

# 1 Model

We consider a static set of  $n$  processes with known identities, communicating by reliable point-to-point channels, in a complete graph. Messages are uniquely identifiable.

**Synchrony.** The network is asynchronous. Processes may crash; at most  $f$  crashes occur.

**Communication.** Processes can exchange through a Reliable Broadcast (RB) primitive (defined below) which's invoked with the functions  $\text{RB-cast}(m)$  and  $\text{RB-received}(m)$ . There exists a shared object called DenyList (DL) (defined below) that is interfaced with the functions  $\text{APPEND}(x)$ ,  $\text{PROVE}(x)$  and  $\text{READ}()$ .

**Notation.** Let  $\Pi$  be the finite set of process identifiers and let  $n \triangleq |\Pi|$ . Two authorization subsets are  $\Pi_M \subseteq \Pi$  (processes allowed to issue  $\text{APPEND}$ ) and  $\Pi_V \subseteq \Pi$  (processes allowed to issue  $\text{PROVE}$ ). Indices  $i, j \in \Pi$  refer to processes, and  $p_i$  denotes the process with identifier  $i$ . Let  $\mathcal{M}$  denote the universe of uniquely identifiable messages, with  $m \in \mathcal{M}$ . Let  $\mathcal{R} \subseteq \mathbb{N}$  be the set of round identifiers; we write  $r \in \mathcal{R}$  for a round. We use the precedence relation  $\prec$  for the DL linearization:  $x \prec y$  means that operation  $x$  appears strictly before  $y$  in the linearized history of DL. For any finite set  $A \subseteq \mathcal{M}$ ,  $\text{ordered}(A)$  returns a deterministic total order over  $A$  (e.g., lexicographic order on  $(\text{senderId}, \text{messageId})$  or on message hashes). For any round  $r \in \mathcal{R}$ , define  $\text{Winners}_r \triangleq \{j \in \Pi \mid (j, \text{prove}(r)) \prec \text{APPEND}(r)\}$ , i.e., the set of processes whose  $\text{PROVE}(r)$  appears before the first  $\text{APPEND}(r)$  in the DL linearization. We denoted by  $\text{PROVE}^{(j)}(r)$  or  $\text{APPEND}^{(j)}(r)$  the operation  $\text{PROVE}(r)$  or  $\text{APPEND}(r)$  invoked by process  $j$ .

## 2 Primitives

### 2.1 Reliable Broadcast (RB)

RB provides the following properties in the model.

- **Integrity:** Every message received was previously sent.  $\forall p_i : \text{RB-received}_i(m) \Rightarrow \exists p_j : \text{RB-cast}_j(m)$ .
- **No-duplicates:** No message is received more than once at any process.
- **Validity:** If a correct process broadcasts  $m$ , every correct process eventually receives  $m$ .

### 2.2 Group Election Object

We consider a Groupe Election object ( $\text{GE}[r]$ ) per round  $r \in \mathcal{R}$  with the following properties.

There are three operations:  $\text{VOTE}(r)$ ,  $\text{COMMIT}(r)$  and  $\text{RESULT}(r)$  such that:

- **Termination.** A  $\text{VOTE}(r)$ ,  $\text{COMMIT}(r)$  or  $\text{RESULT}(r)$  operation invoked by a correct process always returns.
- **Election.** If there exists at least one  $\text{COMMIT}(r)$  operation and let  $\text{COMMIT}(r)^*$  denote the first  $\text{COMMIT}(r)$  in the linearization order. If some correct process  $p$  invokes  $\text{VOTE}(r)$  and the invocation of  $\text{VOTE}(r)$  appears before  $\text{COMMIT}(r)^*$  in the linearization order, then  $\text{Winners}_r \neq \emptyset$ .

- **Prefix Inclusion.** If  $\text{COMMIT}(r)^*$  exists, then there exists a set  $\text{Winners}_r \subseteq \Pi$  such that, for any process  $p_j$ :  $p_j \in \text{Winners}_r$  iff  $p_j$  invokes  $\text{VOTE}(r)$  and its  $\text{VOTE}(r)$  operation is linearized before  $\text{COMMIT}(r)^*$ .
- **Stability.** If  $\text{COMMIT}(r)^*$  exists, then every  $\text{RESULT}(r)$  operation linearized after  $\text{COMMIT}(r)^*$  returns exactly  $\text{Winners}_r$ .
- **READ Validity.** The invocation of  $op = \text{RESULT}(r)$  by a process  $p$  returns the list of valid invocations of  $\text{VOTE}(r)$  that appears before  $op$  in the linearization order along with the names of the processes that invoked each operation.

### 3 Group Election Object Consensus Number

**Definition 1.** We assume a synchronous DenyList (DL) object as in [?] with the following properties.

The DenyList object type supports three operations: APPEND, PROVE, and READ. These operations appear as if executed in a sequence Seq such that:

- **Termination.** A PROVE, an APPEND, or a READ operation invoked by a correct process always returns.
- **APPEND Validity.** The invocation of  $\text{APPEND}(x)$  by a process  $p$  is valid if:
  - $p \in \Pi_M \subseteq \Pi$ ; **and**
  - $x \in S$ , where  $S$  is a predefined set.

Otherwise, the operation is invalid.

- **PROVE Validity.** If the invocation of a  $op = \text{PROVE}(x)$  by a correct process  $p$  is not valid, then:
  - $p \notin \Pi_V \subseteq \Pi$ ; **or**
  - A valid  $\text{APPEND}(x)$  appears before  $op$  in Seq.

Otherwise, the operation is valid.

- **PROVE Anti-Flickering.** If the invocation of a operation  $op = \text{PROVE}(x)$  by a correct process  $p \in \Pi_V$  is invalid, then any  $\text{PROVE}(x)$  operation that appears after  $op$  in Seq is invalid.
- **READ Validity.** The invocation of  $op = \text{READ}()$  by a process  $p \in \pi_V$  returns the list of valid invocations of PROVE that appears before  $op$  in Seq along with the names of the processes that invoked each operation. item **Anonymity.** Let us assume the process  $p$  invokes a  $\text{PROVE}(v)$  operation. If the process  $p'$  invokes a  $\text{READ}()$  operation, then  $p'$  cannot learn the value  $v$  unless  $p$  leaks additional information.

We assume that  $\Pi_M = \Pi_V = \Pi$  (all processes can invoke APPEND and PROVE).

**Lemma 1** (From DenyList to Group Election). *For any fixed value  $r \in S$ , one DenyList object can be used to wait-free implement a Group Election object  $\text{GE}[r]$ .*

*Proof.* We implement the operations of  $\text{GE}[r]$  using the operations of the DenyList as follows.

**function** CANDIDATE( $r$ )  
  PROVE( $r$ )

**end function**

**function** CLOSE( $r$ )

    APPEND( $r$ )

**end function**

**function** READGE( $r$ )

$P \leftarrow \text{READ}()$

**return**  $\{j : (j, \text{prove}(r)) \in P\}$

**end function**

**Termination.** Termination follows from the Termination property of DenyList operations. Consider now a fixed sequential history Seq of the DenyList.

**Prefix Inclusion.** Let APPEND( $r$ )<sup>\*</sup> denote the first valid APPEND( $r$ ) in Seq, if it exists. From the PROVE Validity and anti-flickering properties of the DenyList, a process  $p_j$  has a valid PROVE( $r$ ) in Seq if and only if its PROVE( $r$ ) invocation appears before APPEND( $r$ )<sup>\*</sup> in Seq. Hence, by construction,  $p_j \in \text{Winners}_r$  iff  $p_j$  invokes VOTE( $r$ ) and its VOTE( $r$ ) is linearized before COMMIT( $r$ )<sup>\*</sup> where we identify COMMIT( $r$ )<sup>\*</sup> with APPEND( $r$ )<sup>\*</sup>. This is exactly the Prefix inclusion property.

**Stability.** Moreover, after APPEND( $r$ )<sup>\*</sup>, no new PROVE( $r$ ) can become valid (anti-flickering), so every subsequent READ<sup>\*</sup>() returns the same set of valid PROVE( $r$ ) invocations. Consequently, every RESULT( $r$ ) linearized after COMMIT( $r$ )<sup>\*</sup> returns the same set  $\text{Winners}_r$ , which proves Stability.

**Election.** Finally, if some process invokes VOTE( $r$ ) before COMMIT( $r$ )<sup>\*</sup>, its proof is valid and thus appears in the set returned by READ<sup>\*</sup>(). Hence  $\text{Winners}_r \neq \emptyset$ , which proves Election.

**Validity.** Validity is immediate from the construction of  $\text{Winners}_r$ : a process belongs to  $\text{Winners}_r$  only if it invoked PROVE( $r$ ), i.e., only if it invoked VOTE( $r$ ).

Thus the constructed object satisfies all properties of a Group Election object. □

**Lemma 2** (From Group Election to DenyList). *Fix a value  $r \in S$ . A Group Election object GE[ $r$ ] can be used to wait-free implement an DenyList.*

*Proof.* We implement the DenyList for value  $r$  as follows. We use one Group Election object GE[ $r$ ].

The operations are implemented as follows.

**function** APPEND( $r$ )

    COMMIT( $r$ )

**end function**

**function** PROVE( $r$ )

    VOTE( $r$ )

$W_r \leftarrow \text{RESULT}(r)$

**if**  $p \in W_r$  **then**

**return** True

**else**

**return** False

**end if**  
**end function**

**function** READ()  
 $W \leftarrow \bigcup_{r \in S} \text{RESULT}(r)$   
**return**  $\{(p, r) \mid p \in W_r\}$   
**end function**

**Termination.** Termination follows from the Termination property of Group Election operations. Consider now a fixed sequential history  $\text{Seq}$  of the Group Election.

**APPEND Validity.** By construction, any process invoking  $\text{APPEND}(r)$  invokes  $\text{COMMIT}(r)$ . By definition of Group Election,  $\text{COMMIT}(r)$  is always valid.

**PROVE Validity.** By definition  $\Pi_V = \Pi$ , so any process invoking  $\text{PROVE}(r)$  is in  $\Pi_V$ . So the case  $p \notin \Pi_V$  cannot happen. Now, if some process invokes  $\text{APPEND}(r)$  before the invocation of  $\text{PROVE}(r)$  in  $\text{Seq}$ , then by the Prefix Inclusion property of Group Election, the set  $\text{Winners}_r$  is already fixed and any subsequent  $\text{VOTE}(r)$  cannot be in  $\text{Winners}_r$ . Hence the invocation of  $\text{PROVE}(r)$  is invalid. Conversely, if no  $\text{APPEND}(r)$  appears before  $\text{PROVE}(r)$  in  $\text{Seq}$ , then by the Election property of Group Election, if some correct process invoked  $\text{VOTE}(r)$  before any  $\text{COMMIT}(r)$ , then  $\text{Winners}_r \neq \emptyset$  and hence the invoking process belongs to  $\text{Winners}_r$ . Thus its invocation of  $\text{PROVE}(r)$  is valid. This proves PROVE Validity.

**PROVE Anti-Flickering.** If some invocation of  $\text{PROVE}(r)$  is invalid, then some  $\text{APPEND}(r)$  must have appeared before it in  $\text{Seq}$ . By the Prefix Inclusion property of Group Election, the set  $\text{Winners}_r$  is fixed after the first  $\text{COMMIT}(r)$ , so any subsequent  $\text{VOTE}(r)$  cannot be in  $\text{Winners}_r$ . Hence any subsequent invocation of  $\text{PROVE}(r)$  is also invalid. This proves PROVE Anti-Flickering.

**READ Validity.** Finally, by construction, the invocation of  $\text{READ}()$  returns the list of valid invocations of  $\text{PROVE}$  that appear before it in  $\text{Seq}$  along with the names of the processes that invoked each operation. This proves READ Validity. □

**Theorem 3** (Consensus number of Group Election). *Let  $|\Pi_V| = k$ . The Group Election object type with verifier set  $\Pi_V$  has consensus number  $k$ . In particular, when  $\Pi_V = \Pi$ , the consensus number of Group Election is  $|\Pi|$ .*

*Proof.* We first recall that, for a DenyList object with  $|\Pi_V| = k$  (a  $k$ -DenyList), the consensus number is exactly  $k$ .

**Lower bound.** By Lemma 1, for any fixed value  $r \in S$ , one  $k$ -DenyList object can be used to wait-free implement a Group Election object  $\text{GE}[r]$  over the same set of processes. Since the  $k$ -DenyList has consensus number  $k$ , it follows that the Group Election type has consensus number at least  $k$ .

**Upper bound.** Conversely, by Lemma 2, one Group Election object can be used to wait-free implement a DenyList object restricted to value  $r$ . This restricted DenyList satisfies the same specification as the original  $k$ -DenyList on value  $r$ , and in particular it has consensus number  $k$ . Therefore, the consensus number of the Group Election type cannot exceed  $k$ .

Combining the two bounds, we obtain that the consensus number of the Group Election object type is exactly  $k = |\Pi_V|$ . When we instantiate  $\Pi_V = \Pi$ , we get that the consensus number of Group Election is  $|\Pi|$ .  $\square$

## 4 Target Abstraction: Atomic Reliable Broadcast (ARB)

Processes export  $\text{AB-broadcast}(m)$  and  $\text{AB-deliver}(m)$ . ARB requires total order:

$$\forall m_1, m_2, \forall p_i, p_j : \text{AB-deliver}_i(m_1) < \text{AB-deliver}_i(m_2) \Rightarrow \text{AB-deliver}_j(m_1) < \text{AB-deliver}_j(m_2),$$

plus Integrity/No-duplicates/Validity (inherited from RB and the construction).

## 5 ARB over RB and DL

In this part we're using the consensus number of the GE Object to show that it's possible to implement F-ARB with RB and our defined object. Also we present an algorithm which achieve this in section 7.

**Theorem 4** (RB + Group Election implements F-ARB). *In an asynchronous message-passing system with crash failure. We can wait-free implement a FIFO-Atomic Reliable Broadcast from a Reliable Broadcast (RB) primitive and one Group Election object  $\text{GE}[r]$  per round  $r \in \mathbb{N}$ .*

*Proof.* By Theorem 3, the Group Election object type with verifier set  $\Pi_V$  has consensus number  $|\Pi_V|$ . In particular, when  $\Pi_V = \Pi$ , using one instance  $\text{GE}[r]$  per round  $r$  we can implement wait-free consensus among all processes in  $\Pi$ .

It is well known that, in a crash-prone asynchronous message-passing system, consensus and atomic (total order) broadcast are equivalent (defago et al): given consensus, one can implement atomic broadcast by using an infinite sequence of consensus instances to decide the sequence of messages to deliver, and conversely atomic broadcast can be used to implement consensus by deciding a single value in the first position of the total order.

In our setting, we already have a Reliable Broadcast (RB) primitive, which provides RB-Validity, RB-Agreement, and RB-Integrity for the dissemination of messages. Using the consensus power provided by the Group Election objects, we can therefore apply the standard reduction from consensus to atomic broadcast: each position (or *slot*) in the global delivery sequence is chosen by a consensus instance, whose proposals are messages that have been RB-delivered but not yet ordered. This yields an atomic reliable broadcast (ARB) primitive.

To obtain FIFO-Atomic Reliable Broadcast (F-ARB), it suffices to enforce per-sender FIFO order on top of ARB. This can be done in the usual way by tagging each message broadcast by a process  $p_i$  with a local sequence number  $s \in \mathbb{N}$ , and by ensuring that only the message with the smallest pending sequence number for  $p_i$  is ever proposed to a consensus instance. As a result, for every sender  $p_i$ , messages with tags  $(p_i, s)$  and  $(p_i, t)$  with  $s < t$  are decided (and thus delivered) in this order at all processes.

Hence, RB plus Group Election objects is sufficient to implement FIFO-Atomic Reliable Broadcast.  $\square$

## 6 BFT-ARB over RB and DL

### 6.1 Model extension

We consider a static set of  $n$  processes with known identities, communicating by reliable point-to-point channels, in a complete graph. Messages are uniquely identifiable.

**Synchrony.** The network is asynchronous. Processes may crash or be byzantine; at most  $f = \frac{n}{2} - 1$  processes can be faulty.

**Communication.** Processes can exchange through a Reliable Broadcast (RB) primitive (defined below) which's invoked with the functions `RB-cast( $m$ )` and `RB-received( $m$ )`. There exists a shared object called DenyList (DL) (defined below) that is interfaced with the functions `APPEND( $x$ )`, `PROVE( $x$ )` and `READ()`.

**Byzantine behaviour** A process exhibits Byzantine behavior if it deviates arbitrarily from the specified algorithm. This includes, but is not limited to, the following actions:

- Invoking primitives (`RB-cast`, `APPEND`, `PROVE`, etc.) with invalid or maliciously crafted inputs.
- Colluding with other Byzantine processes to manipulate the system's state or violate its guarantees.
- Delaying or accelerating message delivery to specific nodes to disrupt the expected timing of operations.
- Withholding messages or responses to create inconsistencies in the system's state.

Byzantine processes are constrained by the following:

- They cannot forge valid cryptographic signatures or threshold shares without the corresponding private keys.
- They cannot violate the termination, validity, or anti-flickering properties of the DL object.
- They cannot break the integrity, no-duplicates, or validity properties of the RB primitive.

**Notation.** Let  $\Pi$  be the finite set of process identifiers and let  $n \triangleq |\Pi|$ . Two authorization subsets are  $M \subseteq \Pi$  (processes allowed to issue `APPEND`) and  $V \subseteq \Pi$  (processes allowed to issue `PROVE`). Indices  $i, j \in \Pi$  refer to processes, and  $p_i$  denotes the process with identifier  $i$ . Let  $\mathcal{M}$  denote the universe of uniquely identifiable messages, with  $m \in \mathcal{M}$ . Let  $\mathcal{R} \subseteq \mathbb{N}$  be the set of round identifiers; we write  $r \in \mathcal{R}$  for a round. We use the precedence relation  $\prec$  for the DL linearization:  $x \prec y$  means that operation  $x$  appears strictly before  $y$  in the linearized history of DL. For any finite set  $A \subseteq \mathcal{M}$ , `ordered( $A$ )` returns a deterministic total order over  $A$  (e.g., lexicographic order on  $(senderId, messageId)$  or on message hashes). For any round  $r \in \mathcal{R}$ , define  $Winners_r \triangleq \{j \in \Pi \mid (j, prove(r)) \prec APPEND(r)\}$ , i.e., the set of processes whose `PROVE( $r$ )` appears before the first `APPEND( $r$ )` in the DL linearization. We denote by `PROVE( $j$ )( $r$ )` or `APPEND( $j$ )( $r$ )` the operation `PROVE( $r$ )` or `APPEND( $r$ )` invoked by process  $j$ .

## 6.2 Primitives

### 6.2.1 t-BFT-DL

We consider a t-Byzantine Fault Tolerant DenyList (t-BFT-DL) with the following properties. There are 3 operations : BFT-PROVE( $x$ ), BFT-APPEND( $x$ ), BFT-READ( $x$ ) such that :

**Termination.** Every operation BFT-APPEND( $x$ ), BFT-PROVE( $x$ ), and BFT-READ() invoked by a correct process always returns.

**PROVE Validity.** The invocation of  $op = \text{BFT-PROVE}(x)$  by a correct process is valid iff there exist a set of correct process  $C$  such that  $\forall c \in C, c$  invoke  $op_2 = \text{BFT-APPEND}(x)$  with  $op_2 \prec op_1$  and  $|C| \leq t$

**PROVE Anti-Flickering.** If the invocation of a operation  $op = \text{BFT-PROVE}(x)$  by a correct process  $p \in \Pi_V$  is invalid, then any BFT-PROVE( $x$ ) operation that appears after  $op$  in Seq is invalid.

**READ Liveness.** Let  $op = \text{BFT-READ}()$  invoke by a correct process such that  $R$  is the result of  $op$ . For all  $(i, \text{prove}(x)) \in R$  there exist a valid invocation of BFT-PROVE( $x$ ) by  $p_i$ .

**READ Anti-Flickering.** Let  $op_1, op_2$  two BFT-READ() operations that returns respectively  $R_1, R_2$ . Iff  $op_1 \prec op_2$  then  $R_2 \subseteq R_1$ . Otherwise  $R_1 \subseteq R_2$ .

**READ Safety.** Let  $op_1, op_2$  respectively a valid BFT-PROVE( $x$ ) operation submitted by the process  $p_i$  and a BFT-READ() operation submitted by any correct process such that  $op_1 \prec op_2$ . Let  $R$  the result of  $op_2$  then  $R \ni (i, \text{prove}(x))$

## 6.3 DL $\Rightarrow$ t-BFT-DL

**Lemma 5.** For any fixed value  $t$  such that  $3t < |M|$ , multiple DenyList Object can be used to implement a t-Byzantine Fault Tolerant DenyList Object.

*Proof.* Fix  $3t < |M|$ . Let

$$\mathcal{U} = \{U \subseteq M \mid |U| = |M| - t\}.$$

For each  $U \in \mathcal{U}$ , we instantiate one DenyList object  $DL_U$  whose authorization sets are

$$\Pi_M(DL_U) = S_U = U \quad \text{and} \quad \Pi_V(DL_U) = V.$$

$$|\mathcal{U}| = \binom{|M|}{|M| - t}.$$

---

### Algorithm A BFT-APPEND

---

```
DL1 function BFT-APPEND(x)
DL2   for each  $U \in \mathcal{U}$  st  $i \in U$  do
DL3      $DL_U$ .APPEND(x)
DL4   end for
DL5 end function
```

---

---

**Algorithm B BFT-PROVE**

---

DL6 **function** BFT-PROVE( $x$ )  
DL7      $state \leftarrow false$   
DL8     **for each**  $U \in \mathcal{U}$  **do**  
DL9          $state \leftarrow state$  **OR**  $DL_U.PROVE(x)$   
DL10     **end for**  
DL11     **return**  $state$   
DL12 **end function**

---

---

**Algorithm C BFT-READ**

---

DL13 **function** BFT-READ( $\perp$ )  
DL14      $results \leftarrow \emptyset$   
DL15     **for each**  $U \in \mathcal{U}$  **do**  
DL16          $results \leftarrow results \cup DL_U.READ()$   
DL17     **end for**  
DL18     **return**  $results$   
DL19 **end function**

---

**BFT-PROVE Validity.** Let  $op = \text{BFT-PROVE}(x)$  be an invocation by a correct process  $p_i$ . Let  $A \subseteq M$  be the set of distinct issuers that invoked  $\text{BFT-APPEND}(x)$  before  $op$  in  $\text{Seq}$ .

- **Case (i):**  $|A| \geq t + 1$ . Fix any  $U \in \mathcal{U}$ .  $A \cap U \neq \emptyset$ . Pick  $j \in A \cap U$ . Since  $j \in U$ , the call  $\text{BFT-APPEND}^{(j)}(x)$  triggers  $DL_U.APPEND(x)$ , and because  $\text{BFT-APPEND}^{(j)}(x) \prec op$  in  $\text{Seq}$ , this induces a valid  $DL_U.APPEND(x)$  that appears before the induced  $DL_U.PROVE(x)$  by  $p_i$ . By **PROVE Validity** of DL, the induced  $DL_U.PROVE(x)$  is invalid. As this holds for every  $U \in \mathcal{U}$ , there is *no* component  $DL_U$  where  $\text{PROVE}(x)$  is valid, so the field  $state$  at line DL9 is never becoming true, and  $op$  return false.
- **Case (ii):**  $|A| \leq t$ . There exists  $U^* \in \mathcal{U}$  such that  $A \cap U^* = \emptyset$ . For any  $j \in A$ , we have  $j \notin U^*$ , so  $\text{BFT-APPEND}^{(j)}(x)$  does *not* call  $DL_{U^*}.APPEND(x)$ . Hence no valid  $DL_{U^*}.APPEND(x)$  appears before the induced  $DL_{U^*}.PROVE(x)$ . Since also  $i \in \Pi_V(DL_{U^*})$ , by **PROVE Validity** of DL the induced  $DL_{U^*}.PROVE(x)$  is valid. Therefore, there exists a component with a valid  $\text{PROVE}(x)$ , so  $op$  is valid.

Combining the cases yields the claimed characterization of invalidity.

**BFT-PROVE Anti-Flickering.** Let  $op = \text{BFT-PROVE}(x)$  be an invocation by a correct process  $p_i$  that is *invalid* in  $\text{Seq}$ . By BFT-PROVE Validity, this implies that there exist at least  $t + 1$  *distinct* processes in  $M$  that invoked a *valid*  $\text{BFT-APPEND}(x)$  before  $op$  in  $\text{Seq}$ . Let  $A \subseteq M$  denote that set, with  $|A| \geq t + 1$ .

Fix any  $U \in \mathcal{U}$ . We have  $A \cap U \neq \emptyset$ . Pick  $j \in A \cap U$ . Since  $j \in U$ , the call  $\text{BFT-APPEND}^{(j)}(x)$  triggers a call  $DL_U.APPEND(x)$ . Moreover, because  $\text{BFT-APPEND}^{(j)}(x) \prec op$  in  $\text{Seq}$ , the induced  $DL_U.APPEND(x)$  appears before the induced  $DL_U.PROVE(x)$  of  $op$  in the projection  $\text{Seq}_U$ .

Hence, in  $\text{Seq}_U$ , there exists a *valid*  $DL_U.APPEND(x)$  that appears before the  $DL_U.PROVE(x)$  induced by  $op$ . By **PROVE Validity** the base DL object, the induced  $DL_U.PROVE(x)$  is therefore *invalid* in  $\text{Seq}_U$ .

Let  $op' = \text{BFT-PROVE}(x)$  be any invocation such that  $op \prec op'$  in  $\text{Seq}$ . Fix again any  $U \in \mathcal{U}$ . Hence, the  $DL_U.\text{PROVE}(x)$  induced by  $op'$  appears after the  $DL_U.\text{PROVE}(x)$  induced by  $op$  in  $\text{Seq}_U$ . Since the induced  $DL_U.\text{PROVE}(x)$  of  $op$  is invalid, by **PROVE Anti-Flickering** of DL, every subsequent  $DL_U.\text{PROVE}(x)$  in  $\text{Seq}_U$  is invalid.

As this holds for every  $U \in \mathcal{U}$ , there is no component  $DL_U$  in which the induced  $\text{PROVE}(x)$  of  $op'$  is valid.

**READ Liveness.** Let  $R$  the result of a  $\text{READ}()$  operation submit by any correct process.  $(i, \text{prove}(x)) \in R$  implice that  $\exists U^* \in \mathcal{U}$  such that  $(i, \text{prove}(x)) \in R^{U^*}$  with  $R^{U^*}$  the result of  $DL_{U^*}.\text{READ}()$ . By **READ Validity**  $(i, \text{prove}(x)) \in R^{U^*}$  implice that there exist a valid  $DL_{U^*}.\text{PROVE}^{(i)}(x)$ . The for loop in the  $\text{BFT-PROVE}(x)$  implementation return true iff there at least one valid  $DL_U.\text{PROVE}^{(i)}(x)$  for any  $U \in \mathcal{U}$ .

Hence because there exist a  $U^*$  such that  $DL_{U^*}.\text{PROVE}^{(i)}(x)$ , there exist a valid  $\text{BFT-PROVE}^{(i)}(x)$ .  
 $(i, \text{prove}(x)) \in R \implies \exists \text{BFT-PROVE}^{(i)}(x)$

**READ Anti-Flickering.**

**READ Safety.** □

## 6.4 t-BFT-DL $\implies$ t-BFT-GE

**Lemma 6.** *For any fixed value  $r \in S$ , multiple BFT-DenyList Object can be used to implement a BFT-Group Election Object.*

*Proof.*  $Y[i]$

$\triangleright$  Is a set of  $n$  BFT-DL with  $\Pi_M = \Pi_V = \Pi$

```
function BFTVOTE(j, r)
end function
```

```
function BFTCOMMIT(r)
end function
```

```
function BFTRESULT(r)
  Z  $\leftarrow$   $\emptyset$ 
  for each  $j \in \Pi$  do
    if  $|\{(\_, \text{prove}(\_, r)) \in Y[j].\text{BFT-READ}(r)\}| \geq n - f$  then
      P  $\leftarrow$  BFT-READ()
      return  $\{j : (j, \text{prove}), \}$ 
    end if
  end for
end function
```

□

## 6.5 Algorithm

### 6.5.1 Variables

Each process  $p_i$  maintains the following local variables:

```

current  $\leftarrow$  0
received  $\leftarrow$   $\emptyset$ 
delivered  $\leftarrow$   $\emptyset$ 
prop[r][j]  $\leftarrow$   $\perp$ ,  $\forall r, j$ 
 $W_r \leftarrow$   $\perp$ ,  $\forall r$ 
resolved[r]  $\leftarrow$   $\perp$ ,  $\forall r$ 

```

---

**Algorithm D** ABroadcast( $m$ )

---

```

A1 function ABROADCAST( $m$ )
A2    $r \leftarrow$  current
A3    $S \leftarrow$  (received  $\cup$   $\{m\}$ )
A4   for each  $r \in$  {current, current + 1, ...} do
A5     RB-cast( $i, PROP, S, r$ )
A6     wait until  $|W_r| \geq n - f$  where  $W_r =$  BFT-RESULT[ $r$ ]
A7     BFT-COMMIT( $r$ )
A8     wait until  $|\text{resolved}[r]| \geq n - f$ 
A9      $W_r \leftarrow$  BFT-RESULT[ $r$ ]
A10    if  $i \in W_r \vee (\exists j, r' : j \in W_r \wedge \text{prop}[r'][j] \ni m)$  then
A11      break
A12    end if
A13  end for
A14 end function

```

---



---

**Algorithm E** ADELIVER( $m$ )

---

```

B1 function ADELIVER( $m$ )
B2    $r \leftarrow$  current
B3   if  $|\text{resolved}[r]| < n - f$  then
B4     return  $\perp$ 
B5   end if
B6    $W_r \leftarrow$  BFT-RESULT[ $r$ ]
B7   if  $\exists j \in W_r, \text{prop}[r][j] = \perp$  then
B8     return  $\perp$ 
B9   end if
B10   $M_r \leftarrow \bigcup_{j \in W_r} \text{prop}[r][j]$ 
B11   $m \leftarrow$  ordered( $M_r \setminus$  delivered)[0]  $\triangleright$  Set  $m$  as the smaller message not already delivered
B12  delivered  $\leftarrow$  delivered  $\cup$   $\{m\}$ 
B13  if  $M_r \setminus$  delivered =  $\emptyset$  then  $\triangleright$  Check if all messages from round  $r$  have been delivered
B14    current  $\leftarrow$  current + 1
B15  end if
B16  return  $m$ 
B17 end function

```

---

---

**Algorithm F** RB handlers

---

C1 **function** RRECEIVED( $j$ , PROP,  $S$ ,  $r$ )

C2     received  $\leftarrow$  received  $\cup \{S\}$

C3     prop[ $r$ ][ $j$ ]  $\leftarrow S$

C4     BFT-VOTE( $j$ ,  $r$ )

C5 **end function**

C6 **function** RRECEIVED( $j$ , COMMIT,  $r$ )

C7     received[ $r$ ]  $\cup \{j\}$

C8 **end function**

---

## 7 Example of implementation of ARB with DL and RB

We present below an example of implementation of Atomic Reliable Broadcast (ARB) using a Reliable Broadcast (RB) primitive and a DenyList (DL) object according to the model and notations defined in Section 2.

### 7.1 Algorithm

**Definition 2** (Closed round). Given a DL linearization  $H$ , a round  $r \in \mathcal{R}$  is *closed* in  $H$  iff  $H$  contains an operation APPEND( $r$ ). Equivalently, there exists a time after which every PROVE( $r$ ) is invalid in  $H$ .

#### 7.1.1 Variables

Each process  $p_i$  maintains:

received  $\leftarrow \emptyset$

▷ Messages received via RB but not yet delivered

delivered  $\leftarrow \emptyset$

▷ Messages already delivered

prop[ $r$ ][ $j$ ]  $\leftarrow \perp, \forall r, j$

▷ Proposal from process  $j$  for round  $r$

current  $\leftarrow 0$

**DenyList.** The DL is initialized empty. We assume  $\Pi_M = \Pi_V = \Pi$  (all processes can invoke APPEND and PROVE).

#### 7.1.2 Handlers and Procedures

---

**Algorithm G** RB handler (at process  $p_i$ )

---

A1 **function** RBRECEIVED( $S$ ,  $r$ ,  $j$ )

A2     received  $\leftarrow$  received  $\cup \{S\}$

A3     prop[ $r$ ][ $j$ ]  $\leftarrow S$

▷ Record sender  $j$ 's proposal  $S$  for round  $r$

A4 **end function**

---

---

**Algorithm H** AB-broadcast( $m$ ) (at process  $p_i$ )

---

**B1 function** ABBROADCAST( $m$ )  
**B2**  $P \leftarrow \text{READ}()$   $\triangleright$  Fetch latest DL state to learn recent PROVE operations  
**B3**  $r_{max} \leftarrow \max(\{r' : \exists j, (j, \text{PROVE}(r')) \in P\})$   $\triangleright$  Pick current open round  
**B4**  $S \leftarrow (\text{received} \setminus \text{delivered}) \cup \{m\}$   $\triangleright$  Propose all pending messages plus the new  $m$

**B5 for each**  $r \in \{r_{max}, r_{max} + 1, \dots\}$  **do**  
**B6** RB-cast( $S, r, i$ ); PROVE( $r$ ); APPEND( $r$ );  
**B7**  $P \leftarrow \text{READ}()$   $\triangleright$  Refresh local view of DL  
**B8** **if**  $((i, \text{prove}(r)) \in P \vee (\exists j, r' : (j, \text{prove}(r')) \in P \wedge m \in \text{prop}[r'][j]))$  **then**  
**B9** **break**  $\triangleright$  Exit loop once  $m$  is included in some closed round  
**B10** **end if**  
**B11** **end for**  
**B12 end function**

---

---

**Algorithm I** AB-deliver() at process  $p_i$ 

---

**C1 function** ABDELIVER  
**C2**  $r \leftarrow \text{current}$   
**C3**  $P \leftarrow \text{READ}()$   
**C4** **if**  $\forall j : (j, \text{prove}(r)) \notin P$  **then**  
**C5** **return**  $\perp$   
**C6** **end if**  
**C7** APPEND( $r$ );  $P \leftarrow \text{READ}()$   
**C8**  $W_r \leftarrow \{j : (j, \text{prove}(r)) \in P\}$   
**C9** **if**  $\exists j \in W_r, \text{prop}[r][j] = \perp$  **then**  
**C10** **return**  $\perp$   
**C11** **end if**  
**C12**  $M_r \leftarrow \bigcup_{j \in W_r} \text{prop}[r][j]$   
**C13**  $m \leftarrow \text{ordered}(M_r \setminus \text{delivered})[0]$   $\triangleright$  Set  $m$  as the smaller message not already delivered  
**C14**  $\text{delivered} \leftarrow \text{delivered} \cup \{m\}$   
**C15** **if**  $M_r \setminus \text{delivered} = \emptyset$  **then**  $\triangleright$  Check if all messages from round  $r$  have been delivered  
**C16**  $\text{current} \leftarrow \text{current} + 1$   
**C17** **end if**  
**C18** **return**  $m$   
**C19 end function**

---

## 7.2 Correctness

**Lemma 7** (Stable round closure). *If a round  $r$  is closed, then there exists a linearization point  $t_0$  of APPEND( $r$ ) in the DL, and from that point on, no PROVE( $r$ ) can be valid. Once closed, a round never becomes open again.*

*Proof.* By Definition 2, some APPEND( $r$ ) occurs in the linearization  $H$ .

$H$  is a total order of operations, the set of APPEND( $r$ ) operations is totally ordered, and hence there exists a smallest APPEND( $r$ ) in  $H$ . We denote this operation APPEND<sup>(\*)</sup>( $r$ ) and  $t_0$  its linearization point.

By the validity property of DL, a PROVE( $r$ ) is valid iff PROVE( $r$ )  $\prec$  APPEND<sup>(\*)</sup>( $r$ ). Thus, after  $t_0$ , no PROVE( $r$ ) can be valid.

$H$  is a immutable grow-only history, and hence once closed, a round never becomes open again. Hence there exists a linearization point  $t_0$  of  $\text{APPEND}(r)$  in the DL, and from that point on, no  $\text{PROVE}(r)$  can be valid and the closure is stable.  $\square$

**Definition 3** (First APPEND). Given a DL linearization  $H$ , for any closed round  $r \in \mathcal{R}$ , we denote by  $\text{APPEND}^{(*)}(r)$  the earliest  $\text{APPEND}(r)$  in  $H$ .

**Lemma 8** (Across rounds). *If there exists a  $r$  such that  $r$  is closed,  $\forall r'$  such that  $r' < r$ ,  $r'$  is also closed.*

*Proof. Base.* For a closed round  $k = 0$ , the set  $\{k' \in \mathcal{R}, k' < k\}$  is empty, hence the lemma is true.

*Induction.* Assume the lemma is true for round  $k \geq 0$ , we prove it for round  $k + 1$ .

Assume  $k + 1$  is closed and let  $\text{APPEND}^{(*)}(k + 1)$  be the earliest  $\text{APPEND}(k + 1)$  in the DL linearization  $H$ . By Lemma 1, after an  $\text{APPEND}(k)$  is in  $H$ , any later  $\text{PROVE}(k)$  is rejected also, a process's program order is preserved in  $H$ .

There are two possibilities for where  $\text{APPEND}^{(*)}(k + 1)$  is invoked.

- **Case (B6)** : Some process  $p^*$  executes the loop (lines B5–B11) and invokes  $\text{APPEND}^{(*)}(k + 1)$  at line B6. Immediately before line B6, line B5 sets  $r \leftarrow r + 1$ , so the previous loop iteration (if any) targeted  $k$ . We consider two sub-cases.
  - (i)  $p^*$  is not in its first loop iteration. In the previous iteration,  $p^*$  executed  $\text{PROVE}^{(*)}(k)$  at B6, followed (in program order) by  $\text{APPEND}^{(*)}(k)$ . If round  $k$  wasn't closed when  $p^*$  execute  $\text{PROVE}^{(*)}(k)$  at B9, then the condition at B8 would be true hence the tuple  $(p^*, \text{prove}(k))$  should be visible in  $P$  which implies that  $p^*$  should leave the loop at round  $k$ , contradicting the assumption that  $p^*$  is now executing another iteration. Since the tuple is not visible, the  $\text{PROVE}^{(*)}(k)$  was rejected by the DL which implies by definition an  $\text{APPEND}(k)$  already exists in  $H$ . Thus in this case  $k$  is closed.
  - (ii)  $p^*$  is in its first loop iteration. To compute the value  $r_{max}$ ,  $p^*$  must have observed one or many  $(\_, \text{prove}(k + 1))$  in  $P$  at B2/B3, issued by some processes (possibly different from  $p^*$ ). Let's call  $p_1$  the issuer of the first  $\text{PROVE}^{(1)}(k + 1)$  in the linearization  $H$ . When  $p_1$  executed  $P \leftarrow \text{READ}()$  at B2 and compute  $r_{max}$  at B3, he observed no tuple  $(\_, \text{prove}(k + 1))$  in  $P$  because he's the issuer of the first one. So when  $p_1$  executed the loop (B5–B11), he run it for the round  $k$ , didn't seen any  $(1, \text{prove}(k))$  in  $P$  at B8, and then executed the first  $\text{PROVE}^{(1)}(k + 1)$  at B6 in a second iteration. If round  $k$  wasn't closed when  $p_1$  execute  $\text{PROVE}^{(1)}(k)$  at B6, then the condition at B8 should be true which implies that  $p_1$  sould leave the loop at round  $k$ , contradicting the assumption that  $p_1$  is now executing  $\text{PROVE}^{(1)}(r + 1)$ . In this case  $k$  is closed.
- **Case (C8)** : Some process invokes  $\text{APPEND}(k + 1)$  at C8. Line C8 is guarded by the presence of  $\text{PROVE}(next)$  in  $P$  with  $next = k + 1$  (C5). Moreover, the local pointer  $next$  grow by increment of 1 and only advances after finishing the current round (C17), so if a process can reach  $next = k + 1$  it implies that he has completed round  $k$ , which includes closing  $k$  at C8 when  $\text{PROVE}(k)$  is observed. Hence  $\text{APPEND}^{(*)}(k + 1)$  implies a prior  $\text{APPEND}(k)$  in  $H$ , so  $k$  is closed.

In all cases,  $k + 1$  closed implie  $k$  closed. By induction on  $k$ , if the lemme is true for a closed  $k$  then it is true for a closed  $k + 1$ . Therefore, the lemma is true for all closed rounds  $r$ .  $\square$

**Definition 4** (Winner Invariant). For any closed round  $r$ , define

$$\text{Winners}_r \triangleq \{j : \text{PROVE}^{(j)}(r) \prec \text{APPEND}^*(r)\}$$

as the unique set of winners of round  $r$ .

**Lemma 9** (Invariant view of closure). *For any closed round  $r$ , all correct processes eventually observe the same set of valid tuples  $(\_, \text{prove}(r))$  in their DL view.*

*Proof.* Let's take a closed round  $r$ . By Definition 3, there exists a unique earliest  $\text{APPEND}(r)$  in the DL linearization, denoted  $\text{APPEND}^*(r)$ .

Consider any correct process  $p$  that invokes  $\text{READ}()$  after  $\text{APPEND}^*(r)$  in the DL linearization. Since  $\text{APPEND}^*(r)$  invalidates all subsequent  $\text{PROVE}(r)$ , the set of valid tuples  $(\_, \text{prove}(r))$  observed by any correct process after  $\text{APPEND}^*(r)$  is fixed and identical across all correct processes.

Therefore, for any closed round  $r$ , all correct processes eventually observe the same set of valid tuples  $(\_, \text{prove}(r))$  in their DL view.  $\square$

**Lemma 10** (Well-defined winners). *For any correct process and round  $r$ , if the process computes  $W_r$  at line C9, then :*

- $\text{Winners}_r$  is defined;
- the computed  $W_r$  is exactly  $\text{Winners}_r$ .

*Proof.* Let take a correct process  $p_i$  that reach line C9 to compute  $W_r$ .

By program order,  $p_i$  must have executed  $\text{APPEND}^{(i)}(r)$  at C8 before, which implies by Definition 2 that round  $r$  is closed. So by Definition 4,  $\text{Winners}_r$  is defined.

By Lemma 9, all correct processes eventually observe the same set of valid tuples  $(\_, \text{prove}(r))$  in their DL view. Hence, when  $p_i$  executes the  $\text{READ}()$  at C8 after the  $\text{APPEND}^{(i)}(r)$ , it observes a set  $P$  that includes all valid tuples  $(\_, \text{prove}(r))$  such that

$$W_r = \{j : (j, \text{prove}(r)) \in P\} = \{j : \text{PROVE}^{(j)}(r) \prec \text{APPEND}^*(r)\} = \text{Winners}_r$$

$\square$

**Lemma 11** (No APPEND without PROVE). *If some process invokes  $\text{APPEND}(r)$ , then at least a process must have previously invoked  $\text{PROVE}(r)$ .*

*Proof.* Consider the round  $r$  such that some process invokes  $\text{APPEND}(r)$ . There are two possible cases

- **Case (B6)** : There exists a process  $p^*$  who's the issuer of the earliest  $\text{APPEND}^{(*)}(r)$  in the DL linearization  $H$ . By program order,  $p^*$  must have previously invoked  $\text{PROVE}^{(*)}(r)$  before invoking  $\text{APPEND}^{(*)}(r)$  at B6. In this case, there is at least one  $\text{PROVE}(r)$  valid in  $H$  issued by a correct process before  $\text{APPEND}^{(*)}(r)$ .
- **Case (C8)** : There exist a process  $p^*$  invokes  $\text{APPEND}^{(*)}(r)$  at C8. Line C8 is guarded by the condition at C5, which ensures that  $p$  observed some  $(\_, \text{prove}(r))$  in  $P$ . In this case, there is at least one  $\text{PROVE}(r)$  valid in  $H$  issued by some process before  $\text{APPEND}^{(*)}(r)$ .

In both cases, if some process invokes  $\text{APPEND}(r)$ , then some process must have previously invoked  $\text{PROVE}(r)$ .  $\square$

**Lemma 12** (No empty winners). *Let  $r$  be a round, if  $\text{Winners}_r$  is defined, then  $\text{Winners}_r \neq \emptyset$ .*

*Proof.* If  $\text{Winners}_r$  is defined, then by Definition 4, round  $r$  is closed. By Definition 2, some  $\text{APPEND}(r)$  occurs in the DL linearization.

By Lemma 11, at least a process must have invoked a valid  $\text{PROVE}(r)$  before  $\text{APPEND}^*(r)$ . Hence, there exists at least one  $j$  such that  $\{j : \text{PROVE}^{(j)}(r) \prec \text{APPEND}^*(r)\}$ , so  $\text{Winners}_r \neq \emptyset$ .  $\square$

**Lemma 13** (Winners must propose). *For any closed round  $r$ ,  $\forall j \in \text{Winners}_r$ , process  $j$  must have invoked a  $\text{RB-cast}(S^{(j)}, r, j)$*

*Proof.* Fix a closed round  $r$ . By Definition 4, for any  $j \in \text{Winners}_r$ , there exist a valid  $\text{PROVE}^{(j)}(r)$  such that  $\text{PROVE}^{(j)}(r) \prec \text{APPEND}^*(r)$  in the DL linearization. By program order, if  $j$  invoked a valid  $\text{PROVE}^{(j)}(r)$  at line B6 he must have invoked  $\text{RB-cast}(S^{(j)}, r, j)$  directly before.  $\square$

**Definition 5** (Messages invariant). For any closed round  $r$  and any correct process  $p_i$  such that  $\nexists j \in \text{Winners}_r : \text{prop}^{(i)}[r][j] = \perp$ , define

$$\text{Messages}_r \triangleq \bigcup_{j \in \text{Winners}_r} \text{prop}^{(i)}[r][j]$$

as the unique set of messages proposed by the winners of round  $r$ .

**Lemma 14** (Non-empty winners proposal). *For any closed round  $r$ ,  $\forall j \in \text{Winners}_r$ , for any correct process  $p_i$ , eventually  $\text{prop}^{(i)}[r][j] \neq \perp$ .*

*Proof.* Fix a closed round  $r$ . By Definition 4, for any  $j \in \text{Winners}_r$ , there exist a valid  $\text{PROVE}^{(j)}(r)$  such that  $\text{PROVE}^{(j)}(r) \prec \text{APPEND}^*(r)$  in the DL linearization. By Lemma 13,  $j$  must have invoked  $\text{RB-cast}(S^{(j)}, r, j)$ .

Let take a process  $p_i$ , by *RB Validity*, every correct process eventually receives  $j$ 's RB message for round  $r$ , which sets  $\text{prop}[r][j]$  to a non- $\perp$  value at line A3.  $\square$

**Lemma 15** (Eventual proposal closure). *If a correct process  $p_i$  define  $M_r$  at line C13, then for every  $j \in \text{Winners}_r$ ,  $\text{prop}^{(i)}[r][j] \neq \perp$ .*

*Proof.* Let take a correct process  $p_i$  that computes  $M_r$  at line C13. By Lemma 10,  $p_i$  computes the unique winner set  $\text{Winners}_r$ .

By Lemma 12,  $\text{Winners}_r \neq \emptyset$ . The instruction at line C13 where  $p_i$  computes  $M_r$  is guarded by the condition at C10, which ensures that  $p_i$  has received all RB messages from every winner  $j \in \text{Winners}_r$ . Hence, when  $p_i$  computes  $M_r = \bigcup_{j \in \text{Winners}_r} \text{prop}^{(i)}[r][j]$ , we have  $\text{prop}^{(i)}[r][j] \neq \perp$  for all  $j \in \text{Winners}_r$ .  $\square$

**Lemma 16** (Unique proposal per sender per round). *For any round  $r$  and any process  $p_i$ ,  $p_i$  invokes at most one  $\text{RB-cast}(S, r, i)$ .*

*Proof.* By program order, any process  $p_i$  invokes  $\text{RB-cast}(S, r, i)$  at line B6 must be in the loop B5–B11. No matter the number of iterations of the loop, line B5 always uses the current value of  $r$  which is incremented by 1 at each turn. Hence, in any execution, any process  $p_i$  invokes  $\text{RB-cast}(S, r, j)$  at most once for any round  $r$ .  $\square$

**Lemma 17** (Proposal convergence). *For any round  $r$ , for any correct processes  $p_i$  that define  $M_r$  at line C13, we have*

$$M_r^{(i)} = \text{Messages}_r$$

*Proof.* Let take a correct process  $p_i$  that define  $M_r$  at line C13. That implies that  $p_i$  has defined  $W_r$  at line C9. It implies that, by Lemma 10,  $r$  is closed and  $W_r = \text{Winners}_r$ .

By Lemma 15, for every  $j \in \text{Winners}_r$ ,  $\text{prop}^{(i)}[r][j] \neq \perp$ . By Lemma 16, each winner  $j$  invokes at most one  $\text{RB-cast}(S^{(j)}, r, j)$ , so  $\text{prop}^{(i)}[r][j] = S^{(j)}$  is uniquely defined. Hence, when  $p_i$  computes

$$M_r^{(i)} = \bigcup_{j \in \text{Winners}_r} \text{prop}^{(i)}[r][j] = \bigcup_{j \in \text{Winners}_r} S^{(j)} = \text{Messages}_r.$$

□

**Lemma 18** (Inclusion). *If some correct process invokes  $\text{AB-broadcast}(m)$ , then there exist a round  $r$  and a process  $j \in \text{Winners}_r$  such that  $p_j$  invokes*

$$\text{RB-cast}(S, r, j) \quad \text{with} \quad m \in S.$$

*Proof.* Fix a correct process  $p_i$  that invokes  $\text{AB-broadcast}(m)$  and eventually exits the loop (B5–B11) at some round  $r$ . There are two possible cases.

- **Case 1:**  $p_i$  exits the loop because  $(i, \text{prove}(r)) \in P$ . In this case, by Definition 4,  $p_i$  is a winner in round  $r$ . By program order,  $p_i$  must have invoked  $\text{RB-cast}(S, r, i)$  at B6 before invoking  $\text{PROVE}^{(i)}(r)$  at B7. By line B4,  $m \in S$ . Hence there exist a closed round  $r$  and a correct process  $j = i \in \text{Winners}_r$  such that  $j$  invokes  $\text{RB-cast}(S, r, j)$  with  $m \in S$ .
- **Case 2:**  $p_i$  exits the loop because  $\exists j, r' : (j, \text{prove}(r')) \in P \wedge m \in \text{prop}[r'][j]$ . In this case, by Lemma 13 and Lemma 16  $j$  must have invoked a unique  $\text{RB-cast}(S, r', j)$ . Which set  $\text{prop}^{(i)}[r'][j] = S$  with  $m \in S$ .

In both cases, if some correct process invokes  $\text{AB-broadcast}(m)$ , then there exist a round  $r$  and a correct process  $j \in \text{Winners}_r$  such that  $j$  invokes

$$\text{RB-cast}(S, r, j) \quad \text{with} \quad m \in S.$$

□

**Lemma 19** (Broadcast Termination). *If a correct process invokes  $\text{AB-broadcast}(m)$ , then he eventually exit the function and returns.*

*Proof.* Let a correct process  $p_i$  that invokes  $\text{AB-broadcast}(m)$ . The lemma is true if  $\exists r_1$  such that  $r_1 \geq r_{max}$  and if  $(i, \text{prove}(r_1)) \in P$ ; or if  $\exists r_2$  such that  $r_2 \geq r_{max}$  and if  $\exists j : (j, \text{prove}(r_2)) \in P \wedge m \in \text{prop}[r_2][j]$  (like guarded at B8).

Let admit that there exists no round  $r_1$  such that  $p_i$  invokes a valid  $\text{PROVE}(r_1)$ . In this case  $p_i$  invokes infinitely many  $\text{RB-cast}(S, \_, i)$  at B6 with  $m \in S$  (line B4).

The assumption that no  $\text{PROVE}(r_1)$  invoked by  $p$  is valid implies by DL *Validity* that for every round  $r' \geq r_{max}$ , there exists at least one  $\text{APPEND}(r')$  in the DL linearization, hence by Lemma 12 for every possible round  $r'$  there at least a winner. Because there is an infinite number of rounds, and a finite number of processes, there exists at least a correct process  $p_k$  that invokes infinitely many valid  $\text{PROVE}(r')$  and by extension infinitely many  $\text{AB-broadcast}(\_)$ . By RB *Validity*,  $p_k$  eventually receives  $p_i$ 's RB messages. Let call  $t_0$  the time when  $p_k$  receives  $p_i$ 's RB message.

At  $t_0$ ,  $p_k$  execute Algorithm G and do  $\text{received} \leftarrow \text{received} \cup \{S\}$  with  $m \in S$  (line A2). For the first invocation of  $\text{AB-broadcast}(\_)$  by  $p_k$  after the execution of Algorithm G,  $p_k$  must include  $m$  in his proposal  $S$  at line B4 (because  $m$  is pending in  $j$ 's  $\text{received} \setminus \text{delivered}$  set). There exists a minimum

round  $r_2$  such that  $p_k \in \text{Winners}_{r_2}$  after  $t_0$ . By Lemma 14, eventually  $\text{prop}^{(i)}[r_2][k] \neq \perp$ . By Lemma 16,  $\text{prop}^{(i)}[r_2][k]$  is uniquely defined as the set  $S$  proposed by  $p_k$  at B6, which by Lemma 18 includes  $m$ . Hence eventually  $m \in \text{prop}^{(i)}[r_2][k]$  and  $k \in \text{Winners}_{r_2}$ .

So if  $p_i$  is a winner he cover the condition  $(i, \text{prove}(r_1)) \in P$ . And we show that if the first condition is never satisfied, the second one will eventually be satisfied. Hence either the first or the second condition will eventually be satisfied, and  $p_i$  eventually exits the loop and returns from  $\text{AB-broadcast}(m)$ .  $\square$

**Lemma 20** (Validity). *If a correct process  $p$  invokes  $\text{AB-broadcast}(m)$ , then every correct process that invokes a infinitely often times  $\text{AB-deliver}()$  eventually delivers  $m$ .*

*Proof.* Let  $p_i$  a correct process that invokes  $\text{AB-broadcast}(m)$  and  $p_q$  a correct process that infinitely invokes  $\text{AB-deliver}()$ . By Lemma 18, there exist a closed round  $r$  and a correct process  $j \in \text{Winners}_r$ , such that  $p_j$  invokes

$$\text{RB-cast}(S, r, j) \quad \text{with} \quad m \in S.$$

By Lemma 15, when  $p_q$  computes  $M_r$  at line C13,  $\text{prop}[r][j]$  is non- $\perp$  because  $j \in \text{Winners}_r$ . By Lemma 16,  $p_j$  invokes at most one  $\text{RB-cast}(S, r, j)$ , so  $\text{prop}[r][j]$  is uniquely defined. Hence, when  $p_q$  computes

$$M_r = \bigcup_{k \in \text{Winners}_r} \text{prop}[r][k],$$

we have  $m \in \text{prop}[r][j] = S$ , so  $m \in M_r$ . By Lemma 17,  $M_r$  is invariant so each computation of  $M_r$  by any correct process that defines it includes  $m$ . At each invocation of  $\text{AB-deliver}()$  which deliver  $m'$ ,  $m'$  is add to  $\text{delivered}$  until  $M_r \subseteq \text{delivered}$ . Once this append we're assured that there exist an invocation of  $\text{AB-deliver}()$  which return  $m$ . Hence  $m$  is well delivered.  $\square$

**Lemma 21** (No duplication). *No correct process delivers the same message more than once.*

*Proof.* Let consider two invocations of  $\text{AB-deliver}()$  made by the same correct process which returns  $m$ . Let call these two invocations respectively  $\text{AB-deliver}^{(A)}()$  and  $\text{AB-deliver}^{(B)}()$ .

When  $\text{AB-deliver}^{(A)}()$  occur, by program order and because it reach line C19 to return  $m$ , the process must have add  $m$  to  $\text{delivered}$ . Hence when  $\text{AB-deliver}^{(B)}()$  reach line C14 to extract the next message to deliver, it can't be  $m$  because  $m \notin (M_r \setminus \{\dots, m, \dots\})$ . So a  $\text{AB-deliver}^{(B)}()$  which deliver  $m$  can't occur.  $\square$

**Lemma 22** (Total order). *For any two messages  $m_1$  and  $m_2$  delivered by correct processes, if a correct process  $p_i$  delivers  $m_1$  before  $m_2$ , then any correct process  $p_j$  that delivers both  $m_1$  and  $m_2$  delivers  $m_1$  before  $m_2$ .*

*Proof.* Consider any correct process that delivers both  $m_1$  and  $m_2$ . By Lemma 20, there exist closed rounds  $r'_1$  and  $r'_2$  and correct processes  $k_1 \in \text{Winners}_{r'_1}$  and  $k_2 \in \text{Winners}_{r'_2}$  such that

$$\text{RB-cast}(S_1, r'_1, k_1) \quad \text{with} \quad m_1 \in S_1,$$

$$\text{RB-cast}(S_2, r'_2, k_2) \quad \text{with} \quad m_2 \in S_2.$$

Let consider three cases :

- **Case 1:**  $r_1 < r_2$ . By program order, any correct process must have waited to append in  $\text{delivered}$  every messages in  $M_{r_1}$  (which contains  $m_1$ ) to increment  $\text{current}$  and eventually set  $\text{current} = r_2$  to compute  $M_{r_2}$  and then invoke the  $\text{AB-deliver}()$  that returns  $m_2$ . Hence, for any correct process that delivers both  $m_1$  and  $m_2$ , it delivers  $m_1$  before  $m_2$ .

- **Case 2:**  $r_1 = r_2$ . By Lemma 17, any correct process that computes  $M_{r_1}$  at line C13 computes the same set of messages  $\text{Messages}_{r_1}$ . By line C14 the messages are pulled in a deterministic order defined by  $\text{ordered}(\_)$ . Hence, for any correct process that delivers both  $m_1$  and  $m_2$ , it delivers  $m_1$  and  $m_2$  in the deterministic order defined by  $\text{ordered}(\_)$ .

In all possible cases, any correct process that delivers both  $m_1$  and  $m_2$  delivers  $m_1$  and  $m_2$  in the same order.  $\square$

**Lemma 23** (Fifo Order). *For any two messages  $m_1$  and  $m_2$  broadcast by the same correct process  $p_i$ , if  $p_i$  invokes  $\text{AB-broadcast}(m_1)$  before  $\text{AB-broadcast}(m_2)$ , then any correct process  $p_j$  that delivers both  $m_1$  and  $m_2$  delivers  $m_1$  before  $m_2$ .*

*Proof.* Let take two messages  $m_1$  and  $m_2$  broadcast by the same correct process  $p_i$ , with  $p_i$  invoking  $\text{AB-broadcast}(m_1)$  before  $\text{AB-broadcast}(m_2)$ . By Lemma 20, there exist closed rounds  $r_1$  and  $r_2$  and correct processes  $k_1 \in \text{Winners}_{r_1}$  and  $k_2 \in \text{Winners}_{r_2}$  such that

$$\text{RB-cast}(S_1, r_1, k_1) \quad \text{with} \quad m_1 \in S_1,$$

$$\text{RB-cast}(S_2, r_2, k_2) \quad \text{with} \quad m_2 \in S_2.$$

By program order,  $p_i$  must have invoked  $\text{RB-cast}(S_1, r_1, i)$  before  $\text{RB-cast}(S_2, r_2, i)$ . By Lemma 16, any process invokes at most one  $\text{RB-cast}(S, r, i)$  per round, hence  $r_1 < r_2$ . By Lemma 22, any correct process that delivers both  $m_1$  and  $m_2$  delivers them in a deterministic order.

In all possible cases, any correct process that delivers both  $m_1$  and  $m_2$  delivers  $m_1$  before  $m_2$ .  $\square$

**Theorem 24** (FIFO-ARB). *Under the assumed DL synchrony and RB reliability, the algorithm implements FIFO Atomic Reliable Broadcast.*

*Proof.* We show that the algorithm satisfies the properties of FIFO Atomic Reliable Broadcast under the assumed DL synchrony and RB reliability.

First, by Lemma 19, if a correct process invokes  $\text{AB-broadcast}(m)$ , then it eventually returns from this invocation. Moreover, Lemma 20 states that if a correct process invokes  $\text{AB-broadcast}(m)$ , then every correct process that invokes  $\text{AB-deliver}()$  infinitely often eventually delivers  $m$ . This gives the usual Validity property of ARB.

Concerning Integrity and No-duplicates, the construction only ever delivers messages that have been obtained from the underlying RB primitive. By the Integrity property of RB, every such message was previously  $\text{RB-cast}$  by some process, so no spurious messages are delivered. In addition, Lemma 21 states that no correct process delivers the same message more than once. Together, these arguments yield the Integrity and No-duplicates properties required by ARB.

For the ordering guarantees, Lemma 22 shows that for any two messages  $m_1$  and  $m_2$  delivered by correct processes, every correct process that delivers both  $m_1$  and  $m_2$  delivers them in the same order. Hence all correct processes share a common total order on delivered messages. Furthermore, Lemma 23 states that for any two messages  $m_1$  and  $m_2$  broadcast by the same correct process, any correct process that delivers both messages delivers  $m_1$  before  $m_2$  whenever  $m_1$  was broadcast before  $m_2$ . Thus the global total order extends the per-sender FIFO order of  $\text{AB-broadcast}$ .

All the above lemmas are proved under the assumptions that DL satisfies the required synchrony properties and that the underlying primitive is a Reliable Broadcast (RB) with Integrity, No-duplicates and Validity. Therefore, under these assumptions, the algorithm satisfies Validity, Integrity/No-duplicates, total order and per-sender FIFO order, and hence implements FIFO Atomic Reliable Broadcast, as claimed.  $\square$

### 7.3 Reciprocity

So far, we assumed the existence of a synchronous DenyList (DL) object and showed how to upgrade a Reliable Broadcast (RB) primitive into FIFO Atomic Reliable Broadcast (ARB). We now briefly argue that, conversely, an ARB primitive is strong enough to implement a synchronous DL object (ignoring the anonymity property).

**DenyList as a deterministic state machine.** Without anonymity, the DL specification defines a deterministic abstract object: given a sequence  $\text{Seq}$  of operations  $\text{APPEND}(x)$ ,  $\text{PROVE}(x)$ , and  $\text{READ}()$ , the resulting sequence of return values and the evolution of the abstract state (set of appended elements, history of operations) are uniquely determined.

**State machine replication over ARB.** Assume a system that exports a FIFO-ARB primitive with the guarantees that if a correct process invokes  $\text{AB-broadcast}(m)$ , then every correct process eventually  $\text{AB-deliver}(m)$  and the invocation eventually returns. Following the classical *state machine replication* approach such as described in Schneider [1], we can implement a fault-tolerant service by ensuring the following properties:

**Agreement.** Every nonfaulty state machine replica receives every request.

**Order.** Every nonfaulty state machine replica processes the requests it receives in the same relative order.

Which are covered by our FIFO-ARB specification.

**Correctness.**

**Theorem 25** (From ARB to synchronous DL). *In an asynchronous message-passing system with crash failures, assume a FIFO Atomic Reliable Broadcast primitive with Integrity, No-duplicates, Validity, and the liveness of AB-broadcast. Then, ignoring anonymity, there exists an implementation of a synchronous DenyList object that satisfies the Termination, Validity, and Anti-flickering properties.*

*Proof.* Because the DL object is deterministic, all correct processes see the same sequence of operations and compute the same sequence of states and return values. We obtain:

- **Termination.** The liveness of ARB ensures that each AB-broadcast invocation by a correct process eventually returns, and the corresponding operation is eventually delivered and applied at all correct processes. Thus every APPEND, PROVE, and READ operation invoked by a correct process eventually returns.
- **APPEND/PROVE/READ Validity.** The local code that forms AB-broadcast requests can achieve the same preconditions as in the abstract DL specification (e.g.,  $p \in \Pi_M$ ,  $x \in S$  for  $\text{APPEND}(x)$ ). Once an operation is delivered, its effect and return value are exactly those of the sequential DL specification applied in the common order.
- **PROVE Anti-Flickering.** In the sequential DL specification, once an element  $x$  has been appended, all subsequent  $\text{PROVE}(x)$  are invalid forever. Since all replicas apply operations in the same order, this property holds in every execution of the replicated implementation: after the first linearization point of  $\text{APPEND}(x)$ , no later  $\text{PROVE}(x)$  can return “valid” at any correct process.

Formally, we can describe the DL object with the state machine approach for crash-fault, asynchronous message-passing systems with a total order broadcast layer [1].  $\square$

### 7.3.1 Example executions

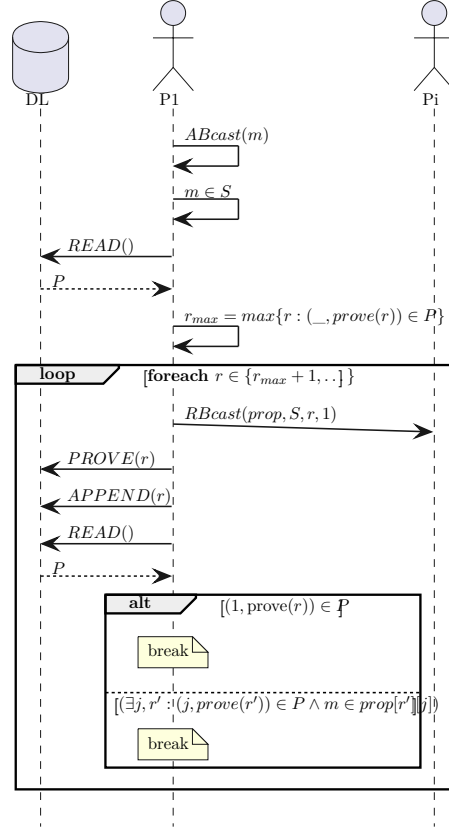


Figure 1: Example execution of the ARB algorithm in a non-BFT setting

## 8 Implementation of BFT-DenyList and Threshold Cryptography

### 8.1 DenyList

**BFT-DenyList** In our algorithm we use multiple DenyList as follows:

- Let  $\mathcal{DL} = \{DL_1, \dots, DL_k\}$  be the set of DenyList used by the algorithm.
- We set  $k = \binom{n}{f}$ .
- For each  $i \in \{1, \dots, k\}$ , let  $M_i$  be the set of moderators associated with  $DL_i$  according to the DenyList definition, so that  $|M_i| = n - f$ .
- Let  $\mathcal{M} = \{M_1, \dots, M_k\}$ . We require that the  $M_i$  are pairwise distinct:

$$\forall i, j \in \{1, \dots, k\}, i \neq j \implies M_i \neq M_j.$$

**Lemma 26.**  $\exists M_i \in \mathcal{M} : \forall p \in M_i$   $p$  is correct.

*Proof.* Let consider the set  $F$  of faulty processes, with  $|F| = f$ . We can construct the set  $M_i = \Pi \setminus F$  such that  $|M_i| = n - |F| = n - f$ . By construction,  $\forall p \in M_i$   $p$  is correct.  $\square$

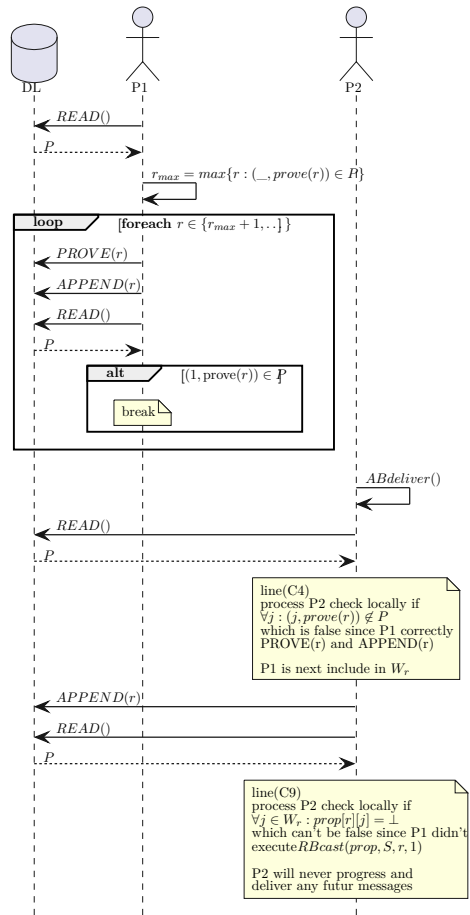


Figure 2: Example execution of the ARB algorithm with a Byzantine process

**Lemma 27.**  $\forall M_i \in M, \exists p \in M_i$  such that  $p$  is correct.

*Proof.*  $\forall i \in \{1, \dots, k\}, |M_i| = n - f$  with  $n \geq 2f + 1$ . We can say that  $|M_i| \geq 2f + 1 - f = f + 1 > f$   $\square$

Each process can invoke the following functions :

- READ' :  $() \rightarrow \mathcal{L}(\mathbb{R} \times \text{prove}(\mathbb{R}))$
- APPEND' :  $\mathbb{R} \rightarrow ()$
- PROVE' :  $\mathbb{R} \rightarrow \{0, 1\}$

Such that :

---

**Algorithm J** READ'( $() \rightarrow \mathcal{L}(\mathbb{R} \times \text{prove}(\mathbb{R}))$ )

---

```

function READ'
   $j \leftarrow$  the process invoking READ'()
   $res \leftarrow \emptyset$ 
  for all  $i \in \{1, \dots, k\}$  do
     $res \leftarrow res \cup DL_i.READ()$ 
  end for
  return  $res$ 
end function

```

---



---

**Algorithm K** APPEND'( $\sigma \rightarrow ()$ )

---

```

function APPEND'( $\sigma$ )
   $j \leftarrow$  the process invoking APPEND'( $\sigma$ )
  for all  $M_i \in \{M_k \in M : j \in M_k\}$  do
     $DL_i.APPEND(\sigma)$ 
  end for
end function

```

---



---

**Algorithm L** PROVE'( $\sigma \rightarrow \{0, 1\}$ )

---

```

function PROVE'( $\sigma$ )
   $j \leftarrow$  the process invoking PROVE'( $\sigma$ )
   $flag \leftarrow false$ 
  for all  $i \in \{1, \dots, k\}$  do
     $flag \leftarrow flag \text{ OR } DL_i.PROVE(\sigma)$ 
  end for
  return  $flag$ 
end function

```

---

## 8.2 Threshold Cryptography

We are using the Boneh-Lynn-Shacham scheme as cryptography primitive to our threshold signature scheme. With :

- $G : \mathbb{R} \rightarrow \mathbb{R} \times \mathbb{R}$
- $S : \mathbb{R} \times \mathcal{R} \rightarrow \mathbb{R}$
- $V : \mathbb{R} \times \mathcal{R} \times \mathbb{R} \rightarrow \{0, 1\}$

Such that :

- $G(x) \rightarrow (pk, sk) : \text{where } x \text{ is a random value such that } \nexists x_1, x_2 : x_1 \neq x_2, G(x_1) = G(x_2)$
- $S(sk, m) \rightarrow \sigma_m$
- $V(pk, m_1, \sigma_{m_2}) \rightarrow k : \text{with } k = 1 \text{ iff } m_1 == m_2 \text{ and } \exists x \in \mathbb{R} \text{ such that } G(x) \rightarrow (pk, sk); \text{ otherwise } k = 0$

**threshold Scheme** In our algorithm we are only using the following functions :

- $G' : \mathbb{R} \times \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{R} \times (\mathbb{R} \times \mathbb{R})^n : \text{with } n \triangleq |\Pi|$
- $S' : \mathbb{R} \times \mathcal{R} \rightarrow \mathbb{R}$
- $C' : \mathbb{R}^n \times \mathcal{R} \times \mathbb{R} \times \mathbb{R}^t \rightarrow \{\mathbb{R}, \perp\} : \text{with } t \leq n$
- $V' : \mathbb{R} \times \mathcal{R} \times \mathbb{R} \rightarrow \{0, 1\}$

Such that :

- $G'(x, n, t) \rightarrow (pk, pk_1, sk_1, \dots, pk_n, sk_n) : \text{let define } pkc = pk_1, \dots, pk_n$
- $S'(sk_i, m) \rightarrow \sigma_m^i$
- $C'(pkc, m_1, J, \{\sigma_{m_2}^j\}_{j \in J}) \rightarrow \sigma : \text{with } J \subseteq \Pi; \text{ and } \sigma = \sigma_{m_1} \text{ iff } |J| \geq t, \forall j \in J : V(pk_j, m_1, \sigma_{m_2}^j) == 1; \text{ otherwise } \sigma = \perp.$
- $V'(pk, m_1, \sigma_{m_2}) \rightarrow V(pk, m_1, \sigma_{m_2})$

## References

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