1 Transparent RDMA NIC Failover: What Can and Cannot Be Supported

1.1 The Scenario

Consider a high-availability RDMA system with a primary NIC and a backup NIC. When the primary NIC fails mid-operation, we want the backup NIC to transparently take over—completing any in-flight operations without the application noticing the failure.

```
┌────────┐        ┌─────────────┐        ┌────────┐
│ Sender │──op──▶│ Primary NIC │───?───│ Memory │
└────────┘        └─────────────┘        └────────┘
    │                 ✗ FAILS                 │
    │             ┌─────────────┐             │
    └────op?─────▶│ Backup NIC  │──────?──────┘
                  └─────────────┘
```

Listing 1: NIC failover scenario: should the backup retry the operation?

The sender detects that the primary NIC has failed. The critical question: **Should the backup NIC re-execute the operation?**

- If the primary executed before failing → backup must NOT retry (double-execution)
- If the primary failed before executing → backup MUST retry (for liveness)

A **transparent failover** mechanism makes this decision without modifying the application protocol—using only the sender's observations and what the backup can read from memory.

1.2 The Question: Which Operations Can Be Supported?

Not all RDMA operations are equal. We analyze which can be transparently failed over:

Operation	**Idempotent?**	**Transparent Failover?**
Send/Recv (two-sided)	Yes	Possible
Write[1]	Yes	Possible
Read	Yes	Possible
FADD	No	Impossible
CAS	Conditional	Impossible in general

The fundamental issue: Determining whether an operation was executed is a 2-consensus problem (shown in Theorem 3). Any operation whose correctness depends on knowing whether it executed cannot be transparently failed over.

For atomic operations (FADD, CAS), the problem is compounded: they are non-idempotent, so incorrect retry corrupts state regardless of ordering concerns.

[1] Write is idempotent in isolation, but if used for memory ordering (e.g., signaling "data ready"), correctness depends on execution knowledge—which requires solving 2-consensus.

1.3 The Core Problem: What Did the Primary Do?

When the primary NIC fails, there are two possible histories:

History H_1: Primary Failed Before Execution

- Sender issued atomic operation to primary NIC
- Primary NIC failed before executing
- Operation was never performed
- **Correct action**: Backup must execute the operation

History H_2: Primary Executed Then Failed

- Sender issued atomic operation to primary NIC
- Primary NIC executed the operation
- Primary NIC failed before sending completion
- **Correct action**: Backup must NOT execute (already done)

The sender observes the same thing in both cases: **the primary NIC failed and no completion was received.** For idempotent operations, this ambiguity is harmless—retry produces the same result. For atomic operations, it is fatal.

1.4 Definitions

> **Definition (Sender View):** The projection π_S extracts only what the sender can observe: operation submissions, completions, and NIC failures. The sender cannot observe whether the primary executed before failing.

> **Definition (Transparent Failover):** A failover mechanism where the backup's decision depends only on: (1) the sender's observations π_S, and (2) reading the current memory state. No persistent metadata or protocol modifications allowed.

> **Definition (Safety and Liveness):**
> - **Safety**: Each operation executes at most once across primary and backup
> - **Liveness**: If an operation was not executed by the primary, the backup eventually executes it

1.5 Theorem 1: Sender Cannot Distinguish Histories

Theorem (Indistinguishability): For any operation, the sender's observations are identical whether the primary executed before failing or failed before executing.

Proof. Consider any operation sent to the primary NIC.

History H_1: Failed Before Execution

1. Sender submits operation to primary NIC
2. Primary NIC fails before processing
3. Sender detects NIC failure
4. Sender's observation: $[\text{Submit(op)}, \text{NICFailure}]$

History H_2: Executed Then Failed

1. Sender submits operation to primary NIC
2. Primary NIC executes operation
3. Primary NIC fails before sending completion
4. Sender detects NIC failure
5. Sender's observation: $[\text{Submit(op)}, \text{NICFailure}]$

Both produce: $\pi_{S(H_1)} = \pi_{S(H_2)} = [\text{Submit(op)}, \text{NICFailure}]$

Any decision rule based solely on π_S must make the same choice for both histories. □

For idempotent operations: This is fine—retry is safe either way.

For atomic operations: This is a problem—H_1 requires retry, H_2 forbids it.

1.6 Theorem 2: Atomic Operations Are Non-Idempotent

The indistinguishability from Theorem 1 only matters because atomic operations cannot tolerate incorrect retry.

Theorem (FADD Non-Idempotency): For $\delta > 0$, executing FADD twice produces different state than executing once.

Proof. Let FADD add δ to address a, starting from $m[a] = 0$.

Scenario	**Final State**	**Return Value**
Execute once (correct)	$m[a] = \delta$	0
Execute twice (incorrect retry)	$m[a] = 2\delta$	2nd returns δ

The states differ: $\delta \neq 2\delta$ for $\delta > 0$. FADD is non-idempotent. □

Consequence: If the backup incorrectly retries FADD after the primary already executed, the application sees 2δ instead of δ—a silent corruption.

Theorem (CAS Can Succeed Twice): With concurrent modification, a CAS retry can succeed even if the original succeeded.

Proof. Consider primary executing CAS($0 \to 1$), then a concurrent process resetting the value:

Step	**Actor**	**Operation**	**Memory**
1	Primary NIC	CAS($0 \to 1$) succeeds	$0 \to 1$
2	Primary NIC	Fails before completion	—
3	Concurrent	CAS($1 \to 0$) succeeds	$1 \to 0$
4	Backup NIC	CAS($0 \to 1$) retry	$0 \to 1$ succeeds!

The backup's retry succeeds because the value returned to 0 (ABA problem). The application's single CAS executed twice. □

The Fallacy: "CAS retry is safe because duplicates fail" assumes no concurrent modification.

1.7 Theorem 3: Memory Inspection Cannot Help

Perhaps the backup NIC can read memory to determine if the primary executed? Theorem 3 shows this fails due to the ABA problem.

Theorem (ABA Defeats Verification): Reading memory cannot distinguish "primary executed then value reset" from "primary never executed."

Proof. Consider CAS($0 \to 1$) where the initial value was 0.

History H_1: Primary Never Executed

- Memory state: $m[a] = 0$ (unchanged)
- Correct decision: Backup should execute

History H_2: Primary Executed, Then ABA Reset

- Primary executed: $0 \to 1$
- Concurrent process reset: $1 \to 0$
- Memory state: $m[a] = 0$ (same as H_1!)
- Correct decision: Backup should NOT execute

The backup reads $m[a] = 0$ in both cases. Any verification function $V : \text{Memory} \to \{\text{Execute}, \text{Skip}\}$ must return the same answer for both, but they require opposite decisions. □

1.8 Why This Is Fundamentally Impossible: The Consensus Hierarchy

The ABA problem is not a bug we can fix with cleverness. It reflects a **fundamental limit** from distributed computing theory: Herlihy's Consensus Hierarchy.

1.8.1 What Is the Consensus Hierarchy?

In 1991, Maurice Herlihy proved that synchronization primitives form a strict hierarchy based on their **consensus number**—the maximum number of processes that can reach agreement using only that primitive.

> **Definition (Consensus Number):** $\mathrm{CN}(X) = n$ means primitive X can solve wait-free consensus among n processes, but not among $n+1$ processes. $\mathrm{CN}(X) = \infty$ means X can solve consensus for any number of processes.

The hierarchy is **strict**: a primitive with $\mathrm{CN} = k$ **cannot implement** any primitive with $\mathrm{CN} > k$.

1.8.2 The Consensus Hierarchy

Primitive	**CN**	**Why**
Read/Write	1	Reads are invisible; writes return no info
FADD	2	Sum is commutative: $\delta_0 + \delta_1 = \delta_1 + \delta_0$
CAS	∞	First CAS wins; all observe the winner

Table 1: Herlihy's Consensus Hierarchy

Key Insight: Each consensus number is **derived** from the primitive's semantics, not arbitrarily assigned:

- **Register CN = 1**: Two processes running solo see the same initial state (empty memory). They must decide differently but observe identically.
- **FADD CN = 2**: Addition is commutative. Process 2 sees $\delta_0 + \delta_1$ whether execution order is $[0, 1, 2]$ or $[1, 0, 2]$—same sum, different winners.
- **CAS CN = ∞**: The first CAS to a sentinel wins, and everyone reads the winner's value. Different winners → different observations → always distinguishable.

1.8.3 Failover Is 2-Process Consensus

The failover decision is **structurally equivalent** to 2-process consensus:

2-Process Consensus	Failover Decision
Two processes P_0, P_1	Two histories H_1, H_2
Each proposes a value	Each requires a decision
P_0 proposes “execute”	H_1 (not executed) requires Execute
P_1 proposes “skip”	H_2 (already executed) requires Skip
Must agree on one value	Must make correct choice
Winner’s value wins	Actual history determines correctness

Table 2: Structural isomorphism between failover and 2-consensus

Theorem (Reduction: Failover Solver ⇒ 2-Consensus): If a correct failover solver F exists, we can solve 2-process consensus:

1. P_0 and P_1 each write their input to `proposed[i]`
2. Both call $F(m)$ where m is the (ABA-ambiguous) memory state
3. If $F(m) = $ Execute: decide `proposed[0]`
4. If $F(m) = $ Skip: decide `proposed[1]`

This satisfies consensus: F determines a unique winner, both processes agree.

1.8.4 Why Read-Only Verification Fails

The backup NIC can only **read** memory to determine if the primary executed. But:

The Consensus Barrier

Failover requires solving 2-process consensus (CN ≥ 2).

Read-only verification has CN $= 1$.

By Herlihy’s impossibility theorem: CN $= 1$ primitives **cannot** solve 2-consensus.

Therefore: **transparent failover for atomics is impossible**.

This is not a limitation of our specific approach—it is a **mathematical impossibility**. No algorithm using only reads can solve this problem, because the consensus hierarchy is a fundamental law of distributed computing.

Theorem (Main Impossibility Result): Transparent failover for atomic operations is impossible because:
1. Failover requires solving 2-process consensus (distinguishing H_1 from H_2)
2. Transparency limits verification to read-only operations
3. CN(Read) $= 1 < 2$
4. By Herlihy’s hierarchy, CN $= 1$ primitives cannot solve 2-consensus

1.9 Summary: What Can and Cannot Be Supported

```
                 ┌──────────────────────────┐
                 │      Primary NIC Fails    │
                 └─────────────┬─────────────┘
                               │
            ┌──────────────────┴──────────────────┐
            │                                     │
            ▼                                     ▼
┌─────────────────┐                     ┌─────────────────┐
│ Idempotent Ops  │                     │ Atomic Ops      │
│ (Read, Write)   │                     │ (FADD, CAS)     │
└────────┬────────┘                     └────────┬────────┘
         │                                       │
         │ Retry is always safe                  │ Retry may corrupt
         │                                       │
         ▼                                       ▼
┌─────────────────┐                     ┌─────────────────┐
│ ✓ SUPPORTED     │                     │ ✗ NOT SUPPORTED │
└─────────────────┘                     └─────────────────┘
```

Listing 2: Transparent failover support depends on operation idempotency

Theorem	What It Shows
1	Sender cannot distinguish "primary executed" from "primary failed before executing"
2	For atomic operations, incorrect retry corrupts state (non-idempotent)
3	Backup cannot determine correct action by reading memory (ABA problem)

Transparent NIC failover cannot support RDMA atomic operations.

FADD and CAS require knowing whether the primary executed—information that is lost when the NIC fails and cannot be recovered by reading memory.

1.10 Implications

Operations That CAN Be Supported:

- Two-sided operations (Send/Recv)—receiver participates explicitly
- RDMA Read (idempotent—reading twice is harmless)
- RDMA Write, **only if** the receiver does not depend on knowing whether the write executed (e.g., no memory ordering for synchronization)
- Any operation where correctness does not depend on execution knowledge

Operations That CANNOT Be Supported Transparently:

- Any operation where the receiver depends on memory ordering (requires knowing if operation executed → 2-consensus)
- FADD (non-idempotent: retry corrupts state)
- CAS (ABA problem: retry can succeed twice)
- Any read-modify-write atomic

Workarounds (Violate Transparency):

- Receiver-side operation logs with deduplication
- Unique operation IDs tracked by receiver
- Application-level acknowledgments
- Two-phase commit protocols

The fundamental impossibility is that determining whether an operation executed requires solving 2-consensus. For truly idempotent operations where correctness does not depend on this knowledge, transparent failover works. For operations with ordering dependencies or non-idempotent semantics, it is impossible.

1.11 Rocq Formalization

All theorems are mechanically verified in Rocq 9.0.

Concept	Module	Key Theorems
Sender view	`Core/Traces.v`	`sender_view`, `SenderObs`
Transparent overlay	`Core/Properties.v`	`TransparentOverlay`
Indistinguishability	`Theorem1/Impossibility.v`	`sender_views_equal`, `impossibility_safe_retransmission`
FADD non-idempotent	`Theorem2/Atomics.v`	`fadd_non_idempotent`
CAS double success	`Theorem2/CAS.v`	`cas_double_success`
ABA problem	`Theorem3/FailoverConsensus.v`	`H0_H1_same_memory`
CN = 1 insufficient	`Theorem3/ConsensusNumber.v`	`readwrite_2consensus_impossible_same_proto`
Atomic failover impossible	`Theorem3/Hierarchy.v`	`transparent_cas_failover_impossible`

1.11.1 Key Theorems

Theorem 1 — Sender observations are identical for both histories:

```
Lemma sender_views_equal :
  sender_view T1_concrete = sender_view T2_concrete.
```

Theorem 2 — FADD is non-idempotent:

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

Theorem 3 — No correct decision function exists for atomics:

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

Main Result — Transparent failover cannot support atomic operations:

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