

# 1 Model 1: Crash

We consider a static set  $\Pi$  of  $n$  processes with known identities, communicating by reliable point-to-point channels, in a complete graph. Messages are uniquely identifiable. At most  $f$  processes can crash, with  $n \geq f$ .

**Synchrony.** The network is asynchronous.

**Communication.** Processes communicate through reliable, error-free point-to-point channels. Messages sent by a correct process to another correct process are eventually delivered without loss or corruption. There exists a shared object called DenyList (DL) (defined below) that is interfaced with a set  $O$  of operations. There exist three types of these operations: **append**( $x$ ), **prove**( $x$ ) and **read**().

**Notation.** For any indice  $x$  we defined by  $\Pi_x$  a subset of  $\Pi$ . We consider two subsets  $\Pi_M$  and  $\Pi_V$  two authorization subsets. Indices  $i \in \Pi$  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 operation  $F \in O$ ,  $F_i(\dots)$  denotes that the operation  $F$  is invoked by process  $p_i$ .

## 2 Primitives

### 2.1 DenyList Object

We assume a linearizable 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 **append**( $x$ ) by a process  $p$  is valid if:
  - $p \in \Pi_M \subseteq \Pi$ ; **and**
  - $x \in S$ , where  $S$  denote the universe of valid entries to be appended to the DenyList.

Otherwise, the operation is invalid.

- **PROVE Validity.** Let  $op$  the invocation of **prove**( $x$ ) by a process  $p_i$ . We said  $op$  to be invalid, if and only if:
  - $p \notin \Pi_V \subseteq \Pi$ ; **or**
  - A valid **append**( $x$ ) appears before  $op$  in **Seq**.

Otherwise, the operation is said to be 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  $\text{Seq}$  is invalid.
- **READ Validity.** The invocation of  $op = \text{read}()$  by a process  $p \in \pi_V$  returns the list of valid invocations of  $\text{prove}$  that appears before  $op$  in  $\text{Seq}$  along with the names of the processes that invoked each operation.

We assume that  $\Pi_M = \Pi_V = \Pi$  (all processes can invoke  $\text{append}$  and  $\text{prove}$ ).

### 3 Atomic Reliable Broadcast (ARB)

Processes export  $\text{ABROADCAST}(m)$  and  $m = \text{ADELIVER}()$ . ARB requires the following properties:

- **Total Order:**

$$\forall m_1, m_2, \forall p_i, p_j : (m_1 = \text{ADELIVER}_i()) \prec (m_2 = \text{ADELIVER}_i()) \Rightarrow (m_1 = \text{ADELIVER}_j()) \prec (m_2 = \text{ADELIVER}_j())$$

- **Integrity:** Every message delivered was previously broadcast.  $\forall p_i : m = \text{ADELIVER}_i() \Rightarrow \exists p_j : \text{ABROADCAST}_j(m)$ .
- **No-duplicates:** No message is delivered more than once at any process.
- **Validity:** If a correct process broadcasts  $m$ , every correct process eventually delivers  $m$ .

### 4 ARB using DL

We present below an example of implementation of Atomic Reliable Broadcast (ARB) using point-to-point reliable, error-free channels and a DenyList (DL) object according to the model and notations defined in Section 2.

#### 4.1 Algorithm

**Definition 1** (Closed round). Given a DL linearization  $H$ , a round  $r \in \mathcal{R}$  is *closed* in  $H$  if  $H$  contains an operation  $\text{append}(r)$ . Equivalently, there exists a time after which every  $\text{prove}(r)$  is invalid in  $H$ .

### 4.1.1 Handlers and Procedures

---

**Algorithm 1:** ARB at process  $p_i$

---

1.1 **Local Variables:**  
1.2  $unordered \leftarrow \emptyset, ordered \leftarrow \epsilon, delivered \leftarrow \epsilon;$   
1.3  $prop[r][j] \leftarrow \perp, \forall r, j;$

1.4 **for**  $r = 1, 2, \dots$  **do**  
1.5  $\quad$  **wait until**  $unordered \setminus ordered \neq \emptyset;$   
1.6  $\quad$   $S \leftarrow (unordered \setminus ordered);$   
1.7  $\quad$  **foreach**  $j \in \Pi$  **do**  $send(PROP, S, \langle r, j \rangle)$  **to**  $p_j$  ;  
1.8  $\quad$   $prove(r); append(r);$   
1.9  $\quad$   $winners[r] \leftarrow \{j : (j, r) \in read()\};$   
1.10  $\quad$  **wait until**  $\forall j \in winners[r], prop[r][j] \neq \perp;$   
1.11  $\quad$   $M \leftarrow \bigcup_{j \in winners[r]} prop[r][j];$   
1.12  $\quad$   $ordered \leftarrow ordered \cdot order(M);$

1.13 **Upon**  $ABROADCAST(m)$   
1.14  $\quad$   $unordered \leftarrow unordered \cup \{m\};$

1.15 **Upon**  $receive(PROP, S, \langle r, j \rangle)$  *from process*  $p_j$   
1.16  $\quad$   $unordered \leftarrow unordered \cup \{S\};$   
1.17  $\quad$   $prop[r][j] \leftarrow S;$

1.18 **Upon**  $ADELIVER()$   
1.19  $\quad$  **if**  $ordered \setminus delivered = \emptyset$  **then return**  $\perp$  ;  
1.20  $\quad$  let  $m$  be the first element in  $(ordered \setminus delivered);$   
1.21  $\quad$   $delivered \leftarrow delivered \cdot m;$   
1.22  $\quad$  **return**  $m$

---

## 4.2 Correctness

**Definition 2** (First APPEND). Given a DL linearization  $H$ , for any closed round  $r \in \mathcal{R}$ , we denote by  $append^{(*)}(r)$  the earliest  $append(r)$  in  $H$ .

*Remark 1* (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 1, 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  $append^{(*)}(r)$  and  $t_0$  its linearization point.

By the validity property of DL, a  $prove(r)$  is valid iff  $prove(r) \prec append^{(*)}(r)$ . 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  $append(r)$  in the DL, and from that point on, no  $prove(r)$  can be valid and the closure is stable.  $\square$

**Lemma 1** (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  $r = 0$ , the set  $\{r' \in \mathcal{R} : r' < r\}$  is empty, so the claim holds.

*Induction step.* Assume  $r + 1$  is closed. By Definition 2, there exists an earliest operation  $\text{append}^{(*)}(r + 1)$  in the DL linearization  $H$ . Let  $p_j$  be the process that invokes this operation.

In Algorithm 1, the call to  $\text{append}(\cdot)$  appears at line 1.8, inside the loop indexed by rounds  $1, 2, \dots$ . Therefore, if  $p_j$  reaches line 1.8 for round  $r + 1$ , then in the previous loop iteration (round  $r$ ) process order implies that  $p_j$  has already invoked  $\text{append}(r)$  at the same line.

Hence an  $\text{append}(r)$  exists in  $H$ , so round  $r$  is closed by Definition 1. We proved:

$$(r + 1 \text{ closed}) \Rightarrow (r \text{ closed}).$$

By repeated application, if some round  $r$  is closed, then every round  $r' < r$  is also closed.  $\square$

**Definition 3** (Winner Invariant). For any closed round  $r$ , define

$$\text{Winners}_r \triangleq \{j : \text{prove}_j(r) \prec \text{append}^{(*)}(r)\}$$

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

**Lemma 2** (Invariant view of closure). *For any closed round  $r$ , all correct processes eventually observe the same set of valid tuples  $(\cdot, r)$  in their DLview.*

*Proof.* Let's take a closed round  $r$ . By Definition 2, there exists  $\text{append}^{(*)}(r)$  the earliest  $\text{append}(r)$  in the DL linearization.

Consider any correct process  $p_i$  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  $(\_, r)$  retrieved by a  $\text{read}()$  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  $(\cdot, r)$  in their DLview.  $\square$

**Lemma 3** (Well-defined winners). *For any correct process  $p_i$  and round  $r$ , if  $p_i$  computes  $\text{winners}[r]$  at line 1.9, then :*

- $\text{Winners}_r$  is defined;
- the computed  $\text{winners}[r]$  is exactly  $\text{Winners}_r$ .

*Proof.* Lets consider a correct process  $p_i$  that reach line 1.9 to compute  $\text{winners}[r]$ .

By program order,  $p_i$  must have executed  $\text{append}_i(r)$  at line 1.8 before, which implies by Definition 1 that round  $r$  is closed at that point. So by Definition 3,  $\text{Winners}_r$  is defined.

By Lemma 2, all correct processes eventually observe the same set of valid tuples  $(\cdot, r)$  in their DLview. Hence, when  $p_i$  executes the  $\text{read}()$  at line 1.9 after the  $\text{append}_i(r)$ , it observes a set  $P$  that includes all valid tuples  $(\cdot, r)$  such that

$$\text{winners}[r] = \{j : (j, r) \in P\} = \{j : \text{prove}_j(r) \prec \text{append}^{(*)}(r)\} = \text{Winners}_r$$

$\square$

**Lemma 4** (Winners are non-empty). *For any closed round  $r$ , there exists at least one process  $p_j$  that invoked  $\text{prove}_j(r) \prec \text{append}^{(*)}(r)$ , so  $\text{Winners}_r \neq \emptyset$ .*

*Proof.* Let  $r$  be a closed round. By Definition 1, some  $\text{append}(r)$  occurs in the DLlinearization  $H$ . Let  $p_i$  be the process invoking the earliest such operation,  $\text{append}^{(*)}(r)$ .

By program order in Algorithm 1, the call to  $\text{append}(\cdot)$  at line 1.8 is immediately preceded by  $\text{prove}(r)$ . Thus  $p_i$  must have invoked  $\text{prove}_i(r)$  before  $\text{append}^{(*)}(r)$ , and by the sequence of code at line 1.8, this  $\text{prove}_i(r)$  executes before  $\text{append}^{(*)}(r)$  in the linearization.

By Definition 3,  $p_i \in \text{Winners}_r$ . Hence  $\text{Winners}_r \neq \emptyset$ .  $\square$

**Lemma 5** (Winners must propose). *For any closed round  $r$ ,  $\forall i \in \text{Winners}_r$ , process  $p_i$  must have sent messages to all processes  $j \in \Pi$ , and hence any correct process  $p_j$  will eventually receive  $p_i$ 's message for round  $r$  and set  $\text{prop}[r][i]$  to a non- $\perp$  value.*

*Proof.* Fix a closed round  $r$ . By Definition 3, for any  $i \in \text{Winners}_r$ , there exists a valid  $\text{prove}_i(r)$  such that  $\text{prove}_i(r) \prec \text{append}^{(*)}(r)$  in the DL linearization. By program order in Algorithm 1,  $p_i$  must have sent messages to all  $j \in \Pi$  at line 1.8 before invoking  $\text{prove}(r)$ .

If  $p_i$  is a correct process that completed sending to all processes, then by the reliable and error-free nature of the communication channels, every correct process  $p_j$  will eventually receive  $p_i$ 's message, which sets  $\text{prop}[r][i] \leftarrow S$  at line 1.17. If  $p_i$  crashes before sending to all processes, then  $p_i$  cannot invoke a valid  $\text{prove}_i(r)$  afterwards, contradicting the assumption that  $i \in \text{Winners}_r$ . Hence  $p_i$  must have completed sending to all processes.  $\square$

**Definition 4** (Messages invariant). For any closed round  $r$  and any correct process  $p_i$  such that  $\forall j \in \text{Winners}_r : \text{prop}^{(i)}[r][j] \neq \perp$ , define

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

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

**Lemma 6** (Eventual proposal closure). *If a correct process  $p_i$  define  $M$  at line 1.11, 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  $M$  at line 1.11. By Lemma 3,  $p_i$  computation is the winner set  $\text{Winners}_r$ .

By Lemma 4,  $\text{Winners}_r \neq \emptyset$ . The instruction at line 1.11 where  $p_i$  computes  $M$  is guarded by the condition at line 1.10, which ensures that  $p_i$  has received messages from every winner  $j \in \text{Winners}_r$ . By Lemma 5, each winner  $j$  has sent messages to all processes including  $p_i$ . Thus, by the reliable and error-free nature of the channels, if  $p_i$  is correct, it will eventually receive  $j$ 's message, setting  $\text{prop}^{(i)}[r][j] \neq \perp$  at line 1.17. Hence,  $\text{prop}^{(i)}[r][j] \neq \perp$  for all  $j \in \text{Winners}_r$ .  $\square$

**Lemma 7** (Unique proposal per sender per round). *For any round  $r$  and any process  $p_i$ ,  $p_i$  sends messages to all processes at most once for each round.*

*Proof.* In Algorithm 1, the only place where a process  $p_i$  can send messages to all processes is at line 1.8, which appears inside the main loop indexed by rounds  $r = 1, 2, \dots$

Each iteration of this loop processes exactly one round value  $r$ , and within that iteration, messages are sent at most once (before the  $\text{prove}(r)$  and  $\text{append}(r)$  calls). Since the loop variable  $r$  takes each value  $1, 2, \dots$  at most once during the execution, process  $p_i$  sends messages at most once for any given round  $r$ .  $\square$

**Lemma 8** (Proposal convergence). *For any round  $r$ , for any correct processes  $p_i$  that execute line 1.11, we have*

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

*Proof.* Let take a correct process  $p_i$  that compute  $M$  at line 1.11. That implies that  $p_i$  has defined  $winner_r$  at line 1.9. It implies that, by Lemma 3,  $r$  is closed and  $winner_r = \text{Winners}_r$ .

By Lemma 27, for every  $j \in \text{Winners}_r$ ,  $prop^{(i)}[r][j] \neq \perp$ . By Lemma 23, each winner  $j$  sends messages to all processes at most once per round. Thus,  $prop^{(i)}[r][j] = S^{(j)}$  is uniquely defined as the messages sent by  $j$  in that round. Hence, when  $p_i$  computes

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

□

**Lemma 9** (Inclusion). *If some correct process invokes  $\text{ABROADCAST}(m)$ , then there exist a round  $r$  and a process  $j \in \text{Winners}_r$  such that  $p_j$  sends a proposal  $S$  to all processes at line 1.8 with  $m \in S$ .*

*Proof.* Let  $p_i$  be a correct process that invokes  $\text{ABROADCAST}(m)$ . By the handler at line 1.14,  $m$  is added to  $unordered$ . Since  $p_i$  is correct, it continues executing the main loop.

Consider any iteration of the loop where  $p_i$  executes line 1.6 while  $m \in (unordered \setminus ordered)$ . At that iteration, for some round  $r$ , process  $p_i$  constructs  $S$  containing  $m$  and sends  $S$  to all processes at line 1.8.

We distinguish two cases:

- **Case 1:  $p_i$  is a winner.** If  $p_i \in \text{Winners}_r$  for this round  $r$ , then by Definition 3 and program order,  $p_i$  has sent proposal  $S$  to all processes with  $m \in S$ , and the lemma holds with  $j = i$ .
- **Case 2:  $p_i$  is not a winner.** If  $p_i \notin \text{Winners}_r$ , then  $p_i$  is still a correct process, so it has sent its proposal  $S$  (containing  $m$ ) to all processes in  $\Pi$ . By the reliable and error-free nature of the communication channels, all correct processes will eventually receive  $p_i$ 's message. By line 1.16, each correct process  $p_k$  adds  $m$  to its own  $unordered$  set. Hence every correct process will eventually attempt to broadcast  $m$  in some subsequent round.

Since there are infinitely many rounds and finitely many processes, and by Lemma 4 every closed round has at least one winner, there must exist a round  $r'$  and a correct process  $p_j \in \text{Winners}_{r'}$  such that  $m \in (unordered \setminus ordered)$  when  $p_j$  constructs its proposal  $S$  at line 1.6 for round  $r'$ . Hence  $p_j$  sends messages  $S$  with  $m \in S$  at line 1.8.

In both cases, there exists a round and a winner whose proposal includes  $m$ . □

**Lemma 10** (Broadcast Termination). *A correct process which invokes  $\text{ABROADCAST}(m)$  eventually exits the function and returns.*

*Proof.* By Algorithm 1, the handler for  $\text{ABROADCAST}(m)$  at line 1.14 performs a single local operation: adding  $m$  to the local set  $unordered$ . This operation terminates immediately and the function returns. □

**Lemma 11** (Validity). *If a correct process  $p$  invokes  $\text{ABROADCAST}(m)$ , then every correct process that invokes a *ininitely many times*  $\text{ADELIVER}()$  eventually delivers  $m$ .*

*Proof.* Let  $p_i$  a correct process that invokes  $\text{ABROADCAST}(m)$  and  $p_q$  a correct process that infinitely invokes  $\text{ADELIVER}()$ . By Lemma 26, there exist a closed round  $r$  and a correct process  $j \in \text{Winners}_r$  such that  $p_j$  sends a proposal  $S$  to all processes with  $m \in S$ .

By Lemma 27, when  $p_q$  computes  $M$  at line 1.11,  $prop[r][j]$  is non- $\perp$  because  $j \in \text{Winners}_r$ . By Lemma 23,  $p_j$  sends messages at most once per round, so  $prop[r][j]$  is uniquely defined as the proposal sent by  $j$ . Hence, when  $p_q$  computes

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

we have  $m \in prop[r][j] = S$ , so  $m \in M$ . By Lemma 8,  $M$  is invariant so each computation of  $M$  by a correct process includes  $m$ . At each invocation of  $m' = \text{ADELIVER}()$ ,  $m'$  is added to  $delivered$  until  $M \subseteq delivered$ . Once this happens we're assured that there exists an invocation of  $\text{ADELIVER}()$  which return  $m$ . Hence  $m$  is well delivered.  $\square$

**Lemma 12** (No duplication). *No correct process delivers the same message more than once.*

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

When  $\text{ADELIVER}^{(A)}()$  occurs, by program order and because it reached line 1.21 to return  $m$ , the process must have add  $m$  to  $delivered$ . Hence when  $\text{ADELIVER}^{(B)}()$  reached line 1.20 to extract the next message to deliver, it can't be  $m$  because  $m \notin (ordered \setminus delivered)$ . So a  $\text{ADELIVER}^{(B)}()$  which delivers  $m$  can't occur.  $\square$

**Lemma 13** (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 a correct process that delivers both  $m_1$  and  $m_2$ . By Lemma 11, there exists 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  $p_{k_1}$  and  $p_{k_2}$  send proposals  $S_1$  and  $S_2$  respectively, with  $m_1 \in S_1$  and  $m_2 \in S_2$ .

Let consider two cases :

- **Case 1:**  $r_1 < r_2$ . By program order, any correct process must have waited to append in  $delivered$  every messages in  $M$  (which contains  $m_1$ ) to increment `current` and eventually set `current` =  $r_2$  to compute  $M$  and then invoke the  $m_2 = \text{ADELIVER}()$ . 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 8, any correct process that computes  $M$  at line 1.11 computes the same set of messages  $\text{Messages}_{r_1}$ . By line 1.12 the messages are pull in a deterministic order defined by  $ordered(\_)$ . 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  $ordered(\_)$ .

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.  $\square$

**Theorem 14** (ARB). *In a crash asynchronous message-passing system with reliable, error-free communication channels, assuming a synchronous DenyList (DL) object, the algorithm implements Atomic Reliable Broadcast.*

*Proof.* We show that the algorithm satisfies the properties of Atomic Reliable Broadcast under the assumed DL synchrony and reliable channel assumption.

First, by Lemma 10, if a correct process invokes  $\text{ABROADCAST}(m)$ , then it eventually returns from this invocation. Moreover, Lemma 11 states that if a correct process invokes  $\text{ABROADCAST}(m)$ ,

then every correct process that invokes `ADELIVER()` 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 processes that constructed and sent them in the algorithm. Every delivered message was previously sent by some process at line 1.8, so no spurious messages are delivered. In addition, Lemma 12 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 28 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.

All the above lemmas are proved under the assumptions that DL satisfies the required synchrony properties and that the communication channels are reliable and error-free (no message loss or corruption). Therefore, under these assumptions, the algorithm satisfies Validity, Integrity/No-duplicates, and total order, and hence implements Atomic Reliable Broadcast, as claimed.  $\square$

### 4.3 Reciprocity

So far, we assumed the existence of a synchronous DenyList (DL) object and showed how to build an Atomic Reliable Broadcast (ARB) primitive using reliable, error-free point-to-point channels. We now briefly argue that, conversely, an ARB primitive is strong enough to implement a synchronous DL object.

**DenyList as a deterministic state machine.** Without anonymity, the DL specification defines a deterministic abstract object: given a sequence `Seq` of operations `append(x)`, `prove(x)`, and `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 `ABROADCAST(m)`, then every correct process eventually `ADELIVER(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 15** (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 ABROADCAST. Then there exists an implementation of a DenyList object that satisfies 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 ABROADCAST 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 ABROADCAST requests can achieve the same preconditions as in the abstract DL specification (e.g.,  $p \in \Pi_M$ ,  $x \in S$  for `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 `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 `append(x)`, no later `prove(x)` can return valid at any correct process.

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

#### 4.3.1 Example executions

## 5 BFT-ARB over RB and DL

### 5.1 Model extension

We consider a static set  $\Pi$  of  $n$  processes with known identities, communicating by reliable point-to-point channels, in a complete graph. Messages are uniquely identifiable. At most  $f$  processes may be Byzantine, and we assume  $n > 3f$ .

**Synchrony.** The network is asynchronous.

**Communication.** Processes can exchange through a Reliable Broadcast (RB) primitive (defined below) which is invoked with the functions `rbcast(m)` and `m = rreceived()`. The Reliable Broadcast is a Byzantine fault-tolerant primitive following Bracha's specification [?]: for any message  $m$  broadcast by a process  $p_j$ , every correct process either receives the message  $m$  or none at all; moreover, all correct processes that receive a broadcast from  $p_j$  for a given sequence number or round receive exactly the same message. This ensures that Byzantine processes cannot send different messages to different correct processes for the same broadcast instance. There exists a shared object called DenyList (DL) (defined below) that is interfaced with a set  $O$  of operations. There exist three types of these operations: `append(x)`, `prove(x)` and `read()`.

**Byzantine behaviour** A process is said to exhibit Byzantine behaviour if it deviates arbitrarily from the prescribed algorithm. Such deviations may, for instance, consist in invoking primitives (`rbcast`, `append`, `prove`, etc.) with invalid inputs or inputs crafted maliciously, colluding with other Byzantine processes in an attempt to manipulate the system state or violate its guarantees, deliberately delaying or accelerating the delivery of messages to selected nodes so as to disrupt the expected timing of operations, or withholding messages and responses in order to induce inconsistencies in the system state.

Byzantine processes are constrained by the following. They cannot forge valid cryptographic signatures or threshold shares without access to the corresponding private keys. They cannot violate

the termination, validity, or anti-flickering properties of the DL object. They also cannot break the integrity, no-duplicates, or validity properties of the RB primitive.

**Notation.** For any indice  $k$  we defined by  $DL[k]$  as the  $k$ -th DenyList object. For a given  $DL[k]$  and any indice  $x$  we defined by  $\Pi_x^k$  a subset of  $\Pi$ . Still for a given  $k$  we consider  $\Pi_M^k \subseteq \Pi$  and  $\Pi_V^k \subseteq \Pi$  two authorization subsets for  $DL[k]$ . Indice  $i \in \Pi$  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  $\prec_k$  for the  $DL[k]$  linearization:  $x \prec_k y$  means that operation  $x$  appears strictly before  $y$  in the linearized history of  $DL[k]$ . 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 operation  $F \in O, F_i(\dots)$  denotes that the operation  $F$  is invoked by process  $p_i$ , and by  $F_i^k(\dots)$  the same operation invoked on the  $DL[k]$  object.

## 5.2 Primitives

### 5.2.1 t-BFT-DL

We consider a t-Byzantine Fault Tolerant DenyList (t-BFT-DL) with the following properties. There are 3 operations :  $BFT-PROVE(x), BFT-APPEND(x), BFT-READ()$  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  $op = BFT-PROVE(x)$  by a correct process is valid iff there exist a set of correct process  $C$  such that  $\forall c \in C, c$  invoke  $op_2 = BFT-APPEND(x)$  with  $op_2 \prec op_1$  and  $|C| \leq t$

**PROVE Anti-Flickering.** If the invocation of a operation  $op = BFT-PROVE(x)$  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  $op = BFT-READ()$  invoke by a correct process such that  $R$  is the result of  $op$ . For all  $(i, x) \in R$  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  $R \ni (i, x)$

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

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 2:** t-BFT-DL implementation using multiple DL objects

---

**2.1 Function** *BFT-APPEND*( $x$ )

**2.2**    **for each**  $U \in \mathcal{U}$  *st*  $i \in U$  **do**

**2.3**    |     $DL_U.append(x)$ ;

  

**2.4 Function** *BFT-PROVE*( $x$ )

**2.5**    |     $state \leftarrow false$ ;

**2.6**    |    **for each**  $U \in \mathcal{U}$  **do**

**2.7**    |    |     $state \leftarrow state$  **OR**  $DL_U.prove(x)$ ;

**2.8**    |    **return**  $state$ ;

  

**2.9 Function** *BFT-READ*()

**2.10**   |     $results \leftarrow \emptyset$ ;

**2.11**   |    **for each**  $U \in \mathcal{U}$  **do**

**2.12**   |    |     $results \leftarrow results \cup DL_U.read()$ ;

**2.13**   |    **return**  $results$ ;

---

**Lemma 16** (BFT-PROVE Validity). *The invocation of  $op = \text{BFT-PROVE}(x)$  by a correct process is invalid iff there exist at least  $t + 1$  distinct processes in  $M$  that invoked a valid  $\text{BFT-APPEND}(x)$  before  $op$  in Seq.*

*Proof.* 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 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 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  $prove(x)$  is valid, so the field  $state$  at line 2.7 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  $prove(x)$ , so  $op$  is valid.

Combining the cases yields the claimed characterization of invalidity. □

**Lemma 17** (BFT-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  $\text{BFT-PROVE}(x)$  operation that appears after  $op$  in Seq is invalid.*

*Proof.* 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  $DL_U.\text{append}(x)$ . Moreover, because  $\text{BFT-APPEND}_j(x) \prec op$  in  $\text{Seq}$ , the induced  $DL_U.\text{append}(x)$  appears before the induced  $DL_U.\text{prove}(x)$  of  $op$  in the projection  $\text{Seq}_U$ .

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

**Lemma 18** (BFT-READ Liveness). *Let  $op = \text{BFT-READ}()$  invoke by a correct process such that  $R$  is the result of  $op$ . For all  $(i, x) \in R$  there exist a valid invocation of  $\text{BFT-PROVE}(x)$  by  $p_i$ .*

*Proof.* Let  $R$  the result of a  $\text{READ}()$  operation submit by any correct process.  $(i, x) \in R$  imply that  $\exists U^* \in \mathcal{U}$  such that  $(i, x) \in R^{U^*}$  with  $R^{U^*}$  the result of  $DL_{U^*}.\text{read}()$ . By **READ Validity**  $(i, x) \in R^{U^*}$  imply 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  $\text{BFT-PROVE}_i(x)$ .  
 $(i, x) \in R \implies \exists \text{BFT-PROVE}_i(x)$   $\square$

**Lemma 19** (BFT-READ Anti-Flickering). *Let  $op_1, op_2$  two  $\text{BFT-READ}()$  operations that returns respectively  $R_1, R_2$ . Iff  $op_1 \prec op_2$  then  $R_2 \subseteq R_1$ . Otherwise  $R_1 \subseteq R_2$ .*

*Proof.* Let  $R_1, R_2$  respectively the output of two  $\text{BFT-READ}()$  operations  $op_1, op_2$  such that  $op_1 \prec op_2$ . By the implementation of  $\text{BFT-READ}$ ,  $R_k = \bigcup_{U \in \mathcal{U}} R_k^U$  where  $R_k^U$  is the result of  $DL_U.\text{read}()$  during  $op_k$ .

Because  $op_1 \prec op_2$  for any  $U \in \mathcal{U}$ , the  $DL_U.\text{read}()$  induced by  $op_1$  happen before the  $DL_U.\text{read}()$  induced by  $op_2$ . Hence we have for all  $U, R_2^U \subseteq R_1^U$ . Therefore

$$\bigcup_U R_2^U \subseteq \bigcup_U R_1^U \implies R_2 \subseteq R_1$$

$\square$

**Lemma 20** (BFT-READ Safety). *Let  $op_1, op_2$  respectively a valid  $\text{BFT-PROVE}(x)$  operation submitted by the process  $p_i$  and a  $\text{BFT-READ}()$  operation submitted by any correct process such that  $op_1 \prec op_2$ . Let  $R$  the result of  $op_2$  then  $R \ni (i, x)$*

*Proof.* Let  $op_1 = \text{BFT-PROVE}_i(x)$  be a valid operation by a correct process  $p_i$  and  $op_2 = \text{BFT-READ}()$  be any  $\text{BFT-READ}()$  operation such that  $op_1 \prec op_2$  in  $\text{Seq}$ . By BFT-PROVE Validity, there exist at most  $t$  distinct processes in  $M$  that invoked a valid  $\text{BFT-APPEND}(x)$  before  $op_1$  in  $\text{Seq}$ . Let  $A \subseteq M$  denote that set, with  $|A| \leq t$ .

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  $DL_{U^*}.\text{append}(x)$ . Hence no valid  $DL_{U^*}.\text{append}(x)$  appears before the induced

$DL_{U^*}.\text{prove}(x)$  of  $op_1$ . Since also  $i \in \Pi_V(DL_{U^*})$ , by **PROVE Validity** of DL the induced  $DL_{U^*}.\text{prove}_i(x)$  is valid.

Now, because  $op_1 \prec op_2$  in Seq, the induced  $DL_{U^*}.\text{prove}_i(x)$  appears before the induced  $DL_{U^*}.\text{read}()$  of  $op_2$  in  $\text{Seq}_{U^*}$ . By **READ Safety** of DL, the result  $R^{U^*}$  of the induced  $DL_{U^*}.\text{read}()$  contains  $(i, x)$ .

Finally, by the implementation of BFT-READ(), we have  $R = \bigcup_{U \in \mathcal{U}} R^U$ , so  $(i, x) \in R$ .  $\square$

**Theorem 21.** *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.* Follows directly from the previous lemmas.  $\square$

## 5.4 Algorithm

---

**Algorithm 3:**  $t$ -BFT ARB at process  $p_i$

---

### 3.1 Local Variables:

3.2  $unordered \leftarrow \emptyset, ordered \leftarrow \epsilon, delivered \leftarrow \epsilon;$   
 3.3  $prop[r][j] \leftarrow \perp, \forall j, r \in \Pi \times \mathbb{N};$   
 3.4  $done[r] \leftarrow \emptyset, \forall r \in \mathbb{N};$

### 3.5 for $r = 1, 2, \dots$ do

3.6 **wait until**  $unordered \setminus ordered \neq \emptyset;$   
 3.7  $S \leftarrow unordered \setminus ordered; \text{rcast}(\text{PROP}, S, \langle i, r \rangle);$   
 3.8 **wait until**  $|\text{VALIDATED}(r)| \geq n - t;$   
 3.9 **foreach**  $j \in \Pi$  **do** BFT-APPEND( $\langle j, r \rangle$ );  
 3.10 **foreach**  $j \in \Pi$  **do** send(DONE,  $r$ ) **to**  $p_j$ ;  
 3.11 **wait until**  $|done[r]| \geq n - t;$   
 3.12 **wait until**  $\forall j \in \text{winners}[r], prop[r][j] \neq \perp$  **with**  $\text{winners}[r] \leftarrow \text{VALIDATED}(r);$   
 3.13  $M \leftarrow \bigcup_{j \in \text{winners}[r]} prop[r][j];$   
 3.14  $ordered \leftarrow ordered \cdot \text{order}(M);$

### 3.15 Function VALIDATED( $r$ )

3.16 **return**  $\{j : |\{k : (k, r) \in \text{BFT-READ}()\}| \geq t + 1\};$

### 3.17 Upon ABROADCAST( $m$ )

3.18  $unordered \leftarrow unordered \cup \{m\};$

### 3.19 Upon rdeliver(PROP, $S, \langle j, r \rangle$ ) from process $p_j$

3.20  $unordered \leftarrow unordered \cup S; prop[r][j] \leftarrow S;$   
 3.21 BFT-PROVE( $\langle j, r \rangle$ );

### 3.22 Upon receive(DONE, $r$ ) from process $p_j$

3.23  $done[r] \leftarrow done[r] \cup \{j\};$

### 3.24 Upon ADELIVER()

3.25 **if**  $ordered \setminus delivered = \emptyset$  **then return**  $\perp$ ;  
 3.26 let  $m$  be the first message in  $(ordered \setminus delivered);$   
 3.27  $delivered \leftarrow delivered \cdot \{m\};$   
 3.28 **return**  $m$

---

## 5.5 Correctness Lemmas

**Definition 5** (Closed Round). A round  $r \in \mathcal{R}$  is said to be *closed* for a correct process  $p_i$  if at the moment  $p_i$  computes  $\text{winners}[r]$ , it satisfies  $|\{k : (k, r) \in \text{BFT-READ}()\}| \geq n - t$ .

**Lemma 22** (Round Monotonicity). *If a round  $r$  is closed, then every round  $r_1 < r$  is also closed.*

*Proof. Base:*  $r = 0$ . Suppose round 1 is closed, the set  $\{r' \in \mathcal{R} : r' < r\}$  is empty, so the claim holds.

**Inductive step:** Let  $r \geq 1$ . Assume that the property holds for all rounds  $r' < r$ : whenever round  $r'$  is closed, all rounds  $r_2 < r'$  are also closed. We show that if round  $r$  is closed, then all rounds  $r_1 < r$  are closed.

Suppose round  $r$  is closed. This means at least one correct process  $p_i$  has reached line 3.8 and satisfied the condition  $|\text{VALIDATED}(r)| \geq n - t$ . Consequently,  $p_i$  has invoked  $\text{BFT-APPEND}(\langle j, r \rangle)$  for each  $j \in \Pi$  at line 3.9. Since these operations are invoked by a correct process, and they depend on valid proofs in the byzantine fault tolerant deny list, at least one of these operations must have succeeded in the DL object.

Moreover, by the algorithm structure, a correct process can only reach the beginning of round  $r$  after completing round  $r - 1$ . In particular, process  $p_i$  reaches line 3.5 for round  $r$  only after having completed round  $r - 1$ , which requires the process to have observed at least  $|\text{done}[r - 1]| \geq n - t$  (line 3.11 for round  $r - 1$ ). This condition can only be satisfied if round  $r - 1$  is closed: at least  $n - t$  processes must have issued DONE messages, and since  $n > 3f$ , at least one of these is correct.

Since  $n - t > 2f + 1$ , among the  $n - t$  processes satisfying the condition for round  $r - 1$ , at least one is correct. A correct process that issued a DONE message for round  $r - 1$  must have previously completed its execution of lines 3.9 for round  $r - 1$ , which in turn required observing  $|\text{VALIDATED}(r - 1)| \geq n - t$  from line 3.8. Thus, round  $r - 1$  is closed.

Now, considering any round  $r_1 < r - 1$ : by the inductive hypothesis applied to round  $r - 1$ , since round  $r - 1$  is closed and  $r_1 < r - 1$ , we have that  $r_1$  is also closed.

By strong induction, if round  $r$  is closed, all rounds  $r_1 < r$  are closed.  $\square$

**Lemma 23** (Uniqueness of Winners' Proposals). *For any closed round  $r$  and any process  $p_j \in \text{winners}[r]$ , there exists a unique set  $S_j \subseteq \mathcal{M}$  such that every correct process  $p_i$  that has received a reliable delivery of a PROP message from  $p_j$  for round  $r$  receives exactly this set  $S_j$ .*

*Proof.* Let  $r$  be a closed round and  $p_j \in \text{winners}[r]$ . Since  $j \in \text{winners}[r]$ , it means that  $(j, r) \in \text{BFT-READ}()$ , i.e., at least  $t + 1$  distinct processes have invoked a valid  $\text{BFT-PROVE}(\langle j, r \rangle)$  before the round  $r$  closed.

By the algorithm, a correct process  $p_i$  invokes  $\text{BFT-PROVE}(\langle j, r \rangle)$  only upon receiving a reliable delivery of a PROP message from  $p_j$  for round  $r$  at line 3.9. Let  $S_j^{(i)}$  denote the set received by  $p_i$ .

Since at least  $t + 1$  distinct processes have performed a valid  $\text{BFT-PROVE}(\langle j, r \rangle)$ , and since  $t < n/3$ , at least one of these processes is correct. Thus, at least one correct process must have received a reliable broadcast from  $p_j$  for round  $r$ .

Now, by Bracha's Byzantine Reliable Broadcast specification [?], for any message broadcast instance by process  $p_j$  in round  $r$ , all correct processes that deliver this broadcast either receive the identical message or none at all. Formally, for all correct processes  $p_i, p'_i$  that received a delivery from  $p_j$  for round  $r$ , we have  $S_j^{(i)} = S_j^{(i')}$ .

Therefore, there exists a unique set  $S_j$  such that every correct process  $p_i$  that receives a reliable delivery from  $p_j$  for round  $r$  receives exactly  $S_j$ .  $\square$

**Lemma 24** (Winners Stability). *For any closed round  $r$ , the set  $\text{winners}[r]$  is stable.*

*Proof.* Let  $r$  be a closed round. By definition, a closed round  $r$  means that at least one correct process  $p_i$  has observed  $|\{k : (k, r) \in \text{BFT-READ}()\}| \geq n - t$  and computed  $\text{winners}[r]$  at line 3.11.

When round  $r$  becomes closed, this implies that at least  $n - t$  distinct processes have invoked  $\text{BFT-APPEND}(\langle j, r \rangle)$  for  $j \in \Pi$  (by the algorithm structure, since  $\text{done}[r]$  contains DONE messages from processes that completed line 3.9). Since  $n > 3f$ , we have  $n - t > 2f + 1 > t + 1$ .

Consider a fixed  $j \in \Pi$ . Since at least  $n - t > t + 1$  processes have invoked a valid  $\text{BFT-APPEND}(\langle j, r \rangle)$ , by Lemma 16, any subsequent invocation of  $\text{BFT-PROVE}(\langle j, r \rangle)$  will be invalid.

By Lemma 17, once a  $\text{BFT-PROVE}(\langle j, r \rangle)$  operation becomes invalid, all subsequent  $\text{BFT-PROVE}(\langle j, r \rangle)$  operations are also invalid.

This holds for all  $j \in \Pi$ . Therefore, after the round  $r$  is closed, the set of valid  $\text{BFT-PROVE}(\langle j, r \rangle)$  operations for each  $j$  cannot grow. By Lemma 19, any subsequent  $\text{BFT-READ}()$  invocation will not return any new  $(k, r)$  pairs beyond those already returned. Thus,  $\text{winners}[r] = \{j : (j, r) \in \text{BFT-READ}()\}$  becomes stable and cannot change.

Since this reasoning applies to all correct processes that compute  $\text{winners}[r]$  after round  $r$  is closed, all correct processes will compute the same stable set  $\text{winners}[r]$ .  $\square$

**Lemma 25** (Message Content Invariance). *For any closed round  $r$  and any correct process  $p_i$ , the set  $M$  computed at line 3.13 by  $p_i$  is identical to the set computed by any other correct process  $p_j$  for the same round  $r$ .*

*Proof.* Let  $r$  be a closed round and let  $p_i, p_{i'}$  be two correct processes. By the algorithm, both processes compute  $M$  as:

$$M^{(i)} = \bigcup_{j \in \text{winners}[r]} \text{prop}^{(i)}[r][j]$$

and

$$M^{(i')} = \bigcup_{j \in \text{winners}[r]} \text{prop}^{(i')}[r][j]$$

at line 3.13.

By Lemma 24, the set  $\text{winners}[r]$  is stable once round  $r$  is closed. Therefore, both  $p_i$  and  $p_{i'}$  compute the same set of winners  $\text{winners}[r]$ .

Now, consider any winner  $j \in \text{winners}[r]$ . By Lemma 23, there exists a unique set  $S_j \subseteq \mathcal{M}$  such that every correct process that receives a reliable delivery of a PROP message from  $p_j$  for round  $r$  receives exactly  $S_j$ .

By the algorithm, each correct process stores this received set in its local variable:  $\text{prop}^{(i)}[r][j] = S_j$  and  $\text{prop}^{(i')}[r][j] = S_j$ .

Since both processes compute the union over the same set of winners, and each winner's proposal is identical for all correct processes:

$$M^{(i)} = \bigcup_{j \in \text{winners}[r]} \text{prop}^{(i)}[r][j] = \bigcup_{j \in \text{winners}[r]} S_j = \bigcup_{j \in \text{winners}[r]} \text{prop}^{(i')}[r][j] = M^{(i')}$$

Therefore, all correct processes compute the same set  $M$  for any closed round  $r$ .  $\square$

**Lemma 26** (Inclusion). *If a correct process  $p_i$  invokes  $\text{ABROADCAST}(m)$ , then there exist a closed round  $r$  and a winner  $j \in \text{winners}[r]$  such that  $p_j$  invoked  $\text{rbcast}(j, \text{PROP}, S, r)$  with  $m \in S$ .*

*Proof.* Let  $p_i$  be a correct process that invokes  $\text{ABROADCAST}(m)$ . By the algorithm,  $p_i$  adds  $m$  to  $\text{unordered}$ . Consequently,  $\text{unordered} \setminus \text{ordered} \neq \emptyset$ , and the process enters the main loop and eventually invokes  $\text{rbcast}(i, \text{PROP}, S_i^{(1)}, r_1)$  for some round  $r_1$  where  $m \in S_i^{(1)}$ .

We distinguish two cases:

**Case 1:  $p_i$  becomes a winner in round  $r_1$ .** If  $p_i$  is elected as a winner for round  $r_1$  (i.e.,  $i \in \text{winners}[r_1]$ ), then the claim holds with  $r = r_1$  and  $j = i$ .

**Case 2:  $p_i$  does not become a winner in round  $r_1$ .** If  $p_i$  is not elected as a winner, by the properties of Reliable Broadcast (Bracha's specification), at least one correct process  $p_{i_1}$  will eventually receive the reliable delivery of the PROP message from  $p_i$  for round  $r_1$ . Process  $p_{i_1}$  adds all messages from  $S_i^{(1)}$  to its *unordered* set at line 3.9. In particular,  $m$  is now in  $p_{i_1}$ 's *unordered* set.

In round  $r_2 > r_1$  (and any subsequent round), process  $p_{i_1}$  computes  $S_{i_1}^{(2)} = \text{unordered} \setminus \text{ordered}$  and invokes  $\text{rbcast}(i_1, \text{PROP}, S_{i_1}^{(2)}, r_2)$  with  $m \in S_{i_1}^{(2)}$ .

This process repeats: either  $p_{i_1}$  becomes a winner and the claim holds, or another correct process receives  $p_{i_1}$ 's proposal and includes  $m$  in its own proposal.

Since  $n > 3f$ , we have  $n - t > t + 1 > f$ . By the pigeonhole principle, there eventually exists a round  $r$  where at least  $n - t$  distinct processes have proposed sets containing  $m$ . Since more than  $2f$  processes have proposed  $m$ , at least one of them must be correct and be elected as a winner. Therefore, there exist a round  $r$  and a winner  $j \in \text{winners}[r]$  such that  $m \in S_j$  was broadcast by  $p_j$ .  $\square$

**Lemma 27 (Eventual Closure).** *For any correct process  $p_i$ , if  $\text{unordered} \setminus \text{ordered} \neq \emptyset$  and if  $r$  is the highest closed round observed by  $p_i$ , then eventually round  $r + 1$  will be closed with  $|\text{winners}[r + 1]| \geq n - t$ .*

**Lemma 28 (Total Order).** *For any two correct processes  $p_i$  and  $p_j$ , the sequence *ordered* maintained locally by  $p_i$  and the sequence maintained by  $p_j$  contain the same messages in the same order, provided that both have reached the same set of closed rounds.*

**Theorem 29.** *The algorithm implements a BFT Atomic Reliable Broadcast.*

## References

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