

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 $m : \text{Addr} \rightarrow \text{Val}$ 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 (`true`, v_{old})
3. Else: memory unchanged, returns (`false`, v_{old})

#rocq_name

Operations.v:29-30

Write execution semantics.

```
Definition exec_write (m : Memory) (a : Addr) (v : Val) : Memory * OpResult :=  
  (mem_write m a v, ResWriteAck).
```

exec_{Write(m,a,v)} writes value v to address a and returns **ResWriteAck**. No informational return value.

#rocq_name

Operations.v:33-34

Read execution semantics.

```
Definition exec_read (m : Memory) (a : Addr) : Memory * OpResult :=  
  (m, ResReadVal (mem_read m a)).
```

exec_{Read(m,a)} returns the current value $m(a)$ without modifying memory.

#rocq_name

Operations.v:51-57

General operation dispatch.

```
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.
```

exec_{op} dispatches to the appropriate execution function based on the operation type.

#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 $\delta \neq 0$.

```
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 $\delta \neq 0$: 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 $\text{EvTimeout}(\text{op}) \in \mathcal{T}$.

#rocq_name

Traces.v:80-89

Memory state after executing events.

```
Fixpoint memory_after_events (m : Memory) (events : list Event) : Memory :=
  match events with
  | [] => m
  | EvExecute op _ :: rest =>
    let (m', _) := exec_op m op in
    memory_after_events m' rest
  | EvAppReuse a v :: rest =>
    memory_after_events (mem_write m a v) rest
  | _ :: rest => memory_after_events m rest
  end.
```

Compute the memory state after a sequence of events. Only `EvExecute` (applying `exec_op`) and `EvAppReuse` (writing a new value) modify memory. Other events (sends, receives, losses, timeouts) leave memory unchanged.

#rocq_name

Traces.v:92-93

Memory state at end of trace.

```
Definition final_memory (t : Trace) : Memory :=
  memory_after_events init_memory t.
```

The *final memory* of a trace is the result of executing all events starting from `init_memory`.

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

#rocq_name

Properties.v:109

An overlay decision function.

```
Definition Overlay := Trace -> Op -> bool.
```

An *overlay* is a function $O : \mathcal{T} \times \text{Op} \rightarrow \{\text{retransmit}, \text{skip}\}$ that can observe the full trace to make retransmission decisions.

#rocq_name

Properties.v:114-117

Transparency constraint on overlays.

```
Definition IsTransparent (overlay : Overlay) : Prop :=
  forall t1 t2 op,
    sender_view t1 = sender_view t2 ->
    overlay t1 op = overlay t2 op.
```

An overlay is *transparent* if its decision depends **only** on the sender view: whenever two traces have identical sender views ($\sigma(\mathcal{T}_1) = \sigma(\mathcal{T}_2)$), the overlay must make the same retransmission decision. This is the key constraint — the overlay cannot use information beyond what the sender observes.

#rocq_name

Properties.v:120-123

A transparent failover mechanism.

```
Record TransparentOverlay := {
  decide_retransmit : Overlay;
  transparency : IsTransparent decide_retransmit;
}.
```

A *transparent overlay* bundles an overlay function with a proof of its transparency. The `transparency` field witnesses that the overlay's decisions depend only on `sender_view`.

#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.
```

`count(\mathcal{T} , 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 Protocol Parameters (**Theorem1/Indistinguishability.v**)

#rocq_name

Indistinguishability.v:22

Data write operation constructor.

```
Definition W_D (v : Val) : Op := OpWrite A_data v .
```

$W_{D(v)}$: an RDMA Write of value v to the data address A_data .

#rocq_name

Indistinguishability.v:25

Flag write operation.

```
Definition W_F : Op := OpWrite A_flag 1 .
```

W_F : an RDMA Write of value 1 to the flag address A_flag , signaling data is ready.

3.2 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**.

3.3 Main Impossibility (Theorem1/Impossibility.v)

#rocq_name

Impossibility.v:65-71

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) t op = false.
```

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

Note: `decide_retransmit` takes the full trace `t`, but the `transparency` proof ensures the decision depends only on `sender_view t`.

#rocq_name

Impossibility.v:76-83

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) 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:48-52

Transparency constraint on overlay.

```
Definition Transparent (overlay : TransparentOverlay) : Prop :=
  IsTransparent overlay.(decide_retransmit).
```

Transparency: if two traces have identical sender views, the overlay must make the same retransmission decision. This is now defined as `IsTransparent` applied to the overlay's decision function, ensuring the overlay cannot use information beyond `sender_view`.

#rocq_name

Impossibility.v:27-29

No application-level ACKs.

```
Definition SilentReceiver : Prop :=
  forall t : Trace, forall e : Event,
  In e t -> ~ AppAckEvent e.
```

Silent receiver: the protocol does not generate application-level acknowledgments. The sender learns nothing beyond transport-level signals.

#rocq_name

Impossibility.v:42-45

Application may reuse memory after consuming.

```
Definition MemoryReuseAllowed : Prop :=
  forall V1 V_new, exists t,
  In (EvAppConsume A_data V1) t /\ In (EvAppReuse A_data V_new) t.
```

Memory reuse: after consuming data, the application may immediately overwrite the buffer with new data V' .

#rocq_name

Impossibility.v:112-115

Concrete packet-loss trace.

```
Definition T1_concrete : Trace :=
[ EvSend op; EvPacketLost op; EvTimeout op ].
```

\mathcal{T}_1 : sender sends `op`, packet is lost, sender times out. Operation **not executed**.

#rocq_name

Impossibility.v:124-130

Concrete ACK-loss trace with memory reuse.

```
Definition T2_concrete : Trace :=
[ EvSend op; EvReceive op; EvExecute op ResWriteAck;
EvAppReuse A_data V_new; EvAckLost op; EvTimeout op ].
```

\mathcal{T}_2 : sender sends `op`, receiver executes it, app reuses memory, ACK lost, sender times out. Operation **was executed**.

#rocq_name

Impossibility.v:167-170

T_1 contains a send event.

```
Lemma T1_has_send : In (EvSend op) T1_concrete.
```

\mathcal{T}_1 contains `EvSend op`.

#rocq_name

Impossibility.v:140-149

T_1 does not execute the operation.

```
Lemma T1_not_executed : ~ op_executed T1_concrete op.
```

\mathcal{T}_1 has no `EvExecute` event for `op` — the operation was **not** executed (packet was lost before reaching receiver).

#rocq_name

Impossibility.v:160-164

T_1 contains a timeout.

```
Lemma T1_has_timeout : sender_saw_timeout T1_concrete op.
```

\mathcal{T}_1 contains `EvTimeout op` — the sender observed a timeout.

#rocq_name

Impossibility.v:152-157

T_2 executes the operation.

```
Lemma T2_executed : op_executed T2_concrete op.
```

\mathcal{T}_2 contains `EvExecute op ResWriteAck` — the operation **was** executed at the receiver.

#rocq_name

Impossibility.v:173-177

T_2 contains memory reuse.

```
Lemma T2_has_reuse : In (EvAppReuse A_data V_new) T2_concrete.
```

\mathcal{T}_2 contains `EvAppReuse A_data V_new` — the application has reused the memory buffer with new data V' .

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

Line-by-line proof annotation:

unfold T1_concrete, T2_concrete.	Initial state — Context: (section variables V1, V_new : Val, op := OpWrite A_data V1). Goal: sender_view T1_concrete = sender_view T2_concrete.
	Inline the definitions of both concrete traces. Goal becomes: sender_view [EvSend op; EvPacketLost op; EvTimeout op] = sender_view [EvSend op; EvReceive op; EvExecute op ResWriteAck; EvAppReuse A_data V_new; EvAckLost op; EvTimeout op].
simpl.	Compute sender_view on each list: the fixpoint filters out non-sender events (EvPacketLost, EvReceive, EvExecute, EvAppReuse, EvAckLost), keeping only EvSend → ObsSent and EvTimeout → ObsTimeout. Goal becomes: [ObsSent op; ObsTimeout op] = [ObsSent op; ObsTimeout op].
reflexivity.	The two sides are syntactically identical. Rocq's definitional equality closes the goal. <i>No goals remaining.</i>

#rocq_name

Impossibility.v:175-229

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

```
Theorem impossibility_safe_retransmission :
  forall overlay : TransparentOverlay,
    Transparent overlay ->
    SilentReceiver -> MemoryReuseAllowed ->
    ~ (ProvidesSafety overlay /\ ProvidesLiveness overlay).
Proof.
  intros overlay Htrans Hsilent Hreuse [Hsafe Hlive].
  pose (V1 := 1). pose (V_new := 2).
  pose (the_op := OpWrite A_data V1).
  pose (t1 := T1_concrete V1). pose (t2 := T2_concrete V1 V_new).
  assert (Hlive_T1 : overlay.(decide_retransmit) t1 the_op = true).
  { apply (Hlive ...). apply T1_has_send. apply T1_not_executed. apply T1_has_timeout. }
  assert (Hsafe_T2 : overlay.(decide_retransmit) t2 the_op = false).
  { apply (Hsafe ...). apply T2_has_reuse. apply T2_executed. }
  assert (Hviews_eq : sender_view t1 = sender_view t2).
  { apply sender_views_equal. }
  assert (Hdec_eq : overlay.(decide_retransmit) t1 the_op =
    overlay.(decide_retransmit) t2 the_op).
  { apply Htrans. exact Hviews_eq. }
  rewrite Hlive_T1 in Hdec_eq. rewrite Hsafe_T2 in Hdec_eq.
  discriminate.
Qed.
```

Theorem 1 (Impossibility of Safe Retransmission): Under transparency, no overlay can provide both safety and liveness.

Line-by-line proof annotation:

```
intros overlay Htrans Hsilent Hreuse [Hsafe Hlive].
```

Initial state — Goal: forall overlay, Transparent overlay -> SilentReceiver -> ... -> ~ (ProvidesSafety overlay /\ ProvidesLiveness overlay).

Introduce all hypotheses. The ~ unfolds to ... -> False. The final [Hsafe Hlive] destructs the conjunction. **Context:** overlay : TransparentOverlay, Htrans : Transparent overlay, Hsilent : SilentReceiver, Hreuse : MemoryReuseAllowed, Hsafe : ProvidesSafety overlay, Hlive : ProvidesLiveness overlay. **Goal:** False.

pose (V1 := 1). pose (V_new := 2).	Bind concrete values. <i>Context adds:</i> V1 := 1 : nat, V_new := 2 : nat. <i>Goal:</i> False (unchanged).
pose (the_op := OpWrite A_data V1).	<i>Context adds:</i> the_op := OpWrite A_data 1 : Op. <i>Goal:</i> False.
pose (t1 := T1_concrete V1).	<i>Context adds:</i> t1 := T1_concrete 1 : Trace (the packet-loss trace). <i>Goal:</i> False.
pose (t2 := T2_concrete V1 V_new).	<i>Context adds:</i> t2 := T2_concrete 1 2 : Trace (the ACK-loss + reuse trace). <i>Goal:</i> False.
assert (Hlive_T1 : ... = true). { apply Hlive ... }	Establishes Hlive_T1 : decide_retransmit overlay t1 the_op = true. The sub-proof applies Hlive (liveness) to \mathcal{T}_1 : sent (T1_has_send), not executed (T1_not_executed), timed out (T1_has_timeout), so the overlay must retransmit. Note: decide_retransmit now takes the trace directly. <i>Goal:</i> False (unchanged after assert is discharged).
assert (Hsafe_T2 : ... = false). { apply Hsafe ... }	Establishes Hsafe_T2 : decide_retransmit overlay t2 the_op = false. The sub-proof applies Hsafe (safety) to \mathcal{T}_2 : memory reused (T2_has_reuse), operation executed (T2_executed), so the overlay must not retransmit. <i>Goal:</i> False.
assert (Hviews_eq : sender_view t1 = sender_view t2).	Establishes Hviews_eq : sender_view t1 = sender_view t2 via sender_views_equal. Both traces yield [ObsSent op; ObsTimeout op]. <i>Goal:</i> False.
assert (Hdec_eq : ...). { apply Htrans. exact Hviews_eq. }	Establishes Hdec_eq : decide_retransmit overlay t1 the_op = decide_retransmit overlay t2 the_op by applying Htrans (transparency): Htrans is IsTransparent decide_retransmit, so identical sender views \Rightarrow identical decisions even though the traces differ. <i>Goal:</i> False.
rewrite Hlive_T1 in Hdec_eq.	Substitute the LHS of Hdec_eq using Hlive_T1. <i>Hdec_eq becomes:</i> true = decide_retransmit overlay t2 the_op. <i>Goal:</i> False.
rewrite Hsafe_T2 in Hdec_eq.	Substitute the RHS using Hsafe_T2. <i>Hdec_eq becomes:</i> true = false. <i>Goal:</i> False.
discriminate.	true and false are distinct constructors of bool. discriminate derives False from true = false. No goals remaining.

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 $\delta > 0$, $FADD(a, \delta)$ 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 $\delta > 0$: $v + \delta \neq v + 2\delta$. \square

#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 \vee 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 $FADD(a, \delta)$ 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. $S.\text{CAS}(0 \rightarrow 1)$ succeeds, $m(a) = 1$
2. Concurrent $P.\text{CAS}(1 \rightarrow 0)$ succeeds, $m(a) = 0$
3. S retries $\text{CAS}(0 \rightarrow 1)$: **succeeds again!**

Single logical CAS executed twice. P 's modification silently overwritten.

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

#rocq_name

Atomics.v:328-355

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 -> overlay.(decide_retransmit) 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). Note: `decide_retransmit` now takes the trace directly; the transparency proof (in the `TransparentOverlay` record) ensures decisions depend only on `sender_view`.

#rocq_name

Atomics.v:358-387

Corollary for atomic operations.

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

Corollary: No transparent overlay supports FADD (with $\delta > 0$) or successful CAS (with $\text{exp} \neq \text{new}$). The inline safety condition requires that the overlay not retransmit when the operation has been executed.

5 Theorem 3: Consensus Hierarchy (Theorem3/)

5.1 Consensus Framework (Theorem3/ConsensusNumber.v)

#rocq_name

ConsensusNumber.v:85-89

Processes that executed before a given process.

```
Fixpoint before_process (exec : list nat) (i : nat) : list nat :=
  match exec with
  | [] => []
  | x :: xs => if Nat.eqb x i then [] else x :: before_process xs i
end.
```

before_process exec i returns the list of processes that executed before process i in execution order exec. Scans the list until finding i , collecting all predecessors.

#rocq_name

ConsensusNumber.v:91

First process to execute.

```
Definition winner (exec : list nat) : nat := hd 0 exec.
```

The *winner* of an execution is the first process in the execution order.

#rocq_name

ConsensusNumber.v:667-668

Prior writers in an execution.

```
Definition writers_before (exec : list nat) (i : nat) : list nat :=
  before_process exec i.
```

writers_before exec i = before_process exec i: the processes that wrote to memory before process i executed. Determines the memory state that i observes.

#rocq_name

ConsensusNumber.v:731-732

Solo execution of process 0.

```
Definition solo_0 : list nat := [0].
Definition solo_1 : list nat := [1].
```

Solo executions: solo_0 = [0] and solo_1 = [1]. In a solo execution, only one process runs. Critically, writers_before solo_0 0 = [] and writers_before solo_1 1 = [] — both solo processes see empty (initial) memory.

#rocq_name

ConsensusNumber.v:400-404

Consensus number type.

```
Definition ConsensusNum := option nat. (* None = infinity *)
Definition cn_one : ConsensusNum := Some 1.
Definition cn_two : ConsensusNum := Some 2.
Definition cn_infinity : ConsensusNum := None.
```

Consensus number is modeled as option nat, where None represents ∞ . Constants: CN = 1, CN = 2, CN = ∞ .

#rocq_name

ConsensusNumber.v:414-419

Strict ordering on consensus numbers.

```

Definition cn_lt (c1 c2 : ConsensusNum) : Prop :=
  match c1, c2 with
  | Some n1, Some n2 => n1 < n2
  | Some _, None => True
  | None, _ => False
end.

```

`cnlt(c1,c2)`: strict ordering. Any finite CN is less than ∞ ; ∞ is not less than anything.

#rocq_name

ConsensusNumber.v:1146-1149

RDMA consensus number assignments.

```

Definition rdma_read_cn : ConsensusNum := cn_one. (* CN(Read) = 1 *)
Definition rdma_write_cn : ConsensusNum := cn_one. (* CN(Write) = 1 *)
Definition rdma_fadd_cn : ConsensusNum := cn_two. (* CN(FADD) = 2 *)
Definition rdma_cas_cn : ConsensusNum := cn_infinity. (* CN(CAS) = inf *)

```

RDMA consensus number assignments: CN(Read) = CN(Write) = 1, CN(FADD) = 2, CN(CAS) = ∞ .

#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:202-210

Prior processes as a multiset.

```

Definition procs_before_as_set (exec : list nat) (i : nat) : list nat :=
  before_process exec i.

Definition same_elements (l1 l2 : list nat) : Prop :=
  forall x, count_occ Nat.eq_dec l1 x = count_occ Nat.eq_dec l2 x.

```

`procs_before_as_set exec i` returns the multiset of processes before i . `same_elements` holds when two lists have identical element counts — i.e., they are permutations of each other.

#rocq_name

ConsensusNumber.v:221-222

Critical executions for FADD CN=2 proof.

```
Definition exec_012 : list nat := [0; 1; 2].
Definition exec_102 : list nat := [1; 0; 2].
```

Two 3-process executions that differ only in the order of P0 and P1. Process P2 executes last in both. Since `procs_before_as_set exec_012 2 = {0,1} = procs_before_as_set exec_102 2`, P2 sees the same FADD observation in both (commutativity of addition).

#rocq_name

ConsensusNumber.v:238

Identity input function.

```
Definition inp (i : nat) : nat := i.
```

Each process i 's input is i itself. So the validity requirement is: the decision must equal the winner's ID.

#rocq_name

ConsensusNumber.v:225-235

P2 sees same prior process set in both executions.

```
Lemma p2_same_prior_set :
  same_elements (procs_before_as_set exec_012 2) (procs_before_as_set exec_102 2).
```

Lemma: `procs_before_as_set exec_012 2 = {0,1}` and `procs_before_as_set exec_102 2 = {1,0}`. These are the same multiset (just reordered), so `same_elements` holds. This is the key commutativity fact.

#rocq_name

ConsensusNumber.v:249-258

P2 sees same observation in both FADD executions.

```
Theorem fadd_p2_indistinguishable_general :
  forall obs : list nat -> nat -> nat,
    valid_fadd_observation obs ->
    obs exec_012 2 = obs exec_102 2.
```

Lemma: For any valid FADD observation function, process 2 sees the same value in `exec_012` and `exec_102`. This follows from `valid_fadd_observation` and `p2_same_prior_set`.

#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.
Proof.
  intros obs Hvalid [decide [Hval0 Hval1]].
  assert (Hsame : obs solo_0 0 = obs solo_1 1).
```

```

{ unfold valid_rw_observation in Hvalid. apply Hvalid. reflexivity. }
rewrite Hsame in Hval0.
rewrite Hval0 in Hval1.
discriminate.
Qed.

```

Theorem (Register CN = 1): No read/write protocol can solve 2-consensus.

Line-by-line proof annotation:

intros obs Hvalid [decide [Hval0 Hval1]].	Initial state — <i>Goal: forall obs, valid_rw_observation obs -> ~ exists decide, decide (obs solo_0 0) = 0 /\ \ decide (obs solo_1 1) = 1.</i> The \sim unfolds to $\dots \rightarrow \text{False}$. Introduce <i>obs</i> , <i>Hvalid</i> , then destruct the existential and conjunction. <i>Context: obs : list nat -> nat -> nat, Hvalid : valid_rw_observation obs, decide : nat -> nat, Hval0 : decide (obs solo_0 0) = 0, Hval1 : decide (obs solo_1 1) = 1. Goal: False.</i>
assert (Hsame : obs solo_0 0 = obs solo_1 1).	Begin sub-proof for the claim that both solo executions produce the same observation. <i>Sub-goal: obs solo_0 0 = obs solo_1 1.</i>
{ unfold valid_rw_observation in Hvalid.	Expand <i>valid_rw_observation</i> in <i>Hvalid</i> . <i>Hvalid becomes: forall exec1 exec2 i j, writers_before exec1 i = writers_before exec2 j -> obs exec1 i = obs exec2 j.</i> <i>Sub-goal: obs solo_0 0 = obs solo_1 1 (unchanged).</i>
apply Hvalid.	Apply <i>Hvalid</i> with <i>exec1 := solo_0, i := 0, exec2 := solo_1, j := 1</i> . <i>Sub-goal becomes: writers_before solo_0 0 = writers_before solo_1 1.</i>
reflexivity. }	<i>writers_before solo_0 0</i> computes to <i>before_process [0] 0 = []</i> (no writers precede P0 in [0]). <i>writers_before solo_1 1</i> computes to <i>before_process [1] 1 = []</i> (no writers precede P1 in [1]). Both are $[]$, so <i>reflexivity</i> closes the sub-goal. <i>This is the fundamental insight: both solo processes see identical (empty) memory.</i> <i>Context adds: Hsame : obs solo_0 0 = obs solo_1 1. Goal: False.</i>
rewrite Hsame in Hval0.	Rewrite <i>obs solo_0 0</i> to <i>obs solo_1 1</i> in <i>Hval0</i> using <i>Hsame</i> . <i>Hval0 becomes: decide (obs solo_1 1) = 0.</i> <i>Goal: False.</i>
rewrite Hval0 in Hval1.	Both <i>Hval0</i> and <i>Hval1</i> now refer to <i>decide (obs solo_1 1)</i> . Substitute the value from <i>Hval0</i> . <i>Hval1 becomes: 0 = 1.</i> <i>Goal: False.</i>
discriminate.	0 and 1 are distinct constructors (0 vs S 0). <i>discriminate</i> derives <i>False</i> . <i>No goals remaining.</i>

#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).

```

Proof.

```

intros obs Hvalid [decide [Hval012 Hval102]].
assert (Hsame : obs exec_012 2 = obs exec_102 2).
{ exact (fadd_p2_indistinguishable_general obs Hvalid). }
rewrite Hsame in Hval012.
simpl in Hval012, Hval102.
rewrite Hval012 in Hval102.
discriminate.
Qed.

```

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

Line-by-line proof annotation:

intros obs Hvalid [decide [Hval012 Hval102]].	Initial state — <i>Goal: forall obs, valid_fadd_observation obs -> ~ exists decide, decide (obs exec_012 2) = inp (winner exec_012) /\ \ decide (obs exec_102 2) = inp (winner</i>
---	--

	exec_102).
assert (Hsame : obs exec_012 2 = obs exec_102 2).	Introduce and destruct. <i>Context</i> : obs : list nat -> nat -> nat, Hvalid : valid_fadd_observation obs, decide : nat -> nat, Hval012 : decide (obs exec_012 2) = inp (winner exec_012), Hval102 : decide (obs exec_102 2) = inp (winner exec_102). <i>Goal</i> : False.
{ exact (fadd_p2_indistinguishable_general obs Hvalid). }	Claim: process 2 sees the same observation in both executions. <i>Sub-goal</i> : obs exec_012 2 = obs exec_102 2.
rewrite Hsame in Hval012.	Apply the indistinguishability lemma directly. It uses valid_fadd_observation and the fact that procs_before_as_set exec_012 2 = {0,1} = procs_before_as_set exec_102 2 (same set, just reordered). Since FADD addition is commutative ($\delta_0 + \delta_1 = \delta_1 + \delta_0$), observations must be equal. <i>Context adds</i> : Hsame : obs exec_012 2 = obs exec_102 2. <i>Goal</i> : False.
simpl in Hval012, Hval102.	Rewrite using Hsame. <i>Hval012 becomes</i> : decide (obs exec_102 2) = inp (winner exec_012). <i>Goal</i> : False.
rewrite Hval012 in Hval102.	Compute winner exec_012 = hd 0 [0;1;2] = 0, inp 0 = 0; winner exec_102 = hd 0 [1;0;2] = 1, inp 1 = 1. <i>Hval102 becomes</i> : decide (obs exec_102 2) = 0. <i>Hval012 becomes</i> : decide (obs exec_102 2) = 1. <i>Goal</i> : False.
discriminate.	Both hypotheses refer to decide (obs exec_102 2). Substitute. <i>Hval102 becomes</i> : 0 = 1. <i>Goal</i> : False. $0 \neq 1$. discriminate derives False. No goals remaining.

#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 \rightarrow different observations \rightarrow always distinguishable. \square

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:45-47

What actually happened.

```
Inductive PastKnowledge :=
| PastExecuted (* CAS was executed at receiver *)
| PastNotExecuted. (* CAS was not executed *)
```

The past process knows the ground truth: was the CAS executed or not?

#rocq_name

FailoverConsensus.v:50-52

What the future process can see.

```
Inductive FutureObservation :=
| FutureSeesTimeout      (* Sender observed timeout *)
| FutureSeesCompletion.  (* Sender received completion ACK *)
```

The *future process* observes only transport-level signals. Critically, `FutureSeesTimeout` occurs in both the packet-loss and ACK-loss scenarios.

#rocq_name

FailoverConsensus.v:206-213

The two failover scenarios.

```
Definition scenario1_future : FutureObservation := FutureSeesTimeout.
Definition scenario1_correct : FailoverDecision := Abort.
Definition scenario2_future : FutureObservation := FutureSeesTimeout.
Definition scenario2_correct : FailoverDecision := Commit.
```

Scenario 1 (packet lost): future sees timeout, correct decision is Abort (retry). Scenario 2 (ACK lost): future sees timeout, correct decision is Commit (don't retry). **Same observation, different correct decisions.**

#rocq_name

FailoverConsensus.v:257

A function from memory to decision.

```
Definition VerificationMechanism := Memory -> bool.
(* Encoding: true = Commit, false = Abort *)
```

A *verification mechanism* is any function $V : \text{Memory} \rightarrow \text{bool}$ that inspects remote memory and returns a failover decision. Under transparency, V is limited to using reads.

#rocq_name

FailoverConsensus.v:261-263

What actually happened to the CAS.

```
Inductive History :=
| HistExecuted : Memory -> History      (* CAS was executed *)
| HistNotExecuted : Memory -> History.    (* CAS was not executed *)
```

A *history* records whether the CAS was executed, along with the memory state. Both variants carry a memory parameter because ABA can reset memory to any state.

#rocq_name

FailoverConsensus.v:265-269

Was the CAS executed?

```
Definition history_executed (h : History) : bool :=
match h with
| HistExecuted _ => true
| HistNotExecuted _ => false
end.
```

Extracts the execution status as a boolean: `true` for executed, `false` for not executed.

#rocq_name

FailoverConsensus.v:271-275

Memory state at end of history.

```
Definition final_memory (h : History) : Memory :=
  match h with
  | HistExecuted m => m
  | HistNotExecuted m => m
  end.
```

The final memory state. **Key ABA property:** both `HistExecuted m` and `HistNotExecuted m` yield the same memory m . This is the formal encoding of the ABA problem.

#rocq_name

FailoverConsensus.v:278-279

The correct failover decision for a history.

```
Definition correct_decision_for (h : History) : bool :=
  history_executed h.
```

The correct decision equals the execution status: `true` (Commit) if executed, `false` (Abort) if not. A correct verification mechanism must satisfy $V(m_{final}) = \text{correct}_{\text{decision}_{\text{for}(h)}}^*$ for all histories.

#rocq_name

FailoverConsensus.v:289-292

The two ABA-indistinguishable histories.

```
Definition H1 : History := HistExecuted init_mem. (* CAS executed, ABA reset *)
Definition H0 : History := HistNotExecuted init_mem. (* CAS not executed *)
```

H_1 (executed) and H_0 (not executed) both have `final_memory = init_mem` due to ABA. But `correct_decision_for H1 = true` while `correct_decision_for H0 = false`.

#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.
```

Proof.

```
intros [f [H1 H2]].
unfold scenario1_future, scenario2_future in *.
unfold scenario1_correct, scenario2_correct in *.
rewrite H1 in H2.
discriminate.
```

Qed.

Theorem: No deterministic function from sender observation to failover decision is correct.

Line-by-line proof annotation:

```
intros [f [H1 H2]].
```

Initial state — Goal: $\sim \exists f, f \text{ scenario1_future} = \text{scenario1_correct} \wedge \wedge f \text{ scenario2_future} = \text{scenario2_correct}$.
 The \sim unfolds to $\dots \rightarrow \text{False}$. Destruct the existential and conjunction.
Context: $f : \text{FutureObservation} \rightarrow \text{FailoverDecision}, H1 : f \text{ scenario1_future} = \text{scenario1_correct}, H2 : f \text{ scenario2_future} = \text{scenario2_correct}$. **Goal:** False .

```
unfold scenario1_future, scenario2_future in *.
```

Inline definitions: $\text{scenario1_future} := \text{FutureSeesTimeout}$,
 $\text{scenario2_future} := \text{FutureSeesTimeout}$. $H1$ becomes: $f \text{ FutureSeesTimeout} = \text{scenario1_correct}$. $H2$ becomes: $f \text{ FutureSeesTimeout} = \text{scenario2_correct}$. **Goal:** False .

```

unfold scenario1_correct, scenario2_correct in *.
  Inline: scenario1_correct := Abort, scenario2_correct := Commit. H1
  becomes: f FutureSeesTimeout = Abort. H2 becomes: f FutureSeesTimeout =
  Commit. Goal: False.

rewrite H1 in H2.
  Since f FutureSeesTimeout appears in both, substitute using H1. H2
  becomes: Abort = Commit. Goal: False.

discriminate.
  Abort and Commit are distinct constructors of FailoverDecision.
  discriminate derives False. No goals remaining.

```

#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 , $V(m_{\text{final}}) = \text{correct decision for } h$.

#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

Direct ABA Argument: No verification mechanism solves failover.

```

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

Proof.
  intros V Hsolves.
  unfold solves_failover in Hsolves.
  specialize (Hsolves H0) as HV0.
  specialize (Hsolves H1) as HV1.
  assert (Hmem_eq : final_memory H0 = final_memory H1).
  { apply H0_H1_same_memory. }
  rewrite Hmem_eq in HV0.
  rewrite HV0 in HV1.
  discriminate.
Qed.

```

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

Line-by-line proof annotation:

```
intros V Hsolves.
```

Initial state — *Context: init_mem : Memory* (section variable). *Goal: forall V : VerificationMechanism, ~ solves_failover V.*
The \sim unfolds to *solves_failover V -> False*. *Context adds: V : VerificationMechanism* (i.e., $\text{Memory} \rightarrow \text{bool}$), *Hsolves : solves_failover V*. *Goal: False*.

```
unfold solves_failover in Hsolves.
```

Expand definition. *Hsolves becomes: forall h : History, V (final_memory h) = correct_decision_for h*. *Goal: False*.

```
specialize (Hsolves H0) as HV0.
```

Instantiate *Hsolves* with $H0 := \text{HistNotExecuted init_mem}$. *Context adds: HV0 : V (final_memory H0) = correct_decision_for H0*.
Computationally: $\text{final_memory}(\text{HistNotExecuted init_mem}) = \text{init_mem}$ and $\text{correct_decision_for}(\text{HistNotExecuted } _) = \text{false}$, so this represents $V \text{ init_mem} = \text{false}$. *Goal: False*.

```
specialize (Hsolves H1) as HV1.
```

Instantiate with $H1 := \text{HistExecuted init_mem}$. *Context adds: HV1 : V (final_memory H1) = correct_decision_for H1*. Computationally:

	final_memory (HistExecuted init_mem) = init_mem and correct_decision_for (HistExecuted _) = true, so this represents V init_mem = true. <i>Goal: False.</i>
assert (Hmem_eq : final_memory H0 = final_memory H1). <i>Sub-goal: final_memory H0 = final_memory H1.</i>	
{ apply H0_H1_same_memory. }	Both final_memory (HistNotExecuted init_mem) and final_memory (HistExecuted init_mem) compute to init_mem. The lemma holds by reflexivity. <i>Context adds: Hmem_eq : final_memory H0 =</i> <i>final_memory H1. Goal: False.</i>
rewrite Hmem_eq in HV0.	Rewrite final_memory H0 to final_memory H1 in HV0. <i>HV0 becomes: V</i> <i>(final_memory H1) = correct_decision_for H0</i> (i.e., V (final_memory H1) = false). <i>Goal: False.</i>
rewrite HV0 in HV1.	Both HV0 and HV1 now refer to V (final_memory H1). Substitute. <i>HV1</i> <i>becomes: correct_decision_for H0 = correct_decision_for H1</i> , which computes to false = true. <i>Goal: False.</i>
discriminate.	false and true are distinct constructors of bool. discriminate derives False. <i>No goals remaining.</i>

#rocq_name

FailoverConsensus.v:677-703

Positive Reduction: Failover solver yields read-based 2-consensus protocol.

```

Lemma failover_solver_yields_2consensus :
  forall V : VerificationMechanism,
    solves_failover V ->
    exists 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.

Proof.
  intros V Hsolves.
  exists (fun _ _ => if V init_memory then 0 else 1).
  split.
  - unfold valid_rw_observation. intros. reflexivity.
  - exists (fun x => if Nat.eqb x 0 then 0 else 1).
    pose proof (Hsolves (HistExecuted init_memory)) as Hexec.
    pose proof (Hsolves (HistNotExecuted init_memory)) as Hnot.
    unfold final_memory, correct_decision_for, history_executed in *.
    rewrite Hexec in Hnot. discriminate.
Qed.

```

Lemma (Positive Reduction): A correct failover solver yields a read-based observation function (satisfying `valid_rw_observation`) and decision function satisfying solo validity for 2-consensus.

Line-by-line proof annotation:

```

intros V Hsolves.

exists (fun _ _ => if V init_memory then 0 else 1).

exists (fun _ _ => if V init_memory then 0 else 1).

```

Initial state — Goal: forall V, solves_failover V
-> exists obs, valid_rw_observation obs /\\
exists decide, decide (obs solo_0 0) = 0 /\\
decide (obs solo_1 1) = 1.
Context: V : VerificationMechanism (i.e., Memory ->
bool), Hsolves : solves_failover V. Goal: exists
obs, valid_rw_observation obs /\ exists decide,
decide (obs solo_0 0) = 0 /\ decide (obs solo_1
1) = 1.

Provide the observation function witness: a
constant function (ignores execution and process
ID), returning 0 if V `init_memory` = true, else 1. This
models the ABA constraint: a read-based mechanism
can only observe `init_memory` regardless of history.
Goal: valid_rw_observation (fun _ _ => ...) /\\
exists decide, decide (if V init_memory then 0
else 1) = 0 /\ decide (if V init_memory then 0
else 1) = 1.

split.	Split the conjunction. <i>Subgoal 1:</i> valid_rw_observation (fun _ _ => if V init_memory then 0 else 1). <i>Subgoal 2:</i> exists decide, decide (if V init_memory then 0 else 1) = 0 /\& decide (if V init_memory then 0 else 1) = 1.
- unfold valid_rw_observation. intros. reflexivity.	<i>Subgoal 1.</i> Unfold: goal becomes forall exec1 exec2 i j, writers_before exec1 i = writers_before exec2 j -> (if V init_memory then 0 else 1) = (if V init_memory then 0 else 1). After intros, both sides are identical regardless of the hypothesis. reflexivity closes it. <i>Subgoal 1 done.</i>
- exists (fun x => if Nat.eqb x 0 then 0 else 1).	<i>Subgoal 2.</i> Provide the decision function: maps 0 → 0, anything else → 1. <i>Goal:</i> (if Nat.eqb (if V init_memory then 0 else 1) 0 then 0 else 1) = 0 /\& (if Nat.eqb (if V init_memory then 0 else 1) 0 then 0 else 1) = 1.
pose proof (Hsolves (HistExecuted init_memory)) as Hexec.	<i>Context adds:</i> Hexec : V (final_memory (HistExecuted init_memory)) = correct_decision_for (HistExecuted init_memory). <i>Goal:</i> unchanged.
pose proof (Hsolves (HistNotExecuted init_memory)) as Hnot.	<i>Context adds:</i> Hnot : V (final_memory (HistNotExecuted init_memory)) = correct_decision_for (HistNotExecuted init_memory). <i>Goal:</i> unchanged.
unfold final_memory, correct_decision_for, history_executed in *.	Compute all definitions in context and goal. final_memory (HistExecuted m) = m, final_memory (HistNotExecuted m) = m, correct_decision_for h = history_executed h, history_executed (HistExecuted _) = true, history_executed (HistNotExecuted _) = false. <i>Hexec becomes:</i> V init_memory = true. <i>Hnot becomes:</i> V init_memory = false. The goal's if expressions also simplify according to V init_memory.
rewrite Hexec in Hnot.	Substitute V init_memory with true (from Hexec) in Hnot. <i>Hnot becomes:</i> true = false. <i>Goal:</i> unchanged.
discriminate.	true = false in context is a contradiction. Since the context is inconsistent, Rocq can prove any goal — including the remaining conjunction. <i>No goals remaining.</i> <i>Key insight:</i> solves_failover V is itself contradictory (V cannot return both true and false on the same input init_memory), so the existential witnesses need not “really” work. The proof constructs them purely to trigger the CN=1 impossibility.

#rocq_name

FailoverConsensus.v:715-726

THEOREM 3 Core: Failover impossible via Register CN=1.

```
Theorem failover_impossible_by_read_cn :
  forall V : VerificationMechanism,
  ~ solves_failover V.
Proof.
  intros V Hsolves.
  destruct (failover_solver_yields_2consensus V Hsolves)
  as [obs [Hvalid [decide [Hval0 Hval1]]]].
  apply (readwrite_2consensus_impossible_same_protocol obs Hvalid).
  exists decide. exact (conj Hval0 Hval1).
Qed.
```

Theorem 3 (Failover Impossible by Register CN=1): No verification mechanism can solve failover. This derives the impossibility FROM the CN=1 theorem, not as a separate ABA argument.

Line-by-line proof annotation:

intros V Hsolves.	Initial state — <i>Goal: forall V : VerificationMechanism, ~ solves_failover V.</i> The \sim unfolds to <i>solves_failover V -> False</i> . <i>Context: V : VerificationMechanism, Hsolves : solves_failover V. Goal: False.</i>
destruct (failover_solver_yields_2consensus V Hsolves)	Apply the positive reduction lemma to <i>V</i> and <i>Hsolves</i> . It returns a proof of <i>exists obs, valid_rw_observation obs /\ exists decide,</i> The <i>destruct</i> extracts the witnesses.
as [obs [Hvalid [decide [Hval0 Hval1]]]].	Destruct the nested existentials and conjunctions. <i>Context adds: obs : list nat -> nat -> nat, Hvalid : valid_rw_observation obs, decide : nat -> nat, Hval0 : decide (obs solo_0 0) = 0, Hval1 : decide (obs solo_1 1) = 1. Goal: False.</i>
apply (readwrite_2consensus_impossible_same_protocol obs Hvalid).	Apply the Register CN=1 impossibility theorem with <i>obs</i> and <i>Hvalid</i> . This theorem has conclusion $\sim \exists \text{ decide}, \text{ decide}(\text{obs solo}_0 0) = 0 \wedge \text{ decide}(\text{obs solo}_1 1) = 1$, which unfolds to <i>(exists decide, ... -> False)</i> . Applying it to the goal <i>False</i> changes the goal to the premise. <i>Goal becomes: exists decide, decide (obs solo_0 0) = 0 /\ decide (obs solo_1 1) = 1.</i>
exists decide.	Provide the <i>decide</i> witness from the destructed positive reduction. <i>Goal becomes: decide (obs solo_0 0) = 0 /\ decide (obs solo_1 1) = 1.</i>
exact (conj Hval0 Hval1).	<i>conj</i> constructs <i>A /\ B</i> from proofs of <i>A</i> and <i>B</i> . Provide <i>Hval0</i> and <i>Hval1</i> as the two conjuncts. <i>No goals remaining.</i> The full chain: <i>Hsolves</i> \rightarrow positive reduction yields <i>obs, decide, Hval0, Hval1</i> \rightarrow CN=1 theorem demands they cannot exist \rightarrow but they do \rightarrow <i>False</i> .

5.3 Main Result (Theorem3/Hierarchy.v)

#rocq_name

Hierarchy.v:31-40

Transparent failover mechanism record.

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

A *transparent failover* mechanism can only: (1) read remote memory, (2) cannot write additional metadata, (3) must decide based solely on read results. This captures the transparency constraint from SHIFT.

#rocq_name

Hierarchy.v:46-48

Reads are genuine memory reads.

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

The mechanism's read operation is a genuine memory read: *can_read_remote addr m = m(addr)*. No side channels or metadata.

#rocq_name

Hierarchy.v:65-67

Reliable CAS via transparent failover.

```
Definition provides_reliable_cas (tf : TransparentFailover) : Prop :=
  exists V : VerificationMechanism, solves_failover V.
```

A transparent failover mechanism *provides reliable CAS* if there exists a `VerificationMechanism` that solves failover for all histories. The main theorem proves this is impossible.

#rocq_name

Hierarchy.v:81-92

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.

Proof.
  intros tf Hreads Hno_meta.
  unfold provides_reliable_cas.
  intros [V Hsolves].
  exact (failover_impossible_by_read_cn V Hsolves).

Qed.
```

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

Line-by-line proof annotation:

intros tf Hreads Hno_meta.	Initial state — <i>Goal</i> : <code>forall tf, verification_via_reads tf -> tf.(no_metadata_writes) -> ~ provides_reliable_cas tf.</code> <i>Context</i> : <code>tf : TransparentFailover, Hreads : verification_via_reads tf, Hno_meta : tf.(no_metadata_writes)</code> . <i>Goal</i> : <code>~ provides_reliable_cas tf</code> .
unfold provides_reliable_cas.	Expand the definition. <i>Goal becomes</i> : <code>~ (exists V : VerificationMechanism, solves_failover V)</code> , i.e., <code>(exists V, solves_failover V) -> False</code> .
intros [V Hsolves].	Assume the existential for contradiction and destruct it. <i>Context adds</i> : <code>V : VerificationMechanism</code> (i.e., <code>Memory -> bool</code>), <code>Hsolves : solves_failover V</code> . <i>Goal</i> : <code>False</code> .
exact (failover_impossible_by_read_cn V Hsolves).	<code>failover_impossible_by_read_cn</code> has type <code>forall V, solves_failover V -> False</code> . Applying it to <code>V</code> and <code>Hsolves</code> yields a proof of <code>False</code> directly. <i>No goals remaining</i> . This single term encapsulates the full reduction chain: <code>Hsolves -> positive reduction -> CN=1 barrier -> contradiction</code> . Note: <code>Hreads</code> and <code>Hno_meta</code> are not used in the proof term — they justify why the mechanism is limited to reads, but <code>failover_impossible_by_read_cn</code> holds for any <code>VerificationMechanism</code> . The transparency constraints narrow the interface; the CN theorem shows even that narrowed interface is insufficient.

#rocq_name

Hierarchy.v:136-137

Reads have consensus number 1.

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

$\text{CN}(\text{Read}) = 1$, 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	$\text{CN}(\text{Read}) = 1$: solo indistinguishable	Herlihy
failover_solver_yields_2consensus	FailoverConsensus.v:680	Solver \rightarrow read-based 2-consensus	Reduction
failover_impossible_by_read_cn	FailoverConsensus.v:725	Thm 3 Core: No V solves failover (via $\text{CN}=1$)	CN chain
transparent_cas_failover_impossible	Hierarchy.v:78	Thm 3 Main: Transparent failover impossible	CN hierarchy

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