eventually install $rcn_j = k$. They increment the remote complaint counter rcn_j to $k + 1$ line 24. Therefore, all the correct processes install $rcn_j = k + 1$.

We conclude the induction proof: for $k \geq 0$, if a correct process install $rcn_j = k$, then all the correct processes install $rcn_j = k$. □

Lemma 2 (Eventual Agreement). All correct processes in the same cluster eventually adopt the same leader.

Proof. We prove this lemma in three steps. Firstly, we prove if a correct process in C_i issue response $new\text{-}leader$ for ts, then eventually all correct process in C_i issue response $new\text{-}leader$ for ts. Secondly, we prove that eventually all the correct process stop changing leader and stay in the same timestamp. Finally, since the leader is deterministically chosen according to the timestamp and cluster membership, we prove that eventually all the correct process eventually adopt the same leader.

For the first statement, le issue response for two type of requests: $complain$ and $next\text{-}leader$. For $complain$ request, we directly use the eventual agreement property of underlying module. For $next\text{-}leader$ request, a correct process p in cluster C_i requests a $next\text{-}leader$ at line 26. Let us assume that p installs $rcn_j = n$ before the $next\text{-}leader$ request at line 24. By Lemma 15, all the correct processes in C_i eventually install $rcn_j = n$. By assumption, this request is apart from the previous remote leader change events and $\Delta - timer_i > \epsilon$. Then all the correct process request the $next\text{-}leader$ for the same $Complaint$ message. Therefore the ts at all correct processes are eventually the same.

For the second statement, correct processes eventually wait long enough for a correct leader to complete inter-broadcast stage: the timer for remote leader change increases exponentially and eventually, all the messages are delivered within a bounded delay after GST. When all $Complaint$ messages have been received, all the correct processes in the same cluster don't issue new complains and by Lemma 3, they stay in the same ts.

For the third statement, by Lemma 13, all the correct processes in the same cluster maintain a consistent group membership for each round. Then all of them deterministically choose the same process as leader based on group member and timestamp. □

Lemma 3 (Overthrow resistance). A correct process does not adopt a new leader unless at least one correct process complains about the previous leader.

Proof. The correct process requests the leader election module to adopt the next leader at Alg. 2, line 26. This request is after receiving a $Complaint$ message at line 23 with the following checks: (1) the expected next complaint counter rcn_j is equal to the received complain number c, and (2) the signatures Σ include at least $2 \times f_j + 1$ signatures from C_j. The first check prevents replay attacks; thus, no complaints about previous leaders can be reused. Therefore, all the signatures in Σ are complaints for the current leader. The second one implies that a correct process in C_j sent the $RComplaint$ message after receiving $2 \times f_j + 1$ $LComplaint$ messages at line 15. Thus, at least $f_j + 1$ correct processes sent $LComplaint$ messages. A correct process sends a $LComplaint$ message at two places: (1) the timer triggers at line 7; (2) the process amplifies the received complaints at line 12. The first case reached the conclusion. In the second case, a correct process only amplifies after receiving $f_j + 1$ $LComplaint$ messages. Thus, at least one correct process sent a $LComplaint$ message with the same two cases as above. This second case is the inductive case, and the first case is the base case. Since the number of processes is finite, by induction, this case is reduced to the first case in a finite number of steps. □

B. Inter-cluster Broadcast

Lemma 4 (Inter Broadcast Termination). In every round, every correct process eventually receives operations from each other cluster.

Proof. We prove the termination property for inter-cluster broadcast with the help of Lemma 1. A leader of cluster i should send $Inter$ message to $f_j + 1$ processes in cluster j for all $i \neq j$ at line 14. By the validity of remote leader change, either this $Inter$ message was delivered to all correct processes in cluster j or all the correct processes in cluster i change a leader. In the first case we conclude the proof. In the second case, eventually the correct processes in cluster i adopt a correct leader. The correct leader sends $Inter$ messages to $f_j + 1$ processes in cluster j. By the validity of apl, at least one correct process p in cluster j delivers the $Inter$ message at line 15. Then p broadcasts the received content in $Local$ message at line 16. By the validity of $abeb$, all the correct processes in cluster j eventually deliver the $Local$ message at line 17. We generalize the same reasoning for all the other cluster and conclude the proof. □

Lemma 5 (Inter Broadcast Agreement). In every round, the operations that every pair of correct processes receive from a cluster are the same.

Proof. Let process p receives $Local(r, j, ops, \Sigma)$ and p' receives $Local(r, j, ops', \Sigma')$. Correct processes only delivery valid $Local$ messages, which means Σ attests ops and Σ' attests ops'. Then Σ and Σ' both contains $2f + 1$ commit signatures for each operation in ops and ops'. By the agreement property of TOB in the first stage and $|ops| = |ops'|$, ops and ops' contains the same set of operations. By the total order property of the TOB, operations in ops and ops' have the same order. Thus, $ops = ops'$. □

C. Byzantine Reliable Dissemination

Lemma 16 (Integrity)**.** *The delivered set contains at least a quorum of messages from distinct processes.*

Proof. A set of messages is delivered at line 36 which is after the delivery of $2f_i+1$ of *Ready* messages (at line 34). At least $f_i + 1$ correct processes sent *Ready* messages since there are only f_i Byzantine processes in a cluster i. A correct process only sends *Ready* message when it receives $2f_i + 1$ *Echo* messages or f_i+1 *Ready* messages. Then by induction, at least $2f_i + 1$ *Echo* messages were received by a correct process. Then at least $f_i + 1$ correct processes sent *Echo* messages. A correct process only sends *Echo* messages when it verifies M is valid (at line 23). A M is valid if and only if Σ includes $2f_i + 1$ distinct signatures and M is the union of all the m sets in those messages; Or M is adopted from the *valid* and Σ contains $2f_i + 1$ *Echo* or $f_i + 1$ *Ready* messages. In the first case, the delivered M contains at least a quorum of m. In the second case, by induction M was in $2f_i+1$ of *Echo* messages and the correct processes who sent the *Echo* message verify that M originally was a union of $2f_i + 1$ m. □

Lemma 17 (Termination)**.** *If all correct processes broadcast messages then every correct process eventually delivers a set of messages.*

Proof. We consider two cases based on whether there is a correct process delivered a set of messages.

Case 1: If there is a correct process that delivers, then eventually all the correct processes deliver. A correct process delivers M after receiving $2f_i + 1$ *Ready* message at line 34. Then at least $f_i + 1$ correct processes broadcast the *Ready* message at line 28. By the validity of *abeb*, eventually all the correct processes deliver $f_i + 1$ *Ready* message at line 30 and broadcast the same message at line 32. Eventually, all the correct processes deliver $2f_i + 1$ *Ready* messages and issue delivery response (at line 36).

Otherwise, Case 2: if no correct process delivers, then each correct process complains about the current leader. Then by the eventual agreement property of the Byzantine leader election, all the correct processes eventually adopt the same correct leader. Upon the last leader election delivered at line 40, all the correct processes send *Valid* or *my-m* to the correct leader at line 47 or line 50. Since the set of correct processes is a quorum, then the correct leader either delivers a quorum of *my-m* messages at line 21 or a *Valid* message at line 51. Then we have two cases, either there is a valid *valid* or not. In the first case, the correct leader adopts M from *valid*. In the second case, the correct leader composes a new set of reconfiguration requests. Both cases can be verified and accepted by correct processes at line 23. Then all the correct processes send *Echo* message at line 25 and eventually $2f_i+1$ *Echo* messages are delivered to all the correct processes. Then all the correct processes send *Ready* message at line 28 and eventually $2f_i + 1$ *Ready* message are delivered to all correct processes. Then all the correct processes issue delivery response at line 36 and we conclude the proof. □

Lemma 18 (Uniformity)**.** *No correct processes deliver different set of messages*

Proof. There are two cases regarding the delivery of messages for p_1 and p_2: either they deliver messages with the same ts or different ts.

In the first case, since any pair of quorums has a correct process in the intersection, if p_1 delivers M_1 and p_2 delivers M_2, $M_1 = M_2$. Otherwise, the correct process sends different *Ready* messages for the same round and ts, which is not permitted by the protocol (at line 28, line 32).

In the second case, let us assume that p_1 delivers first with timestamp ts_1 and then p_2 delivers with another timestamp ts_2. Without losing generality, let us assume that $ts_1 < ts_2$. If p_1 delivers M_1 with ts_1, then p_1 receives at least a quorum of *Ready* messages. A correct process set its *valid* before sending *Ready* messages (at line 29, line 33). Therefore, at least $f_i + 1$ correct processes set their *valid* variable with M_1. For the next timestamp $ts_1 < ts_i \leq ts_2$, it collects a quorum of *my-m* messages or at least one *Valid* message. By assumption, cluster i has $3f_i + 1$ members in total, then at most $2f_i$ processes have not set *valid* and can send *my-m* message, which is not a quorum. Therefore, the leader for ts_i waits for the *Valid* message and adopts its value. Valid *valid* requires either $2f_i + 1$ *Echo* messages or $f_i + 1$ *Ready* messages for the same ts. By induction, since there are only f_i Byzantine processes, a correct process receives $2f_i + 1$ *Echo* messages before sending out *Ready* messages and triggering the amplification. Since any pair of quorums has a correct process in the intersection, there is only one M that can be echoed by a quorum of processes and appears in *valid*. The leader for ts_i can only propose M_1 that will be accepted by correct processes at line 23. From ts_i to ts_2, the *valid* can only be updated to the same M_1. Then when p_2 delivers M_2 in ts_2, $M_2 = M_1$. □

Lemma 19 (No duplication)**.** *Every correct process delivers at most one set of messages*

Proof. This lemma follows directly from the condition (at line 34) before the delivery response is issued at line 36. □

Lemma 20 (Validity)**.** *If a correct process delivers a set of messages containing m from a correct sender p, then m was broadcast by p*

Proof. If a correct process delivers a set of messages, then it receives a quorum of *Ready* messages. A ready message is send by a correct process if it receives a quorum of *Echo* messages or $f + 1$ ready messages. Since there are only f Byzantine processes, then by induction, the first ready message sent by a correct process is because of receiving a quorum of echo messages. A correct process only send echo message if delivers the *Agg* from the leader with valid certificate. A valid *Agg* message states that M is either collected from a quorum of distinct processes through *apl* or adopted from the previous leader. For the first case, by the validity of *apl*, if the sender of m is correct, then it sends m to the leader. For the second case, M can be adopted only if it carries a certificate with a quorum of *Echo* messages for M or $f + 1$ *Ready* messages for m. By the same induction, the messages contained in M is broadcast by its sender p if p is correct. □

D. Reconfiguration

Lemma 11 (Completeness). If a correct process p requests to join (or leave) cluster i, then every correct process will eventually have a configuration C such that $p \in C$ (or $p \notin C$).

Proof. We prove the completeness in two steps: first we prove that all the reconfiguration requests will be in a prepared state which we will formally define later; then we prove that all the prepared reconfiguration requests will be delivered within one round.

We define that a new process prepares a join request when it receives at least a quorum of replies from the existing replicas. Our protocol guarantees that a new process officially joins the system in the round it is prepared. Similarly, we define a leaving process that prepares a leave request when its $RequestLeave$ message has been delivered to a quorum of existing replicas. Our protocol guarantees that a leaving process officially leaves the system in the round it is prepared.

For the first statement, when a correct process p requests to join (or leave) the cluster C_i, it sends out $RequestJoin$ (or $RequestLeave$) messages to all the existing processes at line 7 (or at line 9). If p's request is not installed in a long time line 10, it resends the $RequestJoin$ (or $RequestLeave$) message and doubles the timer at line 12 (or line 14). Therefore $RequestJoin$ (or $RequestLeave$) messages sent out by p at line 7 will be delivered at all the correct processes in C_i in the first stage at line 16 after GST. Upon receiving the $RequestJoin$ and $RequestLeave$ message at line 17 and line 20, correct processes in the system add the reconfiguration request into their $recs$ variable. Since all the correct processes in a cluster is a quorum, p's reconfiguration request is eventually prepared.

We prove the second statement in two steps. First, we prove that any set of installed reconfiguration requests at round r includes p's reconfiguration request. Second, we prove that eventually, all correct processes install a set of reconfiguration requests in round r.

For the first step, at the end of the local ordering stage of each round at line 27, correct processes use Byzantine reliable dissemination module to deliver the reconfiguration requests $recs$ that they have collected. Assume that p's reconfiguration request is prepared in round r. By the integrity of BRD, the delivered set contains a quorum of messages send by distinct processes. Since every pair of quorums have at least one correct process in their intersection, at round r, there is always a correct process which sends p's reconfiguration request in the BRD message and the message is included in the delivered set.

For the second step, we consider the delivery of reconfiguration requests for both local and remote clusters.

For the remote clusters, by Lemma 4 all the correct processes in the remote cluster deliver *Local* message, which is verified to contain reconfiguration requests at line 17. Correct processes eventually receives all the *Local* message at line 20 and install reconfiguration at line 21.

For the local cluster, by the termination property of BRD, all the correct nodes in the local cluster eventually deliver a set of reconfiguration requests through BRD at line 28. They insert the reconfiguration requests at line 29. By Lemma 4, all the correct processes receive enough $Local$ message and install the reconfiguration requests at line 21.

In conclusion, a set of reconfiguration requests is eventually installed at all the correct processes and we conclude the second step. □

Lemma 13 (Uniformity). In every round, the configurations that every pair of correct processes execute are the same.

Proof. Let us assume that two correct processes p_1 and p_2 installed new configurations. The correct process installs new group membership at line 21, which is at the order and execution stage. We prove agreement for correct processes in both local and remote clusters.

For the local cluster, a correct process installs a reconfiguration request from $operations_i$ at line 11. $operations_i$ is updated at line 29, which is after the delivery of an instance of BRD at line 28. By the uniformity property of BRD, all the correct processes deliver the same set of messages. Since the installation of new membership is deterministic and only dependent on the set of reconfiguration requests, we have $C = C'$.

For remote cluster reconfiguration, a correct process in cluster i installs the reconfiguration requests for cluster j at the order and execution stage at line 21. $operations_j$ is updated after verifying the σ at line 17. σ is valid if and only if for each reconfiguration request in T, it contains a quorum of signatures from cluster j in round r. By Lemma 5, the reconfiguration requests installed at cluster j are the same. Therefore, we conclude $C = C'$.

□

Lemma 12 (Accuracy). Consider a correct process p that has a configuration C in a round, and then another configuration C' in a later round. If a correct process $p \in C'_i \setminus C_i$, then p requested to join the cluster i. Similarly, if a correct process $p \in C_i \setminus C'_i$, then p requested to leave the cluster i.

Proof. Since we have $p_n \in C_2 \setminus C_1 \wedge r_2 > r_1$, p_n is not originally a member of this cluster. The cluster membership is updated at line 25, which is after verifying each reconfiguration request is valid: each reconfiguration request is delivered after a quorum of $Ready$ messages. At line 23, every correct process checks the validity of rc, including its signatures from p_n. By the authenticity of apl, if p_n is correct, then it is the sender of the $RequestJoin$ messages and thus requested to join. The same reasoning applies to $leave$ requests.

□

E. Replication System

By Lemma 13, at the beginning of each round, all the correct processes have the same configuration. Thus during the execution of each round, all the correct processes maintain a static membership and we prove termination and total order properties for each round. We prove validity for eventual progress.

Theorem 1 (Validity). Every operation that a correct process requests is eventually executed by a correct process.

Proof. Based on the validity of the underlying TOB protocol in the first stage, if a valid operation o is submitted to a cluster i (at line 15), then o is eventually delivered at a correct process p in C_i at line 17 and included in $operations_i$ (at line 18). By the Lemma 4, each correct process receives $Local$ message from each other cluster and call $execute$ function (at line 3). Since o was included in $operations_i$, it is executed at line 8 and we conclude the proof. □

Theorem 2 (Agreement). If a correct process executes an operation in a round then every correct process executes that operation in the same round.

Proof. A correct process deliver a operation in the execution stage (at Alg. 10), which is stored in $operations$. $operations$ are updated in the inter-cluster stage (at Alg. 1) for remote clusters and in the local ordering stage (at line 18) for the local cluster. Based on whether o is an operation from the local cluster, we prove the termination in two cases.

Case 1: o is from the local cluster. Then we prove that o will be delivered locally and remotely in round r. For the local cluster, we can directly use the termination property provided by the underlying TOB protocol: all the correct processes eventually deliver o. Since correct processes in the local cluster are waiting for a fixed number of operations to be delivered in a batch for each round, they will not move to the next round before they deliver o in round r. Then we proved that o will be delivered locally in round r.

For the remote delivery of o, by the total order and termination property of underlying TOB protocol, o will be delivered at the leader and included in the $Local$ message. Then by the Lemma 4, all the other remote clusters will receive a $Local$ message for each cluster, including the current one. Thus, o will be delivered in the $Local$ message and inserted to $operations$. Finally, after all the $Local$ messages are delivered, o will be executed in the execution stage.

Case 2: o is from a remote cluster. Then we prove that o will be delivered at all the other clusters.

If o is from a remote cluster, then $operations$ is only updated if Σ is valid and the deliver is for the same round. Σ is valid if it contains a quorum of commit certificate for each operation in ops: a quorum of commit messages certify the delivery in the local order protocol. $operations$ is updated when receiving a valid $Local$ message. Then by the Lemma 5 and Lemma 4, o will be delivered at all the other clusters through the same $Local$ message.

□

Theorem 3 (Total order). For every pair of operations o and o', if a correct process executes only o, or executes o before o', then every correct process executes o' only after o.

Proof. We prove the total order property in two steps.

First we prove that if a pair of processes p_1 and p_2 both execute o and o', then they execute o and o' in the same order. Without loss of generality, let us assume that o is executed in p_1 before o' By Theorem 2, all the correct processes deliver the same operations for each cluster. Then they combine the operations in the predefined order based on the cluster identifier. Within each cluster, ops have been ordered across all the correct processes by the total order property of the underlying TOB protocol. Thus the combined operations keeps a total order across all the operations from all the clusters for round r: o is executed before o' at all correct processes including p_2.

In the second step, we prove by contradiction. Assume that process p' executed operation o' before operation o. The process p executed only o, or executed o before o'. In the case that it has executed only o and not o', then, by the Theorem 2, it will eventually execute o' after o. Thus, we will reach a state where p and p' have a different order for the two operations o and o', which contradicts the first statement. □