Distinguished Paper Award

OOPSLA Artifact Evaluated AEC

Hybrid STM/HTM for Nested Transactions on OpenJDK

Keith Chapman
Purdue U

Tony Hosking
ANU/Data61, Purdue U

Eliot Moss
UMass

Motivation

STM has been around for ages

- But STM is slow 🙁

Commodity hardware for transactions available now

- But HTM approaches are only **best effort**

Out goal: Accelerate STM with HTM when possible

Transactions are Good

They deal with concurrency

- Atomic transactions avoid problems with locks
    - Deadlock, wrong lock, priority inversion, etc.

Transactions are Good

They deal with concurrency

- Atomic transactions avoid problems with locks
  - Deadlock, wrong lock, priority inversion, etc.

They handle recovery

- Retry in case of conflict
- Cleanup in face of exceptions/errors

Transactions are Good

They deal with concurrency

- Atomic transactions avoid problems with locks
  - Deadlock, wrong lock, priority inversion, etc.

They handle recovery

- Retry in case of conflict
- Cleanup in face of exceptions/errors

More practical for ordinary programmers than locks for robust concurrent systems

Semantics of Transactions

Offer A, C, I of database ACID properties:

- **Atomicity**: all or nothing
- **Consistency**: single global order
- **Isolation**: intermediate states invisible

In sum, **serialisability**, in face of concurrent execution and transaction failures.

Can be provided by *Transactional Memory*

- Hardware, software, or hybrid

Simple Transactions for Java

Following Harris and Fraser, we offer:

```
atomic { S }
```

- Atomic: Execute `S` entirely or not at all
- Isolated: No other atomic action can see state in the middle, only before `S` or after `S`
- Consistent: All other atomic actions happen logically before `S` or after `S`

Implement with read/write locking/logging, on words or whole objects; optimistic, pessimistic, etc.

A Basic Software Implementation

- Each transaction has an associated log
- Add a version number / owner to each object
- Each read records (read, object, version) in the log
- Each write causes this transaction to try to become the object's owner (use compare-and-swap or similar); records (write, object, version) in the log
- Each write records (object, field, old-value) in the log

Basic Implementation: commit/abort

Commit

- May commit if each object **read** has same version or is owned by this transaction
- Commit increments version number of each object owned by this transaction

Failure / abort:

- Apply write log entries in reverse order
- Release ownership of objects, restoring original version number

Basic Implementation: properties

Reads are **optimistic**: record version number and **validate** at the end; avoids writes/locks on the object itself.

Writes are **pessimistic**: grab “lock” eagerly.

Update-in-place writing strategy

- implying **undo log** failure strategy

Why is this better than locking?

Abstract: Expresses intent without ever over- or under-specifying how to achieve it: correct

Allows unwind and retry: More flexible response to conflict: prevents deadlock

Allows priority without deadlock: Avoids priority inversion (still need to avoid livelock)

Allows more concurrency: synchronises on exact data accessed rather than an object lock* … and distinguishes reads and writes

*The basic strategy is intermediate in granularity

Limitations of Simple Transactions

Long/large transactions either reduce concurrency or are unlikely to commit.

Data structures often have false conflicts

- e.g., reorganising tree nodes

Closed Nesting

[Moss'81]

Each sub-transaction builds its own read/write set.

On commit, **merge** with its parent's sets.

On abort, **discard** its set.

Sub-transaction never conflicts with <u>ancestors</u>

- Conflicts with <u>non-ancestors</u>
- Can see ancestors' intermediate state, etc.

Requires keeping values at each nesting level that writes a data item.

Closed Nesting Helps: partial rollback

When actions conflict, one will be rolled back.

With closed nesting, roll back only up through the youngest conflicting ancestor.

This reduces the amount of work that must be redone when retrying.

Limitations of Closed Nesting

Limitations derive from the original non-nested semantics:

- Aggregates larger and larger conflict sets
  - Still hard to complete long/large transactions
- Synchronises at physical level
  - Gives false conflicts

Open Nesting to the Rescue

Concept and theory developed in the late 1980s

- Comes from the database community
- Partly an explanation/justification of certain real strategies employed in database systems
- Partly an approach to generalising those strategies

Conceptual Backdrop of Open Nesting

Closed nesting has just one level of abstraction:

- Memory contents
  - Basis for concurrency control
  - Basis for rollback

Open nesting has more levels of abstraction

Each level may have a distinct:

- Concurrency control model (style of locks)
- Recovery model (operations for undoing)

Open Nested Transactions

While running, a <u>leaf</u> open nested action

- Operates at the memory word level

When it commits

- Its memory changes are <u>permanent</u>
- Concurrency control and recovery <u>switch levels</u>
    - Give up memory level “locks”, acquire <u>abstract locks</u>
    - Give up memory level unwind, use only <u>inverse operations</u> (undos)

Non-Leaf Open Nested Transactions

A non-leaf open nested action operates at the memory word level, and

- May accumulate abstract locks and undos from committed children

When it commits

- Its memory changes are permanent
- Concurrency control and recovery switch levels
    - Give up memory level “locks” and child locks, acquire abstract locks
    - Give up memory level unwind and child undos, use only inverses (undos)

Abstract Serialisability

Lock parts of abstract state:

- To prevent conflicting forward operations
- To ensure that undo remains applicable

Undo in the abstract:

- Restores changed part of abstract state

Boosting versus Open Nesting

Open nesting is built on top of an assumed TM system (STM or HTM).

Boosting is built on top of assumed “thread-safe” (*linearisable*) data types

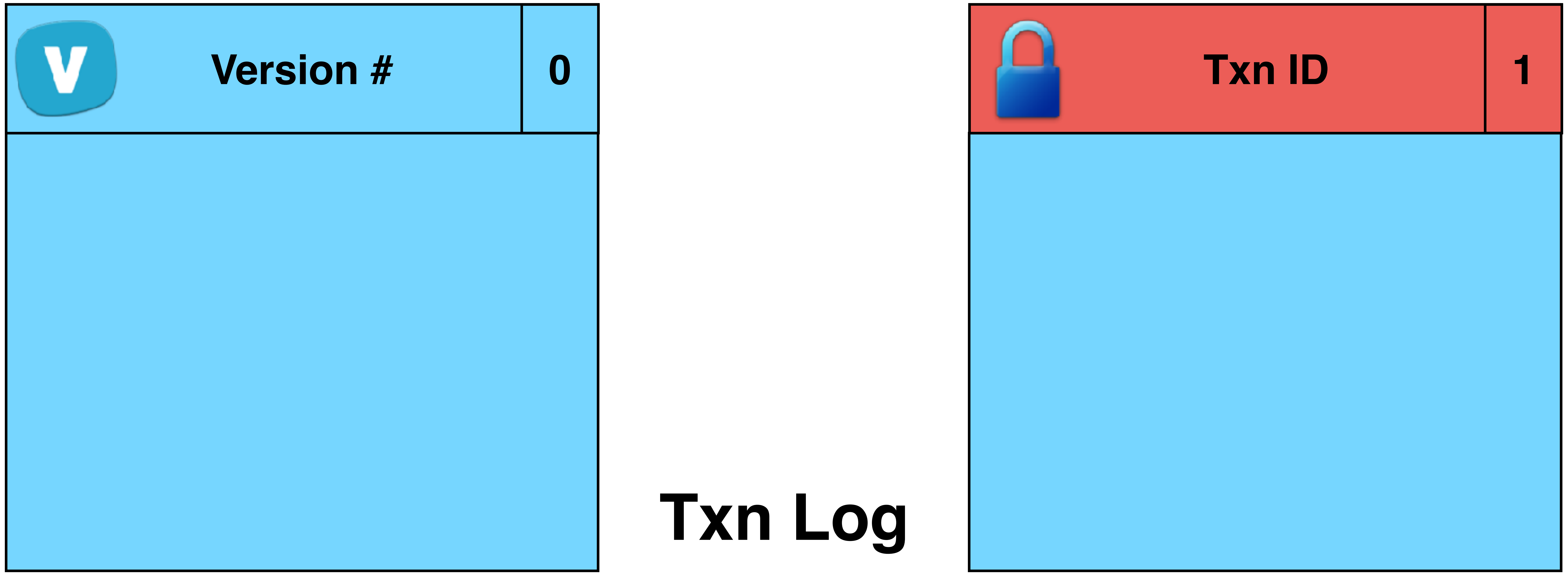
- Implement insides of a concurrent data structure however you like, as long as it is concurrency safe
- Make it transactional by wrapping it with abstract locks and abstract undos

HTM Acceleration

Existing HTM (e.g., Intel TSX) is best-effort flattened transactions.

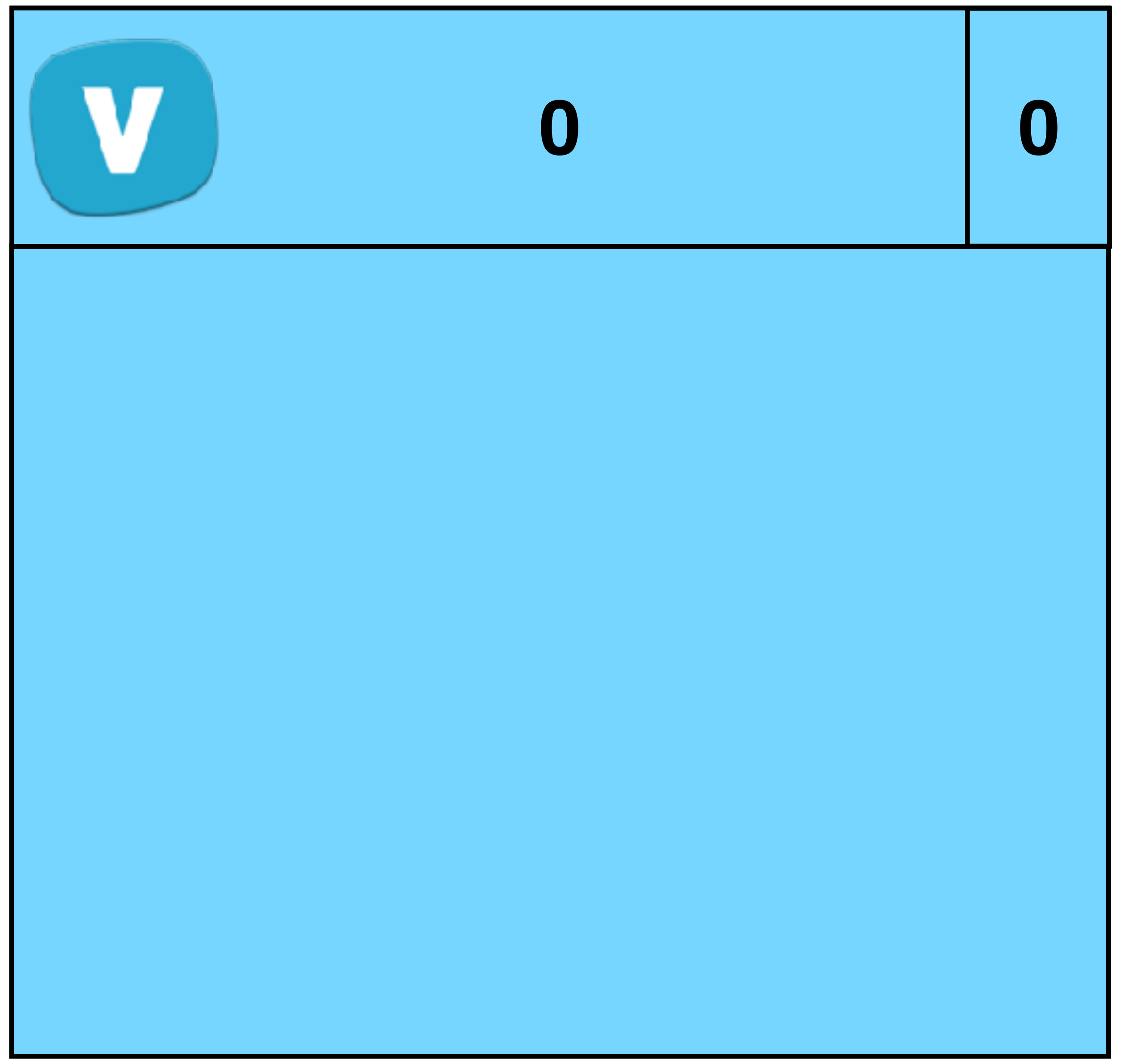
- Top-level in HTM down through all nested children works.
- Closed nested in HTM for children of STM is not useful:
    - Child needs the same STM overhead on behalf of STM ancestors
- Open nested in HTM for children of STM avoids most overhead:
    - HTM handles physical conflicts
    - Abstract locking / undos handle abstract conflicts

TM Metadata



One metadata word for every object

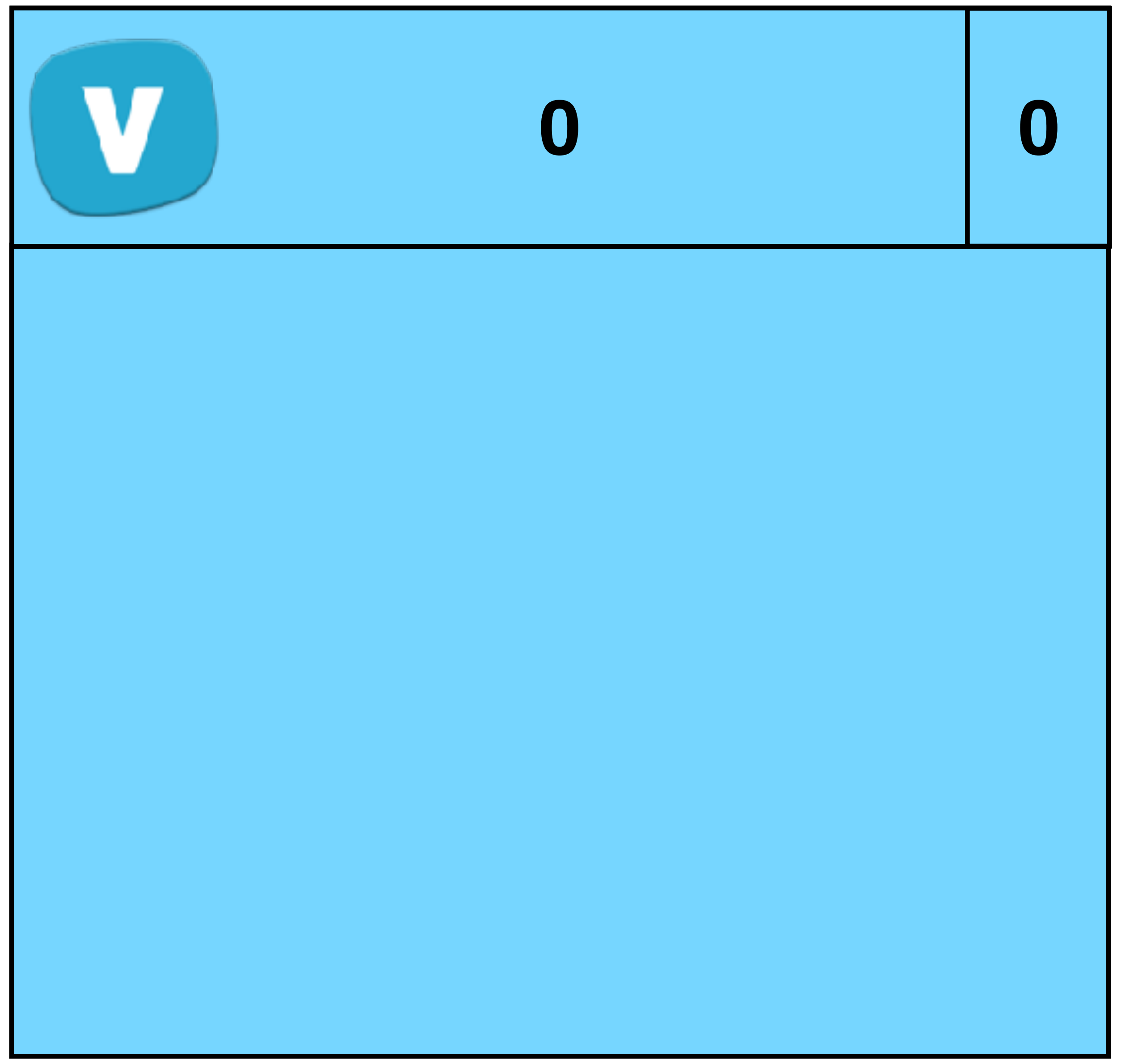
STM Transaction Protocol (Read)



T1 Log

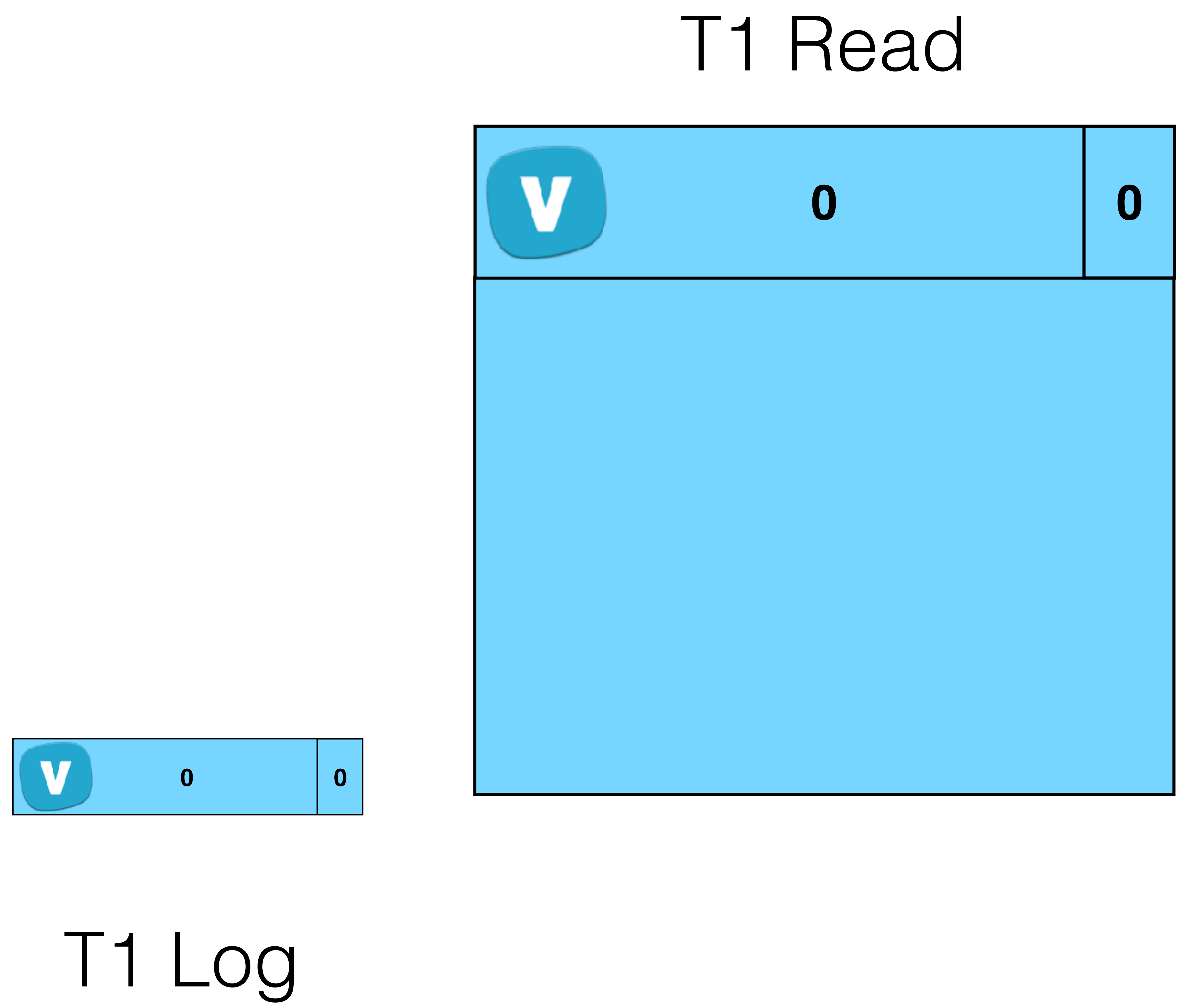
STM Transaction Protocol (Read)

Check Mode

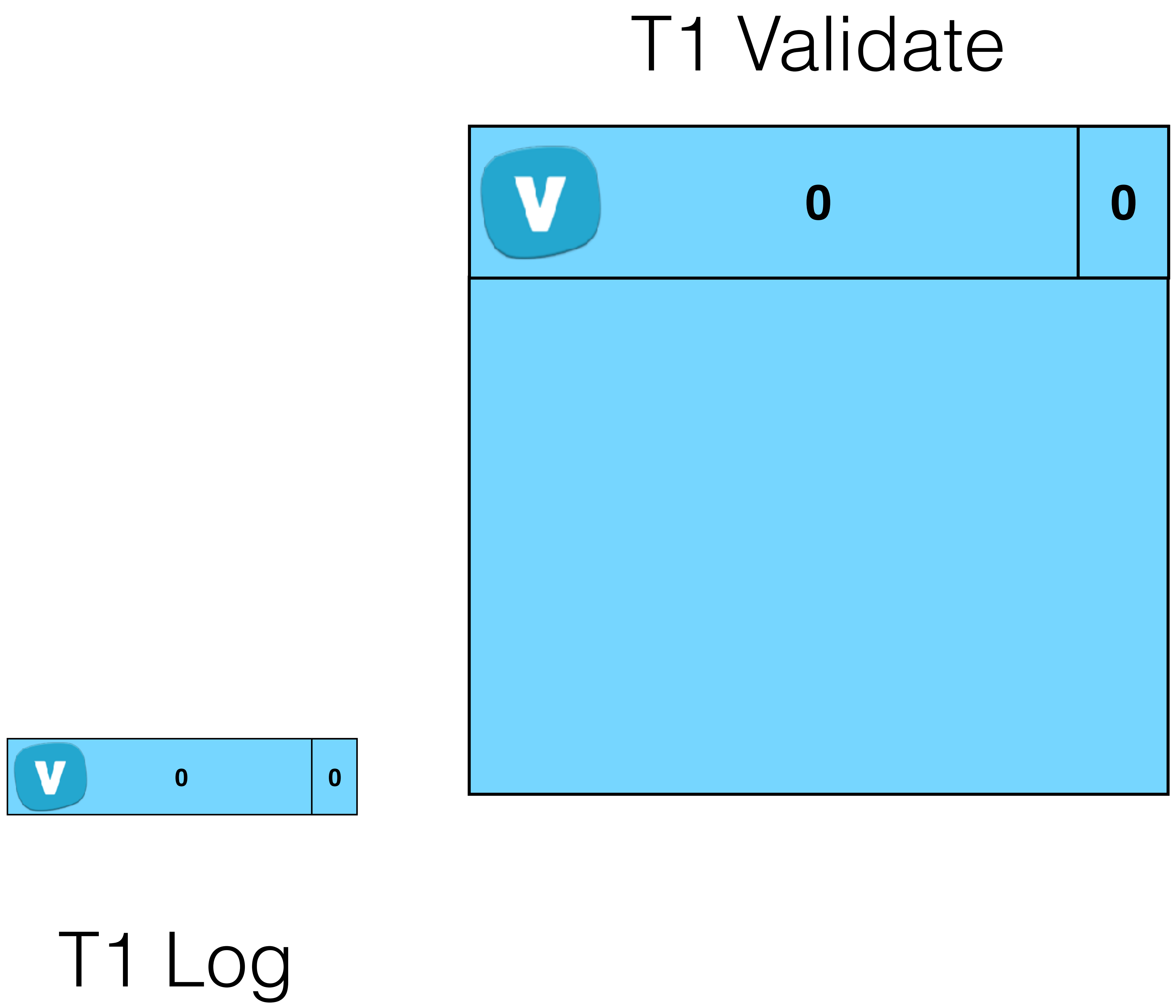


T1 Log

STM Transaction Protocol (Read)

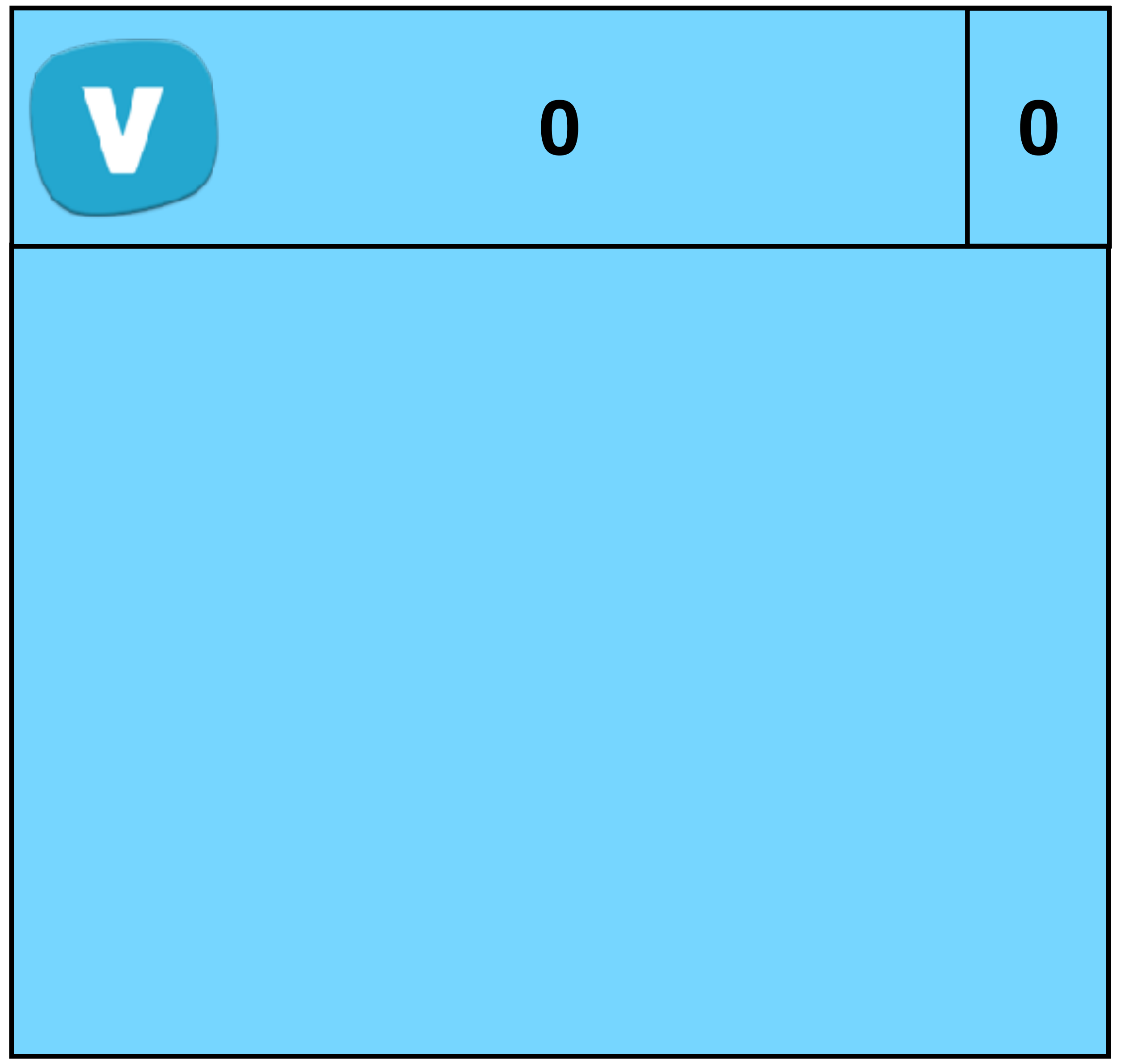


STM Transaction Protocol (Read)



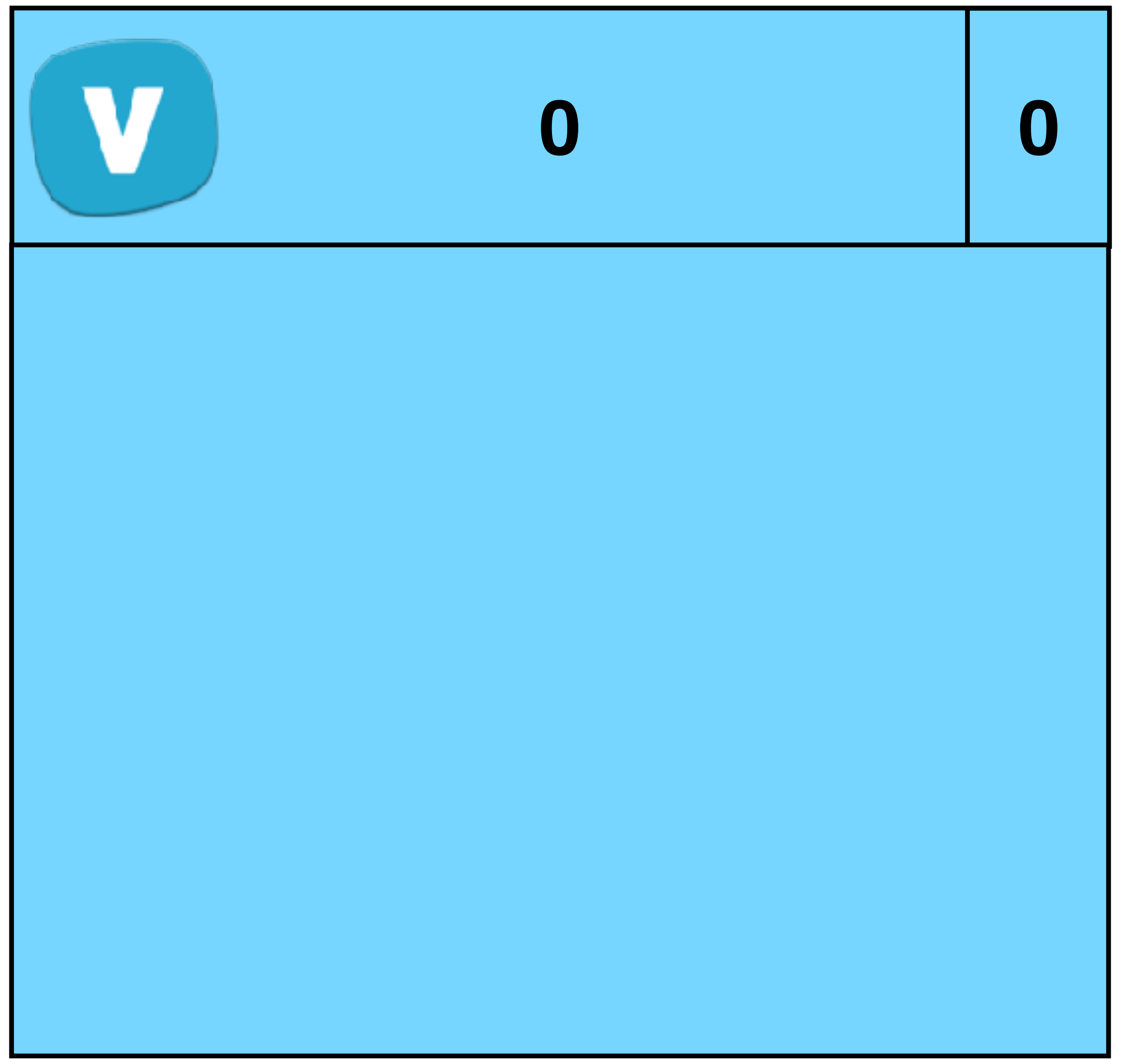
STM Transaction Protocol (Read)

T1 Commit



T1 Log

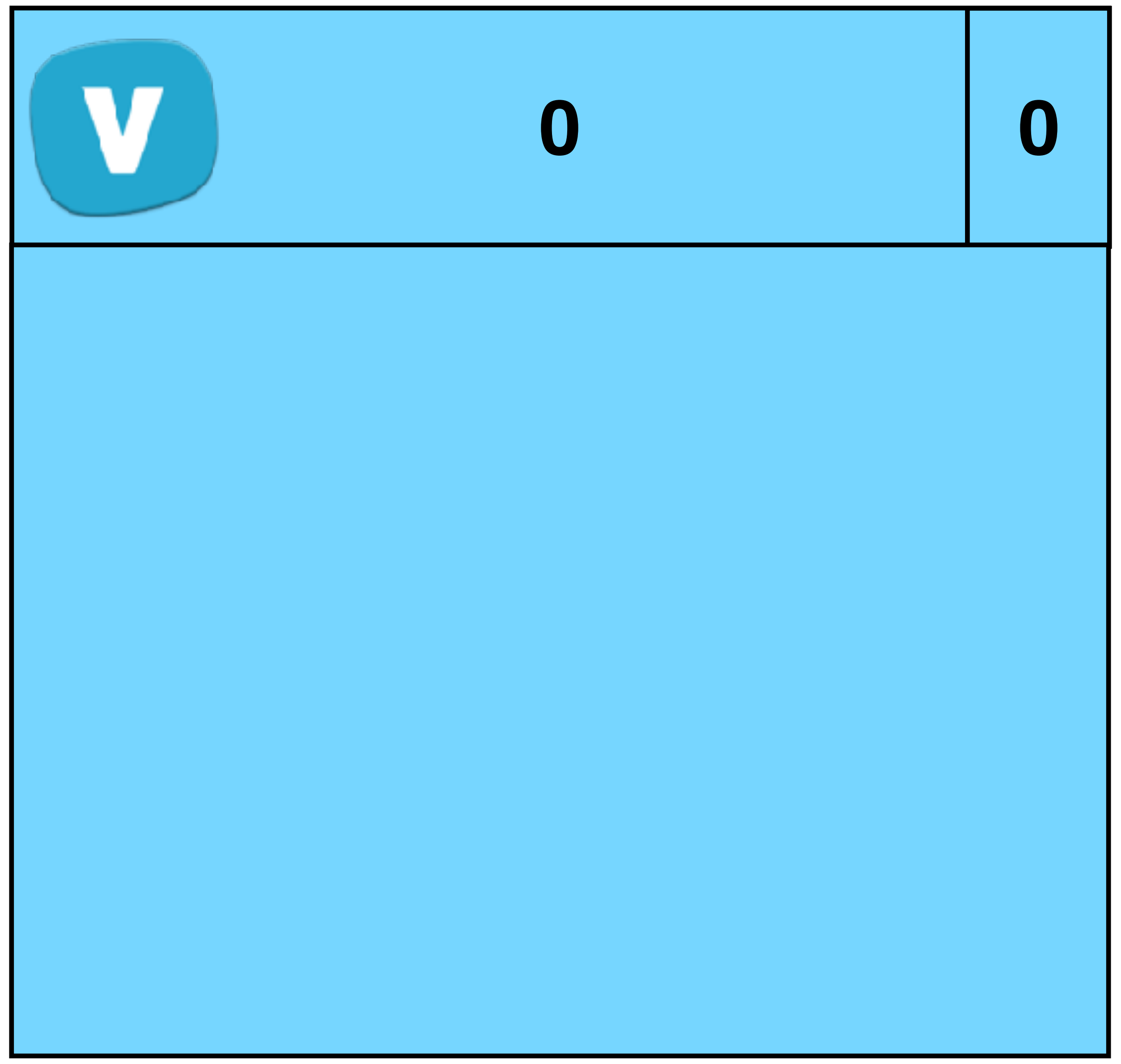
STM Transaction Protocol (Write)



T1 Log

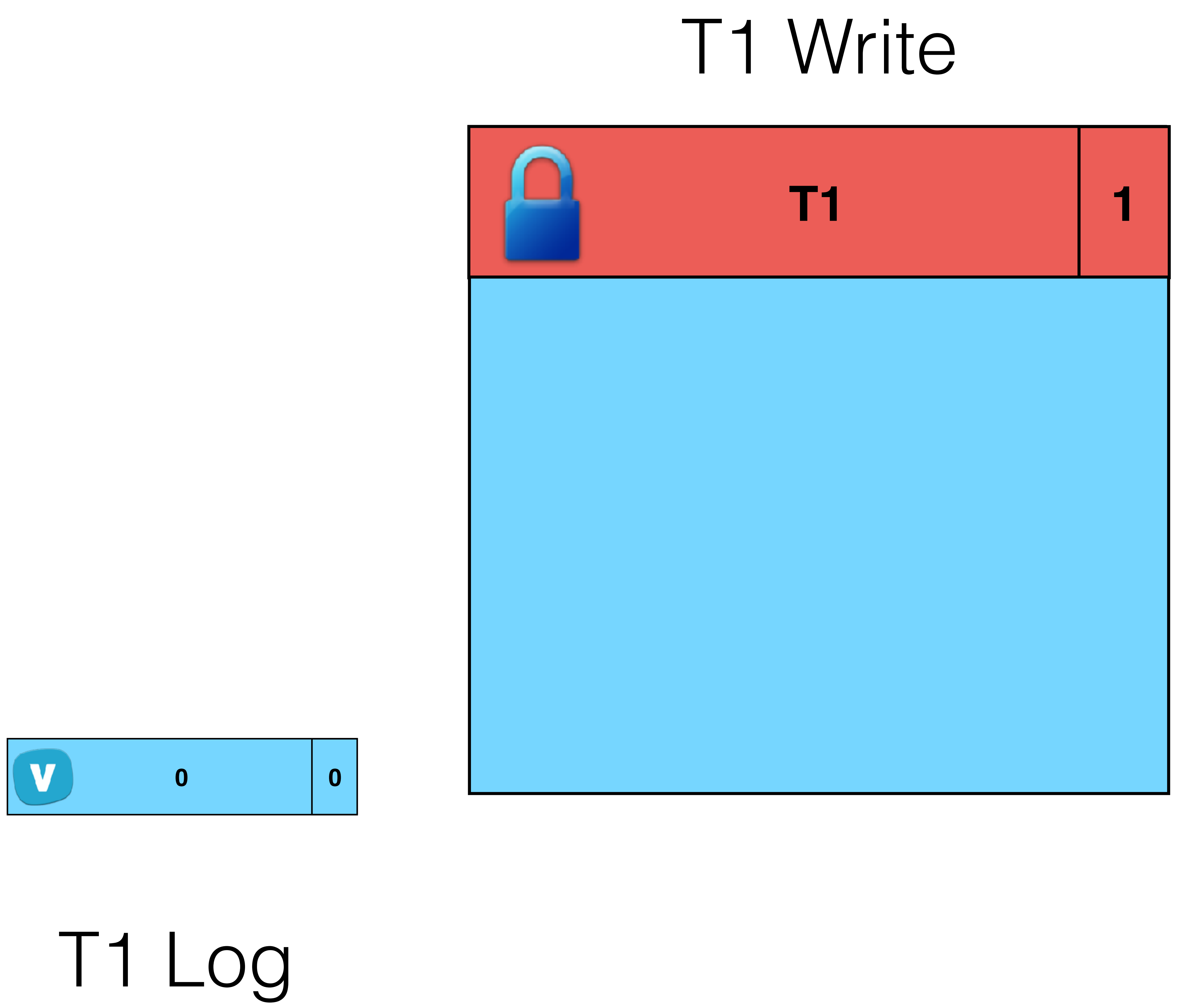
STM Transaction Protocol (Write)

Check Mode

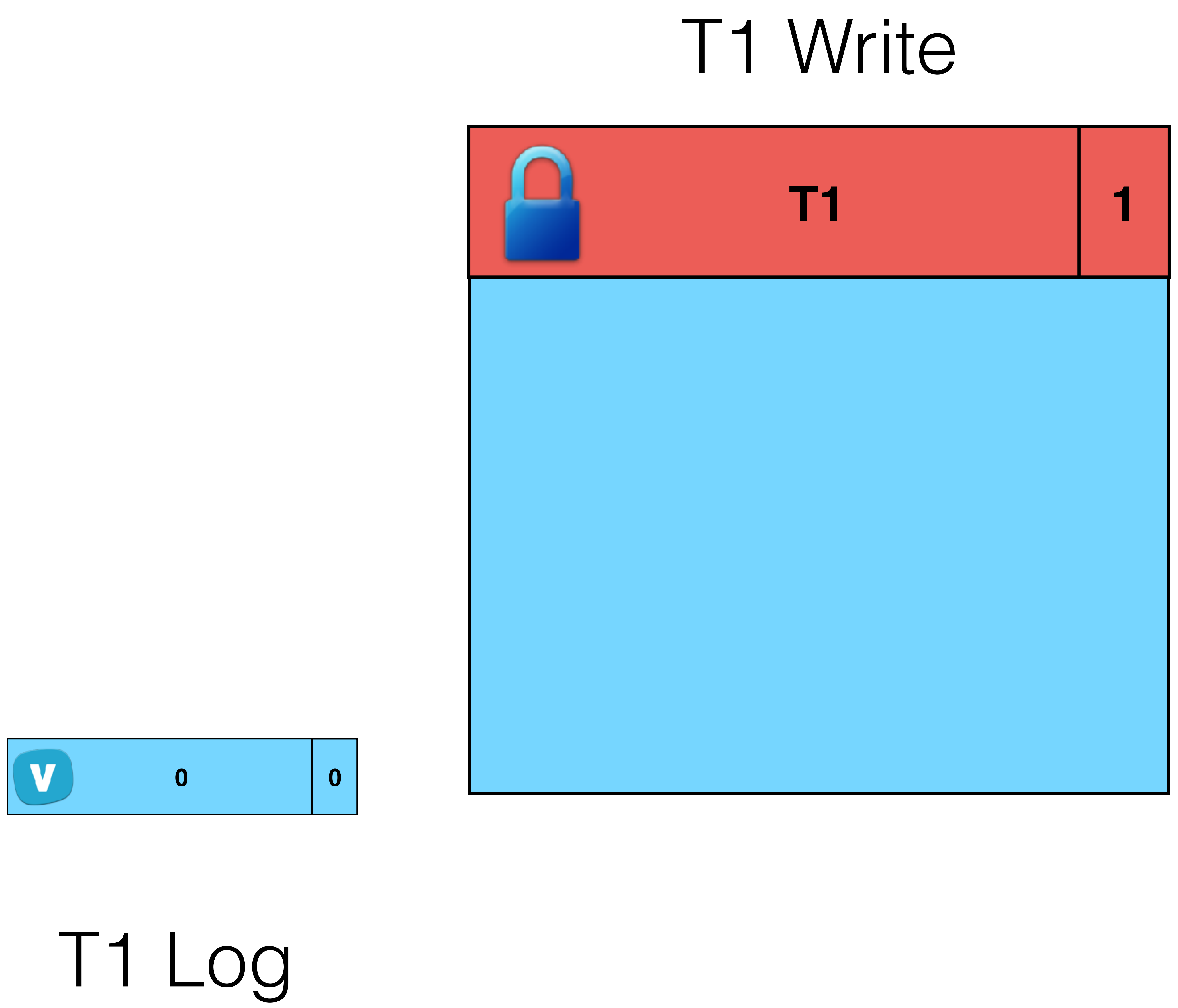


T1 Log

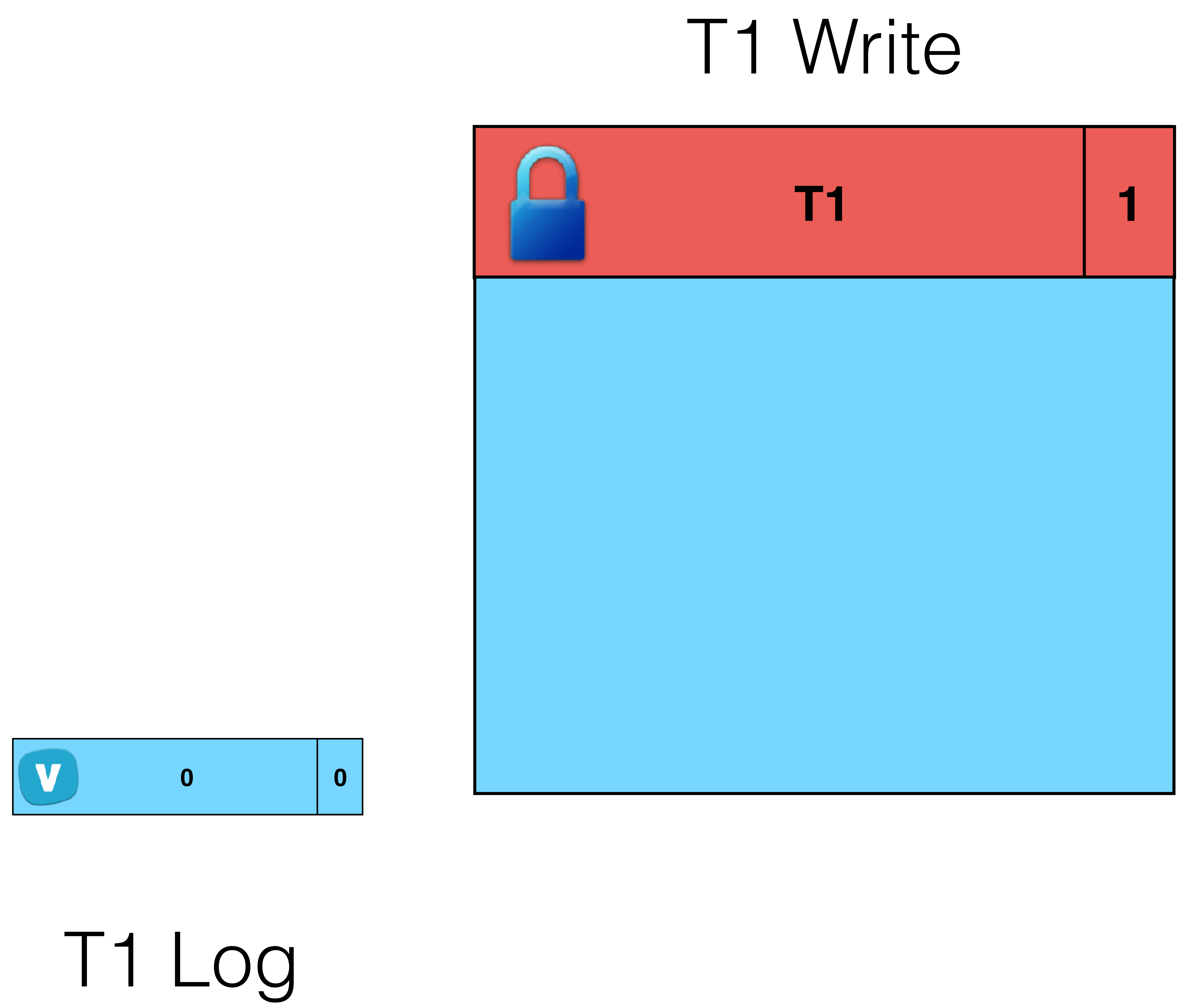
STM Transaction Protocol (Write)



STM Transaction Protocol (Write)

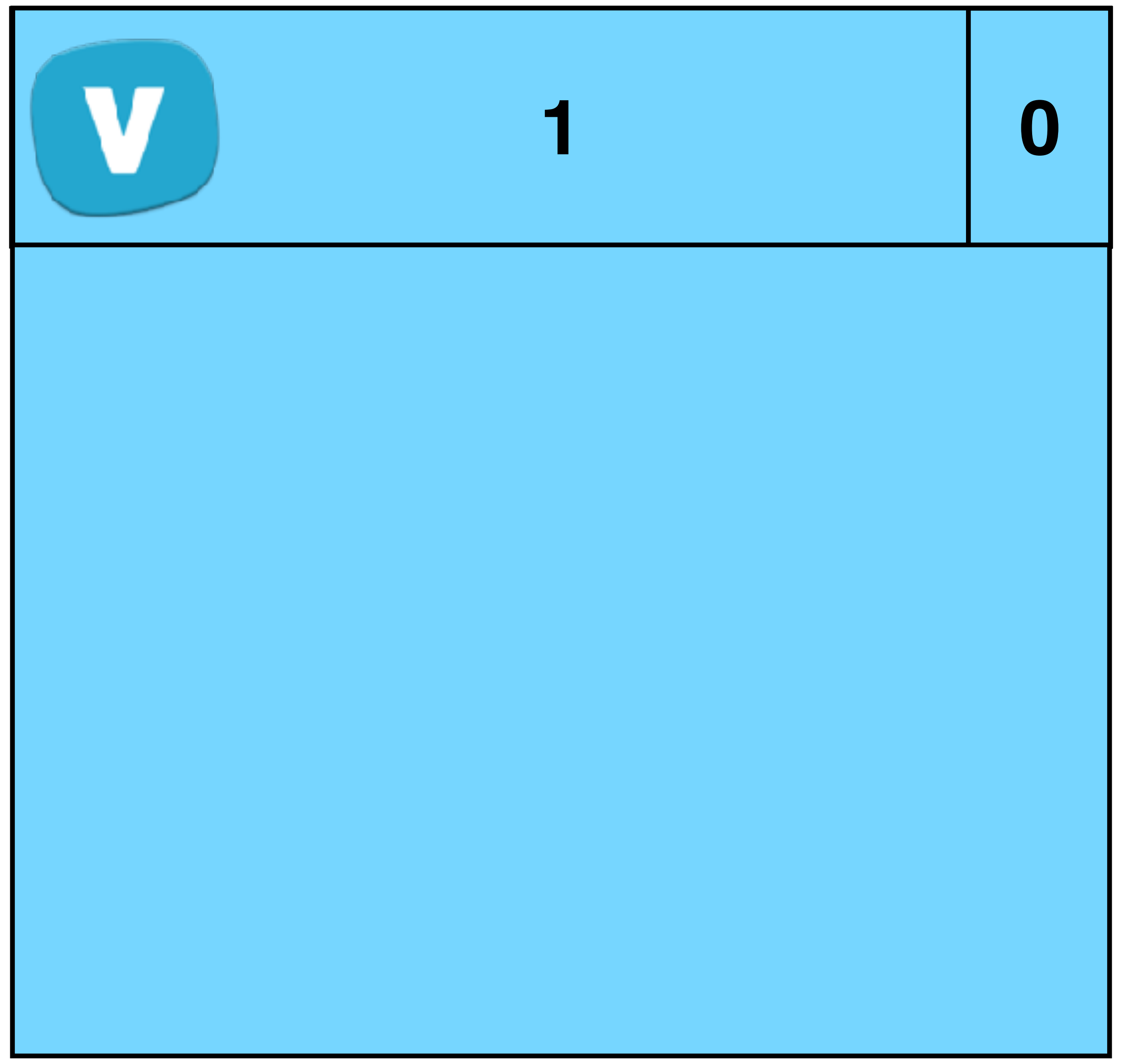


STM Transaction Protocol (Write)



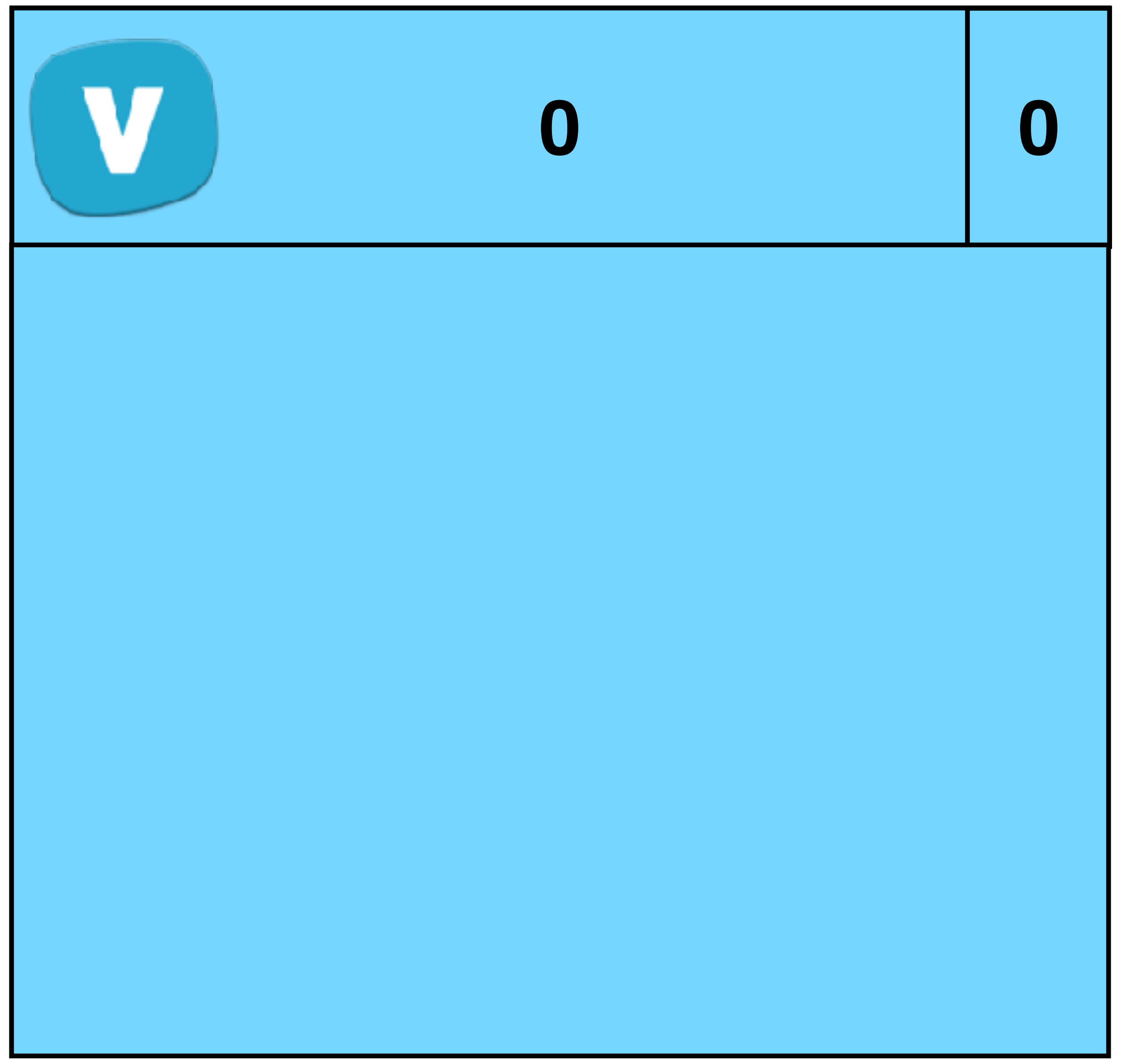
STM Transaction Protocol (Write)

T1 Commit



T1 Log

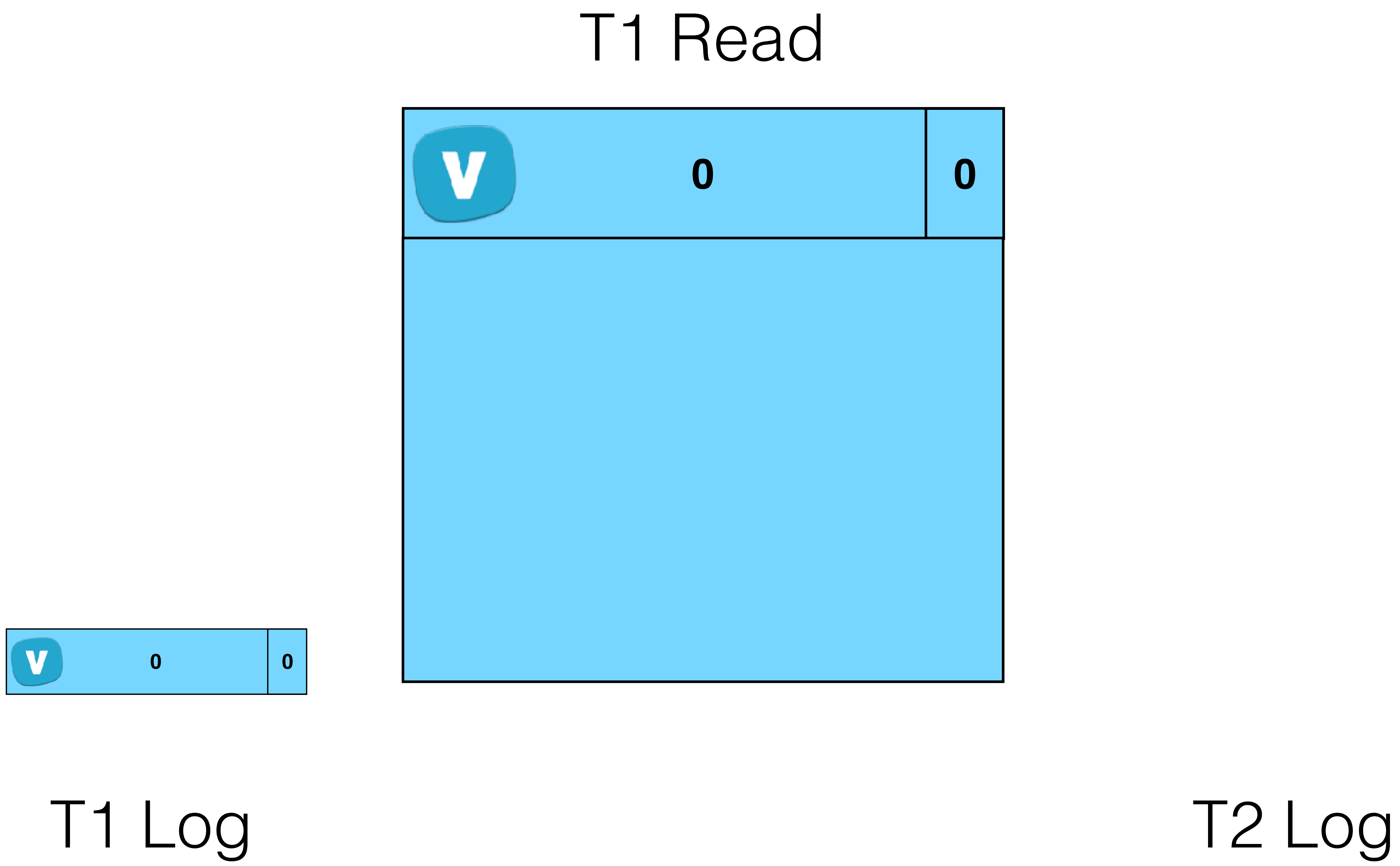
STM Conflict Detection



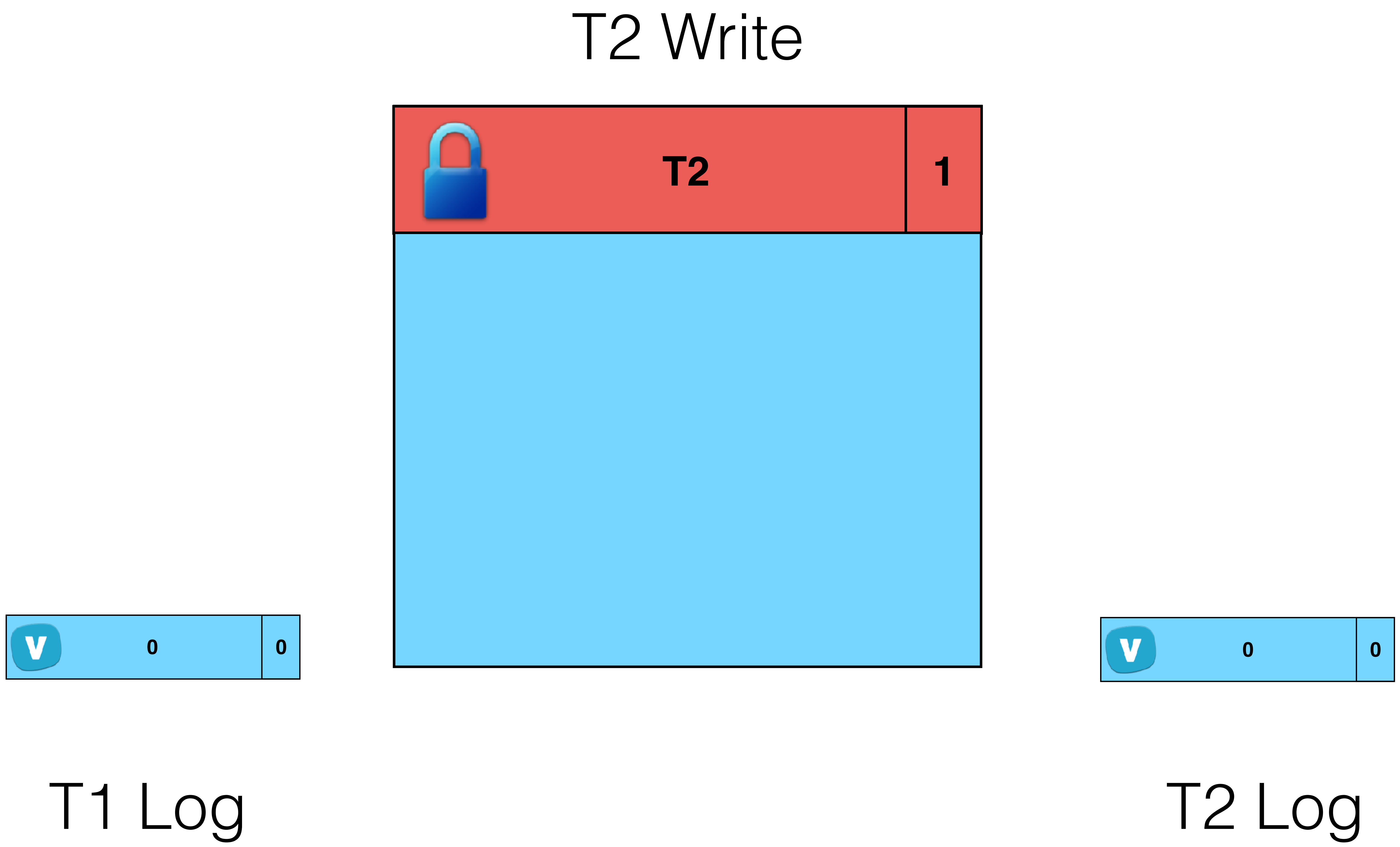
T1 Log

T2 Log

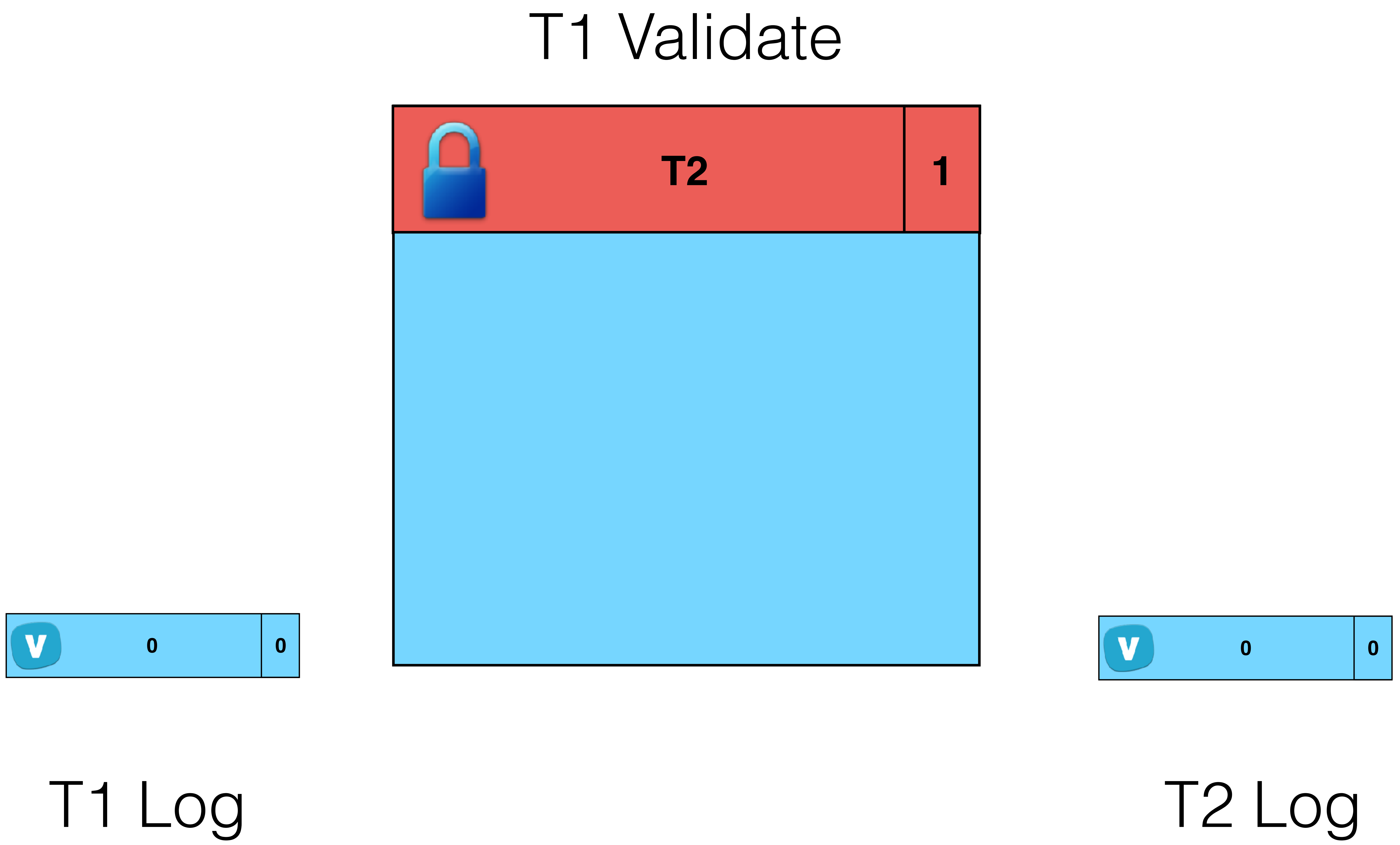
STM Conflict Detection



STM Conflict Detection

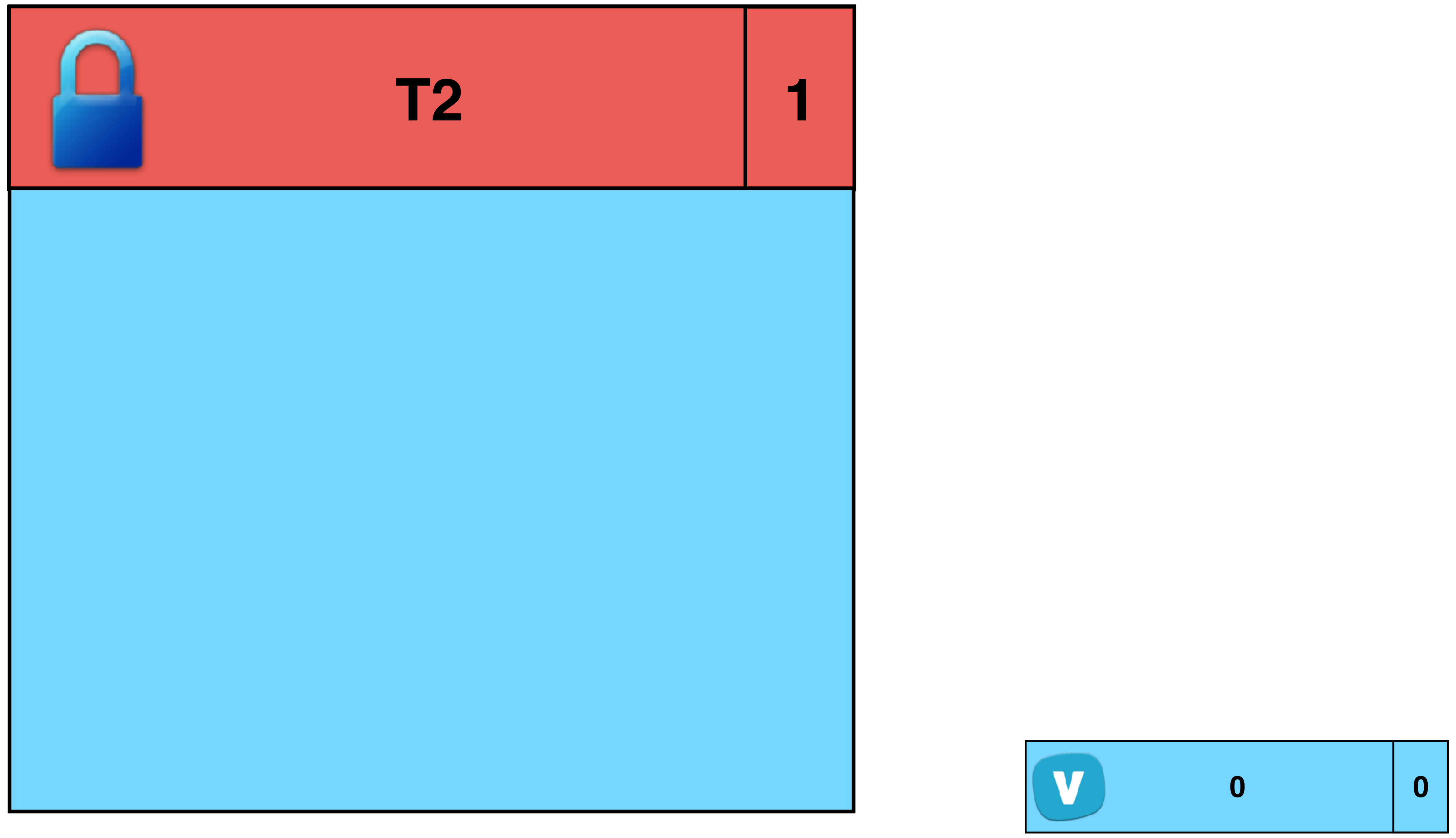


STM Conflict Detection



STM Conflict Detection

T1 Abort



T1 Log

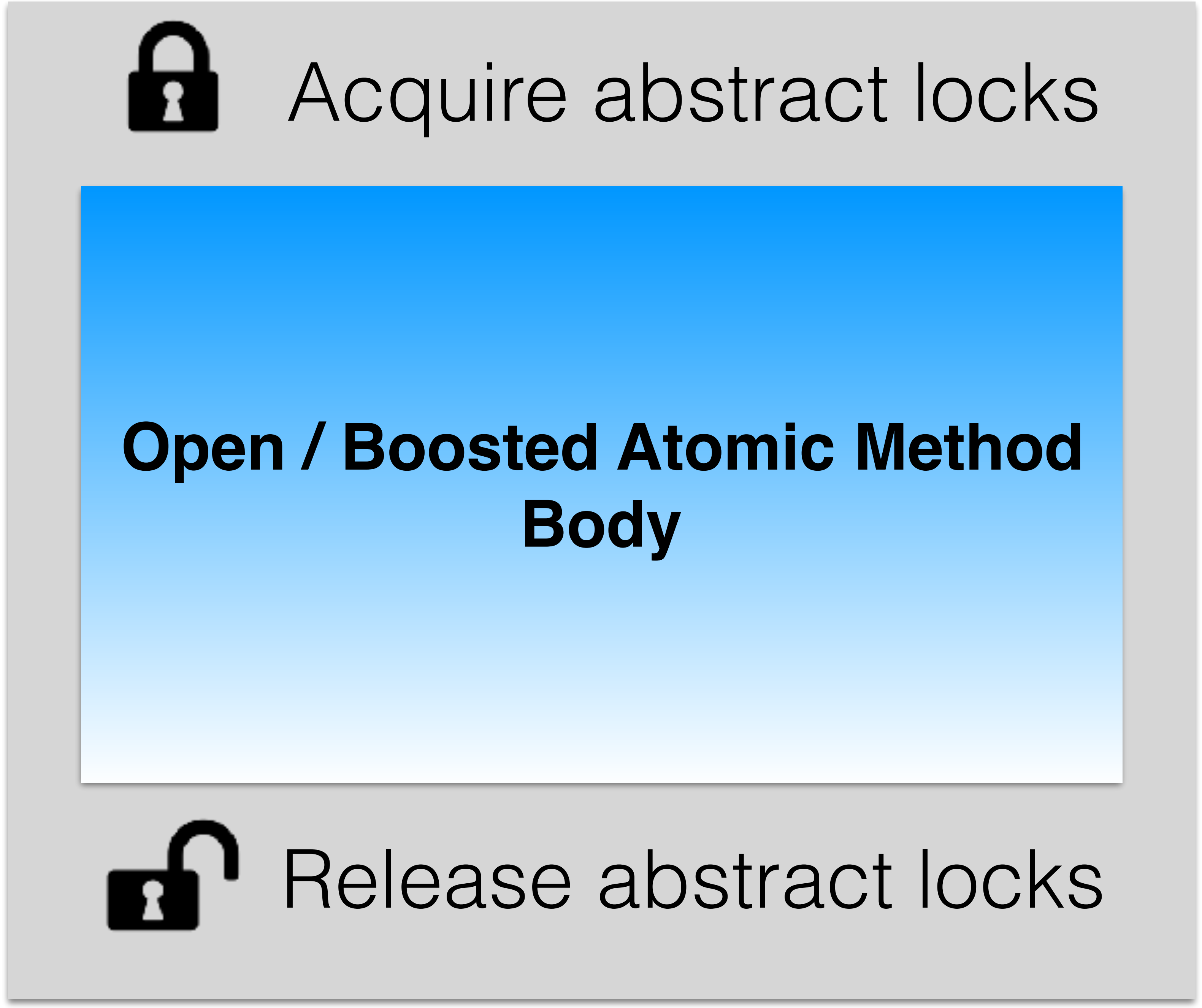
T2 Log

Hybrid Transaction Protocol

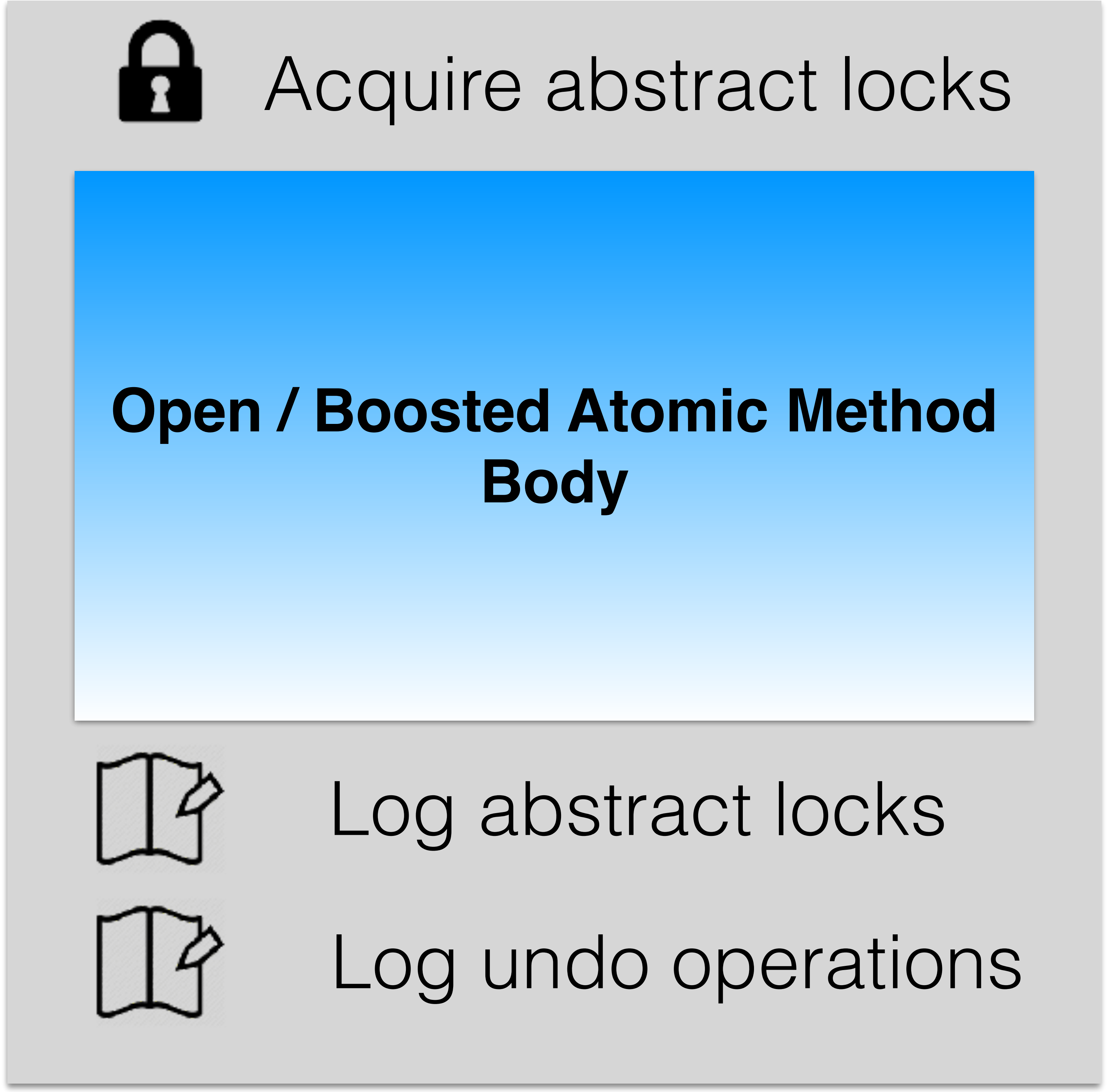
Detect HTM-STM conflicts via lock word accesses

- Explicit XABORT if locked by another transaction
- HTM reads — read the metadata word
  - STM writes modify the metadata word
  - Causes HTM to abort
- HTM writes — increment version number
  - Causes STM read invalidation / HTM abort

Abstract Locks for STM

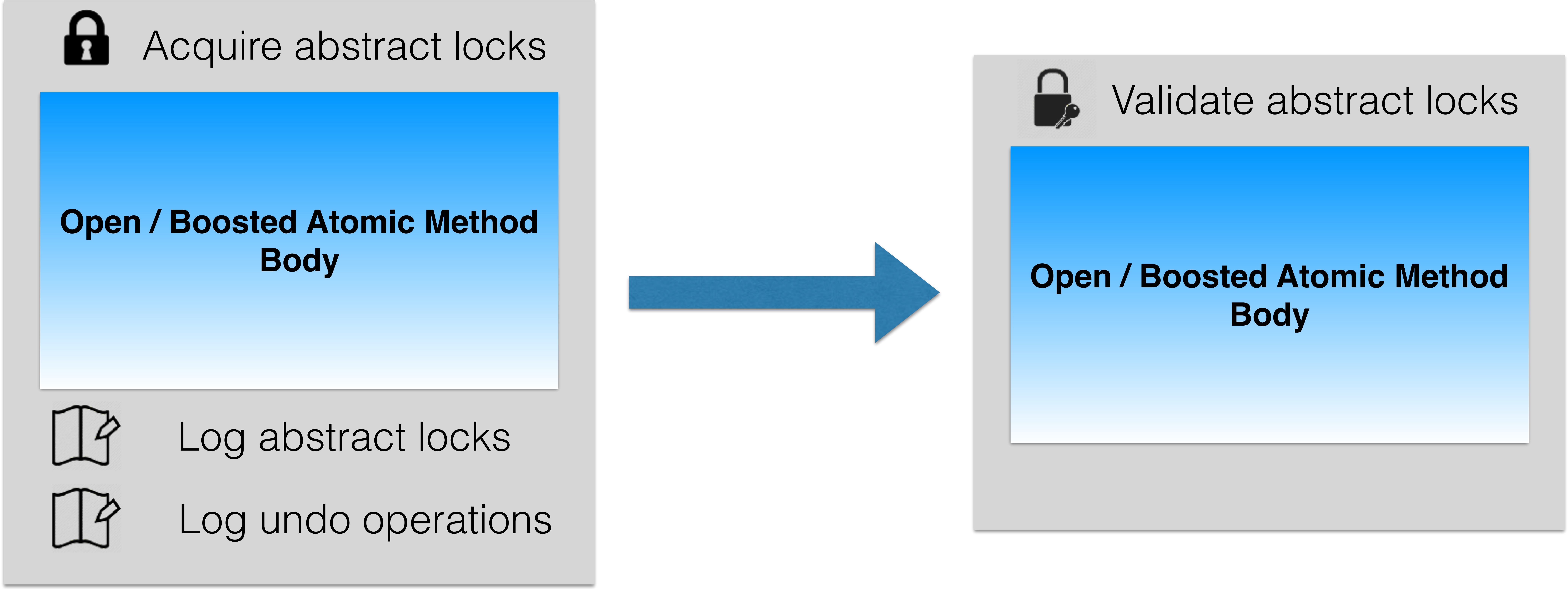


If top level transaction



If nested transaction

Abstract Locks for Hybrid TM

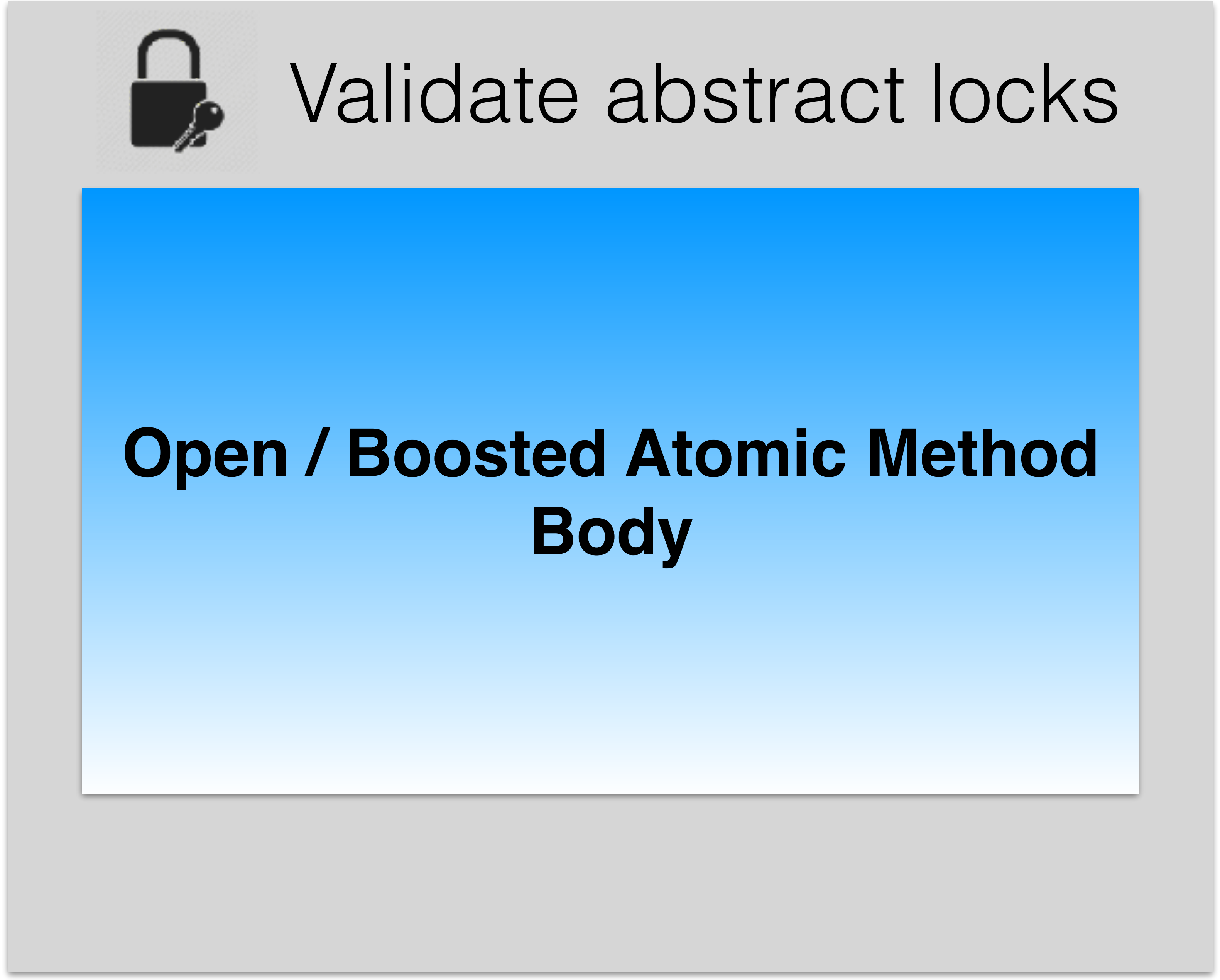


If top level is HTM

Why Validation Works

HTM–STM

- If abstract locks conflict they must touch some same physical words in the abstract locking data structure — otherwise they could not detect the conflict



Why Validation Works

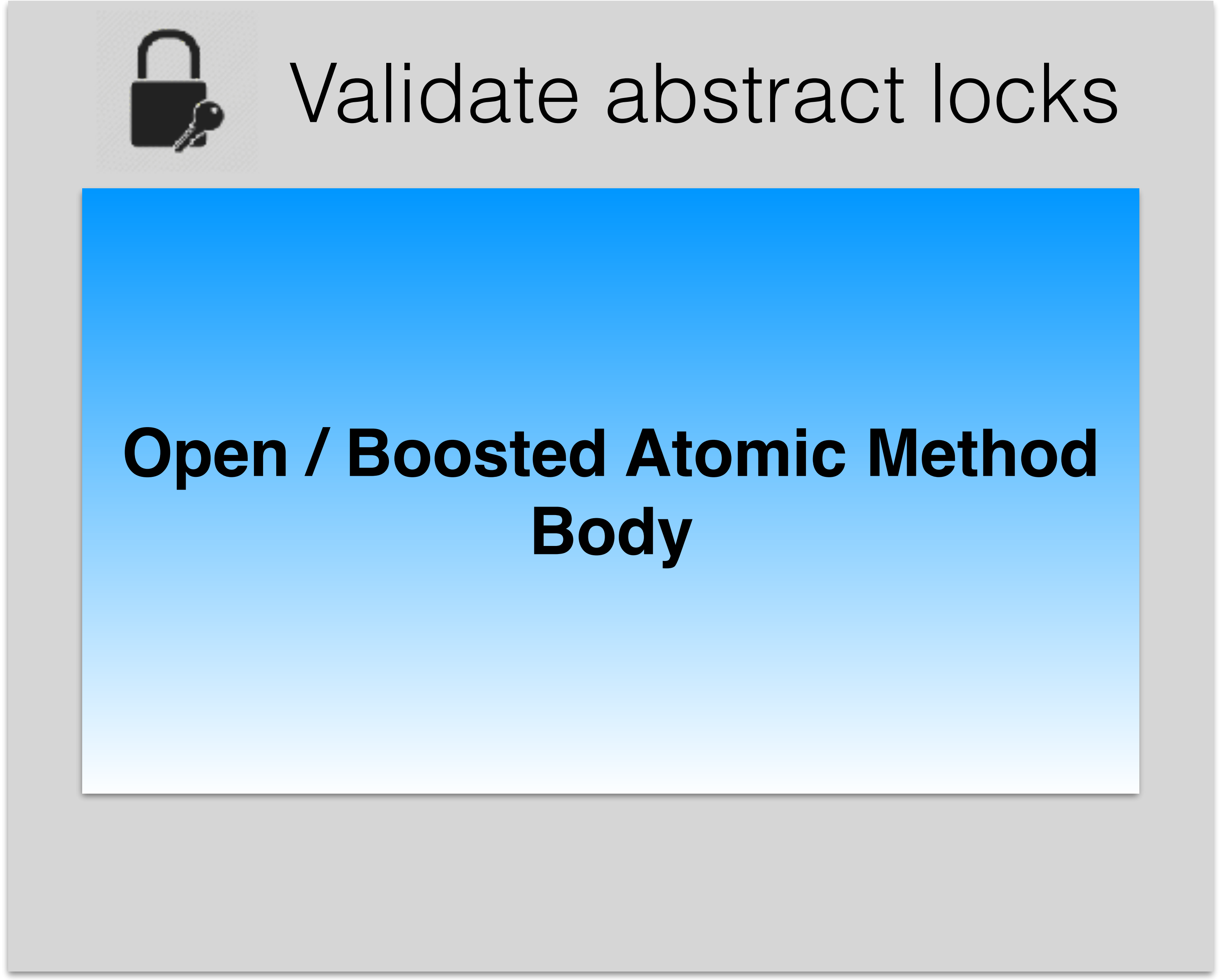
HTM–STM

- If abstract locks conflict they must touch some same physical words in the abstract locking data structure — otherwise they could not detect the conflict

HTM–HTM

- No conflict in the locking data structure because all accesses to it are reads

- Any real conflicts that exist will occur on the actual abstract lock data structure



STM and HTM Methods

STM needs logging HTM doesn't

Different actions during read/write

Different actions for abstract locks

HTM should fall back to STM

STM and HTM Methods

STM needs logging HTM doesn't

Different actions during read/write

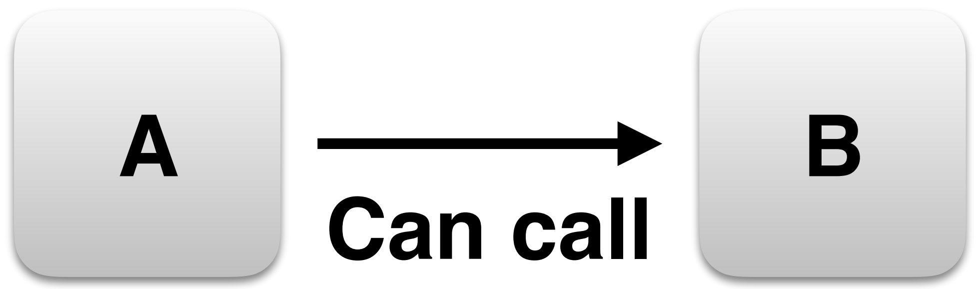
Different actions for abstract locks

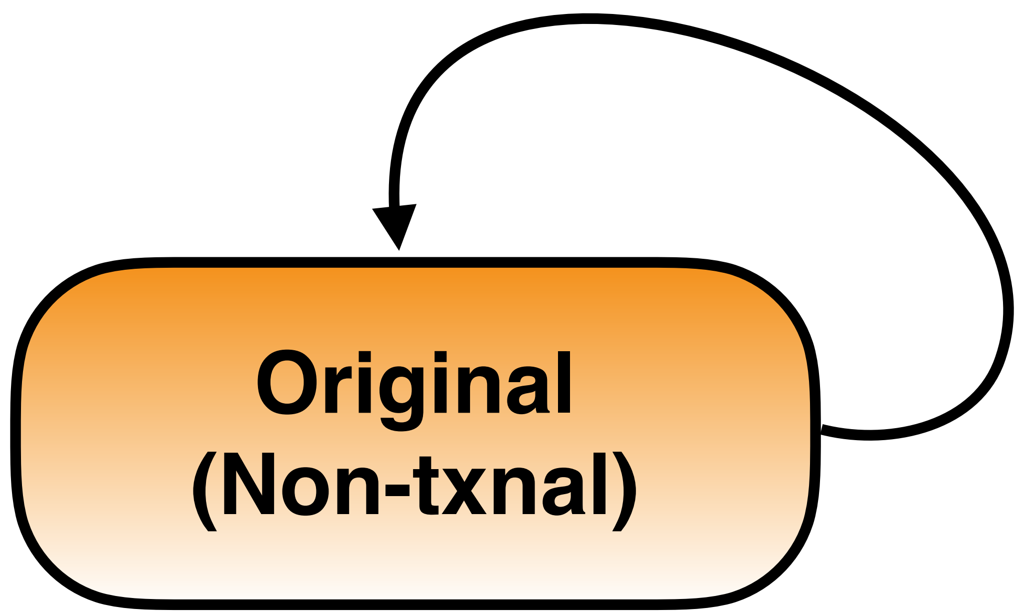
HTM should fall back to STM

Maintain separate HTM and STM versions of methods

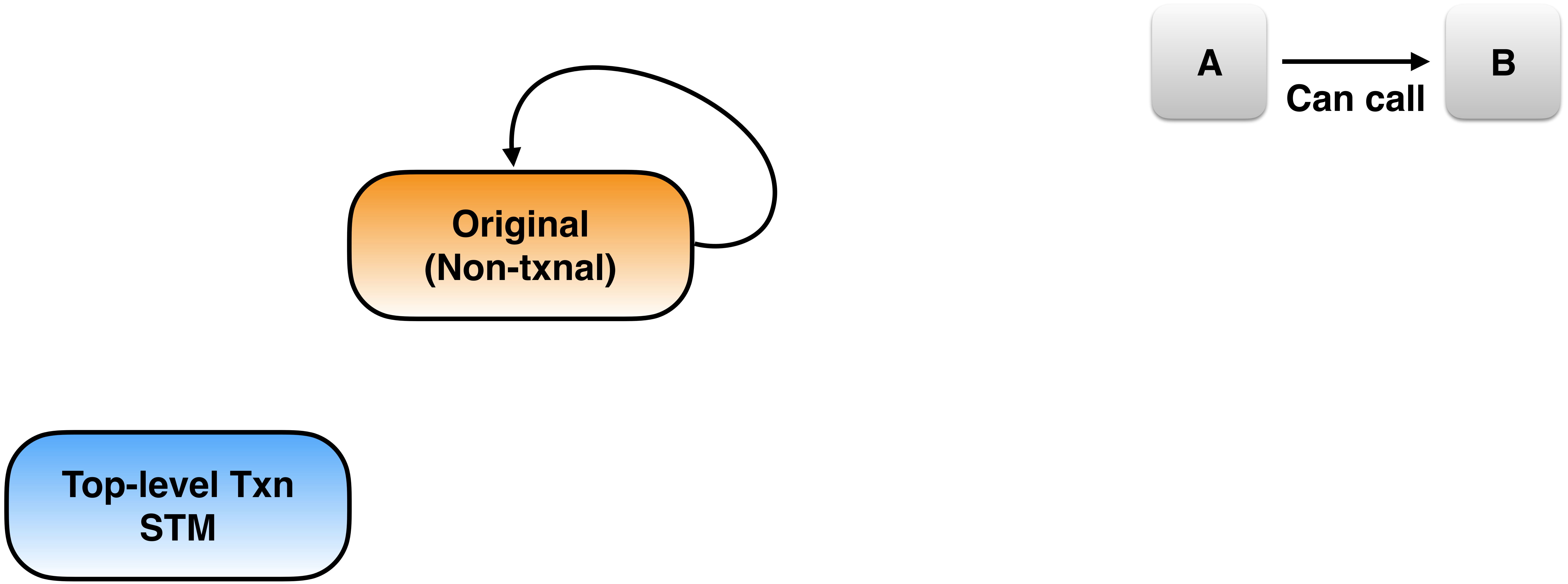
STM Method Variants

STM Method Variants

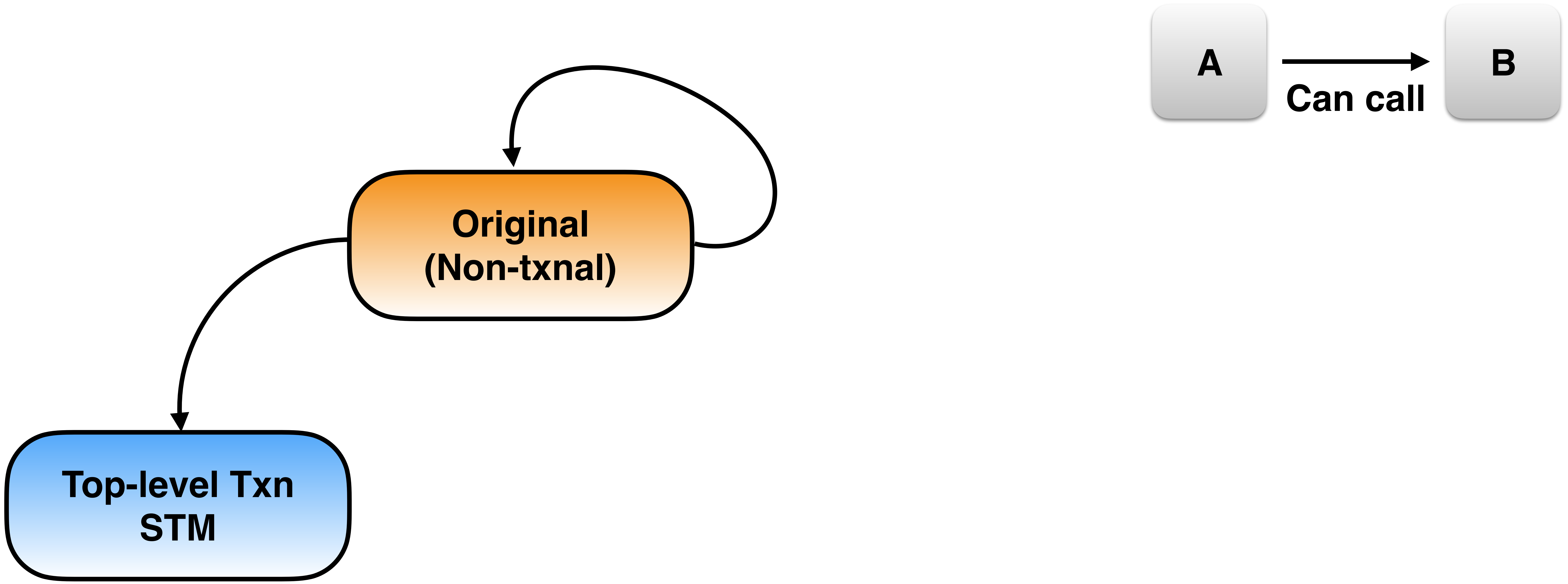




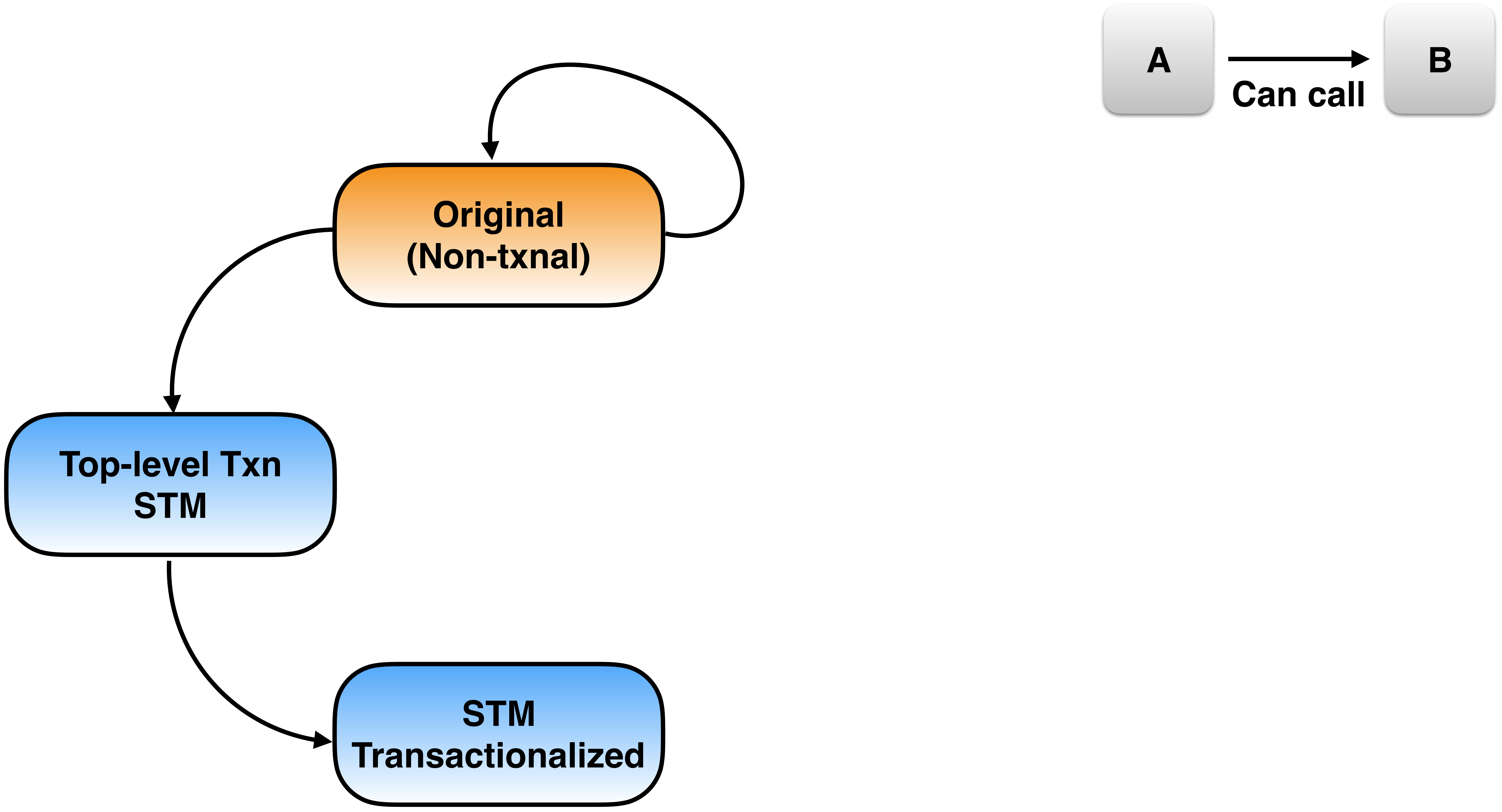
STM Method Variants



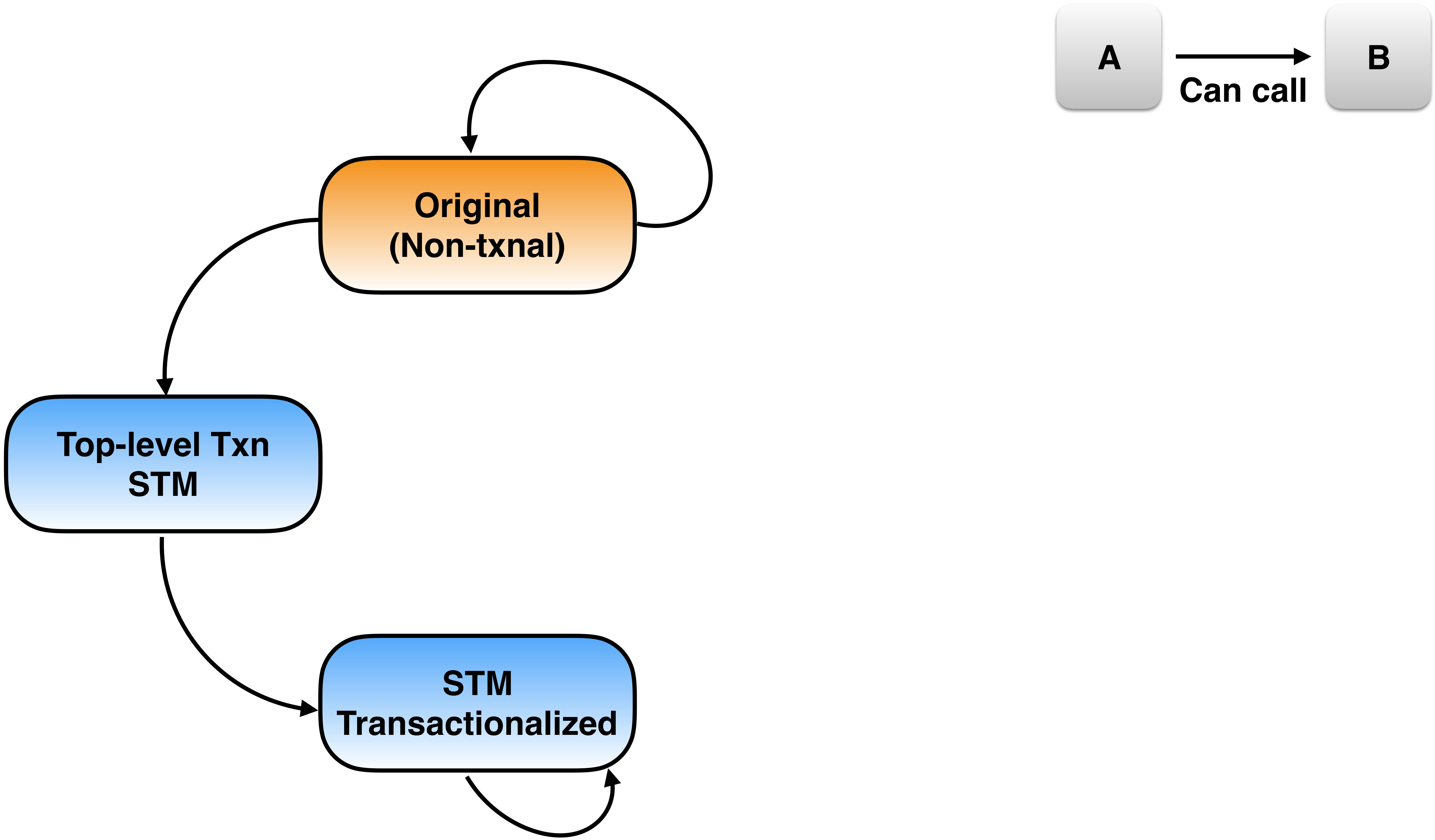
STM Method Variants



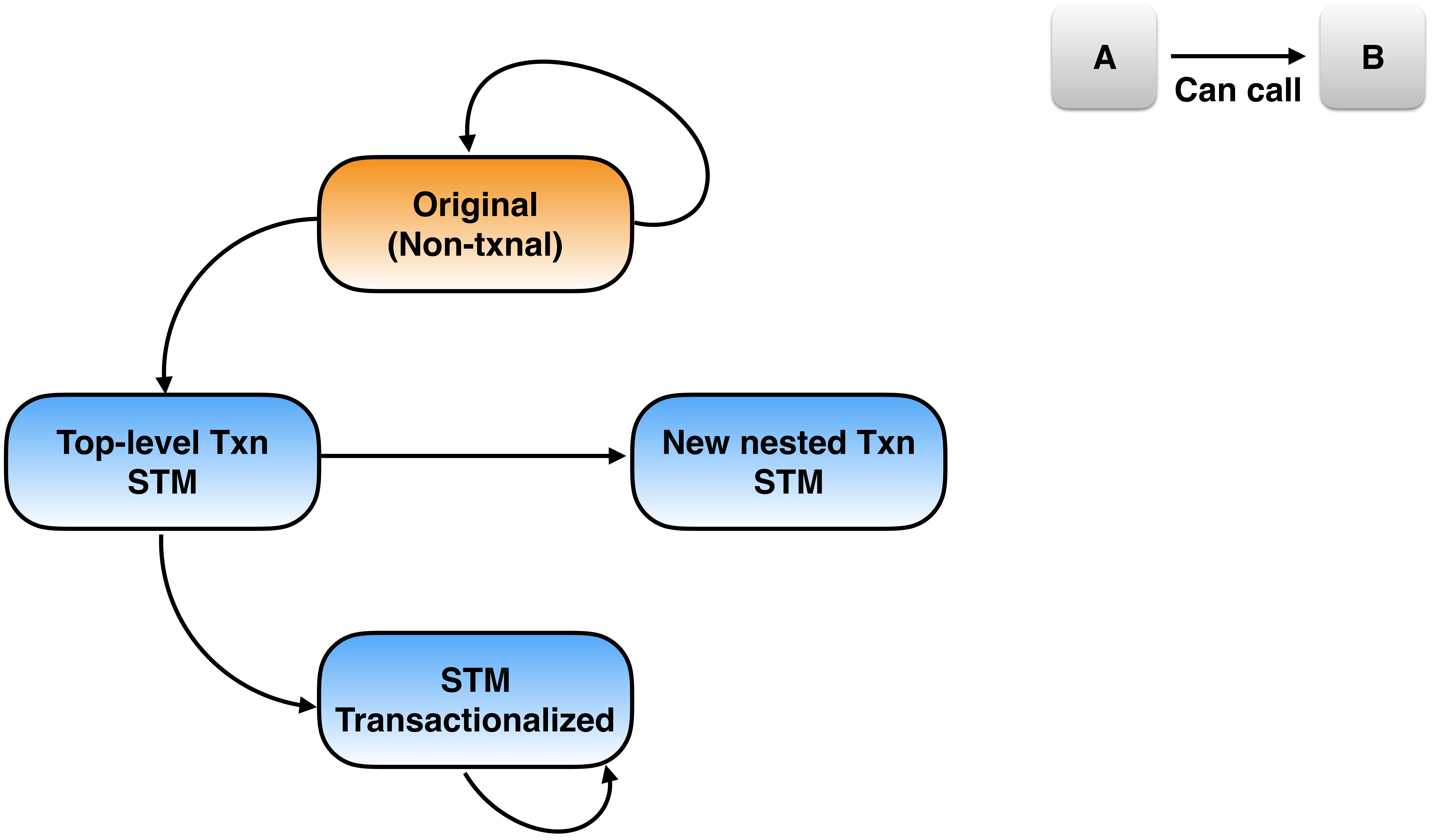
STM Method Variants



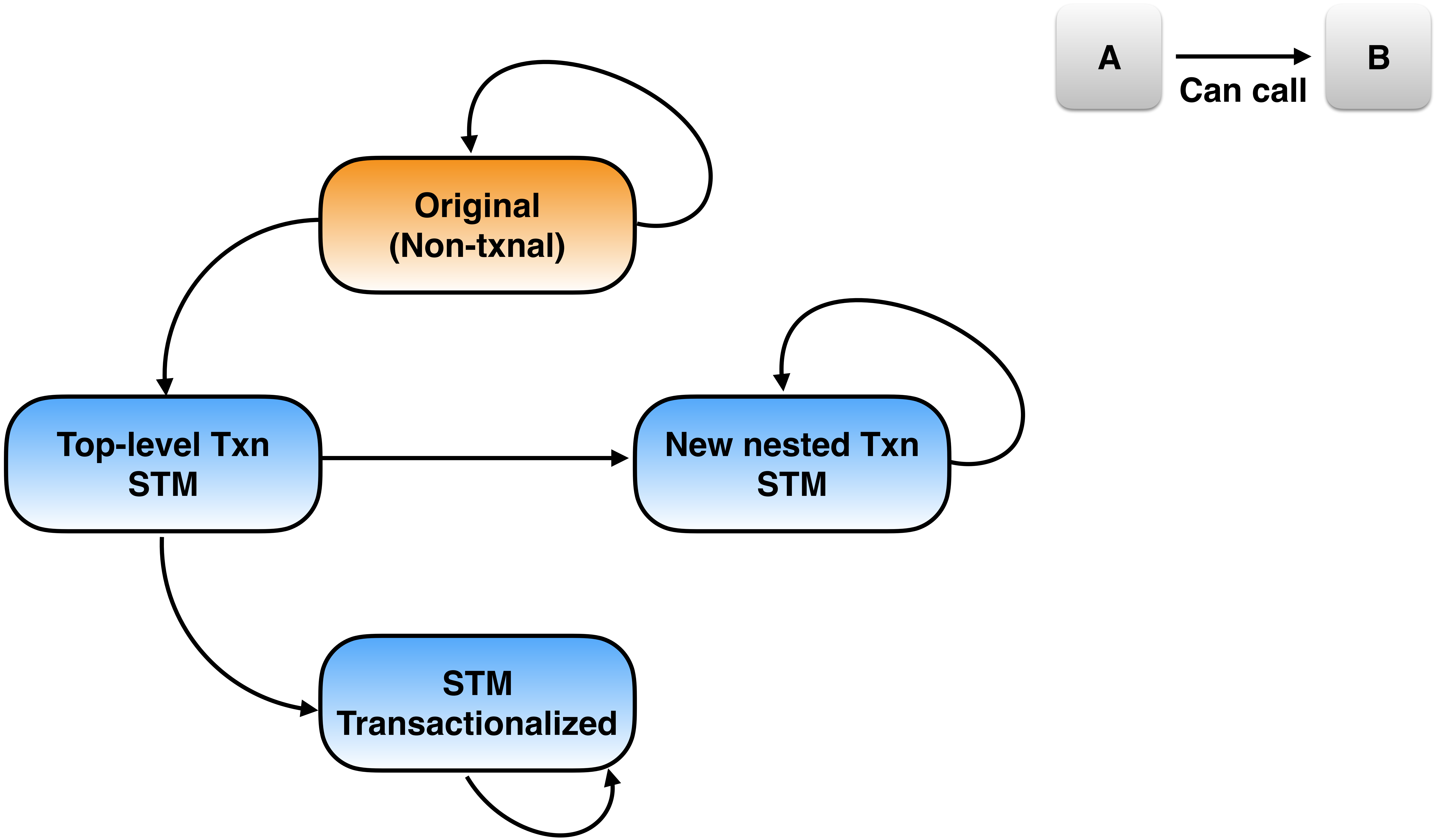
STM Method Variants



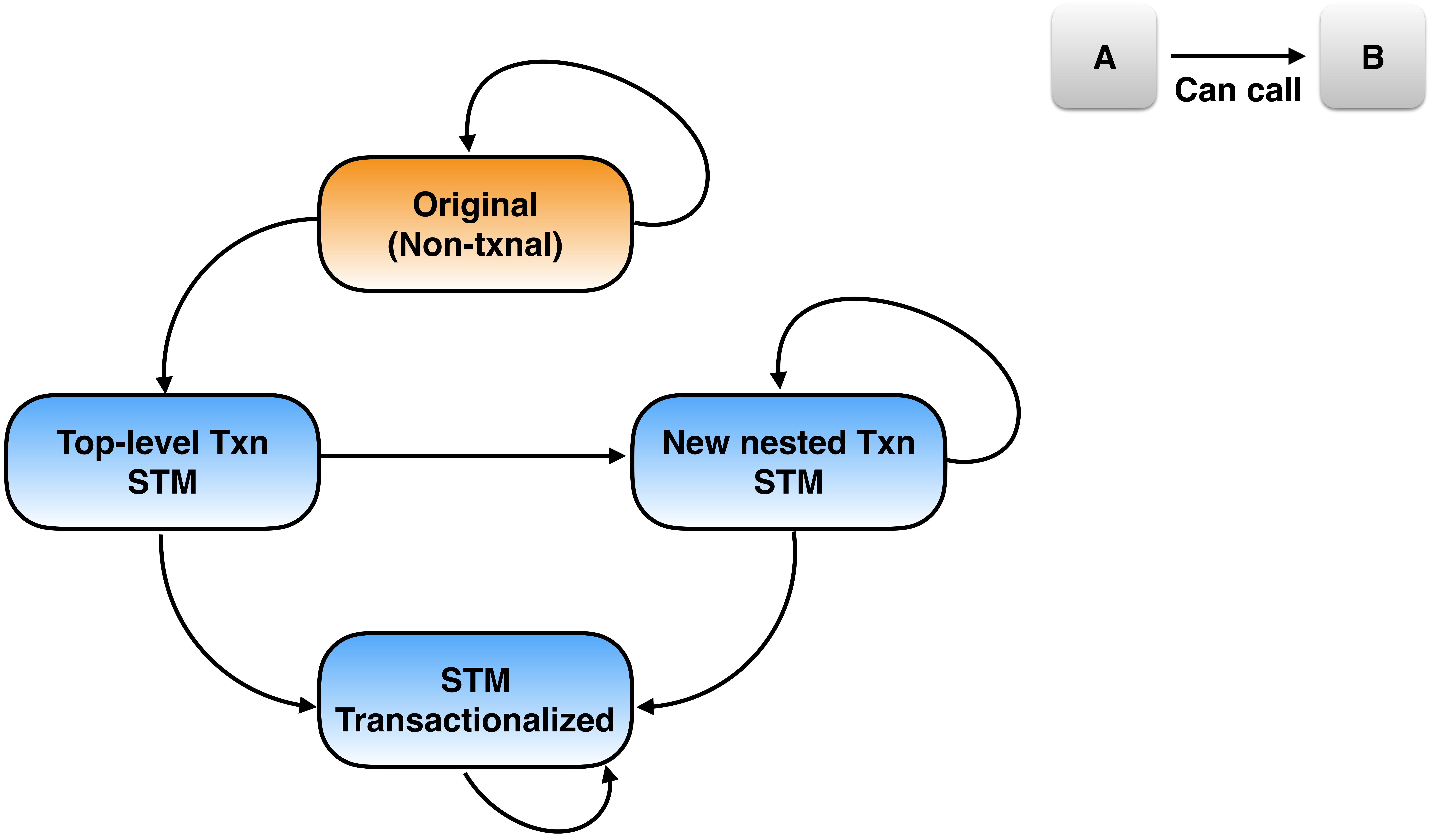
STM Method Variants



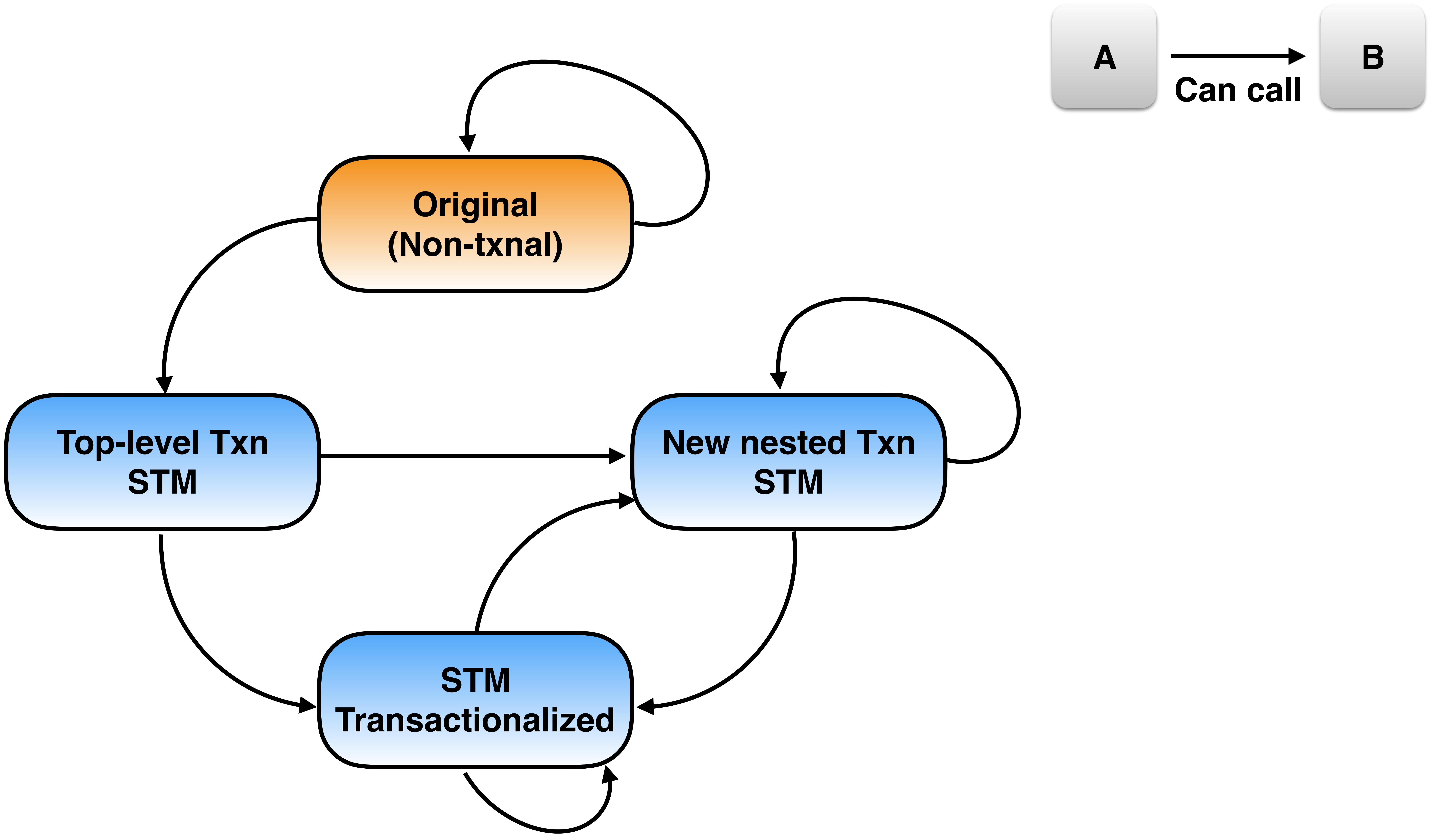
STM Method Variants



STM Method Variants



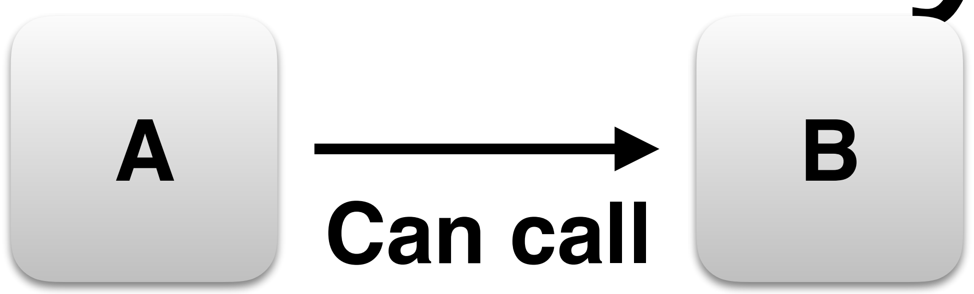
STM Method Variants

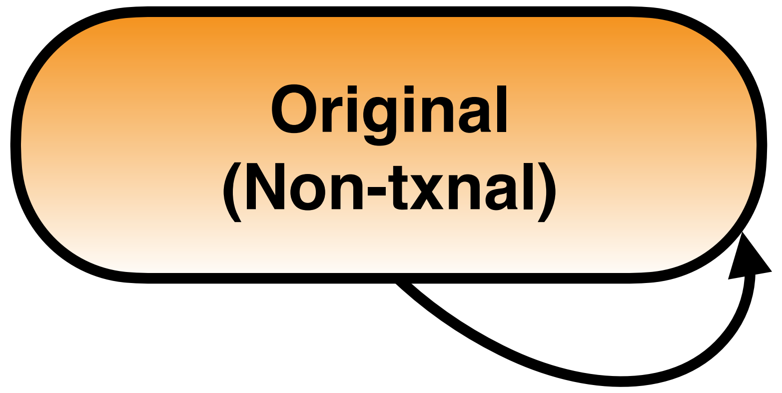


Hybrid TM Method Variants

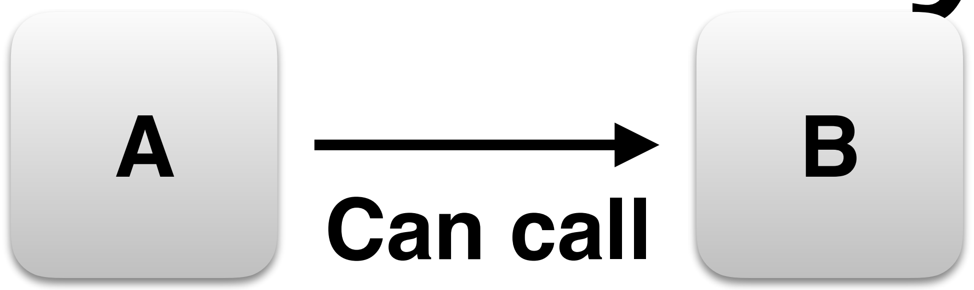
Original
(Non-txnal)

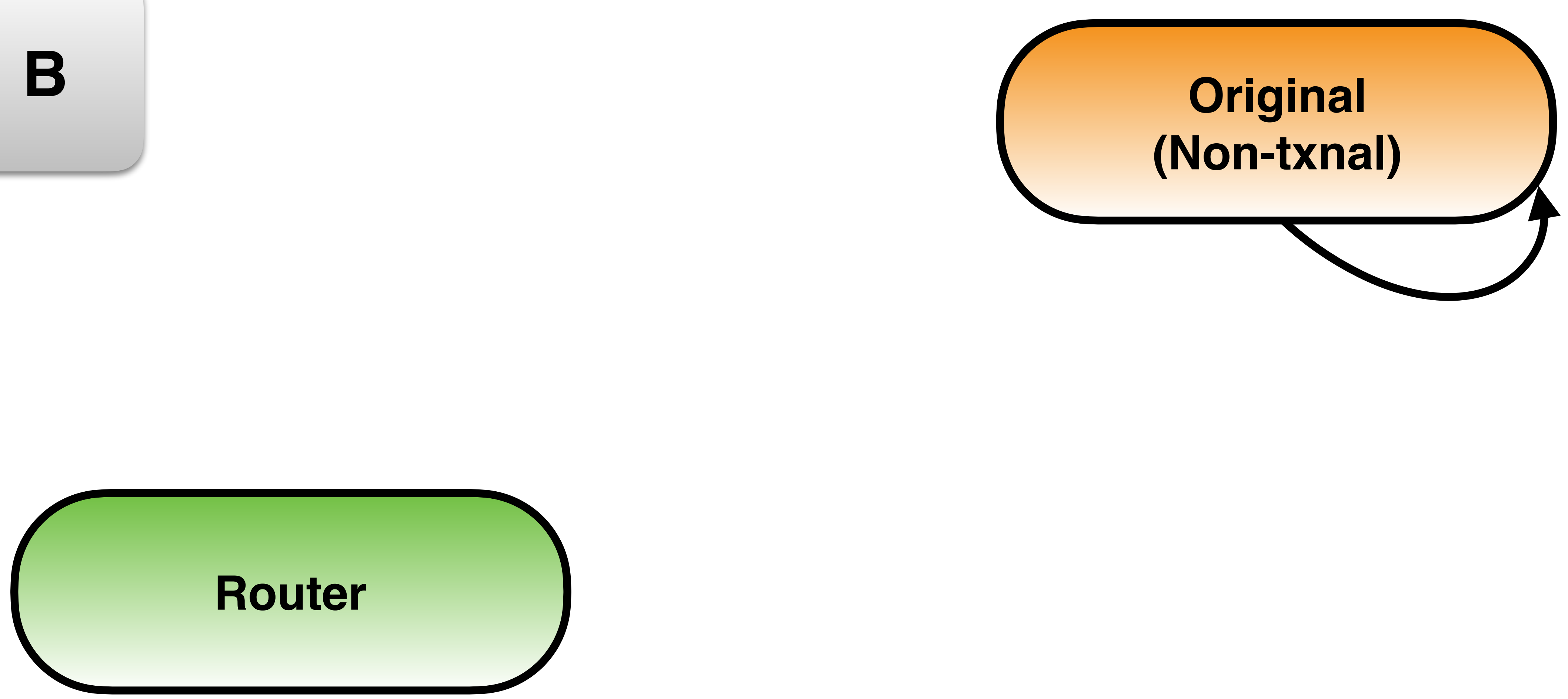
Hybrid TM Method Variants



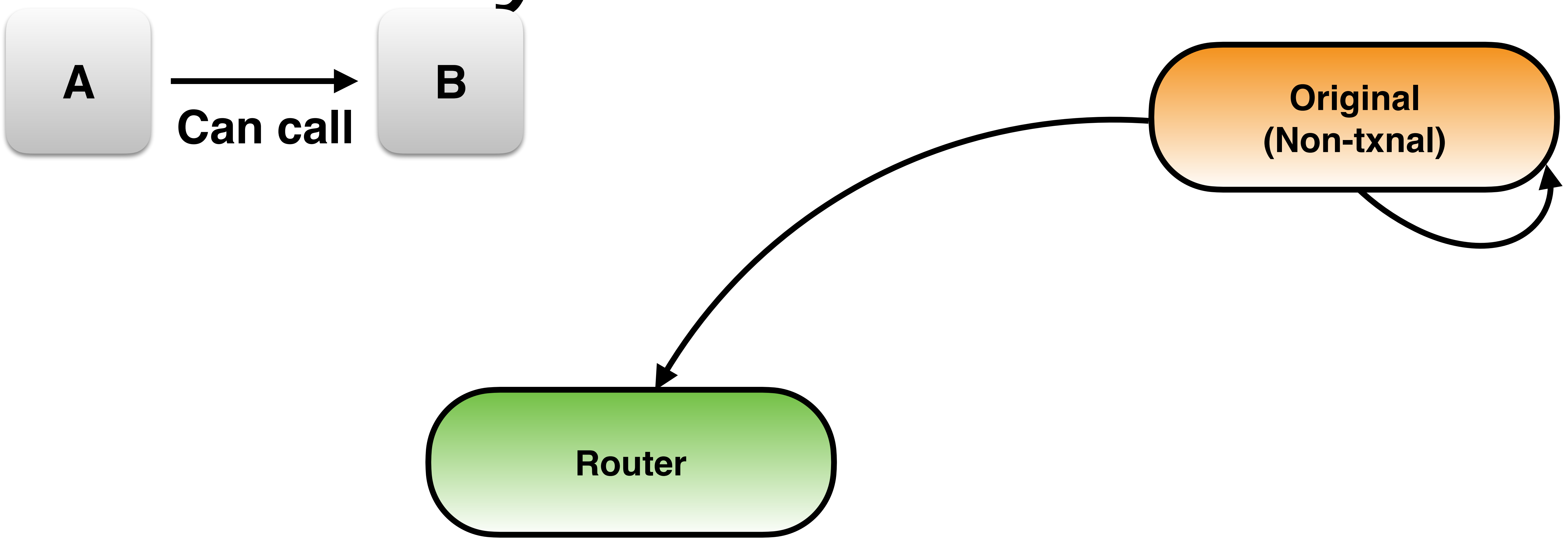


Hybrid TM Method Variants

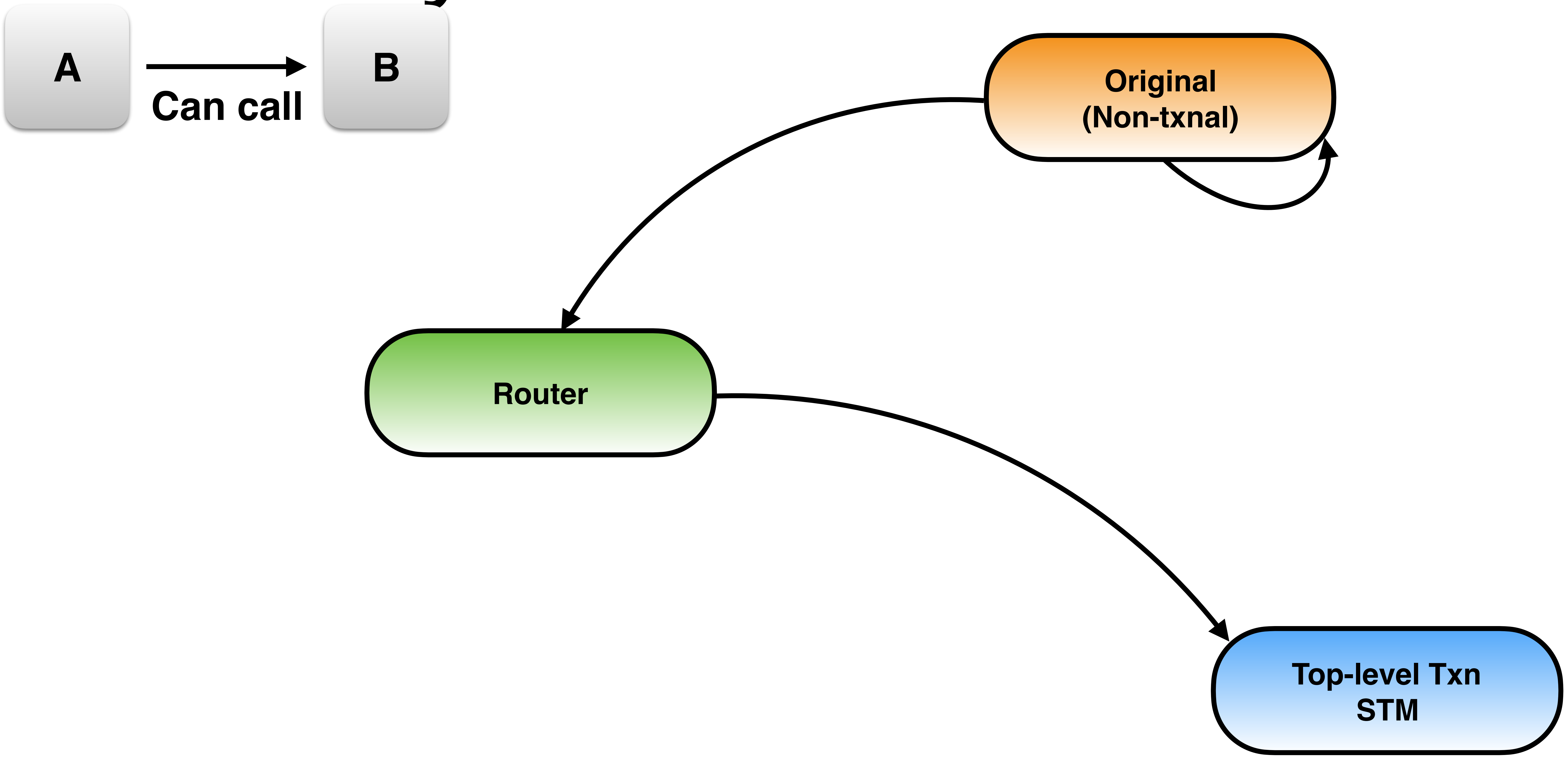




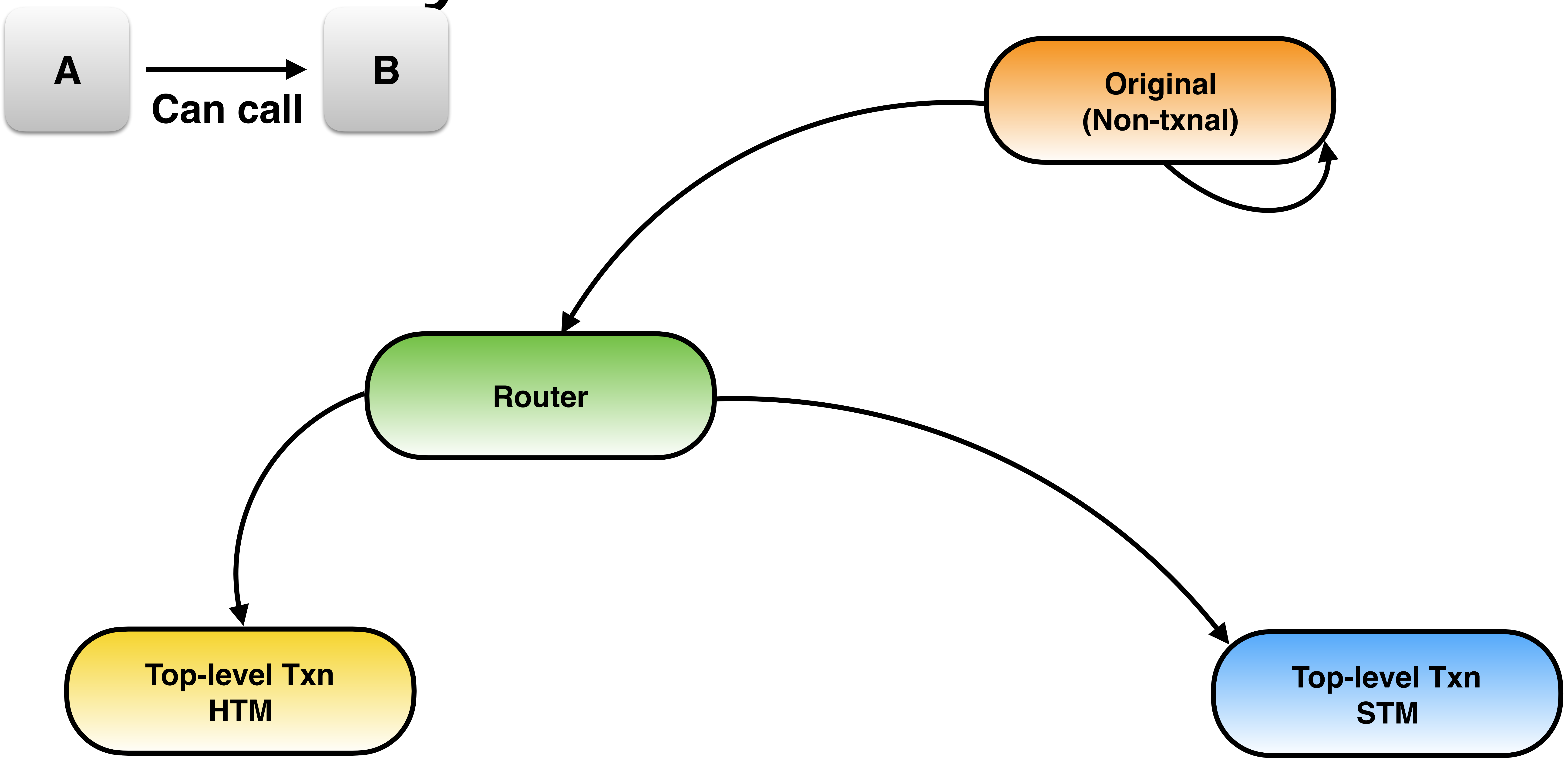
Hybrid TM Method Variants



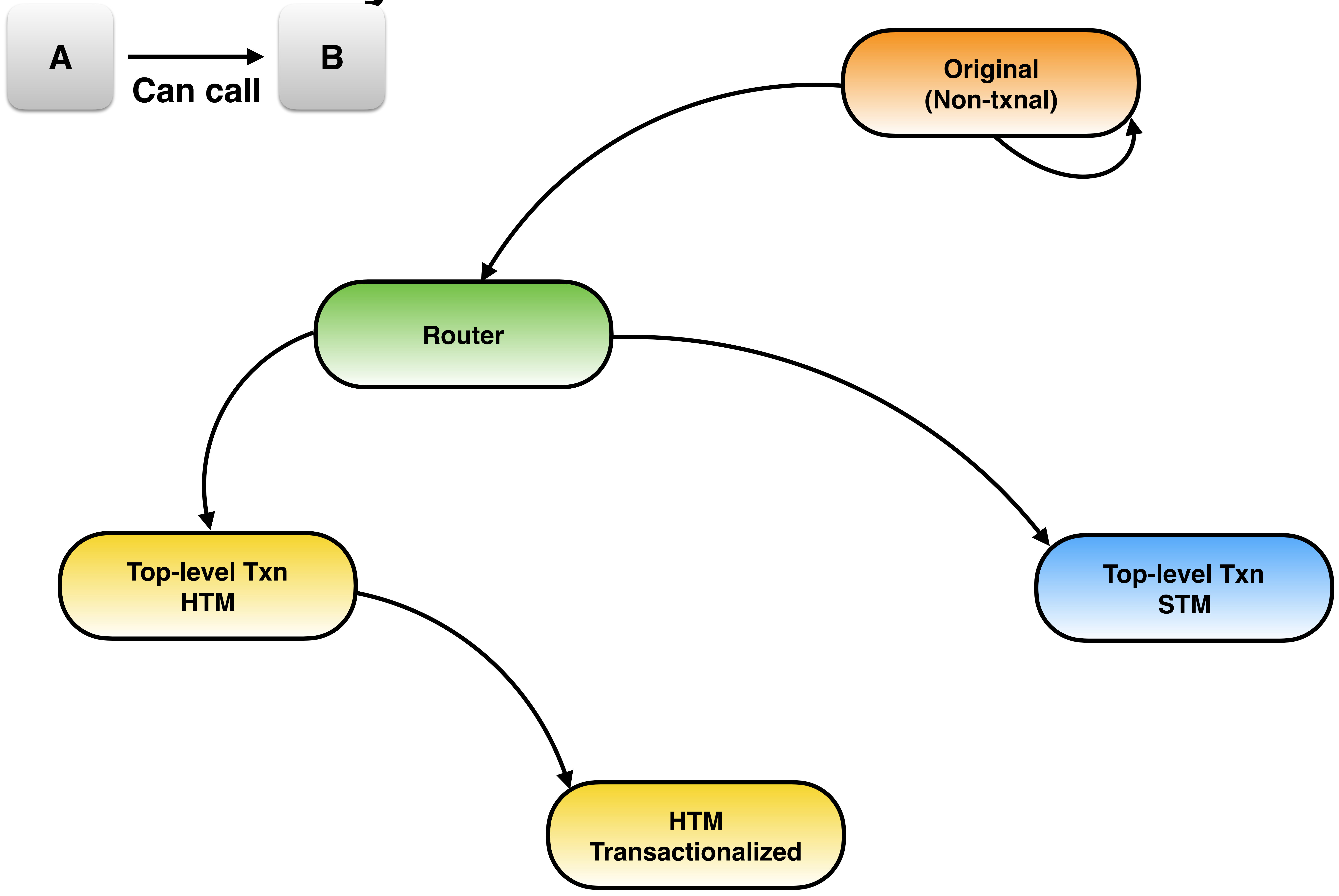
Hybrid TM Method Variants



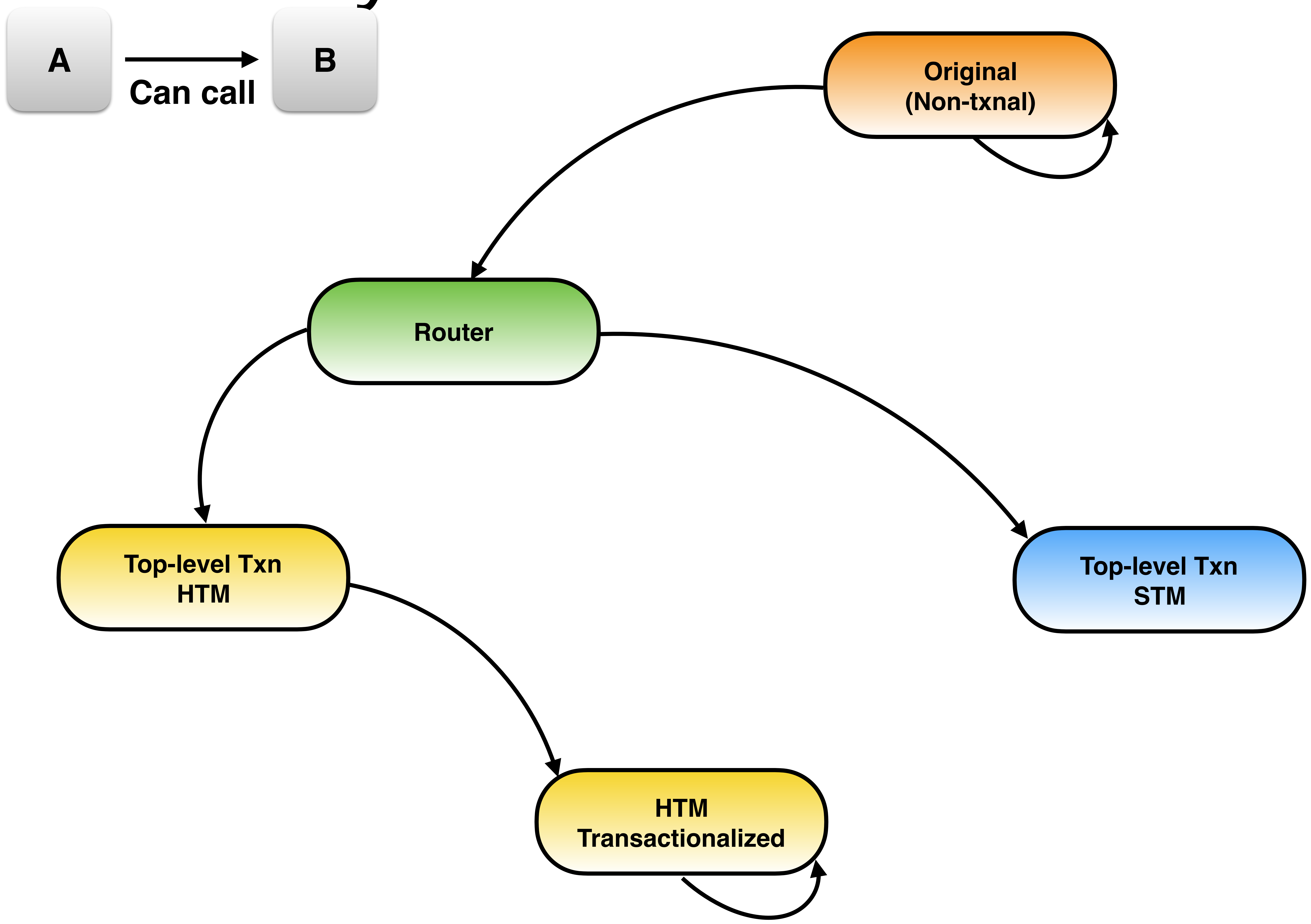
Hybrid TM Method Variants



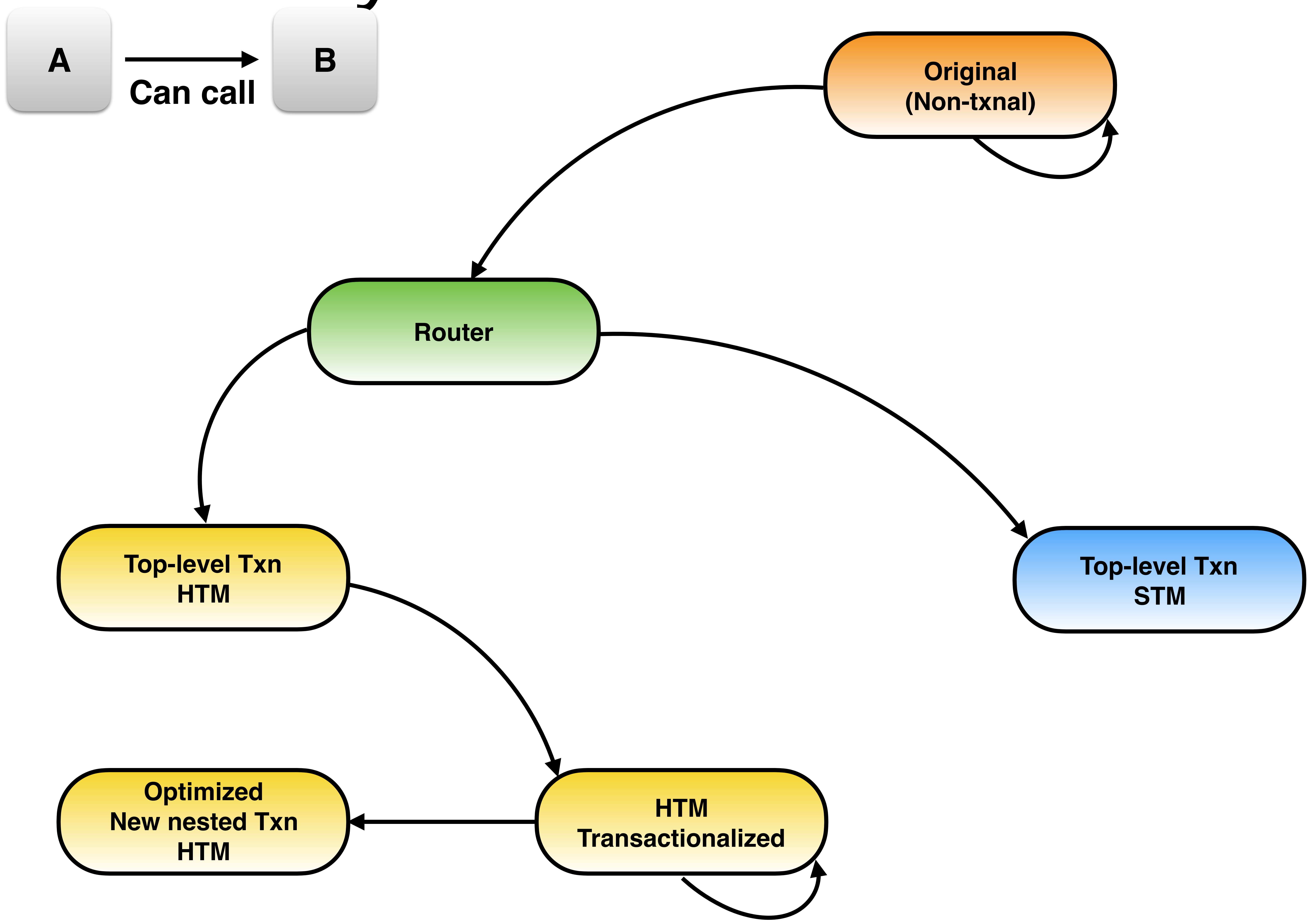
Hybrid TM Method Variants



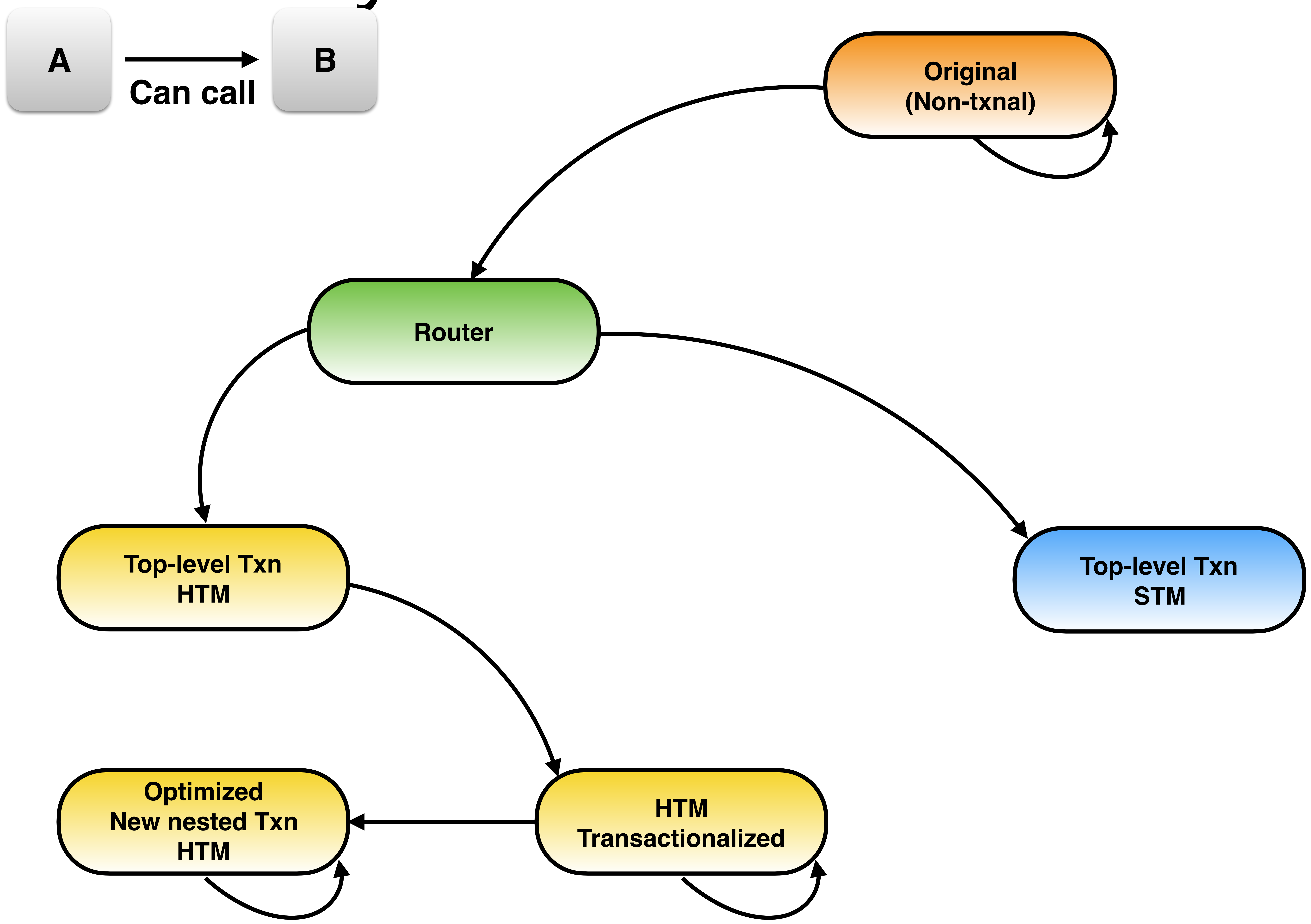
Hybrid TM Method Variants



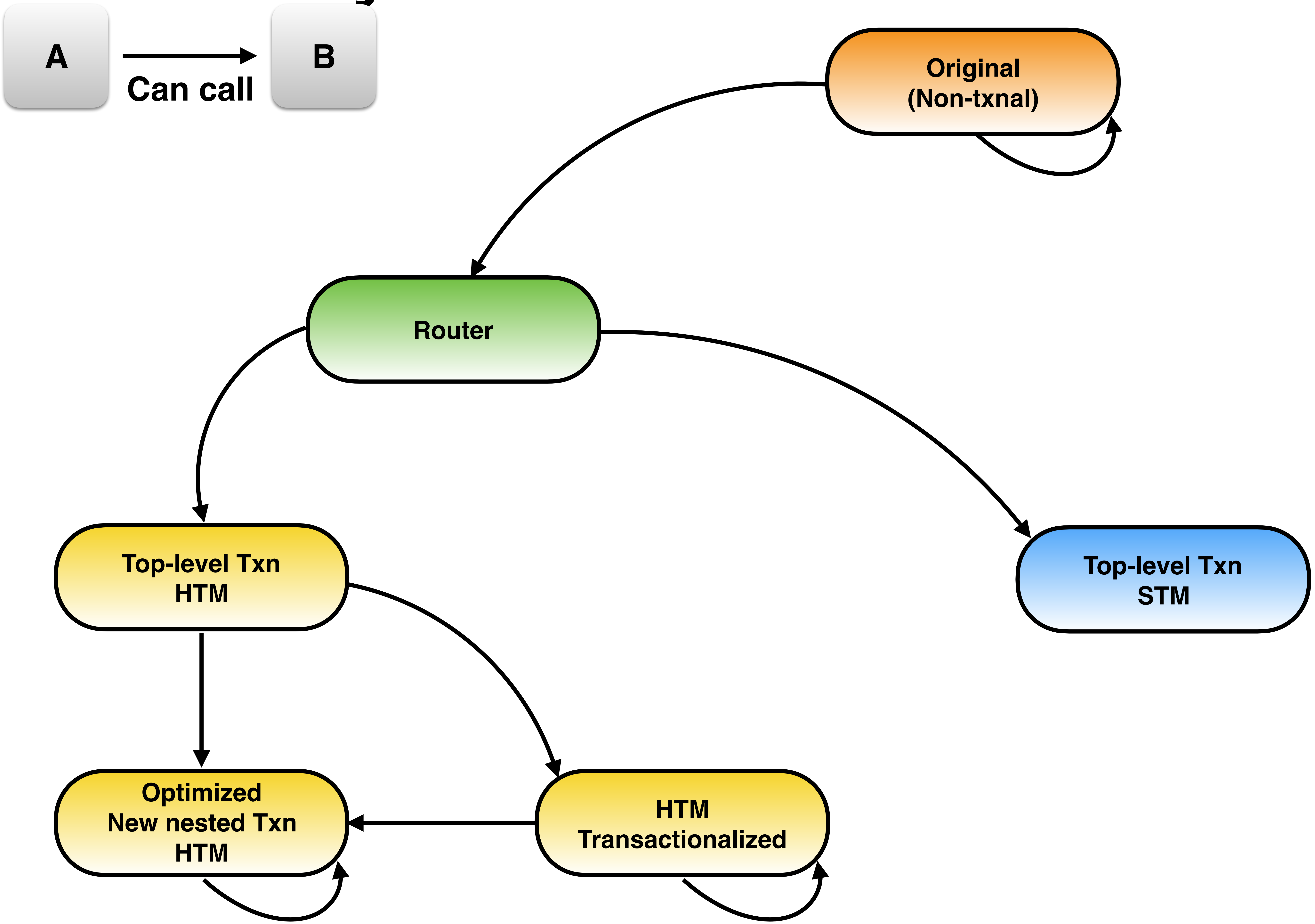
Hybrid TM Method Variants



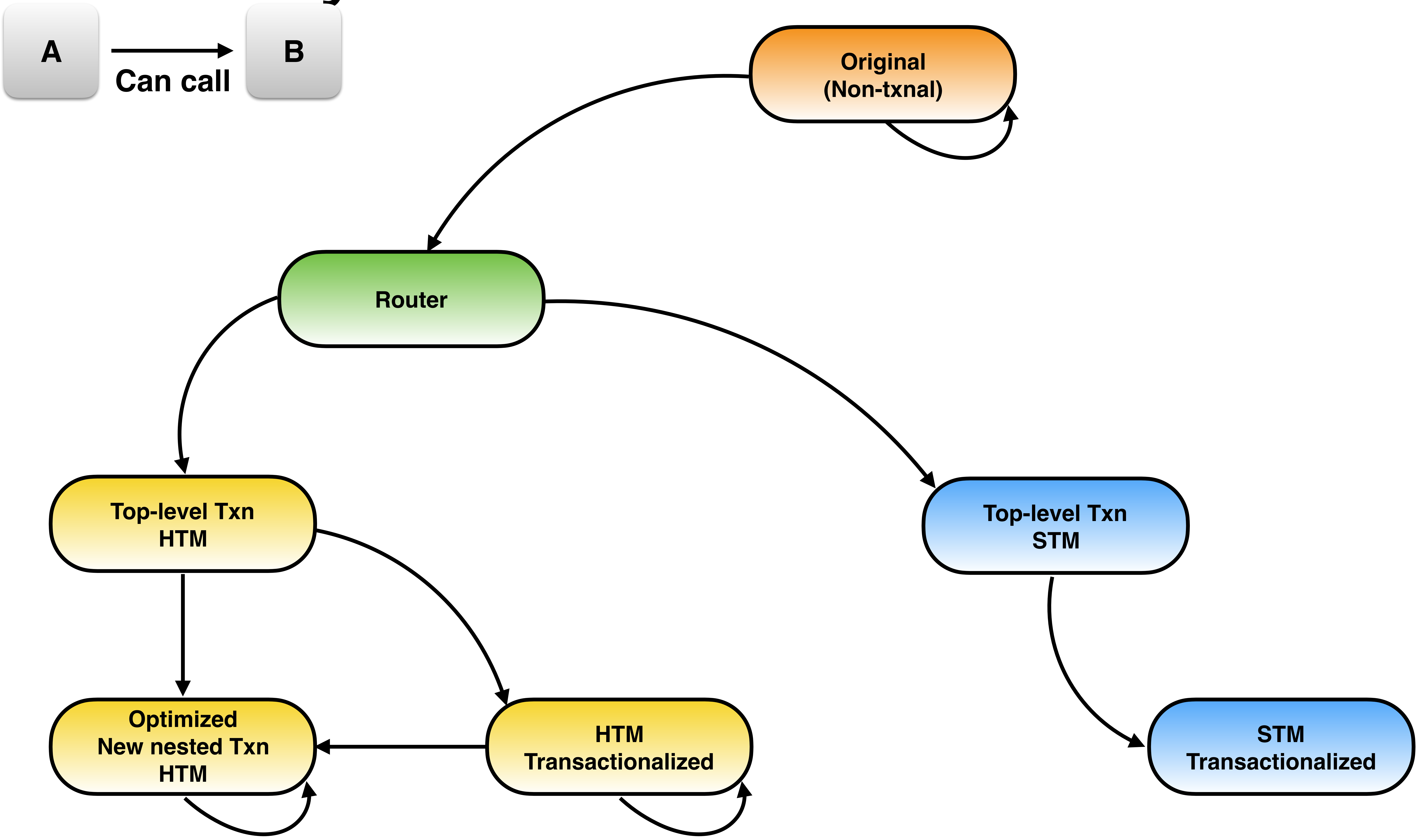
Hybrid TM Method Variants



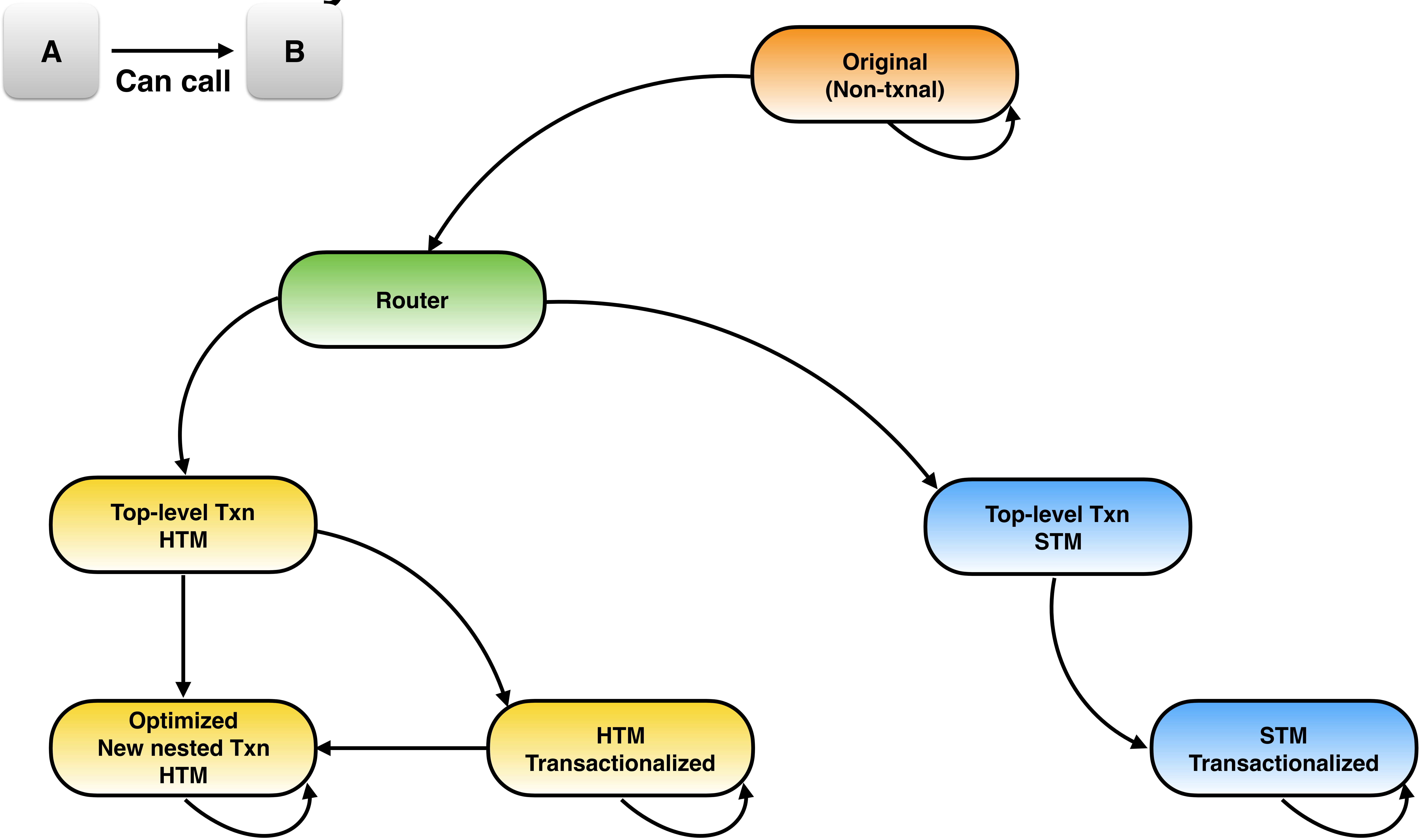
Hybrid TM Method Variants



Hybrid TM Method Variants

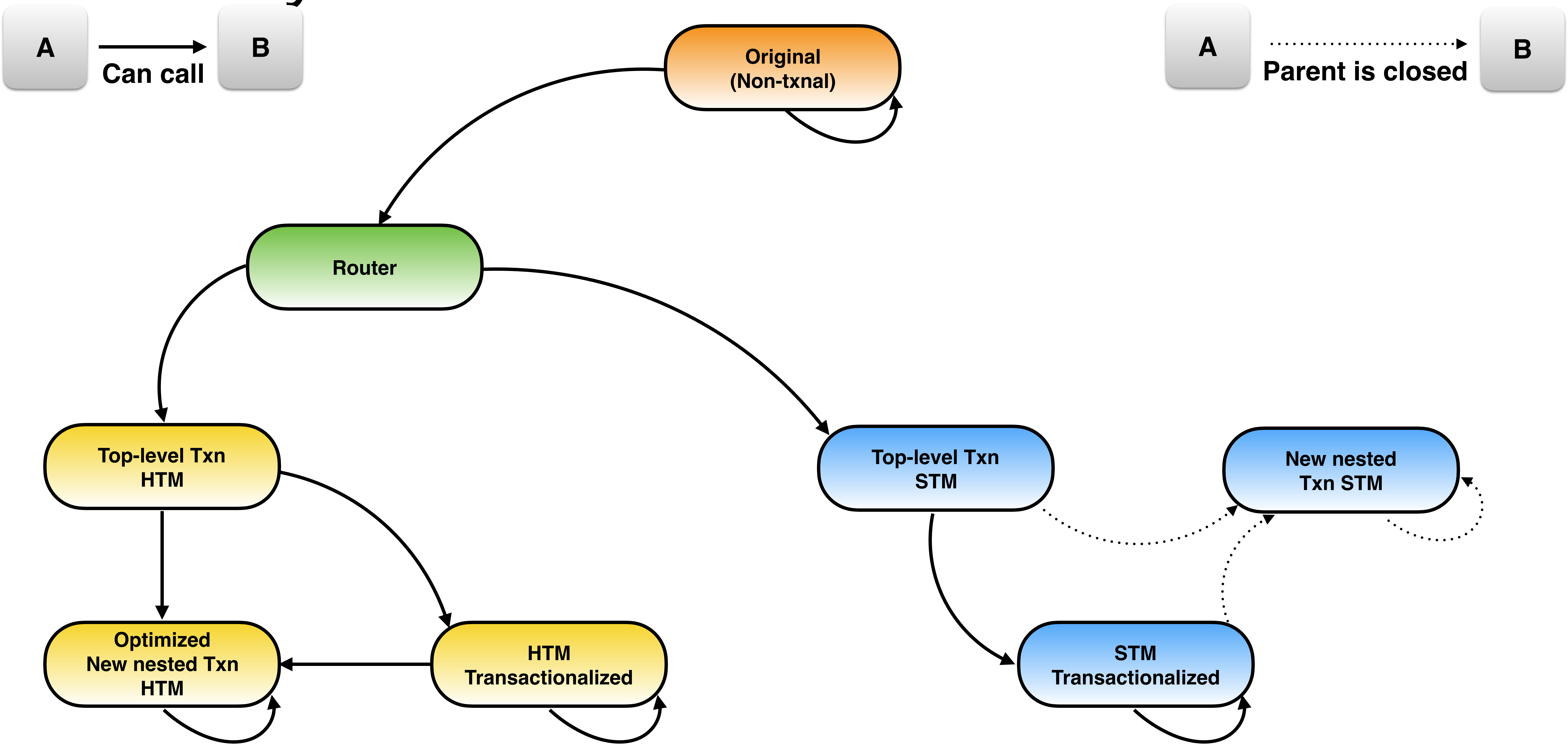


Hybrid TM Method Variants

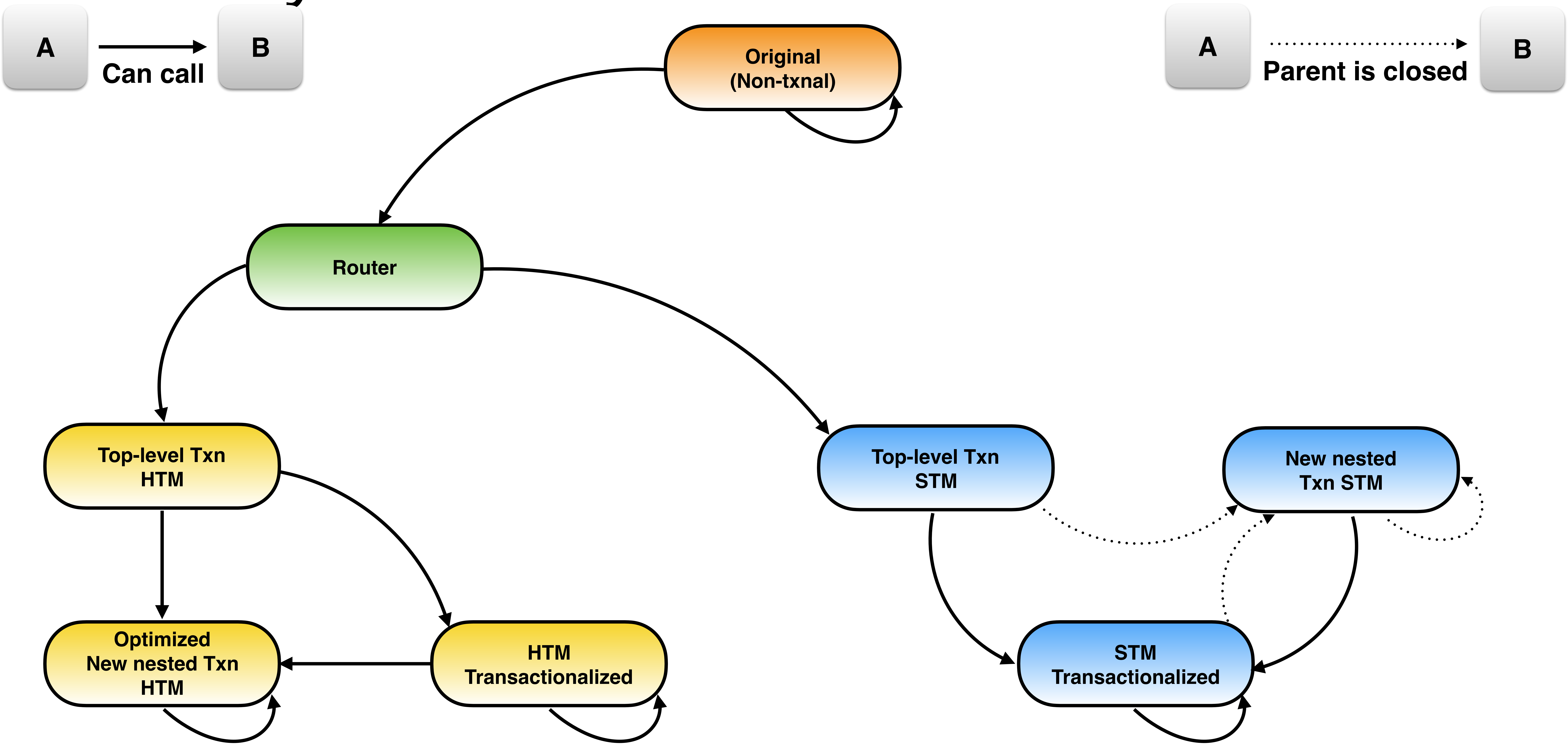


Hybrid TM Method Variants

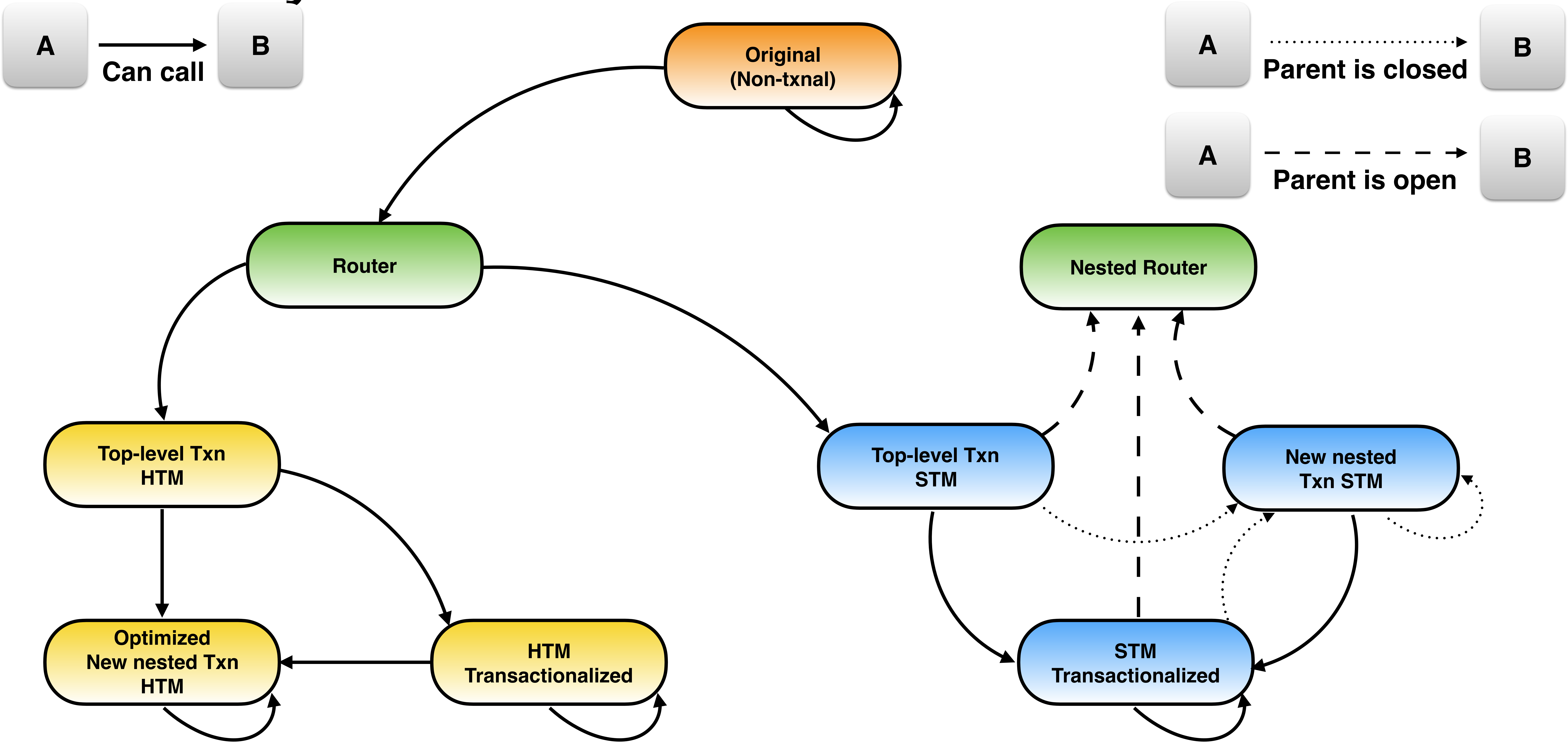
Hybrid TM Method Variants



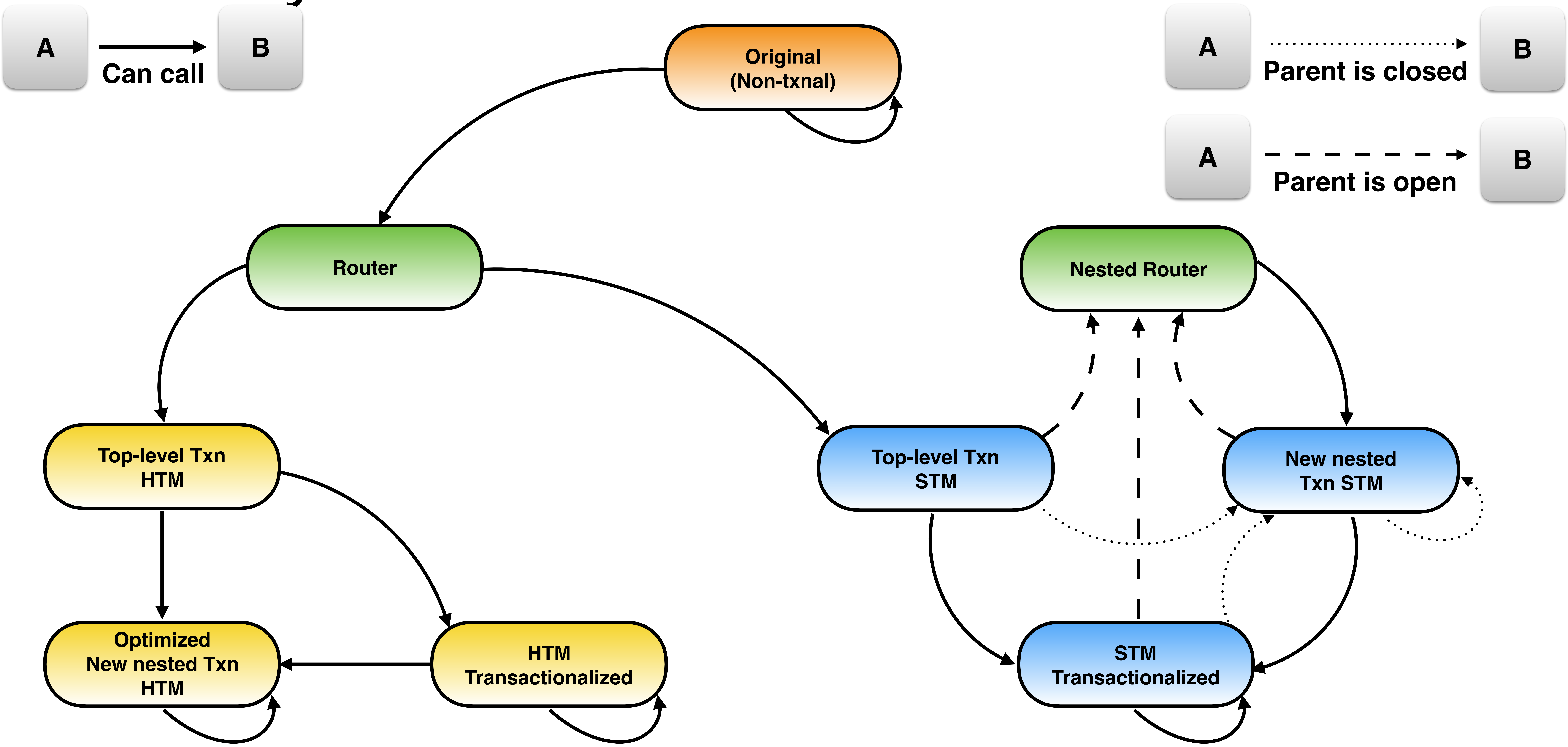
Hybrid TM Method Variants



Hybrid TM Method Variants



Hybrid TM Method Variants



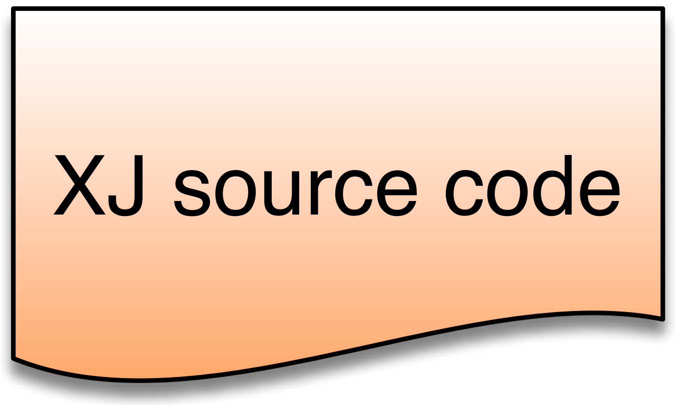
Hybrid TM Method Variants

Hybrid TM Method Variants

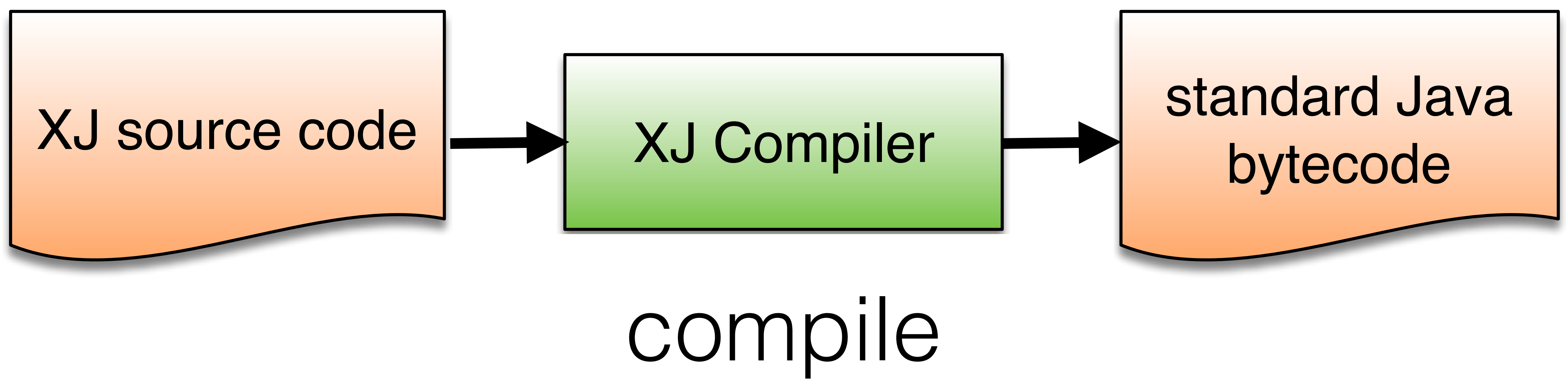
Hybrid TM Method Variants

XJ System Architecture

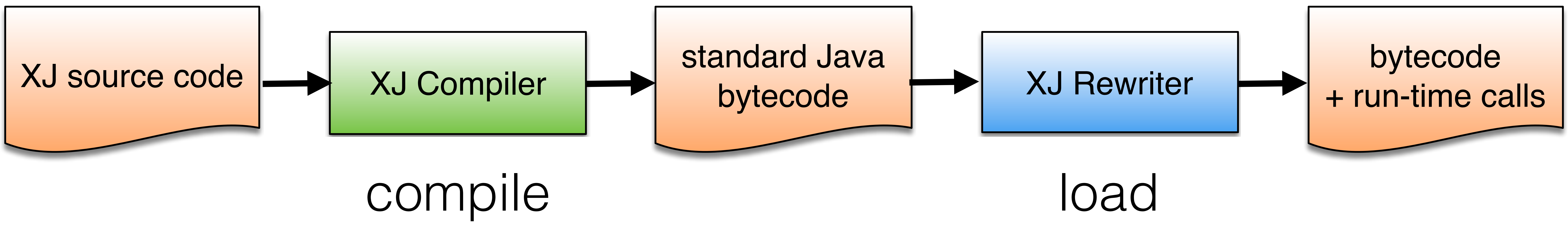
XJ System Architecture



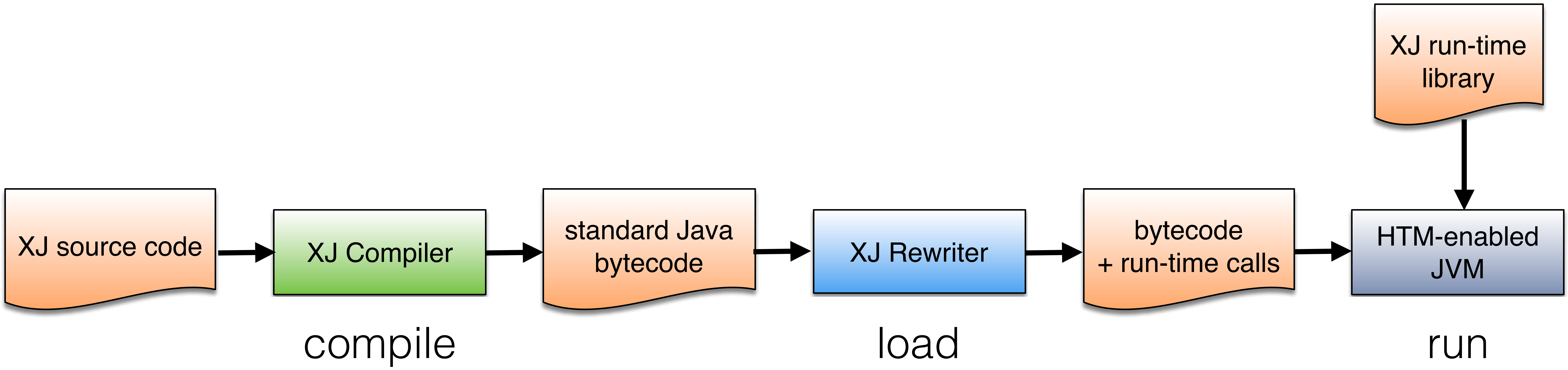
XJ System Architecture



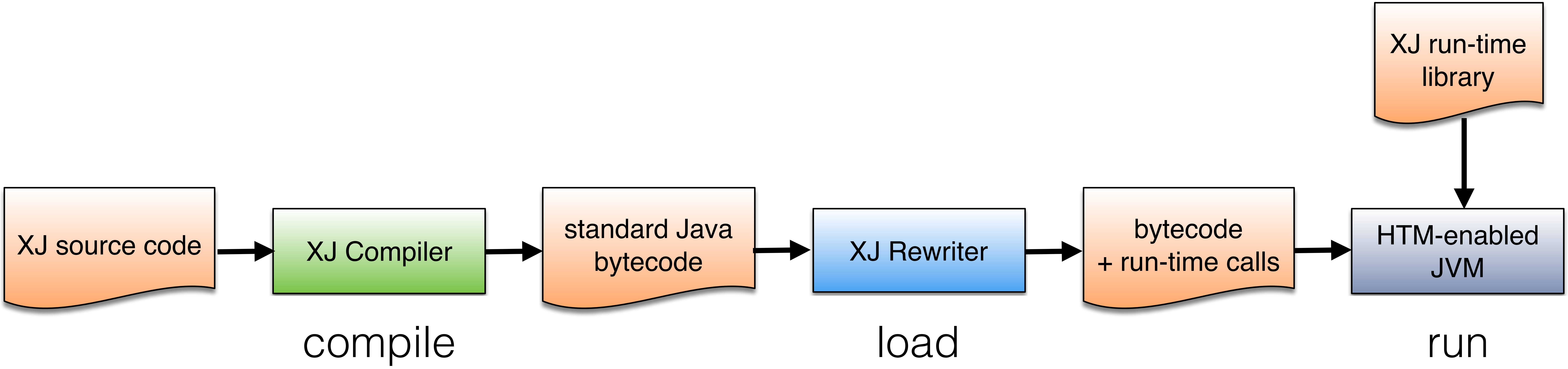
XJ System Architecture



XJ System Architecture



XJ System Architecture



HTM 4-5 times faster than STM

OpenJDK Modifications

Kept to a minimum

- <u>Native methods</u> to begin, end, and abort a HTM transaction
- Made them <u>intrinsic</u> to the HotSpot C1/C2 optimising compilers

Had to jump hoops getting HTM working with the optimising compilers

Results

Synchrobench

Micro-benchmarks to evaluate synchronisation performance on various data structures

Added ability to run multiple operations within a single **transaction** (group size)

Included XJ versions of the benchmarks

- TransactionalFriendlyTreeSet

48-way, Intel Xeon E5-2690 v3 machine with 2 sockets of 12 hyper-threaded cores

Closed nested 5% Updates Open nested

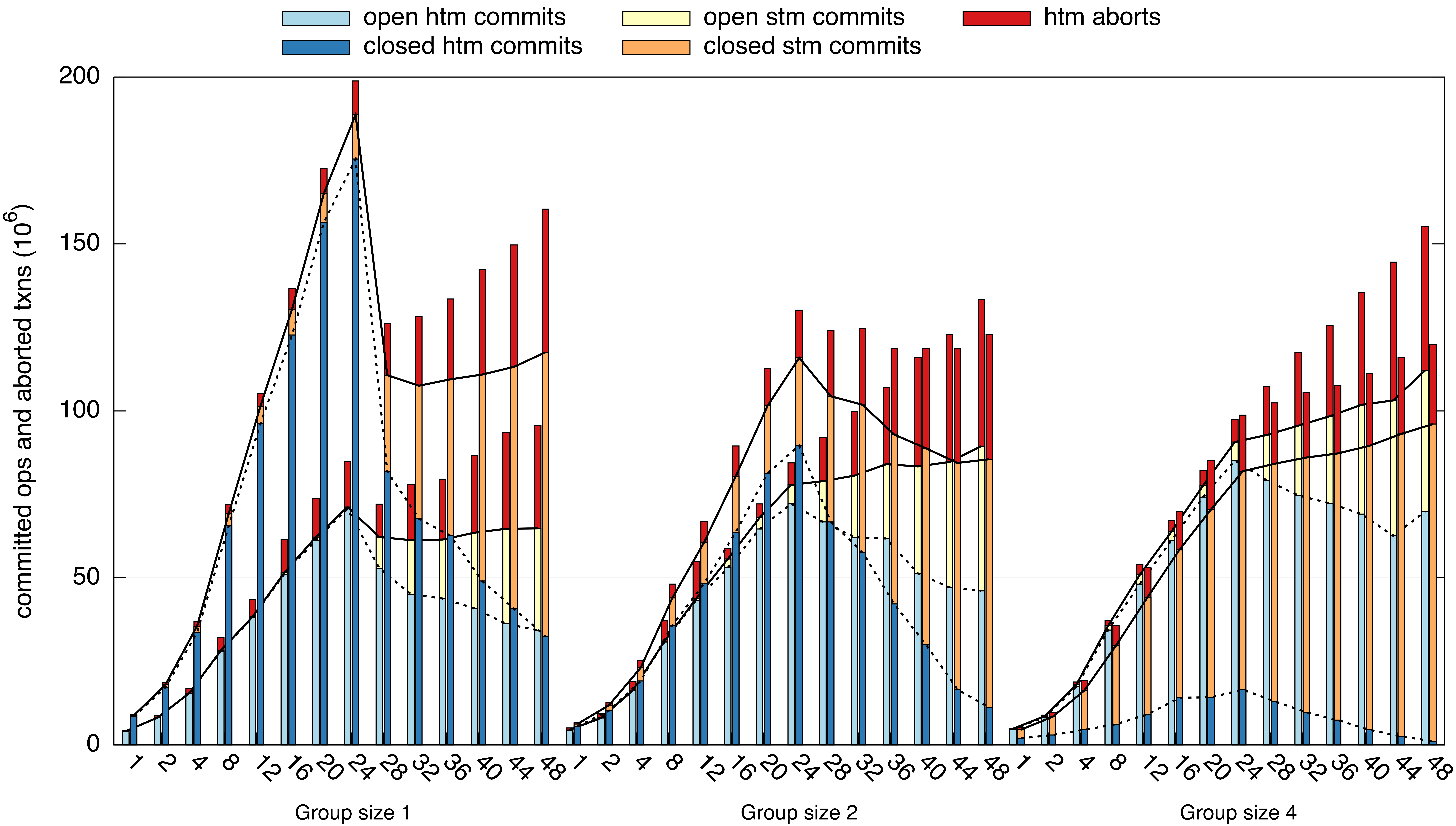

Group size 1
Group size 2
Throughput (Normalized)
Threads
1
0.8
0.6
0.4
0.2
0
1 2 4 8 12 16 20 24 28 32 36 40 44 48
Closed nested
5% Updates
Open nested

Closed nested 5% Updates Open nested

Closed nested 5% Updates Open nested

Closed nested 5% Updates Open nested

Closed nested 5% Updates Open nested


open htm commits
closed htm commits
open stm commits
closed stm commits
htm aborts
committed ops and aborted txns (10^6)
0
50
100
150
200
1 2 4 8 12 16 20 24 28 32 36 40 44 48
Group size 1
Group size 2
Group size 4

Conclusions

STM and HTM can co-exist for nested transactions in Java

- Closed nesting — Similar to previous schemes
- Open nesting — Novel validation mechanism
- Implemented in OpenJDK on Intel TSX — Artifact evaluated

Conclusions

STM and HTM can co-exist for nested transactions in Java

- Closed nesting — Similar to previous schemes
- Open nesting — Novel validation mechanism
- Implemented in OpenJDK on Intel TSX — Artifact evaluated

Conclusions

STM and HTM can co-exist for nested transactions in Java

- Closed nesting — Similar to previous schemes
- Open nesting — Novel validation mechanism
- Implemented in OpenJDK on Intel TSX — Artifact evaluated

When it works, HTM is ~4-5× faster than STM

Conclusions

STM and HTM can co-exist for nested transactions in Java

- Closed nesting — Similar to previous schemes
- Open nesting — Novel validation mechanism
- Implemented in OpenJDK on Intel TSX — Artifact evaluated

When it works, HTM is ~4-5× faster than STM

Open nesting increases the envelope of effectiveness for HTM

Conclusions

STM and HTM can co-exist for nested transactions in Java

- Closed nesting — Similar to previous schemes
- Open nesting — Novel validation mechanism
- Implemented in OpenJDK on Intel TSX — Artifact evaluated

When it works, HTM is ~4-5× faster than STM

Open nesting increases the envelope of effectiveness for HTM

Production VM would need deeper modification