Abstract

We show how to upgrade a Reliable Broadcast (RB) primitive to Atomic Reliable Broadcast (ARB) by leveraging a synchronous DenyList (DL) object. In a purely asynchronous message-passing model with crashes, ARB is impossible without additional power. The DL supplies this power by enabling round closing and agreement on a set of "+winners" for each round. We present the algorithm, its safety arguments, and discuss liveness and complexity under the assumed synchrony of DL.

Keywords Atomic broadcast, total order broadcast, reliable broadcast, consensus, synchrony, shared object, linearizability.

1 Introduction

Atomic Reliable Broadcast (ARB)—a.k.a. total order broadcast—ensures that all processes deliver the same sequence of messages. In asynchronous message-passing systems with crashes, implementing ARB is impossible without additional assumptions, as it enables consensus. We assume a synchronous DenyList (DL) object and demonstrate how to combine DL with an asynchronous RB to realize ARB.

2 Model

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

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

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

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

3 Primitives

3.1 Reliable Broadcast (RB)

RB provides the following properties in the model.

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

3.2 DenyList (DL)

The DL is a *shared, append-only* object that records attestations about opaque application-level tokens. It exposes the following operations:

- APPEND(x)
- PROVE(x): issue an attestation for token x; this operation is *valid* (return true) only if no APPEND(x) occurs earlier in the DL linearization. Otherwise, it is invalid (return false).
- READ(): return a (permutation of the) valid operations observed so far; subsequent reads are monotone (contain supersets of previously observed valid operations).

Validity. APPEND(x) is valid iff the issuer is authorized (in Π_M) and x belongs to the application-defined domain S. PROVE(x) is valid iff the issuer is authorized (in Π_V) and there is no earlier APPEND(x) in the DL linearization.

Progress. If a correct process invokes APPEND(x), then eventually all correct processes will be unable to issue a valid PROVE(x), and READ at all correct processes will (eventually) reflect that APPEND(x) has been recorded.

Termination. Every operation invoked by a correct process eventually returns.

Interface and Semantics. The DL provides a single global linearization of operations consistent with each process's program order. READ is prefix-monotone; concurrent updates become visible to all correct processes within bounded time (by synchrony). Duplicate requests may be idempotently coalesced by the implementation.

4 Target Abstraction: Atomic Reliable Broadcast (ARB)

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

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

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

5 Algorithm

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

5.1 Variables

Each process p_i maintains:

$\mathsf{received} \leftarrow \emptyset$ ▷ Messages received via RB but not yet delivered
$\mathsf{delivered} \leftarrow \emptyset$ ▷ Messages already delivered
$\mathsf{prop}[r][j] \leftarrow \bot,\ \forall r, j$ ▷ Proposal from process j for round r

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

5.2 Handlers and Procedures

Algorithm A RB handler (at process p_i)

```
A1 function RBRECEIVED(S, r, j)
A2     received ← received ∪ {S}
A3     prop[r][j] ← S                          ▷ Record sender j's proposal S for round r
A4 end function
```

Algorithm B AB-broadcast(m) (at process p_i)

```
B1  function ABBROADCAST(m)
B2      P ← READ()                  ▷ Fetch latest DL state to learn recent PROVE operations
B3      r_max ← max({r' : ∃j, (j, PROVE(r')) ∈ P})          ▷ Pick current open round
B4      S ← (received \ delivered) ∪ {m}          ▷ Propose all pending messages plus the new m

B5      for each r ∈ {r_max, r_max + 1, ···} do
B6          RB-cast(S, r, i); PROVE(r); APPEND(r);
B7          P ← READ()                                          ▷ Refresh local view of DL
B8          if (((i, prove(r)) ∈ P  ∨  (∃j, r' : (j, prove(r')) ∈ P ∧  m ∈ prop[r'][j]))) then
B9              break                  ▷ Exit loop once m is included in some closed round
B10         end if
B11     end for
B12 end function
```

Algorithm C AB-deliver() at process p_i

```
C1  next ← 0                                              ▷ Next round to deliver
C2  to_deliver ← ∅                                        ▷ Queue of messages ready to be delivered

C3  function ABdeliver
C4      if to_deliver = ∅ then        ▷ If no message is ready to deliver, try to fetch the next round
C5          P ← READ()                ▷ Fetch latest DL state to learn recent PROVE operations
C6          if ∀j : (j, prove(next)) ∉ P then          ▷ Check if some process proved round next
C7              return ⊥                                ▷ Round next is still open
C8          end if
C9          APPEND(next); P ← READ()                   ▷ Close round next if not already closed
C10         W_next ← {j : (j, prove(next)) ∈ P}        ▷ Compute winners of round next
C11         if ∃j ∈ W_next, prop[next][j] = ⊥ then     ▷ Check if we have all winners' proposals
C12             return ⊥          ▷ Some winner's proposal for round next is still missing
C13         end if
C14         M_next ← ⋃_{j∈W_next} prop[next][j]        ▷ Compute the agreed proposal for round next
C15         for each m ∈ ordered(M_next) do            ▷ Enqueue messages in deterministic order
C16             if m ∉ delivered then
C17                 to_deliver.push(m)                 ▷ Append m to the delivery queue
C18             end if
C19         end for
C20         next ← next + 1                            ▷ Advance to the next round
C21     end if
C22     m ← to_deliver.pop()
C23     delivered ← delivered ∪ {m}
C24     return m
C25 end function
```

6 Correctness

Lemma 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$^{(\star)}(r)$ and t_0 its linearization point.
By the validity property of DL, a PROVE(r) is valid iff PROVE$(r) \prec$ APPEND$^{(\star)}(r)$. 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. □

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

Lemma 2 (Across rounds)**.** *If there exists a r such that r is closed, $\forall r'$ such that $r' < r$, r' is also closed.*

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

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

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

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

- **Case (B6) :** Some process $p^\star$ executes the loop (lines B5–B11) and invokes $\mathsf{APPEND}^{(\star)}(k+1)$ at line B6. Immediately before line B6, line B5 sets $r \leftarrow r+1$, so the previous loop iteration (if any) targeted k. We consider two sub-cases.

  - *(i) $p^\star$ is not in its first loop iteration.* In the previous iteration, $p^\star$ executed $\mathsf{PROVE}^{(\star)}(k)$ at B6, followed (in program order) by $\mathsf{APPEND}^{(\star)}(k)$. If round k wasn't closed when $p^\star$ execute $\mathsf{PROVE}^{(\star)}(k)$ at B9, then the condition at B8 would be true hence the tuple $(p^\star, \mathsf{prove}(k))$ should be visible in P which implies that $p^\star$ should leave the loop at round k, contradicting the assumption that $p^\star$ is now executing another iteration. Since the tuple is not visible, the $\mathsf{PROVE}^{(\star)}(k)$ was rejected by the DL which implies by definition an $\mathsf{APPEND}(k)$ already exists in H. Thus in this case k is closed.
  - *(ii) $p^\star$ is in its first loop iteration.* To compute the value r_{max}, $p^\star$ must have observed one or many $(\_, \mathsf{prove}(k+1))$ in P at B2/B3, issued by some processes (possibly different from $p^\star$). Let's call p_1 the issuer of the first $\mathsf{PROVE}^{(1)}(k+1)$ in the linearization H.
    When p_1 executed $P \leftarrow \mathsf{READ}()$ at B2 and compute r_{max} at B3, he observed no tuple $(\_, \mathsf{prove}(k+1))$ in P because he's the issuer of the first one. So when p_1 executed the loop (B5–B11), he run it for the round k, didn't seen any $(1, \mathsf{prove}(k))$ in P at B8, and then executed the first $\mathsf{PROVE}^{(1)}(k+1)$ at B6 in a second iteration.
    If round k wasn't closed when p_1 execute $\mathsf{PROVE}^{(1)}(k)$ at B6, then the condition at B8 should be true which implies that p_1 sould leave the loop at round k, contradicting the assumption that p_1 is now executing $\mathsf{PROVE}^{(1)}(r+1)$. In this case k is closed.

- **Case (C9) :** Some process invokes $\mathsf{APPEND}(k+1)$ at C9. Line C9 is guarded by the presence of $\mathsf{PROVE}(\mathit{next})$ in P with $\mathit{next} = k+1$ (C6). Moreover, the local pointer next grow by increment of 1 and only advances after finishing the current round (C20), so if a process can reach $\mathit{next} = k+1$ it implies that he has completed round k, which includes closing k at C9 when $\mathsf{PROVE}(k)$ is observed. Hence $\mathsf{APPEND}^\star(k+1)$ implies a prior $\mathsf{APPEND}(k)$ in H, so k is closed.

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

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

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

as the unique set of winners of round r.

Lemma 3 (Invariant view of closure)**.** *For any closed round r, all correct processes eventually observe the same set of valid tuples $(\_, \mathsf{prove}(r))$ in their DL view.*

Proof. Let's take a closed round r. By Definition 2, there exists a unique earliest $\mathsf{APPEND}(r)$ in the DL linearization, denoted $\mathsf{APPEND}^\star(r)$.

Consider any correct process p that invokes $\mathsf{READ}()$ after $\mathsf{APPEND}^\star(r)$ in the DL linearization. Since $\mathsf{APPEND}^\star(r)$ invalidates all subsequent $\mathsf{PROVE}(r)$, the set of valid tuples $(\_, \mathsf{prove}(r))$ observed by any correct process after $\mathsf{APPEND}^\star(r)$ 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 $(\_, \mathsf{prove}(r))$ in their DL view. □

Lemma 4 (Well-defined winners)**.** *For any correct process and round r, if the process computes W_r at line C10, then :*

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

Proof. Let take a correct process p_i that reach line C10 to compute W_r.
By program order, p_i must have executed $\mathsf{APPEND}^{(i)}(r)$ at C9 before, which implies by Definition 1 that round r is closed. So by Definition 3, $\mathsf{Winners}_r$ is defined.
By Lemma 3, all correct processes eventually observe the same set of valid tuples $(\_, \mathsf{prove}(r))$ in their DL view. Hence, when p_i executes the $\mathsf{READ}()$ at C9 after the $\mathsf{APPEND}^{(i)}(r)$, it observes a set P that includes all valid tuples $(\_, \mathsf{prove}(r))$ such that

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

□

Lemma 5 (No APPEND without PROVE)**.** *If some process invokes $\mathsf{APPEND}(r)$, then at least a process must have previously invoked $\mathsf{PROVE}(r)$.*

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

- **Case (B6) :** There exists a process $p^\star$ who's the issuer of the earliest $\mathsf{APPEND}^{(\star)}(r)$ in the DL linearization H. By program order, $p^\star$ must have previously invoked $\mathsf{PROVE}^{(\star)}(r)$ before invoking $\mathsf{APPEND}^{(\star)}(r)$ at B6. In this case, there is at least one $\mathsf{PROVE}(r)$ valid in H issued by a correct process before $\mathsf{APPEND}^{(\star)}(r)$.
- **Case (C9) :** There exist a process $p^\star$ invokes $\mathsf{APPEND}^{(\star)}(r)$ at C9. Line C9 is guarded by the condition at C6, which ensures that p observed some $(\_, \mathsf{prove}(r))$ in P. In this case, there is at least one $\mathsf{PROVE}(r)$ valid in H issued by some process before $\mathsf{APPEND}^{(\star)}(r)$.

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

Lemma 6 (No empty winners)**.** *Let r be a round, if* $\mathsf{Winners}_r$ *is defined, then* $\mathsf{Winners}_r \neq \emptyset$*.*

Proof. If $\mathsf{Winners}_r$ is defined, then by Definition 3, round r is closed. By Definition 1, some $\mathsf{APPEND}(r)$ occurs in the DL linearization.
By Lemma 5, at least a process must have invoked a valid $\mathsf{PROVE}(r)$ before $\mathsf{APPEND}^{(\star)}(r)$. Hence, there exists at least one j such that $\{j : \mathsf{PROVE}^{(j)}(r) \prec \mathsf{APPEND}^\star(r)\}$, so $\mathsf{Winners}_r \neq \emptyset$. □

Lemma 7 (Winners must propose)**.** *For any closed round r, $\forall j \in \mathsf{Winners}_r$, process j must have invoked a $\mathsf{RB\text{-}cast}(S^{(j)}, r, j)$*

Proof. Fix a closed round r. By Definition 3, for any $j \in \mathsf{Winners}_r$, there exist a valid $\mathsf{PROVE}^{(j)}(r)$ such that $\mathsf{PROVE}^{(j)}(r) \prec \mathsf{APPEND}^\star(r)$ in the DL linearization. By program order, if j invoked a valid $\mathsf{PROVE}^{(j)}(r)$ at line B6 he must have invoked $\mathsf{RB\text{-}cast}(S^{(j)}, r, j)$ directly before. □

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

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

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

Lemma 8 (Non-empty winners proposal)**.** *For any closed round r, $\forall j \in \mathsf{Winners}_r$, for any correct process p_i, eventually $\mathsf{prop}^{(i)}[r][j] \neq \bot$.*

Proof. Fix a closed round r. By Definition 3, for any $j \in \mathsf{Winners}_r$, there exist a valid $\mathsf{PROVE}^{(j)}(r)$ such that $\mathsf{PROVE}^{(j)}(r) \prec \mathsf{APPEND}^\star(r)$ in the DL linearization. By Lemma 7, j must have invoked $\mathsf{RB\text{-}cast}(S^{(j)}, r, j)$.

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

Lemma 9 (Eventual proposal closure)**.** *If a correct process p_i define M_r at line C14, then for every $j \in \mathsf{Winners}_r$, $\mathsf{prop}^{(i)}[r][j] \neq \bot$.*

Proof. Let take a correct process p_i that computes M_r at line C14. By Lemma 4, p_i computes the unique winner set $\mathsf{Winners}_r$.

By Lemma 6, $\mathsf{Winners}_r \neq \emptyset$. The instruction at line C14 where p_i computes M_r is guarded by the condition at C11, which ensures that p_i has received all RB messages from every winner $j \in \mathsf{Winners}_r$. Hence, when p_i computes $M_r = \bigcup_{j \in \mathsf{Winners}_r} \mathsf{prop}^{(i)}[r][j]$, we have $\mathsf{prop}^{(i)}[r][j] \neq \bot$ for all $j \in \mathsf{Winners}_r$. □

Lemma 10 (Unique proposal per sender per round)**.** *For any round r and any process p_i, p_i invokes at most one $\mathsf{RB\text{-}cast}(S, r, i)$.*

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

Lemma 11 (Proposal convergence)**.** *For any round r, for any correct processes p_i that define M_r at line C14, we have*

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

Proof. Let take a correct process p_i that define M_r at line C14. That implies that p_i has defined W_r at line C10. It implies that, by Lemma 4, r is closed and $W_r = \mathsf{Winners}_r$.
By Lemma 9, for every $j \in \mathsf{Winners}_r$, $\mathsf{prop}^{(i)}[r][j] \neq \bot$. By Lemma 10, each winner j invokes at most one $\mathsf{RB\text{-}cast}(S^{(j)}, r, j)$, so $\mathsf{prop}^{(i)}[r][j] = S^{(j)}$ is uniquely defined. Hence, when p_i computes

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

□

Lemma 12 (Inclusion)**.** *If some correct process invokes AB-broadcast(m), then there exist a round r and a process $j \in \mathsf{Winners}_r$ such that p_j invokes*

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

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

- **Case 1:** p_i exits the loop because $(i, \mathsf{prove}(r)) \in P$. In this case, by Definition 3, p_i is a winner in round r. By program order, p_i must have invoked RB-cast(S, r, i) at B6 before invoking $\mathsf{PROVE}^{(i)}(r)$ at B7. By line B4, $m \in S$. Hence there exist a closed round r and a correct process $j = i \in \mathsf{Winners}_r$ such that j invokes RB-cast(S, r, j) with $m \in S$.
- **Case 2:** p_i exits the loop because $\exists j, r' : (j, \mathsf{prove}(r')) \in P \wedge m \in \mathsf{prop}[r'][j]$. In this case, by Lemma 7 and Lemma 10 j must have invoked a unique RB-cast(S, r', j). Which set $\mathsf{prop}^{(i)}[r'][j] = S$ with $m \in S$.

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

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

□

Lemma 13 (Broadcast Termination)**.** *If a correct process p invokes AB-broadcast(m), then p eventually quit the function and returns.*

Proof. Let a correct process p_i that invokes AB-broadcast(m). The lemma is true if $(i, \mathsf{prove}(r)) \in P$; or there exists $j : (j, \mathsf{prove}(r')) \in P \wedge m \in \mathsf{prop}[r'][j]$ (like guarded at B8).

- **Case 1:** there exists a round r' such that p_i invokes a valid $\mathsf{PROVE}(r')$. Hence by DL *Progress* and *Semantics* the following READ() at line B7 will defined a P such as $(i, \mathsf{prove}(r')) \in P$. Hence p_i exits the loop at B8.
- **Case 2:** there exists no round r' such that p_i invokes a valid $\mathsf{PROVE}(r')$. In this case p_i invokes infinitely many RB-cast(S, r', i) at B6 with $m \in S$ (line B4).
  The assumption that no $\mathsf{PROVE}(r')$ invoked by p is valid implies by DL *Validity* that for every round r', there exists at least one $\mathsf{APPEND}(r')$ in the DL linearization, hence by Lemma 6 for every possible round r' there at least a winner. Because there is an infinite number of rounds, and a finite number of processes, there exists at least a correct process p_k that invokes infinitely many valid $\mathsf{PROVE}(r')$ and by extension infinitely many AB-broadcast(_). By RB *Validity*, p_k eventually receives p_i 's RB messages. Let call t_0 the time when p_k receives p_i 's RB message.
  At t_0, p_k execute Algorithm A and do $\mathsf{received} \leftarrow \mathsf{received} \cup \{S\}$ with $m \in S$ (line A2). For the first invocation of AB-broadcast(_) by p_k after the execution of Algorithm A, p_k must include m in his proposal S at line B4 (because m is pending in j's $\mathsf{received} \setminus \mathsf{delivered}$ set). There exists a minimum round r_1 such that $p_k \in \mathsf{Winners}_{r_1}$ after t_0. By Lemma 8, eventually $\mathsf{prop}^{(i)}[r_1][k] \neq \bot$. By Lemma 10, $\mathsf{prop}^{(i)}[r_1][k]$ is uniquely defined as the set S proposed by p_k at B6, which by Lemma 12 includes m. Hence eventually $m \in \mathsf{prop}^{(i)}[r_1][k]$ and $k \in \mathsf{Winners}_{r_1}$.

The first case explicit the case where p_i is a winner and also covers the condition $(i, \mathsf{prove}(r')) \in P$. And in the second case, 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 AB-broadcast(m). □

Lemma 14 (Validity)**.** *If a correct process p invokes AB-broadcast(m), then every correct process that invokes a infinitely often times AB-deliver() eventually delivers m.*

Proof. Let p_i a correct process that invokes AB-broadcast(m) and p_q a correct process that infinitely invokes AB-deliver(). By Lemma 12, there exist a closed round r and a correct process $j \in \mathsf{Winners}_r$ such that p_j invokes

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

By Lemma 9, when p_q computes M_r at line C14, $\mathsf{prop}[r][j]$ is non-$\perp$ because $j \in \mathsf{Winners}_r$. By Lemma 10, p_j invokes at most one RB-cast(S, r, j), so $\mathsf{prop}[r][j]$ is uniquely defined. Hence, when p_q computes

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

we have $m \in \mathsf{prop}[r][j] = S$, so $m \in M_r$. By line C16–C18, m is enqueued into *to_deliver* unless it has already been delivered. Therefore, p_q will eventually invokes AB-deliver(), which will returns m. □

Lemma 15 (Across AB-deliver)**.** *A AB-deliver() invocation return m, a non-$\perp$ value, implie that there exist an invocation of AB-deliver() which is the earliest of the round r where m is proposed by a winner and which define M_r.*

Proof. Let take any AB-deliver() invocation which return a non-$\perp$ value, m which was proposed in round r. To return a non-$\perp$ value we can distinguish two cases:

- This actual invocation of AB-deliver have the queue *to_deliver* $\neq \emptyset$ which implies that there exist an earliest invocation of AB-deliver which reach the line C17 to fill this queue for round $next = r$. We can call this invocation AB-deliver$^{(\star r)}$. AB-deliver$^{(\star r)}$ have to fill *to_deliver* at line C17, hence by program order define M_r at C14.
- This actual invocation of AB-deliver have the queue *to_deliver* $= \emptyset$. To return a non-$\perp$ this execution have to pass the two conditions at lines C6 and C11 to compute m_r. Hence AB-deliver() is the AB-deliver$^{(\star r)}$() as described in the first case.

□

Definition 5 (First AB-deliver)**.** For any process which invoke an infinite times AB-deliver(). There exist for any round r an unique earliest invocation which defined M_r and return a non-$\perp$ value. We denote by AB-deliver$^{(\star r)}$() this invocation.

Lemma 16 (No duplication)**.** *No correct process delivers the same message more than once.*

Proof. Let consider two invokations of AB-deliver() made by the same correct process which returns respectively m_1 and m_2. Let call these two invokations respectively AB-deliver$^{(A)}$() and AB-deliver$^{(B)}$().

By Definition 5 we denote AB-deliver$^{(\star A)}$() and AB-deliver$^{(\star B)}$() the two earliest invocations of each AB-deliver() respectively in theirs rounds with A the round where m_1 is delivered and B the round where m_2 is delivered.

Let consider the following cases :

- $A < B$ **:** Let consider $m_1 = m_2 = m$. In the execution of $\mathsf{AB\text{-}deliver}^{(\star A)}$ the process iterate on line C17 and push m in *to_deliver* since m is not in $\mathsf{delivered}$. When the process invoke later $\mathsf{AB\text{-}deliver}^{(\star B)}$ he empties the queue. The only way to empties the queue is at line C22 which implie by program order to add m to the $\mathsf{delivered}$ set. Hence when $\mathsf{AB\text{-}deliver}^{(\star B)}$ iterate on line C17 he will never be able to push m in the queue because the condition $m \notin \mathsf{delivered}$ at line C16 is never satisfied. Hence in this case $\mathsf{AB\text{-}deliver}^{(B)}()$ can't happen if $m_1 = m_2$.
- $A = B$: Let consider $m_1 = m_2 = m$. $\mathsf{AB\text{-}deliver}^{(\star A)}$ and $\mathsf{AB\text{-}deliver}^{(\star B)}$ reference the same invocation of $\mathsf{AB\text{-}deliver}()$. In this unique execution, when the process define M_r he's making a union operation on all the sets which can contains a multiple times the same message m. This operation must result in a unique set which contain a unique time m. Hence when the iteration on C17 is done, m is pushed only once, and can be delivered only once too. So $\mathsf{AB\text{-}deliver}^{(B)}()$ can't happen if $m_1 = m_2$.

□

Lemma 17 (Total order)**.** *For any two messages m_1 and m_2 delivered by correct processes, if a correct process p_i delivers m_1 before m_2, then any correct process p_j that delivers both m_1 and m_2 delivers m_1 before m_2.*

Proof. Consider any correct process that delivers both m_1 and m_2. By Lemma 14, there exist closed rounds r'_1 and r'_2 and correct processes $k_1 \in \mathsf{Winners}_{r'_1}$ and $k_2 \in \mathsf{Winners}_{r'_2}$ such that

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

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

Let consider three cases :

- **Case 1:** $r_1 < r_2$. By program order, any correct process must have waited to empty the queue *to_deliver* (which contains m_1) to exit at round r_1 before invoking $\mathsf{AB\text{-}deliver}()$ that returns m_2. Hence, for any correct process that delivers both m_1 and m_2, it delivers m_1 before m_2.
- **Case 2:** $r_1 = r_2$. By Lemma 11, any correct process that computes M_{r_1} at line C14 computes the same set of messages $\mathsf{Messages}_{r_1}$. By line C16–C18, messages are enqueued into *to_deliver* in a deterministic order defined by $\mathsf{ordered}(\_)$. 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 $\mathsf{ordered}(\_)$.

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. □

Lemma 18 (Fifo Order)**.** *For any two messages m_1 and m_2 broadcast by the same correct process p_i, if p_i invokes $\mathsf{AB\text{-}broadcast}(m_1)$ before $\mathsf{AB\text{-}broadcast}(m_2)$, then any correct process p_j that delivers both m_1 and m_2 delivers m_1 before m_2.*

Proof. Let take two messages m_1 and m_2 broadcast by the same correct process p_i, with p_i invoking $\mathsf{AB\text{-}broadcast}(m_1)$ before $\mathsf{AB\text{-}broadcast}(m_2)$. By Lemma 14, there exist closed rounds r_1 and r_2 and correct processes $k_1 \in \mathsf{Winners}_{r_1}$ and $k_2 \in \mathsf{Winners}_{r_2}$ such that

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

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

By program order, p_i must have invoked $\mathsf{RB\text{-}cast}(S_1, r_1, i)$ before $\mathsf{RB\text{-}cast}(S_2, r_2, i)$. By Lemma 10, any process invokes at most one $\mathsf{RB\text{-}cast}(S, r, i)$ per round, hence $r_1 < r_2$. By Lemma 17, 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. □

Theorem 19 (FIFO-ARB)**.** *Under the assumed DL synchrony and RB reliability, the algorithm implements FIFO Atomic Reliable Broadcast.*

References