9 PROTOCOL PHASES

In the overview § 2 and Fig. 1, we explained the structure and the three phases of the protocol. We presented two sub-protocols in § 3 and § 4. In this section, we elaborate the other sub-protocols.

Phase 1: Phase 1 has two parallel parts. We say the reconfiguration part in § 4. 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 $\mathit{process}(t)$ at any process of any cluster (at line 17), and will later receive a $\mathit{return}(t, v)$ 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 $\mathit{new\text{-}leader}(p, ts)$ requests to install the new leader p with the timestamp ts, and can issue $\mathit{complain}(p)$ responses to complain about the leader p. The protocol is parametric for the total-order broadcast. The total-order broadcast abstracts the classical monolithic Byzantine replication protocols (except the leader election module *le* that we will consider separately.) If a complain is received from *tob*, (at line 15), it is forwarded to the leader election module *le* (in Alg. 8).

Upon receiving a $\mathit{process}(t)$ request (at line 17), the process uses the *tob* to broadcast the transaction in its own cluster (at line 18). Each process stores the operations $\mathit{operations}_j$ that it receives from each cluster C_j. Upon delivery of a transaction t from the *tob* (at line 19), the process appends t to $\mathit{operations}_{\boldsymbol{i}}$ received from this cluster $C_{\boldsymbol{i}}$ (at line 20). 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 21). In the next phase, 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 $\mathit{process}(t)$ 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 phase (phase 1). The reconfigurations are then propagated to other clusters in the inter-cluster broadcast phase (phase 2). In phase 1, a process collects the set of reconfiguration requests *recs*. It then calls the function *send-recs* (at line 23) to send the set of reconfigurations it has collected to the leader who aggregates and uniformly replicates them. In § 4, we presented the Byzantine Reliable Dissemination component that collects and sends reconfigurations to the leader. A process calls this function towards the end of phase 1, *i.e.*, when a large fraction α of the transaction batch is already ordered (at line 22). This leaves ample time in the beginning of phase 1 to accept reconfiguration requests, and also leaves enough time at the end of phase 1 to reach agreement for the reconfigurations. Finally, at the end of phase 1, when $\mathit{operations}_{\boldsymbol{i}}$ contains both the batch of transactions and the reconfigurations (line 24), if the current process (denoted as ***self***) is the leader (line 25), it calls the function *inter-broadcast* (at line 26) to start the inter-cluster broadcast phase (phase 2).

Algorithm 7: Local Ordering

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

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 16) 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 $\mathit{new\text{-}leader}(p, ts)$ at all correct processes to elect a new leader p with the timestamp ts. Further, if the current process sends a *next-leader* request to the module, it issues a response *new-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 $\mathit{new\text{-}leader}(p, ts)$ 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

```
uses:
  le : LeaderElection
    request : complain(p), next-leader
    response : new-leader(p, ts)
vars:
  p-ops, p-certs                    ▷ ops and certs of the previous round
function complain(p)
  le request complain(p)
upon le response new-leader(p, ts')
  ⟨leader, ts⟩ ← ⟨p, ts'⟩
  tob request new-leader(leader, ts)
  brd request new-leader(leader, ts)
  reset $timer_i$
  if leader = self then
    if $|operations_i|$ = batch-size then
      call inter-broadcast(r, $operations_i$, certs)
    if $r > 1$ then
      call inter-broadcast($r - 1$, p-ops, p-certs)
```

Phase 2: We already considered this phase in § 3.

Phase 3: Execution. At the end of the inter-cluster communication phase, 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 phase: 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. 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

Algorithm 9: Leader Election

```
Implements: Leader Election
  request : complain(p)
  response : new-leader(p, ts)
  request : next-leader
uses:
  abeb : AuthenticatedBestEffortBroadcast
vars:
  ts ← 1
  C ← ∅                              ▷ Set of complaining processes
  c ← false                          ▷ Complained
upon request complain(p)
  if ¬c then
    call send-complain()
function send-complain()
  c ← true
  abeb request broadcast(Complaint(ts))
upon abeb response deliver(p, Complaint(ts')) where ts = ts'
  C ← C ∪ {p}
  if $|C| \geq f + 1 \land \neg c$ then
    call send-complain()
  if $|C| \geq 2 \times f(i) + 1$ then
    call change()
function change()
  ts ← ts + 1
  C ← ∅
  c ← false
  response new-leader($p_{ts \bmod N}$, ts)
                                     ▷ Choose leaders in a round robin order.
                                     ▷ N is the number of processes.
upon request next-leader
  call change()
```

state, cluster C, and round r to the received values. It then issues a *joined* response (at line 41) and stops the timer. 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).

Algorithm 10: Phase 3: Execution

```
vars:
  $state$                                              ▷ Process state
function $execute(operations)$
    foreach $operations_j \in order(operations)$
        foreach $o \in operations_j$
            match $o$
                case $Trans(p, t)$ ⇒
                    $\langle state, v\rangle \leftarrow t(state)$
                    if $p = \textbf{\textit{self}}$ then response $return(t, v)$
                case $Reconfig(rc)$ ⇒
                    call $reconfigure(j, rc)$
                    if $j = \textbf{\textit{i}}$ then
                        $irc \leftarrow rc$
    call $kickstart(irc)$
    $p\text{-}ops \leftarrow operations_j$;  $p\text{-}certs \leftarrow certs$
    foreach cluster $C_j$
        reset $timer_j$
        $operations_j \leftarrow \emptyset$;  $certs \leftarrow \emptyset$
        $cn_j \leftarrow rcn_j \leftarrow 0$
    $r \leftarrow r + 1$
function $reconfigure(j, rc)$
    ▷ Function $reconfigure$ is called in Phase 3.
    foreach $o \in rc$
        match $o$
            case $join(p)$ ⇒
                $C_j \leftarrow C_j \cup \{p\}$
            case $leave(p)$ ⇒
                $C_j \leftarrow C_j \setminus \{p\}$
            $f_j = \lfloor(|C_j| - 1)/3\rfloor$
function $kickstart(rc)$
    foreach $o \in rc$    ▷ First joins and then leaves.
        match $o$
            case $join(p)$ ⇒
                $apl$ request $send(p, CurrState(state, C, r))$
            case $leave(p)$ ⇒
                if $p = \textbf{\textit{self}}$ then response $left$
    $recs \leftarrow recs \setminus \{rc\}$
    $echoed \leftarrow readied \leftarrow delivered \leftarrow$ false
    $valid \leftarrow \bot$
upon $apl$ response $deliver(\overline{p}, CurrState(s', C', r'))$ where
  $|\{\overline{p}\}| \geq 2 \times f_i + 1$
    $state \leftarrow s'$;  $r \leftarrow r'$;  $C \leftarrow C'$
    response $joined$
    stop $client\text{-}timer$
```

10 PROOFS

10.1 Remote Leader Change

Lemma 11 (Validity). *Let ops be the locally replicated operations of cluster C at the end of round r. Either ops is delivered to all the correct processes of every other cluster in round r, or correct processes in C eventually change their leader.*

Proof. 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_1 and finally correct processes in C_1 trust a new leader. If none of the correct processes of 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 10. 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 $f_1 + 1$ distinct processes in C_1 at line 17. Thus, at least one correct process in C_1 eventually delivers the *RComplaint* message at line 20 and verifies the validity of the accompanying signatures Σ. Then, it broadcasts the *Complaint* message locally at line 21. By validity of *abeb*, all the correct processes in C_1 deliver the the complain at line 22, and request the leader election module to move to the next leader at line 25. Thus, the leader election module will eventually choose a new leader. Thus, all the correct processes in C_1 will eventually trust a new leader at Alg. 8 line 9-10.

□

Lemma 12 (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 18 after verifying $2f_i + 1$ *LComplaint* messages has been delivered for the same $cn_j = k$ at line 14. 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 11 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 13. 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 14 and they increment cn_j to $k + 1$ at line 18. 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 13 (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 23 after verifying $2f_j + 1$ *LComplaint* messages was in Σ for the same $rcn_j = k$ at line 14. Thus by Lemma 12 and Σ verifies that a correct process in C_j installed $cn_i = k + 1$, all the correct processes in C_j eventually install $cn_i = k+1$ at line 18. 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 $RComplaint(k, i, \Sigma, r)$ 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 20 and broadcasts $Complaint(k, j, \Sigma)$ message at line 21. By the validity of *abeb*, all the correct processes in C_i eventually delivers *Complaint* messages at line 22 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 23. 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 14 (Eventual agreement). *All correct processes in the same cluster eventually trust the same leader.*

Proof. We prove this lemma in three steps. Firstly, we prove if a correct process in C_i issue response *new-leader* for *ts*, then eventually all correct process in C_i issue response *new-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 trust the same leader.

For the first statement, *le* issue response for two type of requests: *complain* and *next-leader*. For *complain* request, we directly use the eventual agreement property of underlying module. For *next-leader* request, a correct process p in cluster C_i requests a *next-leader* at line 25. Let us assume that p installs $rcn_j = n$ before the *next-leader* request at line 23. By Lemma 13, 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 *rec-timer* $> \delta - \theta$. Then all the correct process request the *next-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 phase: 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 do not issue new complains and by Lemma 3, they stay in the same *ts*.

For the third statement, by Lemma 8, 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. □

Lemma 15 (Putsch resistance). *A correct process does not trust a new leader unless at least one correct process's timer is triggered for the previous leader.*

Proof. The correct process requests the leader election module to trust the next leader at Alg. 2, line 25. This request is after receiving a *Complaint* message at line 22 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 14. 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 11. 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. □

10.2 Inter-cluster Broadcast

Lemma 16 (Inter Broadcast Termination). *In each round, every correct process eventually receives a Local message 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 trust 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 17 (Inter Broadcast Agreement). *If a correct process receives a Local message with ops from cluster j, and another correct process receives a Local message with ops′ from cluster j in the same round, then ops = ops′*

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 total order property of the underlying protocol and $|ops| = |ops'|$, they contains the same set of messages with the same order. Thus, $ops = ops'$. □

10.3 Byzantine Reliable Dissemination

Lemma 18 (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 19 (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 trust 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 the reconfiguration requests 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 20 (Uniformity). *No correct processes deliver different set of messages*

Proof. There are two cases regarding the delivery of reconfiguration requests for p_1 and p_2: either they deliver reconfiguration requests 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 rc_1 and p_2 delivers rc_2, $rc_1 = rc_2$. 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 21 (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 22 (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. The 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. □

10.4 Reconfiguration

Lemma 23 (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 phase 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 phase 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 16 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 16, 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 24 (Agreement). *The configurations of every pair of correct processes for the same round is 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 phase. 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 phase 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. As we proved above, the reconfiguration requests installed at cluster j are the same. Therefore, we conclude $C = C'$.

□

Lemma 25 (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.

□

10.5 Replication System

By Lemma 8, 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.

Lemma 26 (Termination). *If a correct process deliver an operation o in round r, then all correct processes deliver o in r.*

Proof. A correct process deliver a operation in the execution phase (at Alg. 10), which is stored in *operations*. *operations* are updated in the inter-cluster phase (at Alg. 1) for remote clusters and in the local ordering phase (at line 20) 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 16, 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 phase.

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 17 and Lemma 16, o will be delivered at all the other clusters through the same *Local* message.

□

Lemma 27 (Total order). *If a correct process deliver an operation o before another o', then all correct processes only deliver o' after the delivery of o.*

Proof. By Lemma 26, 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. □