1 Rocq-to-Text Mapping: SHIFT Impossibility Proofs

This document provides exact correspondence between Rocq definitions/theorems and their textual descriptions for paper integration.

---

2 Core Definitions (`Core/`)

2.1 Memory Model (`Core/Memory.v`)

`#rocq_name` Memory.v:11

Memory addresses, modeled as natural numbers.

```
Definition Addr := nat.
```

> An *address* $a \in \mathbb{N}$ identifies a location in RDMA-accessible memory.

`#rocq_name` Memory.v:14

Values stored in memory, modeled as natural numbers.

```
Definition Val := nat.
```

> A *value* $v \in \mathbb{N}$ represents data stored at a memory address.

`#rocq_name` Memory.v:22

Memory as a total function from addresses to values.

```
Definition Memory := Addr -> Val.
```

> *Memory* is a function $m : \text{Addr} \to \text{Val}$ mapping each address to its current value.

`#rocq_name` Memory.v:25

Initial memory state where all addresses contain 0.

```
Definition init_memory : Memory := fun _ => 0.
```

> The *initial memory* m_0 satisfies $m_0(a) = 0$ for all addresses a.

`#rocq_name` Memory.v:28

Read operation: returns value at address.

```
Definition mem_read (m : Memory) (a : Addr) : Val := m a.
```

> A *read* of address a in memory m returns $m(a)$.

`#rocq_name` Memory.v:31-32

Write operation: creates new memory with updated value.

```
Definition mem_write (m : Memory) (a : Addr) (v : Val) : Memory :=
  fun a' => if Nat.eqb a' a then v else m a'.
```

> A *write* of value v to address a produces memory m' where $m'(a) = v$ and $m'(a') = m(a')$ for $a' \neq a$.

2.2 RDMA Operations (`Core/Operations.v`)

#rocq_name Operations.v:12-16

RDMA operation types.

```
Inductive Op : Type :=
  | OpWrite : Addr -> Val -> Op       (* RDMA Write *)
  | OpRead : Addr -> Op               (* RDMA Read *)
  | OpFADD : Addr -> nat -> Op        (* Fetch-and-Add *)
  | OpCAS : Addr -> Val -> Val -> Op. (* Compare-and-Swap *)
```

> An *RDMA operation* is one of:
> - $\text{Write}(a, v)$: write value v to address a
> - $\text{Read}(a)$: read value at address a
> - $\text{FADD}(a, \delta)$: atomically add δ to address a, return old value
> - $\text{CAS}(a, \text{exp}, \text{new})$: if $m(a) = \text{exp}$, write new; return (success, old value)

#rocq_name Operations.v:20-24

Results returned by operations.

```
Inductive OpResult : Type :=
  | ResWriteAck : OpResult
  | ResReadVal : Val -> OpResult
  | ResFADDVal : Val -> OpResult
  | ResCASResult : bool -> Val -> OpResult.
```

> An *operation result* is the value returned to the sender:
> - Write returns acknowledgment
> - Read returns the value read
> - FADD returns the old value before addition
> - CAS returns (success flag, old value)

#rocq_name Operations.v:37-40

FADD execution semantics.

```
Definition exec_fadd (m : Memory) (a : Addr) (delta : nat) : Memory * OpResult :=
  let old_val := mem_read m a in
  let new_val := old_val + delta in
  (mem_write m a new_val, ResFADDVal old_val).
```

> $\text{exec}_{\text{FADD}(m,a,\delta)}$ atomically:
> 1. Reads $v_{\text{old}} = m(a)$
> 2. Writes $m(a) := v_{\text{old}} + \delta$
> 3. Returns v_{old}

#rocq_name Operations.v:43-48

CAS execution semantics.

```
Definition exec_cas (m : Memory) (a : Addr) (expected new_val : Val) : Memory * OpResult :=
  let old_val := mem_read m a in
  if Nat.eqb old_val expected then
    (mem_write m a new_val, ResCASResult true old_val)
  else
    (m, ResCASResult false old_val).
```

> $\text{exec}_{\text{CAS}(m,a,\text{ exp},\text{ new})}$ atomically:
> 1. Reads $v_{\text{old}} = m(a)$
> 2. If $v_{\text{old}} = \text{exp}$: writes $m(a) := \text{new}$, returns $(\mathsf{true}, v_{\text{old}})$
> 3. Else: memory unchanged, returns $(\mathsf{false}, v_{\text{old}})$

#rocq_name Operations.v:62-66

Write operations are idempotent.

```
Lemma write_idempotent : forall m a v,
  fst (exec_write (fst (exec_write m a v)) a v) = fst (exec_write m a v).
```

> **Lemma (Write Idempotency):** For any memory m, address a, and value v: executing Write(a, v) twice produces the same memory state as executing it once.

#rocq_name Operations.v:69-92

FADD is not idempotent when delta > 0.

```
Lemma fadd_not_idempotent : forall m a delta,
  delta <> 0 ->
  fst (exec_fadd (fst (exec_fadd m a delta)) a delta) <> fst (exec_fadd m a delta).
```

> **Lemma (FADD Non-Idempotency):** For $\delta > 0$: executing FADD(a, δ) twice yields $m(a) = v_{\text{old}} + 2\delta$, not $v_{\text{old}} + \delta$. Therefore FADD is not idempotent.

2.3 Execution Traces (`Core/Traces.v`)

#rocq_name Traces.v:12-28

Events in a distributed execution.

```
Inductive Event : Type :=
  | EvSend : Op -> Event              (* Sender posts operation *)
  | EvTimeout : Op -> Event           (* Sender observes timeout *)
  | EvCompletion : Op -> OpResult -> Event
  | EvPacketLost : Op -> Event        (* Packet lost in network *)
  | EvAckLost : Op -> Event           (* ACK lost in network *)
  | EvReceive : Op -> Event           (* Receiver gets packet *)
  | EvExecute : Op -> OpResult -> Event
  | EvAppConsume : Addr -> Val -> Event
  | EvAppReuse : Addr -> Val -> Event.
```

> An *event* records an occurrence in the distributed system:
> - **Sender-side**: Send, Completion, Timeout
> - **Network**: PacketLost, AckLost
> - **Receiver-side**: Receive, Execute
> - **Application**: Consume (read data), Reuse (overwrite buffer)

#rocq_name Traces.v:31

A trace is a sequence of events.

```
Definition Trace := list Event.
```

> A *trace* $\mathcal{T}$ is a sequence of events representing one possible execution.

#rocq_name Traces.v:36-39

What the sender can observe.

```
Inductive SenderObs : Type :=
  | ObsSent : Op -> SenderObs
  | ObsCompleted : Op -> OpResult -> SenderObs
  | ObsTimeout : Op -> SenderObs.
```

A *sender observation* is one of: operation sent, completion received, or timeout. The sender **cannot** observe network events (PacketLost, AckLost) or receiver events (Execute).

#rocq_name Traces.v:42-49

Projection to sender-observable events.

```
Fixpoint sender_view (t : Trace) : list SenderObs :=
  match t with
  | [] => []
  | EvSend op :: rest => ObsSent op :: sender_view rest
  | EvCompletion op res :: rest => ObsCompleted op res :: sender_view rest
  | EvTimeout op :: rest => ObsTimeout op :: sender_view rest
  | _ :: rest => sender_view rest  (* Cannot observe! *)
  end.
```

The *sender view* $\sigma(\mathcal{T})$ projects a trace to only sender-observable events. This is the **central abstraction**: the sender's decision can only depend on $\sigma(\mathcal{T})$.

#rocq_name Traces.v:54-55

Two traces with identical sender views.

```
Definition sender_indistinguishable (t1 t2 : Trace) : Prop :=
  sender_view t1 = sender_view t2.
```

Traces $\mathcal{T}_1$ and $\mathcal{T}_2$ are *sender-indistinguishable* if $\sigma(\mathcal{T}_1) = \sigma(\mathcal{T}_2)$.

#rocq_name Traces.v:66-67

Operation was executed at receiver.

```
Definition op_executed (t : Trace) (op : Op) : Prop :=
  exists res, In (EvExecute op res) t.
```

Operation op was *executed* in trace $\mathcal{T}$ if $\text{EvExecute}(\text{op}, r) \in \mathcal{T}$ for some result r.

#rocq_name Traces.v:74-75

Sender observed timeout for operation.

```
Definition sender_saw_timeout (t : Trace) (op : Op) : Prop :=
  In (EvTimeout op) t.
```

The sender *saw timeout* for operation op if $\text{EvTimeout}(\text{op}) \in \mathcal{T}$.

2.4 Overlay Properties (`Core/Properties.v`)

#rocq_name Properties.v:110-117

A transparent failover mechanism.

```
Record TransparentOverlay := {
  decide_retransmit : list SenderObs -> Op -> bool;
  decision_deterministic : forall obs1 obs2 op,
    obs1 = obs2 ->
    decide_retransmit obs1 op = decide_retransmit obs2 op;
}.
```

A *transparent overlay* is a retransmission decision function $D : \sigma(\mathcal{T}) \times \mathrm{Op} \to \{\mathrm{retransmit}, \mathrm{skip}\}$ that depends **only** on the sender view. No persistent metadata, no receiver-side modifications.

#rocq_name Properties.v:70-76

Count executions of an operation.

```
Fixpoint execution_count (t : Trace) (op : Op) : nat :=
  match t with
  | [] => 0
  | EvExecute op' _ :: rest =>
      (if op_eq op op' then 1 else 0) + execution_count rest op
  | _ :: rest => execution_count rest op
  end.
```

$\mathrm{count}(\mathcal{T}, \mathrm{op})$ counts how many times operation op was executed in trace $\mathcal{T}$.

#rocq_name Properties.v:79-80

At-most-once execution semantics.

```
Definition AtMostOnce (t : Trace) : Prop :=
  forall op, execution_count t op <= 1.
```

A trace satisfies *at-most-once* if every operation executes at most once.

3 Theorem 1: Indistinguishability (`Theorem1/`)

3.1 Trace Construction (`Theorem1/Indistinguishability.v`)

#rocq_name Indistinguishability.v:32-36

Trace where packet is lost.

```
Definition T1_packet_loss (V1 : Val) : Trace :=
  [ EvSend (W_D V1);
    EvPacketLost (W_D V1);
    EvTimeout (W_D V1) ].
```

> $\mathcal{T}_1$ (Packet Loss): Sender posts write, packet lost in network, sender times out. Operation **not executed**. Correct action: **retransmit**.

#rocq_name Indistinguishability.v:44-55

Trace where ACK is lost but operation executed.

```
Definition T2_ack_loss (V1 V_new : Val) : Trace :=
  [ EvSend (W_D V1);
    EvReceive (W_D V1);
    EvExecute (W_D V1) ResWriteAck;
    EvSend W_F; EvReceive W_F; EvExecute W_F ResWriteAck;
    EvAppConsume A_data V1;
    EvAppReuse A_data V_new;
    EvAckLost (W_D V1);
    EvTimeout (W_D V1) ].
```

> $\mathcal{T}_2$ (ACK Loss): Sender posts write, receiver executes it, flag set, app consumes data and reuses buffer with new value V', ACK lost, sender times out. Operation **was executed**. Correct action: **do not retransmit** (would corrupt V').

#rocq_name Indistinguishability.v:77-101

Key lemma: same sender view, different execution status.

```
Lemma indistinguishable_wrt_WD_execution : forall V1 V_new,
  sender_saw_timeout (T1_packet_loss V1) (W_D V1) /\
  sender_saw_timeout (T2_ack_loss V1 V_new) (W_D V1) /\
  ~ op_executed (T1_packet_loss V1) (W_D V1) /\
  op_executed (T2_ack_loss V1 V_new) (W_D V1).
```

> **Lemma (Indistinguishability):** Both traces produce timeout observation. $\mathcal{T}_1$: operation not executed. $\mathcal{T}_2$: operation executed. The sender view is **identical** but execution status **differs**.

3.2 Main Impossibility (`Theorem1/Impossibility.v`)

#rocq_name Impossibility.v:71-77

Safety: no ghost writes.

```
Definition ProvidesSafety (overlay : TransparentOverlay) : Prop :=
  forall t op V_new,
    In (EvAppReuse A_data V_new) t ->
    op_executed t op ->
    overlay.(decide_retransmit) (sender_view t) op = false.
```

> *Safety*: If operation was executed and memory was reused, the overlay must **not** retransmit (to avoid overwriting new data).

#rocq_name Impossibility.v:82-89

Liveness: lost packets retransmitted.

```
Definition ProvidesLiveness (overlay : TransparentOverlay) : Prop :=
  forall t op,
    In (EvSend op) t ->
    ~ op_executed t op ->
    sender_saw_timeout t op ->
    overlay.(decide_retransmit) (sender_view t) op = true.
```

> *Liveness*: If operation was sent but not executed (packet lost), and sender timed out, the overlay **must** retransmit.

#rocq_name Impossibility.v:133-137

Both traces have identical sender view.

```
Lemma sender_views_equal :
  sender_view T1_concrete = sender_view T2_concrete.
Proof. unfold T1_concrete, T2_concrete. simpl. reflexivity. Qed.
```

> **Lemma:** $\sigma(\mathcal{T}_1) = \sigma(\mathcal{T}_2) = [\text{ObsSent}(W_D), \text{ObsTimeout}(W_D)]$.

#rocq_name Impossibility.v:181-238

THEOREM 1: No transparent overlay provides both safety and liveness.

```
Theorem impossibility_safe_retransmission :
  forall overlay : TransparentOverlay,
    Transparent overlay ->
    SilentReceiver -> MemoryReuseAllowed -> NoExactlyOnce ->
    ~ (ProvidesSafety overlay /\ ProvidesLiveness overlay).
```

> **Theorem 1 (Impossibility of Safe Retransmission):** Under transparency, no overlay can provide both safety and liveness.
>
> **Proof:** By construction of $\mathcal{T}_1, \mathcal{T}_2$:
> - Liveness on $\mathcal{T}_1$ (packet lost): $D(\sigma(\mathcal{T}_1)) = \text{retransmit}$
> - Safety on $\mathcal{T}_2$ (executed, reused): $D(\sigma(\mathcal{T}_2)) = \text{skip}$
> - But $\sigma(\mathcal{T}_1) = \sigma(\mathcal{T}_2)$ and D is a function
> - Contradiction: $\text{retransmit} = \text{skip}$ $\square$

4 Theorem 2: Non-Idempotency (`Theorem2/`)

4.1 Idempotency Definition (`Theorem2/Atomics.v`)

#rocq_name Atomics.v:26-29

Formal definition of idempotency.

```
Definition Idempotent (op : Op) (m : Memory) : Prop :=
  let (m1, r1) := exec_op m op in
  let (m2, r2) := exec_op m1 op in
  m1 = m2 /\ r1 = r2.
```

> Operation op is *idempotent* on memory m iff executing twice yields:
> 1. Same memory state: $m_1 = m_2$
> 2. Same result: $r_1 = r_2$

#rocq_name Atomics.v:33-70

THEOREM 2a: FADD is not idempotent.

```
Theorem fadd_non_idempotent : forall a delta m,
  delta > 0 -> ~ Idempotent (OpFADD a delta) m.
```

> **Theorem 2a (FADD Non-Idempotency):** For $\delta > 0$, $\text{FADD}(a, \delta)$ is not idempotent.
>
> **Proof:** Let $m(a) = v$. After one FADD: $m_1(a) = v + \delta$. After retry: $m_2(a) = v + 2\delta$. Since $\delta > 0$: $v + \delta \neq v + 2\delta$. □

#rocq_name Atomics.v:75-110

CAS idempotency characterization.

```
Theorem cas_idempotent_iff : forall a expected new_val m,
  Idempotent (OpCAS a expected new_val) m <->
  (mem_read m a <> expected \/ expected = new_val).
```

> **Theorem:** CAS is idempotent iff either:
> 1. It fails (current value $\neq$ expected), OR
> 2. expected = new_val (no actual change)
>
> Otherwise (successful CAS with actual change): **not idempotent**.

4.2 FADD Retry (`Theorem2/FADD.v`)

#rocq_name FADD.v:35-40

Value after one FADD.

```
Lemma single_fadd_value :
  mem_read state_after_one target_addr = delta.
```

> After one $\text{FADD}(a, \delta)$ from initial state: $m(a) = \delta$.

#rocq_name FADD.v:42-50

Value after two FADDs.

```
Lemma double_fadd_value :
  mem_read state_after_two target_addr = 2 * delta.
```

> After retry (two FADDs): $m(a) = 2\delta$.

#rocq_name FADD.v:59-65

FADD retry corrupts state.

```
Theorem fadd_retry_state_corruption :
  mem_read state_after_two target_addr <> mem_read state_after_one target_addr.
```

> **Theorem:** FADD retry produces incorrect state: $2\delta \neq \delta$ when $\delta > 0$.

4.3 CAS Concurrent Scenario (`Theorem2/CAS.v`)

#rocq_name CAS.v:106-113

THEOREM 2b: CAS can succeed twice.

```
Theorem cas_double_success :
  result_1 = ResCASResult true 0 /\
  result_3 = ResCASResult true 0.
```

> **Theorem 2b (CAS Double Success):** Under concurrent modification, a CAS retry can succeed even after the original succeeded (ABA problem).
>
> **Construction:**
> 1. S.CAS($0 \to 1$) succeeds, $m(a) = 1$
> 2. Concurrent P.CAS($1 \to 0$) succeeds, $m(a) = 0$
> 3. S retries CAS($0 \to 1$): **succeeds again!**
>
> Single logical CAS executed twice. P's modification silently overwritten.

4.4 Combined Result (`Theorem2/Atomics.v`)

#rocq_name Atomics.v:328-352

No transparent overlay for non-idempotent ops.

```
Theorem no_transparent_overlay_non_idempotent :
  IndistinguishableExecutionStatus ->
  forall (overlay : TransparentOverlay) (op : Op) (m : Memory),
    ~ Idempotent op m ->
    ~ (LiveRetransmit overlay /\
       (forall t, op_executed t op -> decide_retransmit (sender_view t) op = false)).
```

> **Theorem (Combined):** Given indistinguishable execution status (Theorem 1) and non-idempotency (Theorem 2), no transparent overlay can provide both liveness (retry on packet loss) and safety (no retry after execution).

#rocq_name Atomics.v:355-384

Corollary for atomic operations.

```
Corollary no_transparent_overlay_atomics :
  IndistinguishableExecutionStatus ->
  forall (overlay : TransparentOverlay),
    (forall a delta, delta > 0 -> ~ (LiveRetransmit overlay /\ SafeNoRetry overlay (OpFADD a delta))) /\
    (forall a expected new_val, ... -> ~ (LiveRetransmit overlay /\ SafeNoRetry overlay (OpCAS ...))).
```

> **Corollary:** No transparent overlay supports FADD (with $\delta > 0$) or successful CAS (with exp $\neq$ new).

5 Theorem 3: Consensus Hierarchy (`Theorem3/`)

5.1 Consensus Framework (`Theorem3/ConsensusNumber.v`)

#rocq_name ConsensusNumber.v:681-684

Constraint on read/write observations.

```
Definition valid_rw_observation (obs : list nat -> nat -> nat) : Prop :=
  forall exec1 exec2 i j,
    writers_before exec1 i = writers_before exec2 j ->
    obs exec1 i = obs exec2 j.
```

> *Read/Write Observation Constraint*: Observation depends only on memory state (prior writes). If two executions have the same prior write history at positions i, j, observations must be equal. This captures: reads see memory values, writes return only acknowledgment.

#rocq_name ConsensusNumber.v:214-217

Constraint on FADD observations.

```
Definition valid_fadd_observation (obs : list nat -> nat -> nat) : Prop :=
  forall exec1 exec2 i,
    same_elements (procs_before_as_set exec1 i) (procs_before_as_set exec2 i) ->
    obs exec1 i = obs exec2 i.
```

> *FADD Observation Constraint*: Observation depends only on the **set** of prior processes (not order). This captures: FADD returns sum, and addition is commutative ($\delta_0 + \delta_1 = \delta_1 + \delta_0$).

#rocq_name ConsensusNumber.v:1073-1076

Constraint on CAS observations.

```
Definition valid_cas_observation (obs : list nat -> nat -> nat) : Prop :=
  forall exec i, exec <> [] -> obs exec i = winner exec.
```

> *CAS Observation Constraint*: Observation equals the winner (first process to execute). This is **derived** from CAS semantics: first CAS to sentinel wins, all later fail, all read the winner's value.

#rocq_name ConsensusNumber.v:761-778

CN(Register) = 1: Cannot solve 2-consensus.

```
Theorem readwrite_2consensus_impossible_same_protocol :
  forall obs : list nat -> nat -> nat,
    valid_rw_observation obs ->
    ~ exists (decide : nat -> nat),
        decide (obs solo_0 0) = 0 /\
        decide (obs solo_1 1) = 1.
```

> **Theorem (Register CN = 1):** No read/write protocol can solve 2-consensus.
>
> **Proof:** Solo executions [0] and [1] have same prior state (empty). Any valid R/W observation gives same value for both. But validity requires deciding 0 for [0] and 1 for [1]. Contradiction. □

#rocq_name ConsensusNumber.v:261-278

CN(FADD) = 2: Cannot solve 3-consensus.

```
Theorem fadd_cannot_solve_3consensus :
  forall obs : list nat -> nat -> nat,
    valid_fadd_observation obs ->
    ~ exists (decide : nat -> nat),
        decide (obs exec_012 2) = inp (winner exec_012) /\
        decide (obs exec_102 2) = inp (winner exec_102).
```

> **Theorem (FADD CN = 2):** FADD cannot solve 3-consensus.
>
> **Proof:** In executions $[0, 1, 2]$ and $[1, 0, 2]$, process 2 sees the same set $\{0, 1\}$ ran before (sums are equal: $\delta_0 + \delta_1 = \delta_1 + \delta_0$). But winners differ (0 vs 1). □

#rocq_name ConsensusNumber.v:1092-1104

CN(CAS) = ∞: Can solve any n-consensus.

```
Theorem valid_cas_no_ambiguity :
  forall obs : list nat -> nat -> nat,
    valid_cas_observation obs ->
    forall exec1 exec2,
      exec1 <> [] -> exec2 <> [] ->
      winner exec1 <> winner exec2 ->
      forall i, obs exec1 i <> obs exec2 i.
```

> **Theorem (CAS CN = ∞):** Any valid CAS observation allows solving n-consensus for all n.
>
> **Proof:** CAS observations directly reveal the winner. Different winners → different observations → always distinguishable. □

5.2 Failover as 2-Consensus (`Theorem3/FailoverConsensus.v`)

#rocq_name FailoverConsensus.v:38-40

Failover decision space.

```
Inductive FailoverDecision :=
  | Commit  (* Original CAS was executed; do NOT retry *)
  | Abort.  (* Original CAS was NOT executed; retry is SAFE *)
```

> A *failover decision* is either **Commit** (operation executed, skip retry) or **Abort** (operation not executed, do retry).

#rocq_name FailoverConsensus.v:230-243

No correct decision function exists.

```
Theorem no_correct_future_decision :
  ~ exists f : FutureObservation -> FailoverDecision,
      f scenario1_future = scenario1_correct /\
      f scenario2_future = scenario2_correct.
```

> **Theorem:** No deterministic function from sender observation to failover decision is correct.
>
> **Proof:** Both scenarios produce FutureSeesTimeout. Scenario 1 requires Abort. Scenario 2 requires Commit. But $f(\text{Timeout})$ can only return one value. □

#rocq_name FailoverConsensus.v:282-283

Definition of solving failover.

```
Definition solves_failover (V : VerificationMechanism) : Prop :=
  forall h : History, V (final_memory h) = correct_decision_for h.
```

> A verification mechanism V *solves failover* if for every possible history h, $V(m_{\text{final}}) =$ correct decision for h.

#rocq_name FailoverConsensus.v:295-296

ABA: both histories have same memory.

```
Lemma H0_H1_same_memory : final_memory H0 = final_memory H1.
```

Lemma (ABA Problem): History H_0 (not executed) and H_1 (executed then reset) have **identical** final memory.

#rocq_name FailoverConsensus.v:304-331

THEOREM 3 Core: No verification mechanism solves failover.

```
Theorem failover_unsolvable :
  forall V : VerificationMechanism, ~ solves_failover V.
```

Theorem 3 (Failover Unsolvable): No verification mechanism can solve failover.

Proof:

- H_0: CAS not executed $\to V(m) = \text{false}$ (Abort)
- H_1: CAS executed, ABA reset $\to V(m) = \text{true}$ (Commit)
- But $\text{final\_memory}(H_0) = \text{final\_memory}(H_1) = m$
- So $V(m) = \text{true}$ AND $V(m) = \text{false}$. Contradiction. □

5.3 Main Result (`Theorem3/Hierarchy.v`)

#rocq_name Hierarchy.v:78-89

THEOREM 3 Main: Transparent failover impossible.

```
Theorem transparent_cas_failover_impossible :
  forall tf : TransparentFailover,
    verification_via_reads tf ->
    tf.(no_metadata_writes) ->
    ~ provides_reliable_cas tf.
```

Theorem 3 (Main Impossibility): Transparent failover for CAS is impossible.

Proof structure:

1. Failover requires solving 2-consensus (distinguishing H_0 from H_1)
2. Transparency limits verification to read-only operations
3. $\text{CN(Read)} = 1 < 2$
4. By Herlihy's hierarchy, CN=1 primitives cannot solve 2-consensus □

#rocq_name Hierarchy.v:136-137

Reads have consensus number 1.

```
Theorem reads_have_cn_1_verified : rdma_read_cn = cn_one.
```

$\text{CN(Read)} = 1$, verified via `valid_rw_observation`.

#rocq_name FailoverConsensus.v:633-637

Failover needs $\text{CN} \geq 2$.

```
Theorem failover_needs_cn_2 : cn_lt cn_one (Some 2).
```

Failover requires $\text{CN} \geq 2$.

---

6 Summary Table

Rocq Theorem	**File:Line**	**Paper Statement**	**Technique**
`impossibility_safe_retransmission`	Impossibility.v:181	Thm 1: No overlay provides safety + liveness	Trace construction
`fadd_non_idempotent`	Atomics.v:33	Thm 2a: FADD retry yields $2\delta \neq \delta$	Arithmetic
`cas_double_success`	CAS.v:106	Thm 2b: CAS retry succeeds twice (ABA)	Interleaving
`no_transparent_overlay_atomics`	Atomics.v:355	Thm 2 Combined: No overlay for atomics	Thm 1 + 2
`readwrite_2consensus_impossible...`	ConsensusNumber.v:761	CN(Read) = 1: solo indistinguishable	Herlihy
`failover_unsolvable`	FailoverConsensus.v:304	Thm 3 Core: No V solves failover	ABA
`transparent_cas_failover_impossible`	Hierarchy.v:78	Thm 3 Main: Transparent failover impossible	CN hierarchy

Table 1: One-to-one mapping of key theorems