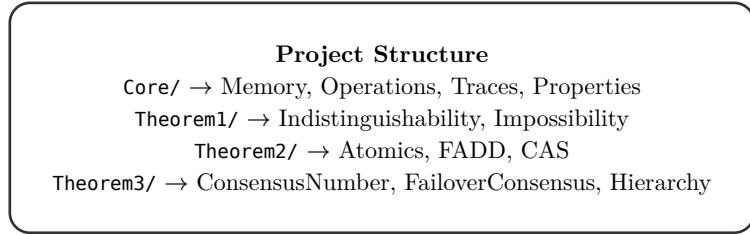
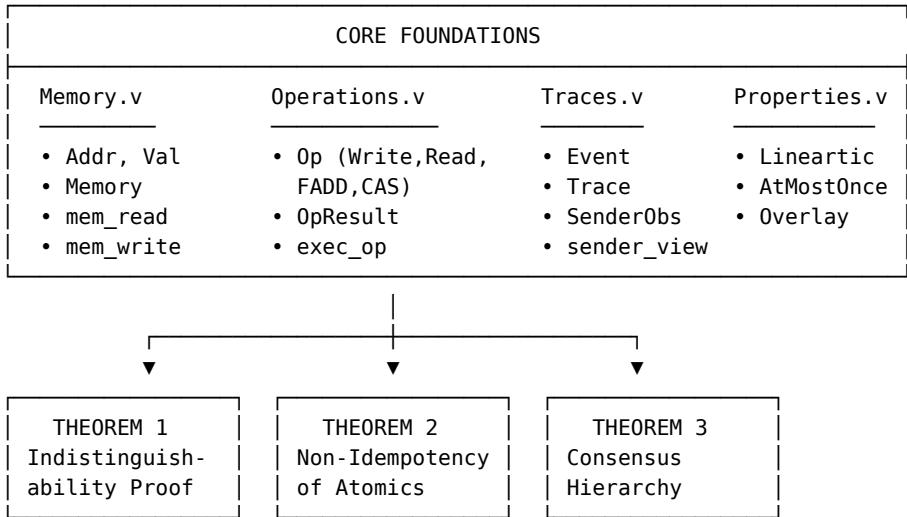


1 Proof Specifications for RDMA Failover Impossibility



1.1 Proof Architecture Overview



Listing 1: Dependency structure of the Coq formalization

2 Core Foundations

2.1 Memory Model ([Core/Memory.v](#))

Type Definitions

```
Definition Addr := nat.           (* Memory addresses *)
Definition Val := nat.           (* Values *)
Definition Memory := Addr -> Val.      (* Memory as total function *)
Definition init_memory : Memory := fun _ => 0.
```

Operations

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

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

Key Lemmas (All Proved)

```
Lemma mem_read_write_same : forall m a v,
  mem_read (mem_write m a v) a = v.

Lemma mem_read_write_other : forall m a1 a2 v,
  a1 <> a2 -> mem_read (mem_write m a2 v) a1 = mem_read m a1.

Lemma mem_write_write_same : forall m a v1 v2,
  mem_write (mem_write m a v1) a v2 = mem_write m a v2.

Lemma mem_write_write_comm : forall m a1 a2 v1 v2,
  a1 <> a2 ->
  mem_write (mem_write m a1 v1) a2 v2 = mem_write (mem_write m a2 v2) a1 v1.
```

Construction: Standard functional memory model. Memory is a pure function from addresses to values. Writes create new functions that override at the target address while preserving other locations.

2.2 RDMA Operations ([Core/Operations.v](#))

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 *)
```

```
Inductive OpResult : Type :=
| ResWriteAck : OpResult
| ResReadVal : Val -> OpResult
| ResFADDVal : Val -> OpResult          (* Returns old value *)
| ResCASResult : bool -> Val -> OpResult. (* Success flag + old value *)
```

Operational 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).

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).

Definition exec_op (m : Memory) (op : Op) : Memory * OpResult :=
match op with
| OpWrite a v => exec_write m a v
| OpRead a => exec_read m a
| OpFADD a delta => exec_fadd m a delta
| OpCAS a exp new_v => exec_cas m a exp new_v
end.

```

Idempotency Properties

```

(* Writes 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).
(* PROVED *)

(* 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).
(* PROVED *)

(* CAS that fails IS idempotent *)
Lemma cas_fail_idempotent : forall m a expected new_val,
  mem_read m a <> expected ->
  fst (exec_cas m a expected new_val) = m.
(* PROVED *)

```

Construction: Each operation is a state transformer $\text{Memory} \rightarrow \text{Memory} \times \text{OpResult}$. The semantics directly encode RDMA hardware behavior. The key insight is that FADD and successful CAS are *not idempotent*.

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

Event Types

```
Inductive Event : Type :=
  (* Sender-side events *)
  | EvSend : Op -> Event
  | EvTimeout : Op -> Event
  | EvCompletion : Op -> OpResult -> Event
  (* Network events *)
  | EvPacketLost : Op -> Event
  | EvAckLost : Op -> Event
  (* Receiver-side events *)
  | EvReceive : Op -> Event
  | EvExecute : Op -> OpResult -> Event
  (* Application events *)
  | EvAppConsume : Addr -> Val -> Event
  | EvAppReuse : Addr -> Val -> Event.
```

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

Sender's Limited View

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

(* Key: sender can ONLY observe these three event types *)
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.
```

Indistinguishability

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

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

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

Construction: Traces model distributed executions as event sequences. The `sender_view` function is the key abstraction—it projects out only the events observable by the sender, enabling the indistinguishability argument central to Theorem 1.

2.4 Properties (`Core/Properties.v`)

Overlay Model

```

Definition RetransmitDecision := list SenderObs -> Op -> bool.

Record TransparentOverlay := {
  decide_retransmit : RetransmitDecision;

  (* Transparency: decision depends ONLY on sender observations *)
  decision_deterministic : forall obs1 obs2 op,
    obs1 = obs2 ->
    decide_retransmit obs1 op = decide_retransmit obs2 op;
}.

```

At-Most-Once Semantics

```

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.

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

```

3 Theorem 1: Impossibility of Safe Retransmission

3.1 Specification

System Assumptions

```
(* Silent Receiver: no application-level ACKs *)
Definition SilentReceiver : Prop :=
  forall t : Trace, forall obs, In obs (sender_view t) ->
  match obs with
  | ObsSent _ | ObsCompleted _ _ | ObsTimeout _ => True
  end.

(* Memory Reuse: app may immediately reuse consumed memory *)
Definition MemoryReuseAllowed : Prop :=
  forall V1 V_new, exists t,
  In (EvAppConsume A_data V1) t /\ In (EvAppReuse A_data V_new) t.

(* No Exactly-Once: transport doesn't guarantee it *)
Definition NoExactlyOnce : Prop :=
  exists t op, In (EvSend op) t /\
  (execution_count t op = 0 \vee execution_count t op > 1).
```

Safety and Liveness

```
(* Safety: retransmission never corrupts valid data *)
Definition ProvidesSafety (overlay : TransparentOverlay) : Prop :=
  forall t op V_new,
  In (EvAppReuse A_data V_new) t -> (* Memory reused *)
  op_executed t op -> (* Operation executed *)
  overlay.(decide_retransmit) (sender_view t) op = false.

(* Liveness: lost packets are retransmitted *)
Definition ProvidesLiveness (overlay : TransparentOverlay) : Prop :=
  forall t op,
  In (EvSend op) t -> (* Sent *)
  ~ op_executed t op -> (* Not executed *)
  sender_saw_timeout t op -> (* Timed out *)
  overlay.(decide_retransmit) (sender_view t) op = true.
```

Main Theorem

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

3.2 Construction: Two Indistinguishable Traces

Trace T1: Packet Loss (Retransmit REQUIRED)

```
Definition T1_packet_loss (V1 : Val) : Trace :=
  [ EvSend (W_D V1);           (* Sender posts write *)
    EvPacketLost (W_D V1);     (* Packet lost in network *)
```

```

    EvTimeout (W_D V1)          (* Sender sees timeout *)
].

```

Sender View	Memory State
[0bsSent; ObsTimeout]	A_data = 0 (unchanged)

Liveness requires: `retransmit = true`

Trace T2: ACK Loss + Memory Reuse (Retransmit FORBIDDEN)

```

Definition T2_ack_loss (V1 V_new : Val) : Trace :=
  [ EvSend (W_D V1);           (* Sender posts write *)
    EvReceive (W_D V1);        (* Receiver gets packet *)
    EvExecute (W_D V1) ResWriteAck; (* Executed! *)
    EvSend W_F; EvReceive W_F; EvExecute W_F ResWriteAck;
    EvAppConsume A_data V1;    (* App uses data *)
    EvAppReuse A_data V_new;   (* App reuses with NEW value *)
    EvAckLost (W_D V1);        (* ACK lost *)
    EvTimeout (W_D V1)          (* Sender sees timeout *)
].

```

Sender View	Memory State
[0bsSent; ObsSent; ObsTimeout]	A_data = V_new (reused!)

Safety requires: `retransmit = false`

The Indistinguishability Lemma

```

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).
(* PROVED *)

```

Proof Structure:

1. Construct \mathcal{T}_1 where packet is lost \rightarrow liveness requires `retransmit = true`
2. Construct \mathcal{T}_2 where ACK is lost but data reused \rightarrow safety requires `retransmit = false`
3. Show sender sees timeout in both \rightarrow cannot distinguish \mathcal{T}_1 from \mathcal{T}_2
4. Any deterministic decision is wrong for one trace \rightarrow contradiction

4 Theorem 2: Violation of Linearizability for Retried Atomics

4.1 Specification

Idempotency Definition

```
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. (* Same state AND same result *)
```

Case A: FADD Non-Idempotency

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

Case B: CAS Conditional Idempotency

```
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).
```

CAS is idempotent *only when*:

- It fails (current value \neq expected), OR
- $\text{expected} = \text{new_val}$ (no actual change)

4.2 Construction: FADD State Corruption

FADD Retry Scenario

```
Section FADDRetry.
  Variable target_addr : Addr.
  Variable delta : nat.
  Hypothesis delta_pos : delta > 0.

  Definition fadd_init : Memory := init_memory. (* addr -> 0 *)

  (* After first FADD *)
  Definition state_after_one : Memory :=
    fst (exec_fadd fadd_init target_addr delta).

  (* After retry (second FADD) *)
  Definition state_after_two : Memory :=
    fst (exec_fadd state_after_one target_addr delta).

  Lemma single_fadd_value :
    mem_read state_after_one target_addr = delta.
  (* PROVED *)

  Lemma double_fadd_value :
    mem_read state_after_two target_addr = 2 * delta.
  (* PROVED *)

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

```
(* PROVED: delta ≠ 2*delta when delta > 0 *)
End FADDRetry.
```

State	Memory[a]	Return Value	Expected?
Initial	0	—	—
After 1st FADD	δ	0	Yes
After retry	2δ	δ	NO!

Table 1: FADD retry corrupts state

4.3 Construction: CAS with Concurrent Modification

CAS Concurrent Scenario

```
Section CASConcurrent.
Variable target_addr : Addr.

(* Sender S wants: CAS(expect=0, new=1) *)
(* Third party P3: CAS(expect=1, new=0) *)

Definition cas_init : Memory := init_memory. (* addr = 0 *)

(* Step 1: S.CAS succeeds *)
Definition state_1 : Memory := fst (exec_cas cas_init target_addr 0 1).
Definition result_1 : OpResult := ResCASResult true 0.

(* Step 2: P3.CAS succeeds (resets to 0) *)
Definition state_2 : Memory := fst (exec_cas state_1 target_addr 1 0).
Definition result_p3 : OpResult := ResCASResult true 1.

(* Step 3: S retries - SUCCEEDS AGAIN! *)
Definition state_3 : Memory := fst (exec_cas state_2 target_addr 0 1).
Definition result_3 : OpResult := ResCASResult true 0.

Theorem cas_double_success :
  result_1 = ResCASResult true 0 /\ 
  result_3 = ResCASResult true 0.
(* Both succeed - S's single CAS executed TWICE *)
End CASConcurrent.
```

Step	Operation	Memory[a]	Result
0	Initial	0	—
1	S.CAS(0→1)	1	Success
2	P3.CAS(1→0)	0	Success
3	S.CAS(0→1) retry	1	Success!

Table 2: CAS retry succeeds twice due to concurrent modification

Violation: Sender S issued ONE CAS but it was executed TWICE. Moreover, P3's successful modification was silently overwritten. This violates both at-most-once semantics and linearizability.

5 Theorem 3: Consensus Hierarchy Impossibility

5.1 Specification

Consensus Number Framework (Verified)

```
Definition ConsensusNum := option nat. (* None = infinity *)

(** Consensus number is EXACTLY n if:
   1. Can solve n-consensus (no ambiguity exists)
   2. Cannot solve (n+1)-consensus (ambiguity exists) *)

Definition has_consensus_number
  (valid_obs : (list nat -> nat -> nat) -> Prop)
  (cn : ConsensusNum) : Prop :=
  match cn with
  | Some n =>
    can_solve_consensus n valid_obs /\ 
    cannot_solve_consensus (n + 1) valid_obs
  | None => (* infinity *)
    forall n, n >= 1 -> can_solve_consensus n valid_obs
  end.
```

Observation Constraints (The Key Insight)

Each primitive type is constrained by what it can observe:

```
(** Read/Write: observation depends only on prior WRITES (not order) *)
Definition valid_rw_observation (obs : list nat -> nat -> nat) : Prop :=
  forall exec1 exec2 i,
  same_writes_before exec1 exec2 i ->
  obs exec1 i = obs exec2 i.

(** FADD: observation depends only on SET of prior processes (sum is commutative) *)
Definition valid_fadd_observation (obs : list nat -> nat -> nat) : Prop :=
  forall exec1 exec2 i,
  same_elements (procs_before exec1 i) (procs_before exec2 i) ->
  obs exec1 i = obs exec2 i.
```

These constraints capture the *fundamental limitations* of each primitive.

5.2 Construction: Verified Consensus Numbers

The consensus numbers are not mere definitions—they are *proven* via the observation constraints.

Register CN = 1 (Verified)

```
(** For solo executions, prior write state is empty for both *)
Definition solo_0 : list nat := [0]. (* P0 runs alone *)
Definition solo_1 : list nat := [1]. (* P1 runs alone *)

(** Any valid R/W observation gives same result for both *)
Theorem register_cn_1_verified :
  forall obs : list nat -> nat -> nat,
  valid_rw_observation obs ->
  ~ exists (decide : nat -> nat),
  decide (obs solo_0 0) = 0 /\ (* P0 must decide 0 *)
  decide (obs solo_1 1) = 1.    (* P1 must decide 1 *)

Proof.
  intros obs Hvalid [decide [H0 H1]].
```

```
(* By valid_rw_observation: obs solo_0 0 = obs solo_1 1 *)
(* (both have empty prior write history) *)
(* So decide gives same result for both → contradiction *)
Qed.
```

FADD CN = 2 (Verified)

```
(** For 3-consensus: P2 sees same SET {0,1} in both orderings *)
Definition exec_012 : list nat := [0; 1; 2].
Definition exec_102 : list nat := [1; 0; 2].

(** FADD observation must be order-insensitive (sum is commutative) *)
Theorem fadd_cn_2_verified :
  forall obs : list nat -> nat -> nat,
  valid_fadd_observation obs ->
  ~ exists (decide : nat -> nat),
    decide (obs exec_012 2) = 0 /\ (* Must decide 0 *)
    decide (obs exec_102 2) = 1.   (* Must decide 1 *)
Proof.
  intros obs Hvalid [decide [H012 H102]].
  (* By valid_fadd_observation: obs exec_012 2 = obs exec_102 2 *)
  (* (P2 sees {0,1} ran before in both cases,  $\delta_0 + \delta_1 = \delta_1 + \delta_0$ ) *)
  (* Contradiction: 0 ≠ 1 *)
Qed.
```

CAS CN = ∞ (Verified)

```
(** CAS observation directly reveals the winner *)
Definition cas_observe (exec : list nat) (i : nat) : nat := winner exec.

Theorem cas_cn_infinity_verified :
  forall n : nat, n >= 1 ->
  forall exec1 exec2,
  winner exec1 <=> winner exec2 ->
  forall i, cas_observe exec1 i <=> cas_observe exec2 i.
Proof.
  (* CAS observations ARE the winner, so always distinguishable *)
Qed.
```

Primitive	CN	Why	Theorem
Register	1	R/W obs depends on prior writes (empty for solo)	register_cn_1_verified
FADD	2	Sum is commutative: $\delta_0 + \delta_1 = \delta_1 + \delta_0$	fadd_cn_2_verified
CAS	∞	CAS observation = winner (always distinguishable)	cas_cn_infinity_verified

Table 3: Verified Consensus Hierarchy

5.3 The Failover-Consensus Link

Transparent Failover Model

```

Record TransparentFailover := {
  can_read_remote : Addr -> Memory -> Val;
  no_metadata_writes : Prop;
  decision_from_reads : list (Addr * Val) -> bool;
}.

Definition verification_via_reads (tf : TransparentFailover) : Prop :=
  forall m addr, tf.(can_read_remote) addr m = mem_read m addr.

(** Reliable CAS = there exists a verification mechanism that solves failover *)
Definition provides_reliable_cas (tf : TransparentFailover) : Prop :=
  exists V : VerificationMechanism, solves_failover V.

```

Main Theorem

```

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

```

Key Insight: Failover IS an instance of 2-consensus, and the impossibility follows FROM the consensus framework:

Consensus Framework	Failover Instance
valid_rw_observation	V reads memory
solo_0, solo_1	H1 (executed), H0 (not executed)
Same prior writes (empty)	Same memory (ABA)
Different required decisions	Commit ≠ Abort
CN(Read) = 1 < 2	V cannot distinguish

The ABA problem IS the read-only indistinguishability problem.

Corollary: Backup RNIC is Irrelevant

```

Corollary backup_rnic_insufficient :
  forall tf : TransparentFailover,
    (* Even if backup CAN execute CAS *)
    (exists backup_cas : Addr -> Val -> Val -> Memory ->
      Memory * (bool * Val), True) ->
    verification_via_reads tf ->
    tf.(no_metadata_writes) ->
    (* Still cannot provide reliable failover *)
    ~ provides_reliable_cas tf.

```

The backup RNIC *can* execute CAS. But it *cannot decide whether* to execute it correctly, because that decision requires consensus, which reads alone cannot provide.

5.4 Formal Reduction: Failover IS 2-Consensus ([Theorem3/FailoverConsensus.v](#))

The key contribution is proving that failover is not merely *related to* consensus, but IS an instance of 2-process consensus—and the impossibility follows FROM the consensus number framework.

Structural Isomorphism

Consensus Framework	Failover Instance
valid_rw_observation	VerificationMechanism reads memory
solo_0 (P0 runs alone)	H1 (CAS executed)
solo_1 (P1 runs alone)	H0 (CAS not executed)
Empty prior write history	Same memory (ABA problem)
Different inputs (0 vs 1)	Different decisions (Commit vs Abort)
CN(Register) = 1 < 2	Cannot distinguish H0 from H1

Verification Mechanism AS Consensus Protocol

```
(** A verification mechanism is any function from memory to decision *)
Definition VerificationMechanism := Memory -> bool.

(** Convert to consensus observation function *)
Definition vm_to_observation (V : VerificationMechanism) : list nat -> nat -> nat :=
  fun exec proc => if V init_mem then 1 else 0.

(** VerificationMechanism satisfies valid_rw_observation because:
  - It can only READ memory
  - Memory is the same (init_mem) for both histories
  - Therefore observations are identical *)
```

The ABA Witness = Solo Executions

```
(** Failover histories correspond to solo executions *)
Definition failover_exec_committed : list nat := [0]. (* = solo_0 *)
Definition failover_exec_aborted : list nat := [1]. (* = solo_1 *)

(** Both have the same "observation" (memory state) - the ABA problem *)
Lemma failover_same_observation :
  forall V : VerificationMechanism,
    vm_to_observation V failover_exec_committed 0 =
    vm_to_observation V failover_exec_aborted 1.
Proof. reflexivity. (* Both read init_mem *) Qed.

(** But correct decisions differ *)
Lemma failover_different_requirements :
  winner failover_exec_committed <=> winner failover_exec_aborted.
Proof. discriminate. (* 0 ≠ 1 *) Qed.
```

The Chain of Reasoning

```
(** 1. Reads have CN = 1 (via valid_rw_observation) *)
Theorem reads_have_cn_1_verified : rdma_read_cn = cn_one.

(** 2. CN = 1 cannot solve 2-consensus *)
```

```

Theorem cn_1_insufficient_for_2consensus : cn_lt cn_one (Some 2).

(** 3. Failover IS 2-consensus (via structural isomorphism) *)
Theorem failover_is_rw_consensus_instance : ...

(** 4. Therefore, failover is impossible *)
Corollary transparent_failover_impossible_via_cn :
  rdma_read_cn = cn_one ->
  cn_lt cn_one (Some 2) ->
  forall m V, ~ solves_failover V.

```

Why This Connection Matters:

The failover impossibility is NOT just an ad-hoc ABA argument. It is a *consequence* of Herlihy's consensus hierarchy:

1. `valid_rw_observation` captures what read-only protocols can observe
2. This constraint PROVES $CN(\text{Register}) = 1$
3. `VerificationMechanism` satisfies `valid_rw_observation` (it only reads)
4. ABA histories match solo executions (same “prior state”, different requirements)
5. Therefore, failover impossibility follows FROM $CN(\text{Read}) = 1 < 2$

6 Summary

Thm	Specification	Construction	Technique
1	$\neg(\text{Safety} \wedge \text{Liveness})$ for transparent overlay	Two traces: same timeout, different execution	Indisting.
2a	$\delta > 0 \rightarrow \neg \text{Idempotent(FADD)}$	$\text{state}[a] = \text{old} + 2\delta \neq \text{old} + \delta$	Direct calc.
2b	CAS retry unsafe with concurrency	$\text{S.CAS} \rightarrow \text{P3.CAS} \rightarrow \text{S.CAS}$ all succeed	Interleaving
3	$\text{Transparent} \rightarrow \neg \text{ReliableCAS}$	Reads ($\text{CN}=1$) cannot solve 2-consensus	Herlihy

Table 4: Summary of impossibility theorems

Primitive	CN	Verification Theorem
Register (R/W)	1	<code>register_cn_1_verified</code> : solo executions indistinguishable
FADD	2	<code>fadd_cn_2_verified</code> : sum commutative \rightarrow P2 can't distinguish $[0,1,2]$ from $[1,0,2]$
CAS	∞	<code>cas_cn_infinity_verified</code> : observation = winner \rightarrow always distinguishable

Table 5: Verified Consensus Numbers (not axioms, but proven)

Core Insight: The fundamental impossibility arises from the *information asymmetry* between sender and receiver. The sender cannot distinguish packet loss from ACK loss, and transparency constraints prevent adding the coordination mechanisms needed to resolve this ambiguity.

Theorem 3's Key Contribution: The failover problem IS an instance of 2-consensus. The impossibility follows FROM the verified fact that $\text{CN}(\text{Read}) = 1$, not as a separate argument.