Formal Verification of RDMA Failover Impossibility

We formalize and mechanically verify three impossibility theorems for transparent RDMA failover using the Rocq proof assistant (formerly Coq). All proofs are available at github.com/taooceros/shift-verification.

Theorem 1: Indistinguishability of Packet Loss and ACK Loss

Definition (Sender View)**.** Let $\mathcal{T}$ be an execution trace. The *sender view* $\sigma(\mathcal{T})$ is the projection containing only sender-observable events: operation sends, completions, and timeouts.

Definition (Transparent Overlay)**.** A failover mechanism is *transparent* if its retransmission decision $D : \sigma(\mathcal{T}) \times \mathrm{Op} \to \{0, 1\}$ depends only on the sender view.

Theorem (Impossibility of Safe Retransmission)**.** For any transparent overlay D, there exist executions $\mathcal{T}_1$ (packet lost) and $\mathcal{T}_2$ (ACK lost, memory reused) such that:

$$\sigma(\mathcal{T}_1) = \sigma(\mathcal{T}_2) \tag{1}$$

but safety requires $D(\sigma(\mathcal{T}_1)) = 1$ (retransmit) while $D(\sigma(\mathcal{T}_2)) = 0$ (do not retransmit).

Proof. We construct two traces with identical sender views but opposite correctness requirements:

$\mathcal{T}_1$: $[\mathrm{Send}(W_D), \mathrm{PacketLost}(W_D), \mathrm{Timeout}(W_D)]$
$\mathcal{T}_2$: $[\mathrm{Send}(W_D), \mathrm{Receive}, \mathrm{Execute}, \mathrm{AppConsume}, \mathrm{AppReuse}(V'), \mathrm{AckLost}, \mathrm{Timeout}(W_D)]$

Both produce sender view $[\mathrm{ObsSent}(W_D), \mathrm{ObsTimeout}(W_D)]$. In $\mathcal{T}_1$, the operation was never executed (liveness requires retry). In $\mathcal{T}_2$, the operation executed and memory was reused with value $V' \neq V_1$ (safety forbids retry). Since D is a function, $D(\sigma(\mathcal{T}_1)) = D(\sigma(\mathcal{T}_2))$, contradicting the requirements. □

Theorem 2: Non-Idempotency of Operations

Theorem (FADD Non-Idempotency)**.** For any $\delta > 0$ and memory state m, FADD is not idempotent:

$$\mathrm{exec}_{\mathrm{FADD}\left(\mathrm{exec}_{\mathrm{FADD}(m,a,\delta)},a,\delta\right)} \neq \mathrm{exec}_{\mathrm{FADD}(m,a,\delta)} \tag{2}$$

Theorem (Queue Sliding (Two-Sided Non-Idempotency))**.** Retrying a SEND operation consumes an additional Receive WQE, corrupting the message-to-buffer mapping.

Proof. Let the receiver queue $Q_R = [R_1, R_2, ...]$. Trace 1 (Success): Message M_1 consumes R_1. $Q_{R'} = [R_2, ...]$. ACK lost. Trace 2 (Retry): Message M_1 (retry) consumes R_2. $Q_{R''} = [R_3, ...]$. Result: M_1 is duplicated, and R_2 (intended for M_2) is lost. The streams are permanently misaligned. □

Theorem (CAS Retry Violation)**.** Under concurrent modification, a CAS retry can succeed twice, violating at-most-once semantics.

Proof. Consider sender S with $\mathrm{CAS}(a, 0, 1)$ and concurrent process P with $\mathrm{CAS}(a, 1, 0)$:

State 0: $m[a] = 0$
State 1: $S.\mathrm{CAS}(0, 1)$ succeeds $\to m[a] = 1$
State 2: $P.\mathrm{CAS}(1, 0)$ succeeds $\to m[a] = 0$
State 3: S retries $\mathrm{CAS}(0, 1)$ $\to$ succeeds again!

S's single CAS executed twice, and P's successful modification was silently overwritten. □

Theorem 3: Consensus Hierarchy Barrier

We prove that failover coordination is equivalent to 2-process consensus, which read-only verification cannot solve.

Unified Observation Constraint Framework

Definition (Observation Constraint)**.** Each synchronization primitive defines a constraint on what protocols can observe:

Primitive	**Constraint**
Register	$\text{valid}_{\text{rw}} : \text{obs}(\text{exec}, i)$ depends only on writes before i
FADD	$\text{valid}_{\text{fadd}} : \text{obs}(\text{exec}, i)$ depends only on $\{j : j \text{ before } i\}$ (set, not order)
CAS	$\text{valid}_{\text{cas}} : \text{obs}(\text{exec}, i) = \text{winner}(\text{exec})$ (first process)

The constraints are *derived* from primitive semantics:

- **Register**: Reads are invisible; only writes affect observable state
- **FADD**: Returns sum of prior deltas; $\delta_0 + \delta_1 = \delta_1 + \delta_0$
- **CAS**: First CAS to sentinel wins; all subsequent fail; all read same value

Reduction: Failover Solver ⇒ 2-Consensus Protocol

Following Herlihy's methodology (cf. Theorem 5.4.1 for FIFO queues), we prove failover requires CN $\geq$ 2 by constructing a 2-consensus protocol from a hypothetical failover solver.

Definition (Failover Solver)**.** A failover solver F : Memory $\to$ {`true`, `false`} returns `true` (Commit) if CAS was executed, `false` (Abort) otherwise.

Theorem (2-Consensus Protocol from Failover Solver)**.** Given a correct failover solver F, the following protocol solves 2-consensus:

FIFO Protocol (Herlihy)

```
Queue := [WIN, LOSE]
proposed[i] := v_i
result := dequeue()
if result = WIN
  then decide(proposed[me])
  else decide(proposed[other])
```

Failover Protocol (Ours)

```
Memory := m (ABA state)
proposed[i] := v_i
result := F(m)
if result = true
  then decide(proposed[0])
  else decide(proposed[1])
```

The failover solver F acts like `dequeue()`: it reveals who "won".

Proof. We verify the three conditions:

- **Wait-free**: The protocol contains no loops. Each process executes a finite number of steps. ✓
- **Agreement**: Both processes call the same F on the same memory m. They get the same result and select from the same `proposed[]` array. Therefore they decide the same value. ✓
- **Validity**: If P_0 won (CAS executed), then $F(m) =$ `true` by correctness of F, so both decide `proposed[0]` $= P_0$'s input. Similarly for P_1. The decision is always the winner's input. ✓

□

Theorem (Failover Solver Yields 2-Consensus)**.** A correct failover solver F yields a read-based observation/decision pair that would solve 2-consensus.

Proof. Given F satisfying `solves_failover`, construct:

- $\text{obs}(e, i) := \text{if } F(m_0) \text{ then } 0 \text{ else } 1$ (constant; satisfies `valid_rw_observation` trivially)
- $\text{decide}(x) := \text{if } x = 0 \text{ then } 0 \text{ else } 1$

Solo validity for P_0: by `solves_failover` applied to `HistExecuted`(m_0), $F(m_0) =$ `true`, so $\text{obs} = 0$ and $\text{decide}(0) = 0$.
Solo validity for P_1: by `solves_failover` applied to `HistNotExecuted`(m_0), $F(m_0) =$ `false`, so $\text{obs} = 1$ and $\text{decide}(1) = 1$.

(These two directions of `solves_failover` are contradictory; the CN theorem combines them.) □

Theorem (Failover Impossible by Register CN=1)**.** No verification mechanism solves failover.

Proof. By formal reduction through the consensus hierarchy:

1. **Positive reduction** (`failover_solver_yields_2consensus`): A correct failover solver yields `obs` satisfying `valid_rw_observation` and `decide` satisfying solo validity for both P_0 and P_1.
2. **CN barrier** (`readwrite_2consensus_impossible_same_protocol`): No read-based observation function admits a decision function satisfying both solo validities (Register CN=1).
3. **Combined** (`failover_impossible_by_read_cn`): Contradiction. □

□

Theorem (Main Result)**.** Transparent RDMA failover for atomic operations is impossible because:
1. A correct failover solver would yield a read-based 2-consensus protocol (positive reduction)
2. Register CN=1 makes such a protocol impossible (CN barrier)
3. The main theorem lifts this to the `TransparentFailover` interface

Mechanization

Component	**Lines**	**Key Theorems**
Core definitions	400	Memory model, RDMA operations, traces
Theorem 1	200	`impossibility_safe_retransmission`
Theorem 2	300	`fadd_not_idempotent`, `send_queue_sliding`, `cas_double_success`
Theorem 3	2200	`register_cn_1_verified`, `fadd_cn_2_verified`, `valid_cas_no_ambiguity`, `failover_impossible_by_read_cn`, `transparent_cas_failover_impossible`

Table 1: Rocq formalization statistics

All proofs are constructive and fully mechanized in Rocq 9.0 (3,900 lines). The consensus number framework provides a unified treatment where each primitive's limitation is derived from its operational semantics. The failover impossibility is formally derived FROM the Register CN=1 theorem via `failover_impossible_by_read_cn`, which chains the positive reduction (`failover_solver_yields_2consensus`) with the CN barrier (`readwrite_2consensus_impossible_same_protocol`). This connection is mechanized, not merely informal.