1 Introduction

1.1 Model

1.1.1 Model Properties

The system consists of n asynchronous processes communicating via reliable point-to-point message passing.
Each process has a unique, unforgeable identifier and knows the identifiers of all other processes.
Up to $f < n$ processes may crash (fail-stop).
The network is reliable : if a correct process sends a message to another correct process, it is eventually delivered.
Messages are uniquely identifiable : two messages sent by distinct processes or at different rounds are distinguishable
2 messages sent by the same processus in two differents rounds are differents

Property 1 (Message Uniqueness) *If two messages are sent by different processes, or by the same process in different rounds, then the messages are distinct.*
Formally :

$$\forall p_1, p_2,\ \forall r_1, r_2,\ \forall m_1, m_2,\ \left(\begin{array}{l} send(p_1, r_1, m_1) \land send(p_2, r_2, m_2) \\ \land\ (p_1 \neq p_2 \lor r_1 \neq r_2) \end{array} \right) \Rightarrow m_1 \neq m_2$$

1.1.2 Reliable Broadcast Properties

Property 2 *Integrity Every message received was previously sent.*
Formally :
$\forall p_i : bc\text{-}recv_i(m) \Rightarrow \exists p_j : bc\text{-}send_j(m)$

Property 3 *No Duplicates No message is received more than once at any single processor.*
Formally :
$\forall m, \forall p_i : bc\text{-}recv_i(m)$ *occurs at most once*

Property 4 *Validity All messages broadcast by a correct process are eventually received by all non faulty processors.*
Formally :
$\forall m, \forall p_i : correct(p_i) \land bc\text{-}send_i(m) => \forall p_j : correct(p_j) \Rightarrow bc\text{-}recv_j(m)$

1.1.3 AtomicBroadcast Properties

Property 5 *AB Totally ordered* $\forall m_1, m_2, \forall p_i, p_j\ :\ ab\text{-}recv_{p_i}(m_1) < ab\text{-}recv_{p_i}(m_2) \Rightarrow ab\text{-}recv_{p_j}(m_1) < ab\text{-}recv_{p_j}(m_2)$

1.1.4 DenyList Properties

Let Π_M be the set of processes authorized to issue `APPEND` operations, and Π_V the set of processes authorized to issue `PROVE` operations.
Let S be the set of valid values that may be appended. Let `Seq` be the linearization of operations recorded in the DenyList.

Property 6 *APPEND Validity An operation* ***`APPEND`***(x) *is valid iff : the issuing process* $p \in \Pi_M$*, and the value* $x \in S$

Property 7 *PROVE Validity An operation* $\texttt{PROVE}(x)$ *is valid iff : the issuing process* $p \in \Pi_V$*, and there exists no* $\texttt{APPEND}(x)$ *that appears earlier in* $\texttt{Seq}$.

Property 8 *PROGRESS If an APPEND(x) is invoked by a correct process, then all correct processes will eventually be unable to PROVE(x).*

Property 9 *READ Validity READ() return a list of tuples who is a random permutation of all valids PROVE() associated to the identity of the emiter process.*

1.2 Algorithms

We consider a set of processes communicating asynchronously over reliable point-to-point channels. Each process maintains the following local or shared variables :

- ***received*** : the set of messages that have been received via the reliable broadcast primitive but not yet ordered.
- ***delivered*** : the set of messages that have been ordered.
- ***prop***$[r][j]$: the proposal set announced by process j at round r. It contains a set of messages that process j claims to have received but not yet delivered.
- ***winner***^{r} : the set of processes that have issued a valid `PROVE` for round r, as observed through the registry.
- ***window*** : the list of the ids from the $f + 1$ last rounds. *window.pop()* remove the first value of the array. *window.push(x)* append x as the last value of the array.
- $\texttt{RB-cast}(\texttt{PROP}, S, r, j)$: a reliable broadcast invocation that disseminates the proposal S from process j for round r.
- $\texttt{RB-delivered}(\texttt{PROP}, S, r, j)$: the handler invoked upon reception of a `RB-cast`, which stores the received proposal S into $prop[r][j]$.
- `READ()` : returns the current view of all valid operations stored in the DenyList registry.
- $\texttt{ordered}(S)$: returns a deterministic total order over a set S of messages.
- $\texttt{hash}(T, r)$: returns the identifier of the next round as a deterministic function of the delivered set T and current round r.

1.3 Round mecansism

We assume that the hash function is deterministic and without collisions. Because we're sure that the round contains at least f + 1 processes as winners, the next round ID is unpredictable by a process who would not follow the protocol and would drop messages legally sent by non-byzantine process. Also, it ensures that if a byzantine process try to go faster than the others, he will at least wait the faster non-byzantine process to progress.

1.4 proof

Theorem 1 (Integrity) *If a message* m *is delivered by any process, then it was previously broadcast by some process via the* `AB-broadcast` *primitive.*

Proof 1

Theorem 2 (No Duplication) *No message is delivered more than once by any process.*

Proof 2

Theorem 3 (Validity) *If a correct process invokes* $\texttt{AB-Broadcast}_j(m)$*, then all correct processes eventually deliver* m.

Proof 3

Algorithm 1 Atomic Broadcast with DenyList

```
proves ← ∅
received ← ∅
delivered ← ∅
window ← [⊥]^{f+1}
r_1 ← 0

AB-Broadcast_j(m)
   RB-Broadcast_j(m)

RB-delivered_j(m)
   received ← received ∪ {m}
   repeat while received \ delivered ≠ ∅
      S ← received \ delivered
      RB-broadcast(PROP, S, r_1, j)
      proves ← READ()
      PROVE[j](r_1)
      APPEND[j](r_1)
      S ← {1, ..., n}
      repeat while |S| ≤ n − f
         forall i ∈ S
            if ¬PROVE[i](r_1)
               S ← S \ i
      winner[r_1] ← READ_ALL()
      wait  ∀j ∈ winner[r_1], |prop[r_1][j] ≠ ⊥| ≥ f + 1
      T ← ⋃_{j∈winner[r_1]} prop[r_1][j] \ delivered
      for each  m ∈ ordered(T)
         delivered ← delivered ∪ {m}
         AB-deliver_j(m)
      r_1 ← hash(T, r_1)

READ_ALL(r)
   for each  j ∈ (1, ..., n)
      win[j] ← {j_1 : READ_{j_1}() ∋ (j, PROVE(r))}
   for i ∈ (1, ..., n)
      for j ∈ (1, ..., n)
         if i ∈ win[j]
            count[i] + +
   return {i : count[i] ≥ n − f}
```

Theorem 4 (Total Order) *If two correct processes deliver two messages* m_1 *and* m_2*, then they deliver them in the same order.*

Proof 4