

1 Model 1: Crash

We consider a static set Π 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 can exchange through a Reliable Broadcast (RB) primitive (defined below) which is invoked with the functions $\text{RBroadcast}(m)$ and $m = \text{RReceived}()$. 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: $\text{APPEND}(x)$, $\text{PROVE}(x)$ and $\text{READ}()$.

Notation. For any indice x we defined by Π_x a subset of Π . We consider two subsets Π_M and Π_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 $A \subseteq \mathcal{M}$, $\text{ordered}(A)$ returns a deterministic total order over A (e.g., lexicographic order on $(\text{senderId}, \text{messageId})$ or on message hashes). For any operation $F \in O$, $F_i(\dots)$ denotes that the operation F is invoked by process p_i .

2 Primitives

2.1 Reliable Broadcast (RB)

RB provides the following properties in the model.

- **Integrity:** Every message received was previously sent. $\forall p_i : m = \text{RReceived}_i() \Rightarrow \exists p_j : \text{RBroadcast}_j(m)$.
- **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 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 $\text{APPEND}(x)$ 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 $\text{PROVE}(x)$ by a process p_i . We said op to be invalid, if and only if:
 - $p \notin \Pi_V \subseteq \Pi$; **or**
 - A valid $\text{APPEND}(x)$ 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 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.

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

3 Atomic Reliable Broadcast (ARB)

Processes export $\text{ABroadcast}(m)$ and $m = \text{ADeliver}()$. ARB requires total order:

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

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

4 ARB over RB and DL

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.

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 Variables

Each process p_i maintains:

```

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

```

4.1.2 Handlers and Procedures

Algorithm 1: RB handler (at process p_i)

1.1 **Function** $RBreceived(S, r, j)$
 1.2 received \leftarrow received $\cup \{S\}$;
 1.3 prop[r][j] $\leftarrow S$;

Algorithm 2: ABroadcast(m) (at process p_i)

2.1 **Function** $ABbroadcast(m)$
 2.2 $P \leftarrow$ READ();
 2.3 $r_{max} \leftarrow \max(\{r' : \exists j, (j, r') \in P\})$;
 2.4 $S \leftarrow$ (received \setminus delivered) $\cup \{m\}$;
 2.5 $r \leftarrow r_{max} - 1$;
 2.6 **while** $((i, r) \notin P \wedge (\nexists j, r' : (j, r') \in P \wedge m \in \text{prop}[r'][j]))$ **do**
 2.7 $r \leftarrow r + 1$;
 2.8 RBroadcast(S, r, i); PROVE(r); APPEND(r);
 2.9 $P \leftarrow$ READ();

Algorithm 3: ADeliver() at process p_i

3.1 **Function** $ADeliver(\perp)$
 3.2 $r \leftarrow$ current;
 3.3 $P_r \leftarrow \{(j, r) : (j, r) \in \text{READ}()\}$;
 3.4 **if** $P_r = \emptyset$ **then**
 3.5 **return** \perp
 3.6 APPEND(r); $P \leftarrow$ READ();
 3.7 $W_r \leftarrow \{j : (j, \text{prove}(r)) \in P\}$;
 3.8 **if** $\exists j \in W_r, \text{prop}[r][j] = \perp$ **then**
 3.9 **return** \perp
 3.10 $M_r \leftarrow \bigcup_{j \in W_r} \text{prop}[r][j]$;
 3.11 $m \leftarrow \text{ordered}(M_r \setminus \text{delivered})[0]$;
 3.12 delivered \leftarrow delivered $\cup \{m\}$;
 3.13 **if** $M_r \setminus \text{delivered} = \emptyset$ **then**
 3.14 current \leftarrow current + 1
 3.15 **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 $\text{APPEND}^{(*)}(r)$ 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 $\text{APPEND}(r)$ in H . We denote this operation $\text{APPEND}^{(*)}(r)$ and t_0 its linearization point.

By the validity property of DL, a $\text{PROVE}(r)$ is valid iff $\text{PROVE}(r) \prec \text{APPEND}^{(*)}(r)$. Thus, after t_0 , no $\text{PROVE}(r)$ 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

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, hence the lemma is true.

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

Assume $r + 1$ is closed and by Definition 2 $\text{APPEND}^{(*)}(r + 1)$ be the earliest $\text{APPEND}(r + 1)$ in the DL linearization H . By Lemma 1, after an $\text{APPEND}(r)$ were in H , any later $\text{PROVE}(r)$ will be invalid also, a process's program order is preserved in H .

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

- **Case Algorithm 3 :** Some process executes the loop at 2.6 and invokes $\text{PROVE}(r + 1)$; $\text{APPEND}^{(*)}(r + 1)$ at line 2.8. Let p_i be this process. Immediately before line 2.8, line 2.7 sets $r \leftarrow r + 1$, so the previous loop iteration targeted r . We consider two sub-cases.
 - (i) p_i is not in its first loop iteration. In the previous iteration, p_i executed $\text{PROVE}_i(r)$ at 2.8, followed (in program order) by $\text{APPEND}_i(r)$. If round r wasn't closed when p_i execute $\text{PROVE}_i(r)$ at 2.8, then the condition at 2.6 would be true hence the tuple (i, r) should be visible in P which implies that p_i should leave the loop at round r , contradicting the assumption that p_i is now executing another iteration. Since the tuple is not visible, the $\text{PROVE}_i(r)$ was invalid which implies by definition that an $\text{APPEND}(r)$ already exists in H . Thus in this case r is closed.
 - (ii) p_i is in its first loop iteration. To compute the value r_{max} , p_i must have observed one or many $(_, r + 1)$ in P at 2.3, issued by some processes (possibly different from p_i). Let's call p_j the issuer of the first $\text{PROVE}_j(r + 1)$ in the linearization H . When p_j executed $P \leftarrow \text{READ}()$ at 2.2 and compute r_{max} at 2.3, he observed no tuple $(_, r + 1)$ in P because he's the issuer of the first one. So when p_j executed the loop (2.6), he run it for the round r , didn't seen any (j, r) in P at B6, and then executed the first $\text{PROVE}_j(r + 1)$ at 2.8 in a second iteration. If round r wasn't closed when p_j execute $\text{PROVE}_j(r)$ at 2.8, then the (j, r) will be in P and the condition at 2.6 should be true which implies that p_j could leave the loop at round r , contradicting the assumption that p_j is now executing $\text{PROVE}_j(r + 1)$. In this case r is closed.
- **Case Algorithm 4 :** Some process invokes $\text{APPEND}(r + 1)$ at 3.6. Let p_i be this process. Line 3.6 is guarded by the condition at 3.4, which ensures that p_i observed some $(_, r + 1)$ in P . Let p_j be the issuer of the first $\text{PROVE}_j(r + 1)$ in the linearization H . When p_j executed $\text{PROVE}_j(r + 1)$ at 2.8, he observed no tuple $(_, r + 1)$ in P at 3.4 because he's the issuer of the first one. By the same reasoning as in the Case Algorithm 3 (ii), p_j must have observed that the round r was closed.

In all cases, $r + 1$ closed implies r closed. By induction on r , if the lemma is true for a closed r then it is true for a closed $r + 1$. Therefore, the lemma is true for all closed rounds r . \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 $(_, \text{prove}(r))$ 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 $(_, \text{prove}(r))$ in their DLview. \square

Lemma 3 (Well-defined winners). *For any correct process p_i and round r , if p_i computes W_r at line 3.7, then :*

- Winners_r is defined;
- the computed W_r is exactly Winners_r .

Proof. Lets consider a correct process p_i that reach line 3.7 to compute W_r .

By program order, p_i must have executed $\text{APPEND}_i(r)$ at line 3.6 before, which implies by Definition 1 that round r is closed at that point. So by Definition 3, Winners_r is defined.

By Lemma 2, all correct processes eventually observe the same set of valid tuples $(_, r)$ in their DLview. Hence, when p_i executes the $\text{READ}()$ at line 3.6 after the $\text{APPEND}_i(r)$, it observes a set P that includes all valid tuples $(_, r)$ such that

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

\square

Lemma 4 (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 (2.8) :** There exists a process p_i who's the issuer of the earliest $\text{APPEND}^{(*)}(r)$ in the DL linearization H . By program order, p_i must have previously invoked $\text{PROVE}_i(r)$ before invoking $\text{APPEND}^{(*)}(r)$ at line 2.8. In this case, there is at least one $\text{PROVE}(r)$ valid in H issued by a correct process before $\text{APPEND}^{(*)}(r)$.
- **Case (3.6) :** There exists a process p_i that invoked $\text{APPEND}^{(*)}(r)$ at . Line 3.6 is guarded by the condition at 3.4, which ensures that p observed some $(_, r)$ 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 5 (No empty winners). *Let r be a closed round, then $\text{Winners}_r \neq \emptyset$.*

Proof. By Definition 1, some $\text{APPEND}(r)$ occurs in the DL linearization.

By Lemma 4, at least one process must have invoked a valid $\text{PROVE}(r)$ before $\text{APPEND}^*(r)$. Hence, there exists at least one p_j such that p_j invoked $\text{PROVE}_j(r) \prec \text{APPEND}^*(r)$, so $\text{Winners}_r \neq \emptyset$. \square

Lemma 6 (Winners must propose). *For any closed round r , $\forall i \in \text{Winners}_r$, process p_i must have invoked a $\text{RBroadcast}(S^{(i)}, r, i)$ and hence any correct will eventually 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 exist a valid $\text{PROVE}_i(r)$ such that $\text{PROVE}_i(r) \prec \text{APPEND}^*(r)$ in the DL linearization. By program order, if i invoked a valid $\text{PROVE}_i(r)$ at line 2.8 he must have invoked $\text{RBroadcast}(S^{(i)}, r, i)$ directly before.

Let take a correct process p_j , by *RB Validity*, every correct process eventually receives i 's RB message for round r , which sets $\text{prop}[r][i]$ to a non- \perp value at line 1.3. \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 7 (Eventual proposal closure). *If a correct process p_i define M_r at line 7.11, 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_r at line 7.11. By Lemma 3, p_i computation is the winner set Winners_r .

By Lemma 5, $\text{Winners}_r \neq \emptyset$. The instruction at line 7.11 where p_i computes M_r is guarded by the condition at line 3.8, which ensures that p_i has received all RB messages from every winner $j \in \text{Winners}_r$. Hence, $M_r = \bigcup_{j \in \text{Winners}_r} \text{prop}^{(i)}[r][j]$, we have $\text{prop}^{(i)}[r][j] \neq \perp$ for all $j \in \text{Winners}_r$. \square

Lemma 8 (Unique proposal per sender per round). *For any round r and any process p_i , p_i invokes at most one $\text{RBroadcast}(S, r, i)$.*

Proof. By program order, any process p_i invokes $\text{RBroadcast}(S, r, i)$ at line 2.8 must be in the loop 2.6. No matter the number of iterations of the loop, line 2.6 always uses the current value of r which is incremented by 1 at each turn. Hence, in any execution, any process p_i invokes $\text{RBroadcast}(S, r, j)$ at most once for any round r . \square

Lemma 9 (Proposal convergence). *For any round r , for any correct processes p_i that execute line 7.11, we have*

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

Proof. Let take a correct process p_i that compute M_r at line 7.11. That implies that p_i has defined W_r at line 3.7. It implies that, by Lemma 3, r is closed and $W_r = \text{Winners}_r$.

By Lemma 7, for every $j \in \text{Winners}_r$, $\text{prop}^{(i)}[r][j] \neq \perp$. By Lemma 8, each winner j invokes at most one $\text{RBroadcast}(S^{(j)}, r, j)$, 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 10 (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 invokes*

$$\text{RBroadcast}(S, r, j) \quad \text{for same } S \text{ with } m \in S.$$

Proof. Fix a correct process p_i that invokes $\text{ABroadcast}(m)$ and eventually exits the loop (2.6) at some round r . There are two possible cases.

- **Case 1:** p_i exits the loop because $(i, r) \in P$. In this case, by Definition 3, p_i is a winner in round r . By program order, p_i must have invoked $\text{RBroadcast}(S, r, i)$ at 2.8 before invoking $\text{PROVE}_i(r)$. By line 2.4, $m \in S$. Hence there exist a closed round r and p_i is a correct process such that $i \in \text{Winners}_r$. Hence, i invokes $\text{RBroadcast}(S, r, i)$ with $m \in S$.
- **Case 2:** p_i exits the loop because $\exists j, r' : (j, r') \in P \wedge m \in \text{prop}[r'][j]$. In this case, by Lemma 6 and Lemma 8 p_j must have invoked a unique $\text{RBroadcast}(S, r', j)$. Which set $\text{prop}^{(i)}[r'][j] = S$ with $m \in S$.

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

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

□

Lemma 11 (Broadcast Termination). *A correct process which invokes $\text{ABroadcast}(m)$, eventually exits the function and returns.*

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

Consider 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 $\text{RBroadcast}(S, _, i)$ at line 2.8 with $m \in S$ (line 2.4).

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 5 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 one process p_k that invokes infinitely many valid $\text{PROVE}(r')$ and by extension infinitely many $\text{ABroadcast}(_)$. 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 executes Algorithm 1 and sets $\text{received} \leftarrow \text{received} \cup \{S\}$ with $m \in S$ (line 1.2). For the first invocation of $\text{ABroadcast}(_)$ by p_k after the execution of $\text{RReceived}()$, p_k must include m in his proposal S at line 2.4 (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 6, eventually $\text{prop}^{(i)}[r_2][k] \neq \perp$. By Lemma 8, $\text{prop}^{(i)}[r_2][k]$ is uniquely defined as the set S proposed by p_k at B6, which by Lemma 10 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 satisfies the condition $(i, r_1) \in P$. 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 $\text{ABroadcast}(m)$. \square

Lemma 12 (Validity). *If a correct process p invokes $\text{ABroadcast}(m)$, then every correct process that invokes a infinitely 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 10, there exist a closed round r and a correct process $j \in \text{Winners}_r$ such that p_j invokes

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

By Lemma 7, when p_q computes M_r at line 7.11, $\text{prop}[r][j]$ is non- \perp because $j \in \text{Winners}_r$. By Lemma 8, p_j invokes at most one $\text{RBroadcast}(S, r, j)$, 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 9, M_r is invariant so each computation of M_r by a correct process includes m . At each invocation of $m' = \text{ADeliver}()$, m' is added to delivered until $M_r \subseteq \text{delivered}$. Once this happens we're assured that there exists an invocation of $\text{ADeliver}()$ which return m . Hence m is well delivered. \square

Lemma 13 (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 C18 to return m , the process must have add m to delivered . Hence when $\text{ADeliver}^{(B)}()$ reached line 3.11 to extract the next message to deliver, it can't be m because $m \notin (M_r \setminus \{\dots, m, \dots\})$. So a $\text{ADeliver}^{(B)}()$ which delivers m can't occur. \square

Lemma 14 (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 12, there exists a 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

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

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

Let consider three cases :

- **Case 1:** $r_1 < r_2$. By program order, any correct process must have waited to append in delivered every messages in M_{r_1} (which contains m_1) to increment current and eventually set $\text{current} = r_2$ to compute M_{r_2} 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 9, any correct process that computes M_{r_1} at line 7.11 computes the same set of messages Messages_{r_1} . By line 3.11 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 15 (Fifo Order). *For any two distinct messages m_1 and m_2 broadcast by the same correct process p_i , if p_i invokes $\text{ABroadcast}(m_1)$ before $\text{ABroadcast}(m_2)$, then any correct process p_j that delivers both m_1 and m_2 delivers m_1 before m_2 .*

Proof. Let take m_1 and m_2 broadcast by the same correct process p_i , with p_i invoking $\text{ABroadcast}(m_1)$ before $\text{ABroadcast}(m_2)$. By Lemma 12, there exist closed rounds r_1 and r_2 such that $r_1 \leq r_2$ and correct processes $k_1 \in \text{Winners}_{r_1}$ and $k_2 \in \text{Winners}_{r_2}$ such that

$$\text{RBroadcast}(S_1, r_1, k_1) \quad \text{with} \quad m_1 \in S_1,$$

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

By program order, p_i must have invoked $\text{RBroadcast}(S_1, r_1, i)$ before $\text{RBroadcast}(S_2, r_2, i)$. By Lemma 8, any process invokes at most one $\text{RBroadcast}(S, r, i)$ per round, hence $r_1 < r_2$. By Lemma 14, 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 16 (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 11, if a correct process invokes $\text{ABroadcast}(m)$, then it eventually returns from this invocation. Moreover, Lemma 12 states that if a correct process invokes $\text{ABroadcast}(m)$, then every correct process that invokes $\text{ADeliver}()$ 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 the underlying RB primitive. By the Integrity property of RB, every such message was previously RBroadcast by some process, so no spurious messages are delivered. In addition, Lemma 13 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 14 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 15 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 ABroadcast .

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

4.3 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.

DenyList as a deterministic state machine. Without anonymity, the DLspecification 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 cover by our FIFO-ARB specification.

Correctness.

Theorem 17 (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 DLObject 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 DLspecification (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 DLspecification applied in the common order.
- **PROVE Anti-Flickering.** In the sequential DLspecification, 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

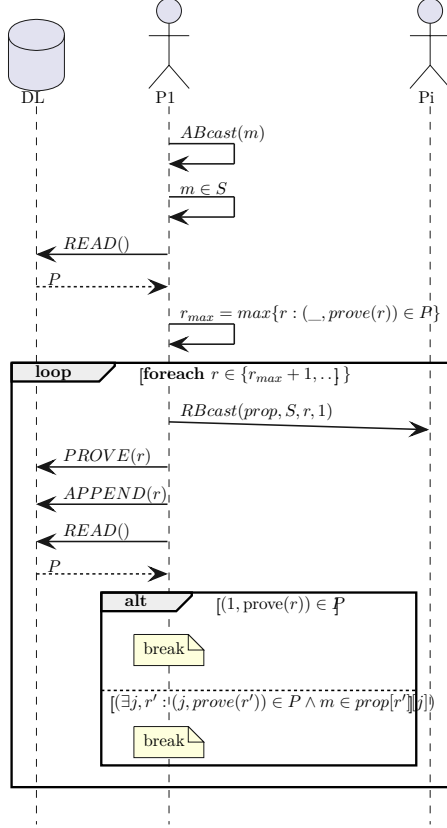


Figure 1: Example execution of the ARB algorithm in a non-BFT setting

4.3.1 Example executions

5 BFT-ARB over RB and DL

5.1 Model extension

We consider a static set Π 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 $\text{RBroadcast}(m)$ and $m = \text{RReceived}()$. 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: $\text{APPEND}(x)$, $\text{PROVE}(x)$ and $\text{READ}()$.

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 (RBroadcast, 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

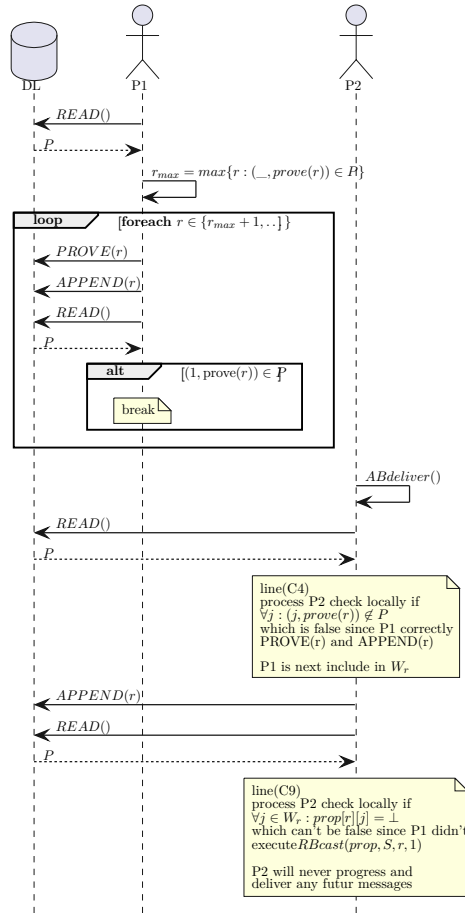


Figure 2: Example execution of the ARB algorithm with a Byzantine process

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 Π_x^k a subset of Π . 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 $A \subseteq \mathcal{M}$, $\text{ordered}(A)$ returns a deterministic total order over A (e.g., lexicographic order on $(\text{senderId}, \text{messageId})$ or on message hashes). For any 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 : $\text{BFT-PROVE}(x), \text{BFT-APPEND}(x), \text{BFT-READ}()$ such that :

Termination. Every operation $\text{BFT-APPEND}(x), \text{BFT-PROVE}(x),$ and $\text{BFT-READ}()$ 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 $\text{BFT-PROVE}(x)$ 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 $\text{BFT-PROVE}(x)$ by p_i .

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 $R_1 \subseteq R_2$.

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 $R \ni (i, x)$

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

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

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

Algorithm 4: t-BFT-DL implementation using multiple DL objects

```

4.1 Function BFT-APPEND( $x$ )
4.2   for each  $U \in \mathcal{U}$  st  $i \in U$  do
4.3      $DL_U$ .APPEND( $x$ );

4.4 Function BFT-PROVE( $x$ )
4.5    $state \leftarrow false$ ;
4.6   for each  $U \in \mathcal{U}$  do
4.7      $state \leftarrow state$  OR  $DL_U$ .PROVE( $x$ );
4.8   return  $state$ ;

4.9 Function BFT-READ()
4.10   $results \leftarrow \emptyset$ ;
4.11  for each  $U \in \mathcal{U}$  do
4.12     $results \leftarrow results \cup DL_U$ .READ();
4.13  return  $results$ ;

```

Lemma 18 (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 4.7 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. \square

Lemma 19 (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 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 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 Seq , the induced $DL_U.\text{APPEND}(x)$ appears before the induced $DL_U.\text{PROVE}(x)$ of op in the projection Seq_U .

Hence, in 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 Seq_U .

Let $op' = \text{BFT-PROVE}(x)$ be any invocation such that $op \prec op'$ in 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 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 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 20 (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. $(i, \text{prove}(x)) \in R$ implice 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** $(i, x) \in R^{U^*}$ 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 $\text{BFT-PROVE}_i(x)$. $(i, x) \in R \implies \exists \text{BFT-PROVE}_i(x)$ \square

Lemma 21 (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 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 22 (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 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 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 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 $\text{BFT-READ}()$, we have $R = \bigcup_{U \in \mathcal{U}} R^U$, so $(i, x) \in R$. \square

Theorem 23. *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

5.4.1 Variables

Each process p_i maintains the following local variables:

```

last_committed  $\leftarrow$  0;
last_delivered  $\leftarrow$  0;
received  $\leftarrow$   $\emptyset$ ;
delivered  $\leftarrow$   $\emptyset$ ;
prop[r][j]  $\leftarrow$   $\perp$ ,  $\forall r, j$ ;
W[r]  $\leftarrow$   $\perp$ ,  $\forall r$ ; this is the set of the winners for the round  $r$ 
B[r]  $\leftarrow$   $\perp$ ,  $\forall r$ ; this is the set of processes who have behaved maliciously for round  $r$ 
resolved[r]  $\leftarrow$   $\perp$ ,  $\forall r$ ;
Y a BFT-DL such that the value space is  $\mathcal{R} \times \Pi$ ;
V a BFT-DL such that the value space is  $(\mathcal{R} \times \mathcal{M} \times \Pi)$ ;

```

Algorithm 5: $\text{ABroadcast}(m)$ at process p_i

```

5.1 Function  $\text{ABroadcast}(m)$ 
5.2   | received  $\leftarrow$  received  $\cup \{m\}$ ;
5.3   | Propose();
5.4 EndFunction

```

Algorithm 6: Propose(\perp) at process p_i

```
6.1 Function Propose( $\perp$ )
6.2    $r \leftarrow \text{last\_committed};$ 
6.3   while  $S \neq \emptyset$  with  $S \leftarrow \text{received} \setminus (\text{delivered} \cup (\bigcup_{r' < r} \bigcup_{\substack{(j,S') \in W[r'] \\ j \notin B[r'] \\ S' \in \text{prop}[r'][j]}} S'))$  do
      |
      |    $V.\text{BFT-PROVE}((r, S, i));$  /* PROP PHASE */
      |    $\text{RBroadcast}(i, \text{PROP}, S, r);$ 
      |   wait until  $|\{j : \exists S, |\{k : (k, (r, S, j)) \in V.\text{BFT-READ}()\}| \geq t + 1\}| \geq n - t;$ 
      |   /* COMMIT PHASE */
      |   for each  $j \in \Pi$  do
      |     |  $V.\text{BFT-APPEND}((r, S, j));$ 
      |     |  $Y.\text{BFT-APPEND}((r, j));$ 
      |   end
      |    $\text{RBroadcast}(i, \text{COMMIT}, r);$ 
      |   wait until  $|\text{resolved}[r]| \geq n - t;$ 
      |    $W[r] \leftarrow \{(j, S) : |\{k : (k, (r, S, j)) \in V.\text{BFT-READ}()\}| \geq t + 1\};$ 
      |    $B[r] \leftarrow \{j : |\{k : (k, (r, j)) \in Y.\text{BFT-READ}()\}| \geq t + 1\};$ 
      |    $r \leftarrow r + 1;$ 
6.16  end
6.17   $\text{last\_committed} \leftarrow r;$ 
6.18 EndFunction
```

Algorithm 7: ADeliver() at process p_i

```
7.1 Function ADeliver( $\perp$ )
7.2    $r \leftarrow \text{last\_delivered};$ 
7.3   if  $|\text{resolved}[r]| < n - t$  then
7.4     | return  $\perp$ 
7.5   end
7.6    $W \leftarrow \{(j, S) : |\{k : (k, (r, S, j)) \in V.\text{BFT-READ}()\}| \geq t + 1\};$ 
7.7    $B \leftarrow \{j : |\{k : (k, (r, j)) \in Y.\text{BFT-READ}()\}| \geq t + 1\};$ 
7.8   if  $\exists (j, S) \in W, j \notin B, S \notin \text{prop}[r][j]$  then
7.9     | return  $\perp$ 
7.10  end
7.11   $M \leftarrow \bigcup_{\substack{(j,S) \in W \\ j \notin B \\ S \in \text{prop}[r][j]}} S;$ 
7.12   $m \leftarrow \text{ordered}(M \setminus \text{delivered})[0];$ 
7.13   $\text{delivered} \leftarrow \text{delivered} \cup \{m\};$ 
7.14  if  $M \setminus \text{delivered} = \emptyset$  then
7.15    |  $\text{last\_delivered} \leftarrow \text{last\_delivered} + 1;$ 
7.16  end
7.17  return  $m$ 
7.18 EndFunction
```

Algorithm 8: RB handler at process p_i

```
8.1 Upon  $Rdeliver(j, PROP, S, r)$ 
8.2   | received  $\leftarrow$  received  $\cup \{S\}$ ;
8.3   | if  $\text{prop}[r][j] = \perp \vee (\{S' : (j, (r, S', j)) \in V.BFT-READ()\} = \{S\})$  then
8.4     |    $V.BFT-PROVE((r, S, j))$ ;
8.5   | else
8.6     |    $Y.BFT-PROVE((r, j))$ ;
8.7   | end
8.8   |  $\text{prop}[r][j] \leftarrow \text{prop}[r][j] \cup S$ ;
8.9   |  $\text{Propose}()$ ;
8.10 EndUpon

8.11 Upon  $Rdeliver(j, COMMIT, r)$ 
8.12 |  $\text{resolved}[r] \leftarrow \text{resolved}[r] \cup \{j\}$ ;
8.13 EndUpon
```

Everything below has to be updated

Definition 5 (BFT Closed round for k). Given Seq^k the linearization of the BFT-DL $Y[k]$, a round $r \in \mathcal{R}$ is *closed* in Seq iff there exist at least $n - f$ distinct processes $j \in \Pi$ such that $\text{BFT-APPEND}_j(r)$ appears in Seq^k . Let call $\text{BFT-APPEND}(r)^*$ the $(n - f)^{\text{th}}$ $\text{BFT-APPEND}(r)$.

Definition 6 (BFT Closed round). A round $r \in \mathcal{R}$ is *closed* iff for all $\text{DL}[k]$, r is closed in Seq^k .

5.5 Proof of correctness

Remark 2 (BFT Stable round closure). If a round r is closed, no more $\text{BFT-PROVE}(r)$ can be valid and thus linearized. In other words, once $\text{BFT-APPEND}(r)^*$ is linearized, no more process can make a proof on round r , and the set of valid proofs for round r is fixed. Therefore Winners_r is fixed.

Proof. By definition r closed means that for all process p_i , there exist at least $n - f$ distinct processes $j \in \Pi$ such that $\text{BFT-APPEND}_j(r)$ appears in Seq^k . By BFT-PROVE Validity, any subsequent $\text{BFT-PROVE}(r)$ is invalid because at least $n - f$ processes already invoked a valid $\text{BFT-APPEND}(r)$ before it. Thus no new valid $\text{BFT-PROVE}(r)$ can be linearized after $\text{BFT-APPEND}(r)^*$. Hence the set of valid proofs for round r is fixed, and so is Winners_r . \square

Lemma 24 (BFT Across rounds). *For any r, r' such that $r < r'$, if r' is closed, r is also closed.*

Proof. Let $r \in \mathcal{R}$. By Definition 6, if $r + 1$ is closed, then for all $\text{DL}[k]$, $r + 1$ is closed in Seq^k . By the implementation, a process can only invoke $\text{BFT-APPEND}(r + 1)$ after observing at least $n - f$ valid $\text{BFT-PROVE}(r)$, which means that for all $\text{DL}[k]$, r is closed in Seq^k . Hence by Definition 6, r is closed.

Because r is monotonically increasing, we can recursively apply the same argument to conclude that for any r, r' such that $r < r'$, if r' is closed, r is also closed. \square

Lemma 25 (BFT Progress). *For any correct process p_i such that*

$$\text{received} \setminus (\text{delivered} \cup (\cup_{r' < r} \cup_{j \in W[r']} \text{prop}[r'][j])) \neq \emptyset$$

with r the highest closed round in the DL linearization. Eventually $r + 1$ will be closed.

Lemma 26 (BFT Winners invariant). *For any closed round r , define*

$$\text{Winners}_r = \{j : \text{BFT-PROVE}_j(r) \prec \text{BFT-APPEND}^*(r)\}$$

called the unique set of winners of round r .

Lemma 27 (BFT n-f lower-bounded Winners). *Let r a closed round, $|W[r]| \geq n - f$.*

Remark 3. Because we assume $n \geq 2f + 1$, if $|W[r]| \geq n - f$ at least 1 correct have to be in $W[r]$ to progress.

Lemma 28 (BFT Winners must purpose). *Let r a closed round, for all process p_j such that $j \in W[r]$, p_j must have executed $\text{RBroadcast}(j, \text{PROP}, _, r)$ and hence any correct will eventually set $\text{prop}[r][j]$ to a non- \perp value.*

Lemma 29 (BFT 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*

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

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

Lemma 30 (BFT E ventual proposal closure). *If a correct process p_i define M at line 7.11, then for every $j \in \text{Winners}_r$, $\text{prop}^{(i)}[r][j] \neq \perp$.*

Lemma 31 (BFT Unique proposal per sender per round). *For any round r and any process p_i , if p_i invokes two RBroadcast call for the same round, such that $\text{RBroadcast}(i, \text{PROP}, S, r) \prec \text{RBroadcast}(i, \text{PROP}, S', r)$. Then for any correct process p_j , $\text{prop}^{(j)}[r][i] \in \{\perp, S\}$*

Theorem 32. *The algorithm implements a BFT Atomic Reliable Broadcast.*

6 Implementation of BFT-DenyList and Threshold Cryptography

6.1 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 33. $\exists M_i \in \mathcal{M} : \forall p \in M_i$ 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

Lemma 34. $\forall M_i \in \mathcal{M}, \exists p \in M_i$ 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 :

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

Such that :

Algorithm 9: $\text{READ}'() \rightarrow \mathcal{L}(\mathbb{R} \times \text{prove}(\mathbb{R}))$

```

9.1  $j \leftarrow$  the process invoking  $\text{READ}'()$ ;
9.2  $res \leftarrow \emptyset$ ;
9.3 forall  $i \in \{1, \dots, k\}$  do
9.4    $res \leftarrow res \cup DL_i.\text{READ}()$ ;
9.5 return  $res$ ;
```

Algorithm 10: $\text{APPEND}'(\sigma) \rightarrow ()$

```

10.1  $j \leftarrow$  the process invoking  $\text{APPEND}'(\sigma)$ ;
10.2 forall  $M_i \in \{M_k \in M : j \in M_k\}$  do
10.3    $DL_i.\text{APPEND}(\sigma)$ ;
```

Algorithm 11: $\text{PROVE}'(\sigma) \rightarrow \{0, 1\}$

```

11.1  $j \leftarrow$  the process invoking  $\text{PROVE}'(\sigma)$ ;
11.2  $flag \leftarrow false$ ;
11.3 forall  $i \in \{1, \dots, k\}$  do
11.4    $flag \leftarrow flag \text{ OR } DL_i.\text{PROVE}(\sigma)$ ;
11.5 return  $flag$ ;
```

6.2 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 :

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

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

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

Such that :

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