

Proposal of Transaction on RAMCloud

rev0.64

24 Oct. 2013

Satoshi Matsushita

Objectives

- Introduce "**Transaction**" to RAMCloud
- What is "**Transaction**" ?
 - Wikipedia 'Database Transaction':
 - To provide **reliable units of work** that allow correct recovery from failures and **keep a database consistent** even in cases of system failure, when execution stops (completely or partially) and many operations upon a database remain uncompleted, with unclear status.
 - To provide **isolation** between programs accessing a database **concurrently**. If this isolation is not provided, the program's outcome are possibly erroneous.
- User declares a partial sequence of data (object) access as "**a Transaction**", to which RAMCloud provides 'Database Transaction' feature.

Characteristics of Transaction

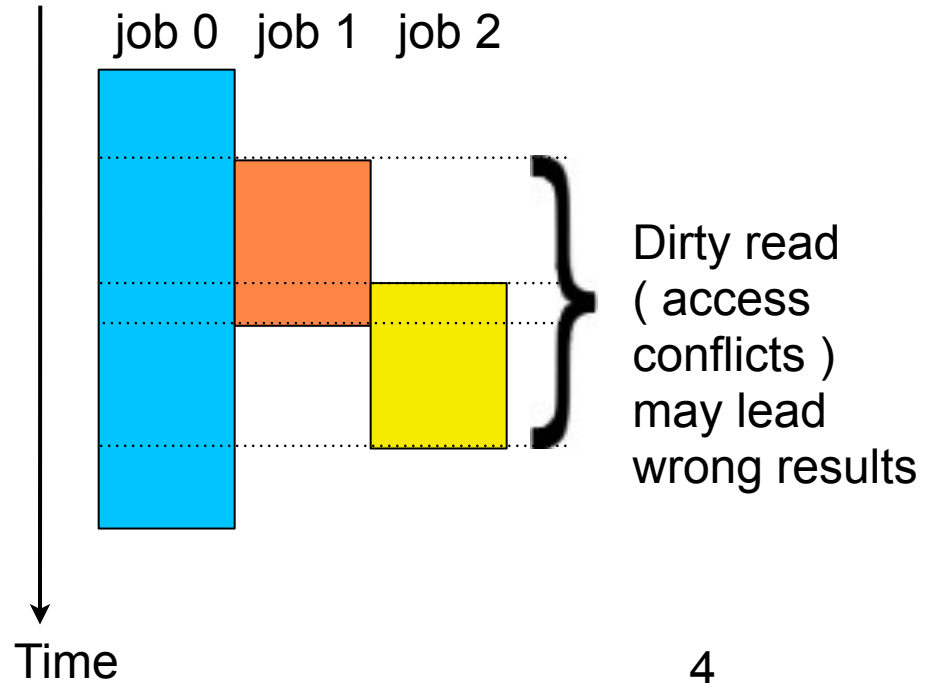
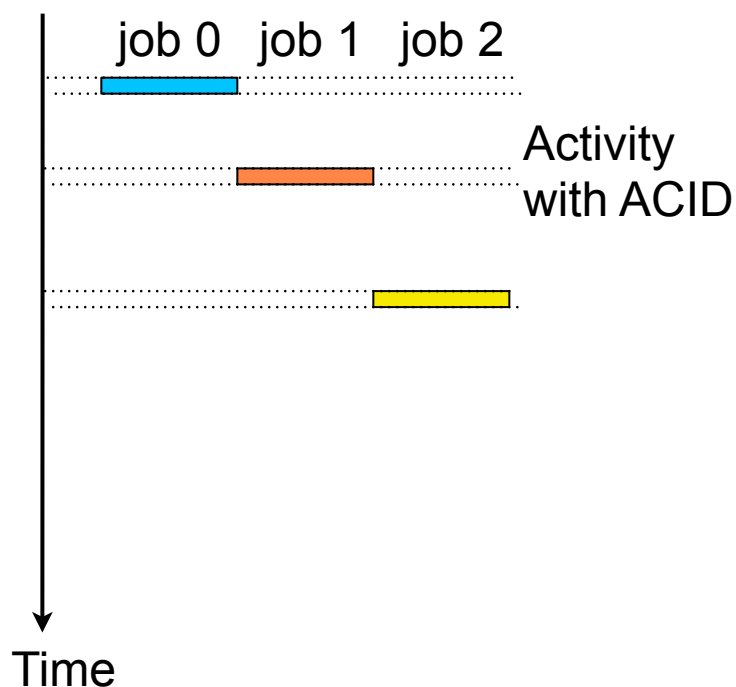
1. Duration varies from short to long: 0.1ms to 100ms
2. Very small chance of conflict to other transactions
3. Too many conflicts are data/control design issue

Example	Duration	Chance of Conflict
Analytic (Data analysis)	min. to hours	none after start
Ticket or seat reservation	to a few sec	small
Banking	to a few sec	small, at money transfer
Online shopping	to a few sec	small, can split to many independent
Stock trading	1 to 100ms	small or medium
SNS	100 to 1000 ms	small
Other web services	100 to 1000 ms	small

Issues in Parallel Execution

- Resource access conflict occurs in parallel execution
- Requirement to avoid the problem (Reality)
 - ACID: (atomicity, consistency, isolation, durability)
 - CAP Theorem: (Can relax partition tolerance) - discuss later

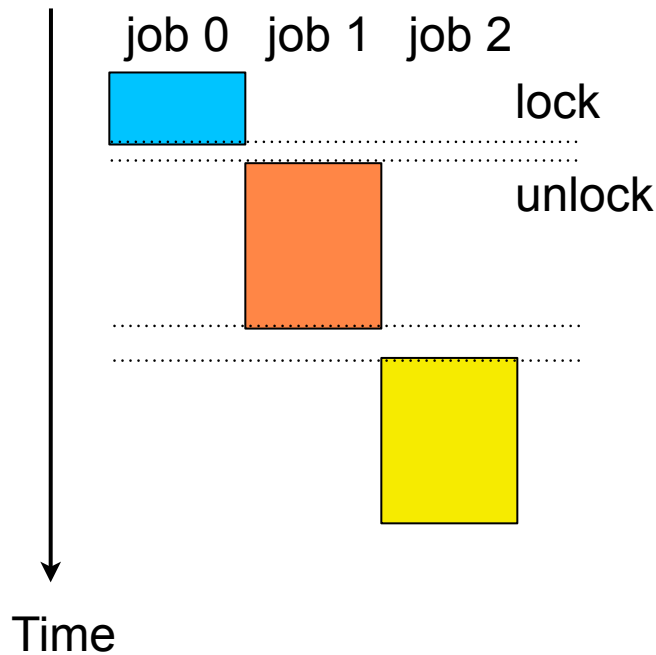
Ideal Parallel Execution)



4

Conflict Solutions

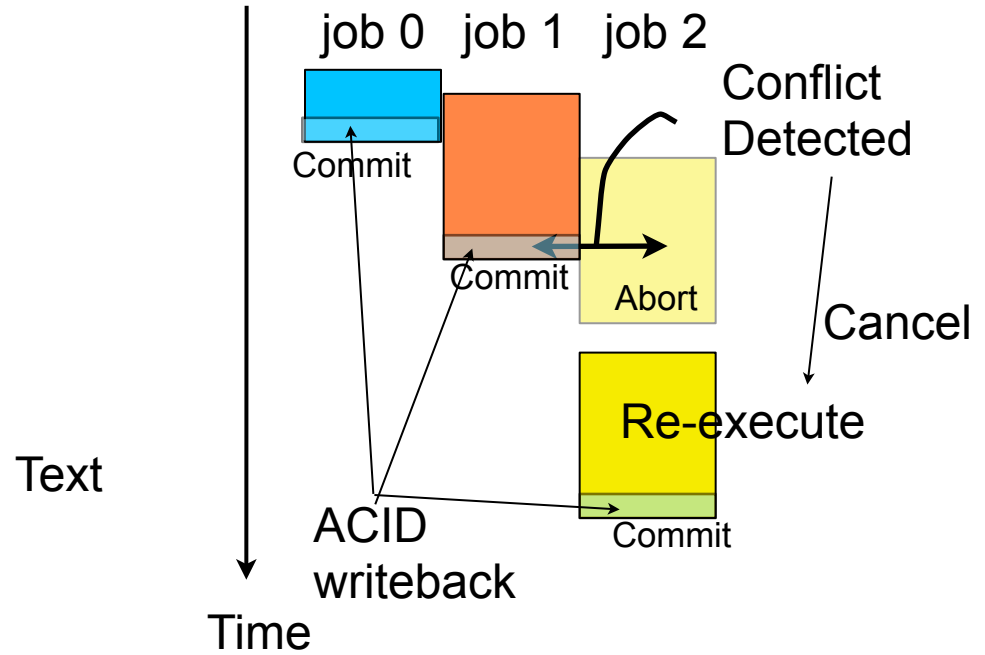
Pessimistic Lock)



Pros & Cons)

- Lower parallelism with giant locks
- Dead lock prone with fine locks
- Need releasing lock with node crash

Optimistic Lock)

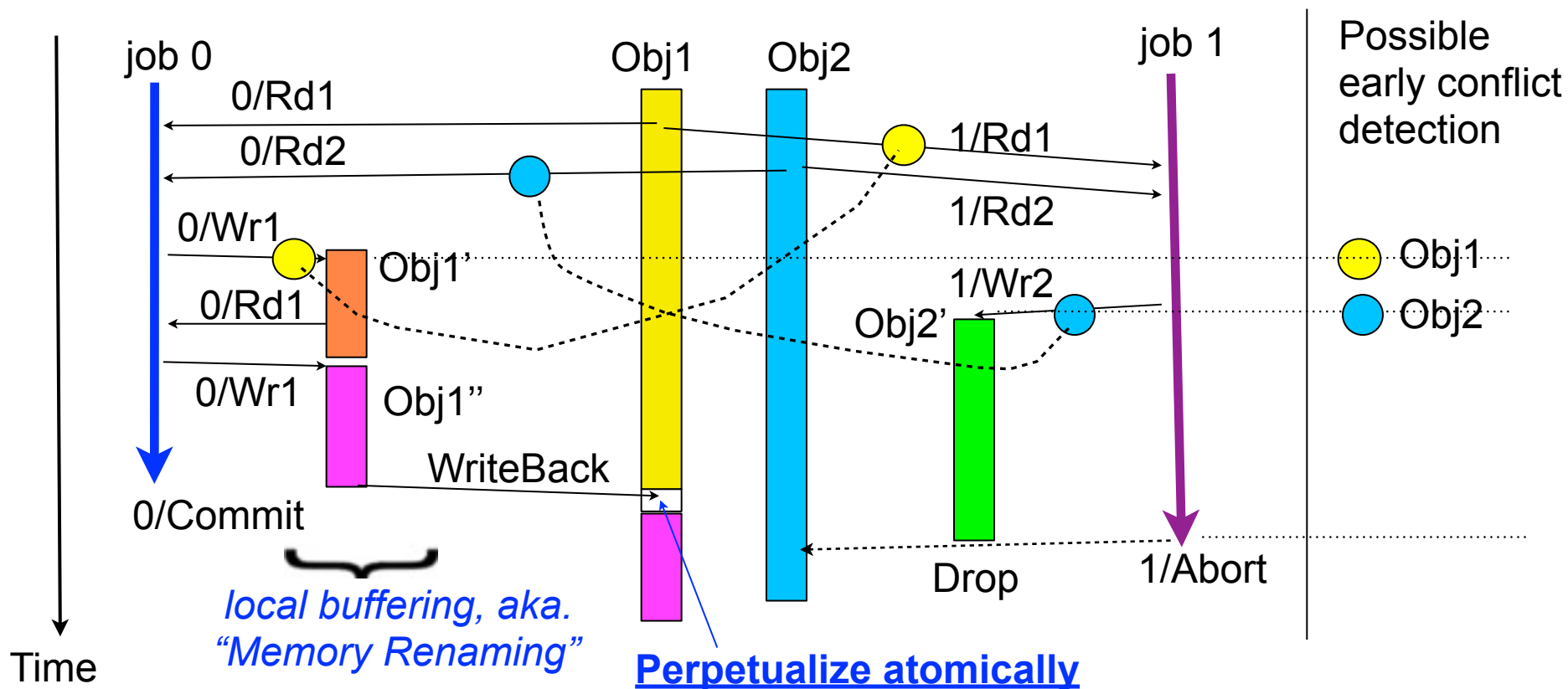


Pros & Cons)

- Need conflict detection logic
- Lower Performance loss by frequent conflicts
- Alternatives in abort detection

Optimistic Lock: General Solution

- Conflict detection of true dependencies: RAW (Read after Write)
- Renaming false dependencies : WAR, WAR
 - Common technique in parallel execution such as Speculative MT, Transactional Mem., RDBM



Assumptions and Strategies

Application Specific)

- Transaction life varies between **short to long**
 - Try early conflict detection avoiding livelock
- **Small probability** of conflicts
 - Use optimistic lock based design
 - Otherwise create pessimistic lock at user level
- Well designed application shares appropriate amounts of data in a transaction
 - Involve small number of nodes to **reduce probability of relevant node crash** for a transaction

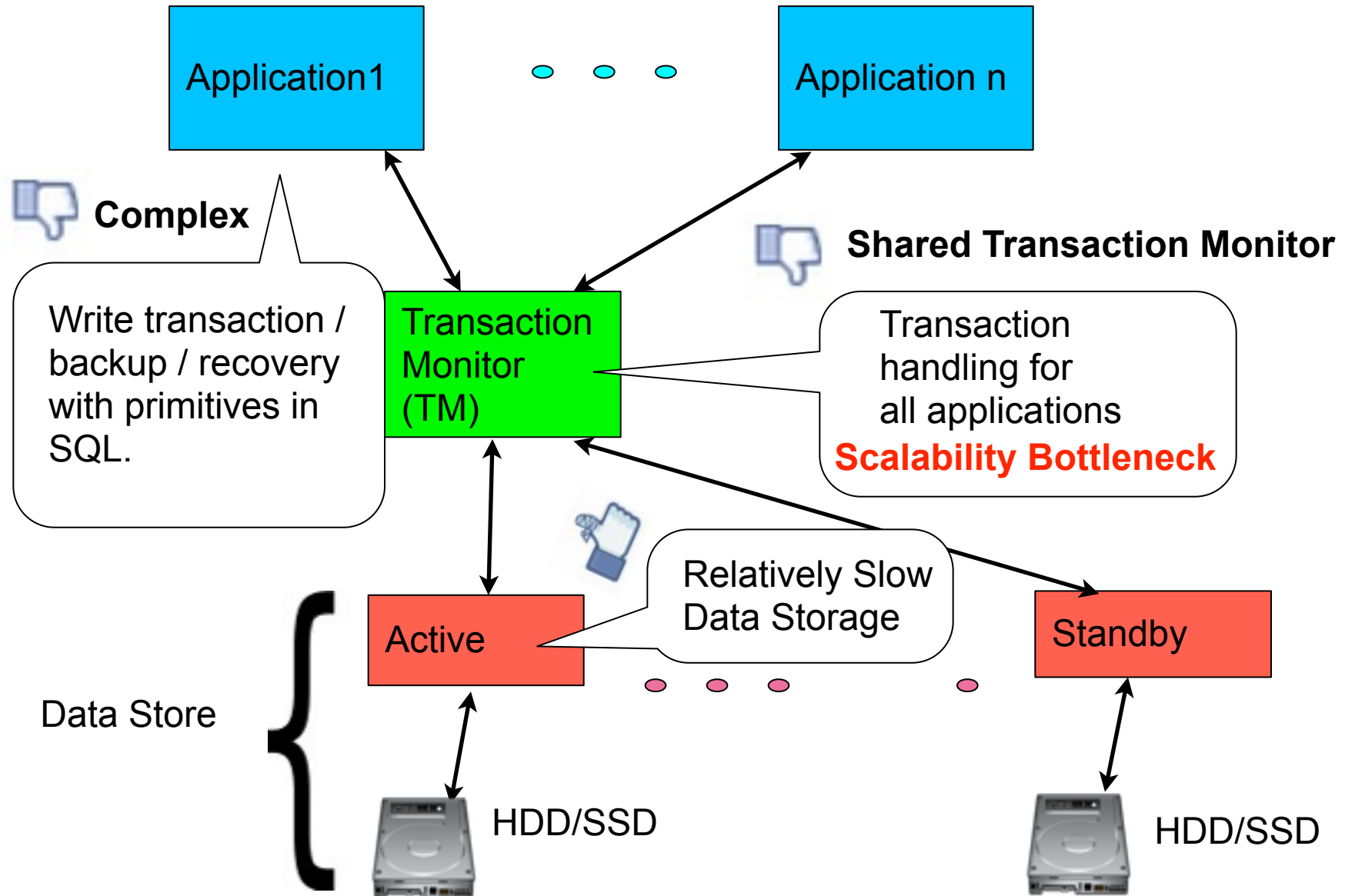
RAMCloud Specific)

- Faster crash recovery around 1 sec
 - Can yield to **blocking algorithm** without corner cases
- **A separate log** on each master - advantage of **scalability**

Note)

- CAP Theorem
 - Means: Consistency, Availability, Partition-tolerance
 - RAMCloud natively does not have partition tolerance, only the partition where coordinator exists works.
- Multiphase Commit
 - If we can allow waiting for node recovery, two phase commit works.
 - Since the blockage is not realistic, couple of non-blocking commit algorithm have been introduced:
 - Consensus (Paxos, Raft): Always live majority hides node crash
 - Multiphase Commit - prevent commit blockage
 - Quorum Commit: Majority side works during partitioning
 - Three phase commit - still it is not easy to detect failure mode.
 - Paxos commit, etc

Traditional Transaction System



Traditional Transaction: Sharding

- Distribute database into several servers for scalability
- Micro-Sharding: implement SQL's transaction on KVS by confining a transaction related fields in a single row.

Problems)

- Need a good design of fields in record
- Not always possible to allocate independent sharding

To Be Added

Ref: Microsharding: Mapping Relational Workloads on Key-Value Stores,
Junichi Tatemura, Hakan Hacigumus, et. al., NEC Lab. America

Traditional Transaction: Sinfonia

To Be Reviewed

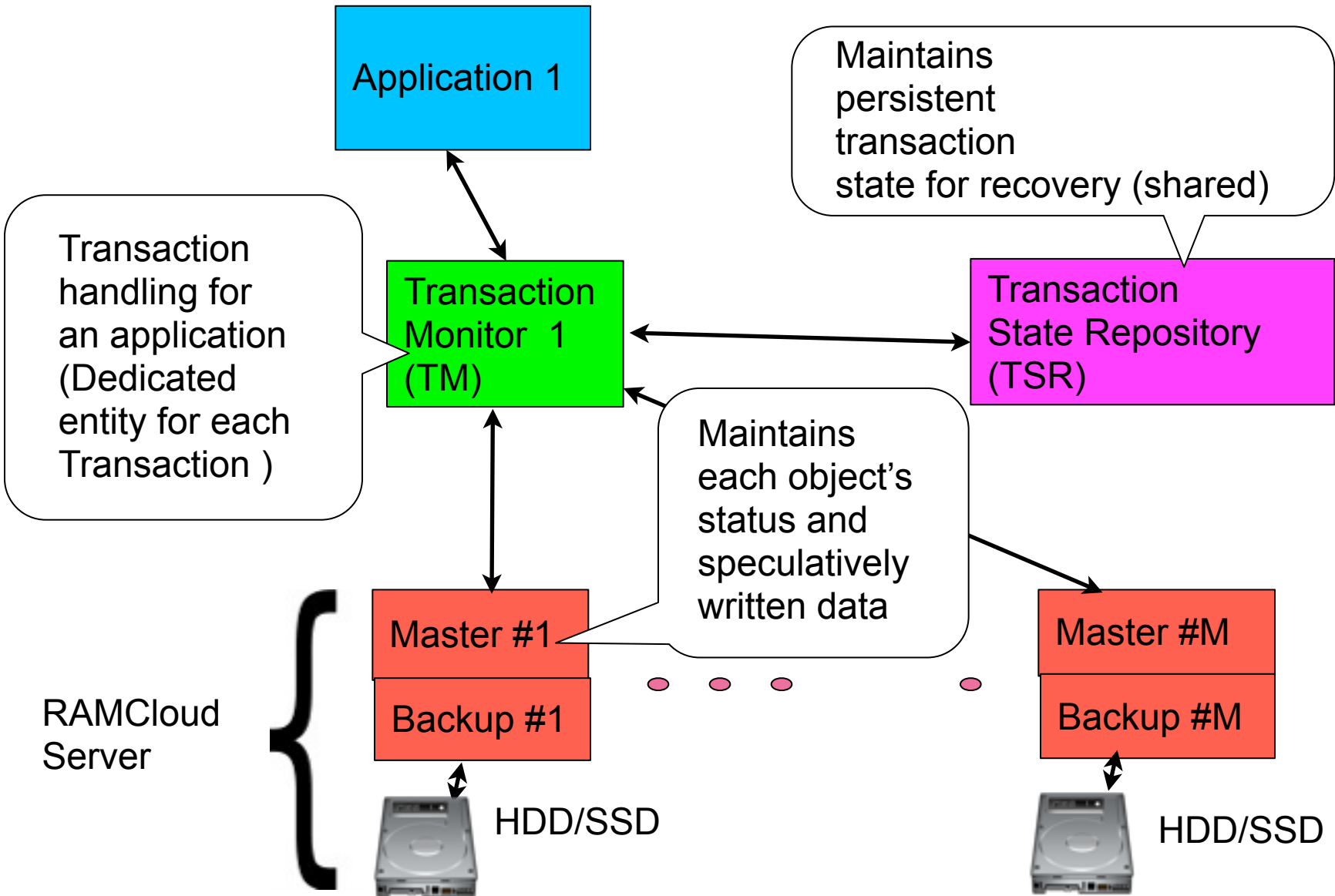
- Distributed design: user library manages transaction - distributed transaction monitor
- Light: memory based, so that fail detection and abort
 - Memory node recovery with redo-log
 - Abundant APIs
- Static: pack compare/write data in an operation
 - Two phase commit
 - Compare and conditional store at commit time
- Trans. coordinator recovery mechanism not included
 - Node failure detection
 - Recovery coordinator for coordinator crash

Ref: Sinfonia: A New Paradigm of Building Scalable Distributed Systems, Macros K. Aguilera (HP Lab.), et. al. , SOSP, Oct. 07

Proposal: Key Idea

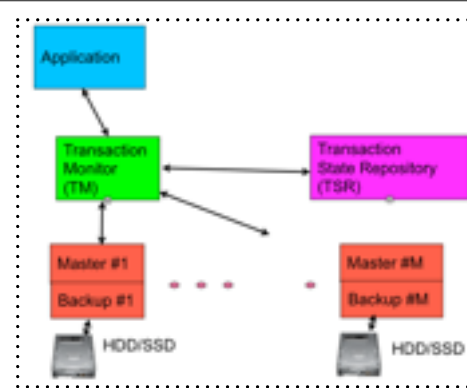
- Distributed TM (transaction monitor) for **scalability**
 - Library based design for **low latency**
- **Integrated crash recovery**
 - triggered by RAMCloud coordinator which is always available by consensus algorithm
- Taking advantage of **distributed log** in RAMCloud master
 - Natively all the checkpoints are available and durable
- **Natural** transaction API
 - **Dynamic**: enclose a part of native code with StartTx / {Commit,Abort} primitive - needs bad code safety
 - **Smaller design efforts** in database scheme
 - **Not expose**:
 - Node crash/recovery
 - Data structure such as log, checkpoint

Proposal: Components



Components - Functions

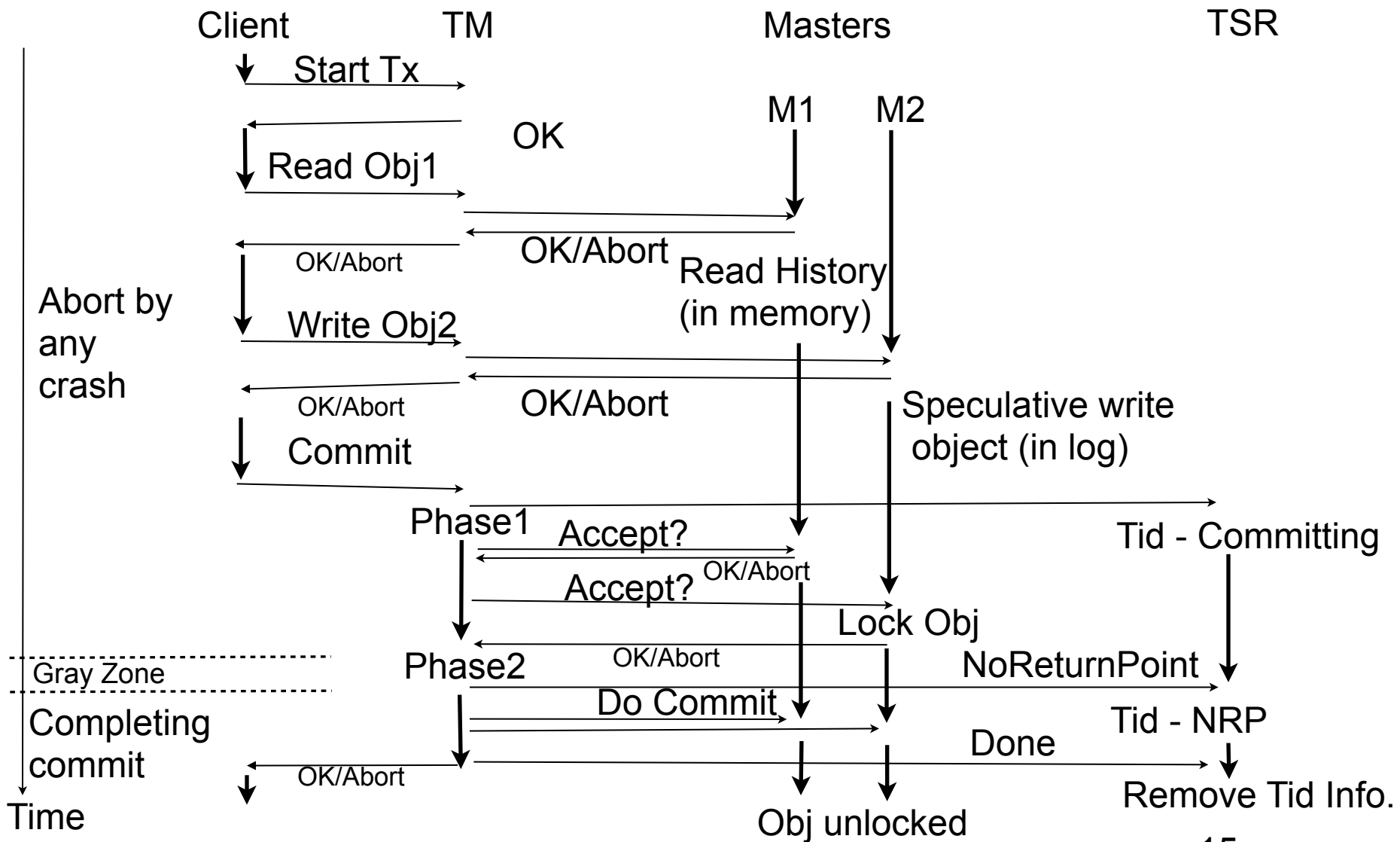
- If client application is restarted immediately (by coordinator, etc), TM can be implemented in client library.



Functions	TM:Trans. Monitor	TSR:Trans. State Repo.	Master	Coordinator
Normal Op.	Generate unique Transaction ID. Keep track objects states. 2phase commit coordination.	Store global status of a transaction persistently	Keep object s' status and temporal data, return appropriate data	Maintain crash information and TM identifier.
At Recovery	Continue 2phase commit (resource unlock)	TM accesses the transaction status	Respond TM to complete commit/abort	Restart TM, or notice TM crashed node.
Possible location	Client library, Client node, or Master	Master node as a normal table.	Master node	Coordinator

Basic Flow: Life of a Transaction

- Define Transaction priority uniquely with Tid: Transaction ID



Outline of Detailed Discussion

1. Client API
2. State transition of transaction and objects
3. Conflict Management
 - i. Resolution at object access with transaction priority
 - ii. TMid/Tid for unique global transaction order
 - iii. Timeout to avoid deadlock
4. Commit - transition from non-blocking to blocking
5. Recovery
 - i. Cleaning up by abort or completing commit
 - ii. TM implementation
 - service process or library - depends on client recovery
 - iii. TSR implementation - in a normal table
6. Implementation Control / Data structure
7. Optimization
 - i. Callback instead of piggyback
 - ii. Separate key/state and data for objects in log

1. Client API - Simple, Minimum

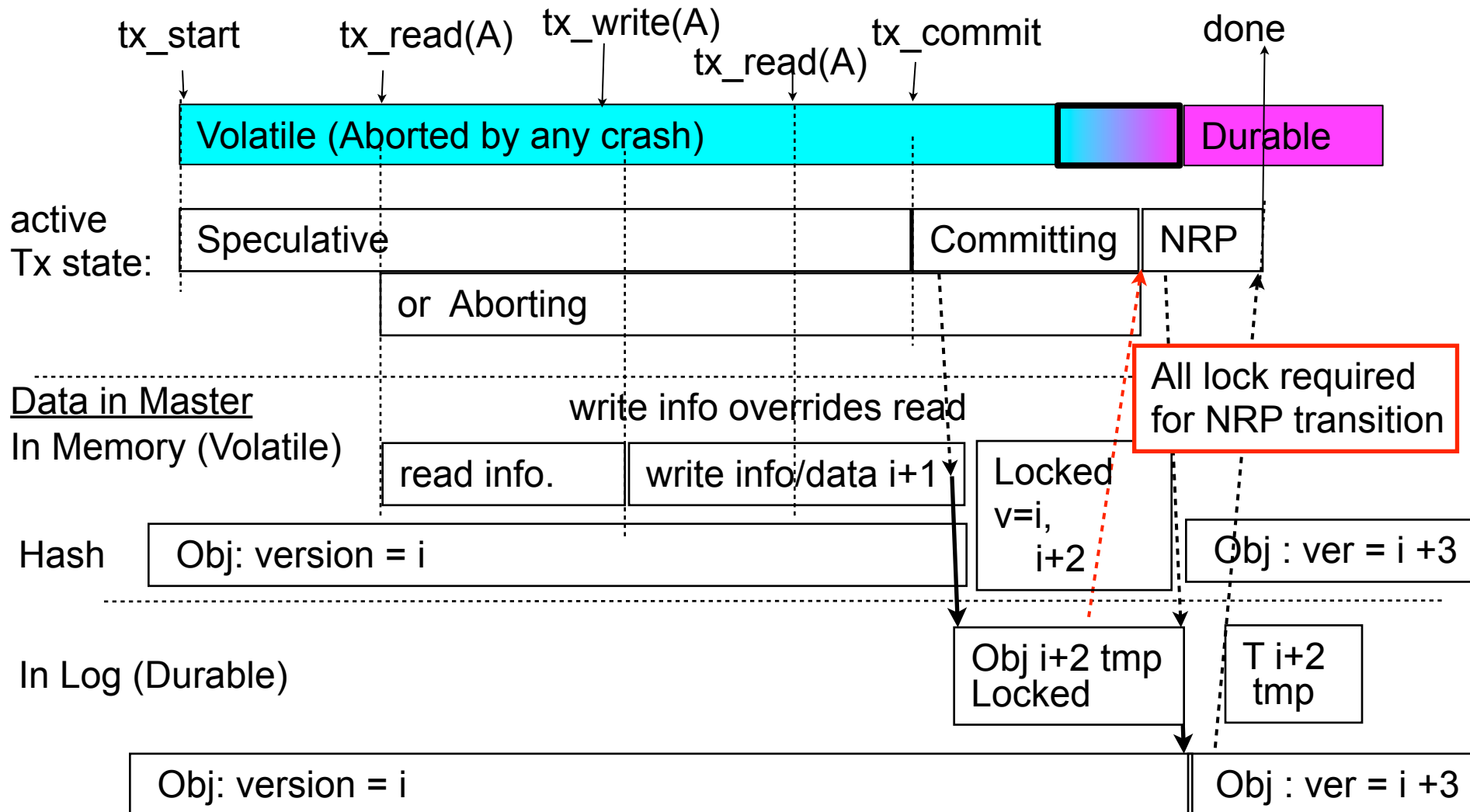
- Start Transaction
 - `tx_start(&tid); // return new tid`
- Object Access
 - `tx_read(tid, tableId, key, &buf, &state...);`
 - `tx_write (tid, tableId, key, &buf, &state...);`
 - `tx_remove(), tx_multi-...()`,
 - ▶ Can define `tid=0` as non-transactional operation
 - ▶ Still need compare & swap for multi-threading app?
- Commit & Status of Transaction
 - `tx_commit(tid, &state);`
 - `tx_abort(tid, &state);`
 - `tx_status(tid, &state);`

2. Transaction States

Detail

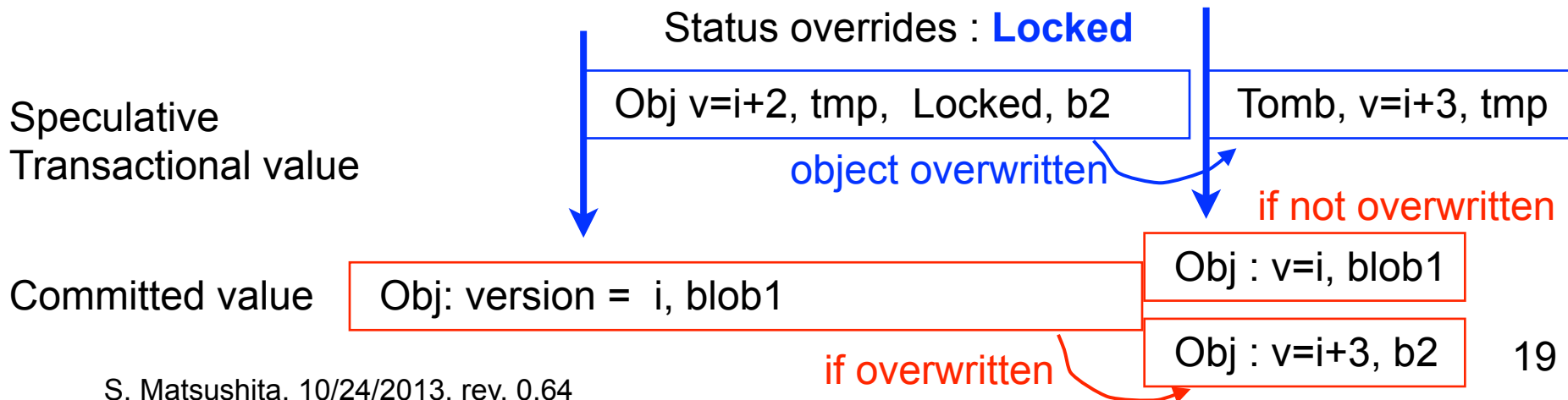
- Assume a transaction accesses single object 'A' for simplicity

Time



2. New Object Type

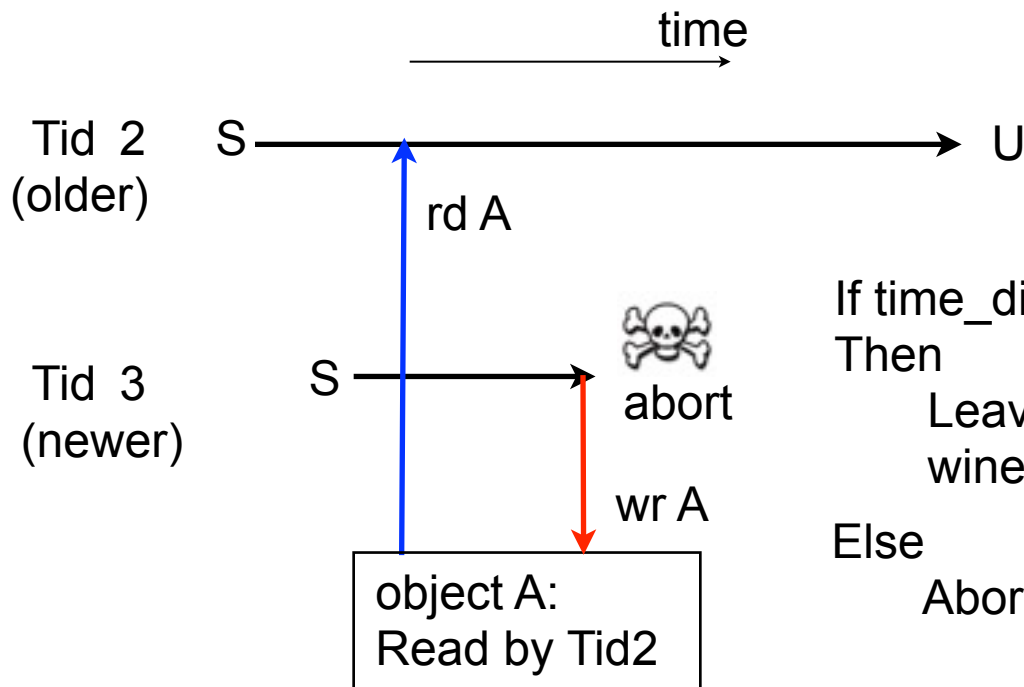
- How to declare locked object in log?
 - ✓ 1. Define version grouping, let status override other groups
 - 2. Define multi-data object containing both $v=i$, $i+2$
 - Note) 2 is simple but max object size becomes multiple
- Grouping object by group field
 - Compare version within the same group
- Define temporal object as group=tmp
 - $\text{Object}(\text{key}, \text{ver}, \text{group=tmp}, \text{tid}, \text{status=locked}, \text{blob})$
 - $\text{Tomb}(\text{key}, \text{ver}+1, \text{group=tmp}, \text{tid})$



3. Conflict management at object access

- Compares Tid in Master. Abort newer Tid immediately.
(Traditional technique in DBMS)
- Timeout to avoid deadlock by incorrect code or client crash, which freezes the oldest transaction.

Notation)
 S: Started
 A: Aborted
 U: Uncommitted
 (Speculatively running)



If $\text{time_difference}(\text{Tid3}, \text{Tid2}) > \text{Tout}$
 Then
 Leaves both alive and decides winner at commit time.
 Else
 Abort transaction with newer Tid

2. Truth Table of Conflicts Management

- Older transaction id wins at data access
- Provides only shared reads: can detect Read/Read conflict with dummy write: Rd (Obj1) with Wr(Dummy1)

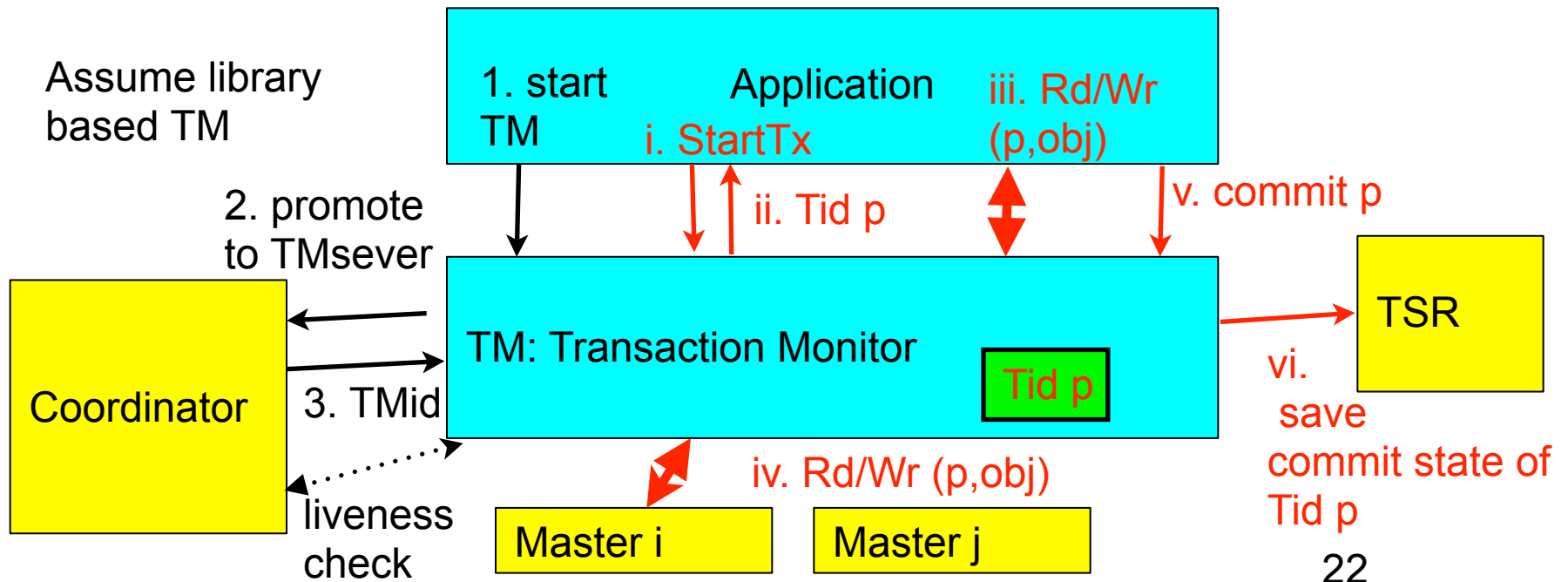
Tid 1 (Older) < Tid 2 (Younger)

operation mode	Tid 1	Tid 2	winner
mode 1	read	read	both
mode 2	Not Supported	read	Tid 1
both modes	read	write	Tid 1
both modes	write	read	Tid 1
both modes	write	write	Tid 1

21

Tid, TMid

- TMid (TM identifier) is given by coordinator at TM startup
- Tid (Transaction identifier)
 - Define Tid = [TMid, TM-localtime] at a transaction generation // note: [a, b] = concatenation of 'a' and 'b'
 - Compare TMid only when local time is the same
 - Preciseness is not required, because Tid is just a priority to decide a winner transaction at object access time.

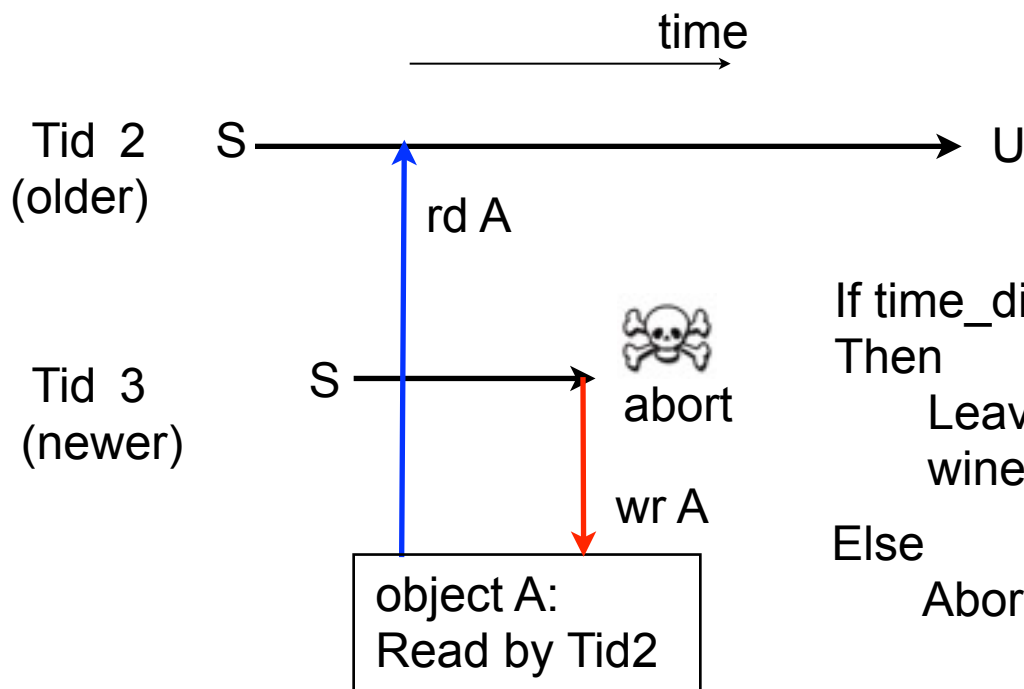


22

Conflict management at object access

- Compares Tid in Master. Abort newer Tid immediately.
(Traditional technique in DBMS)
- Timeout to avoid deadlock by incorrect code or client crash, which freezes the oldest transaction.

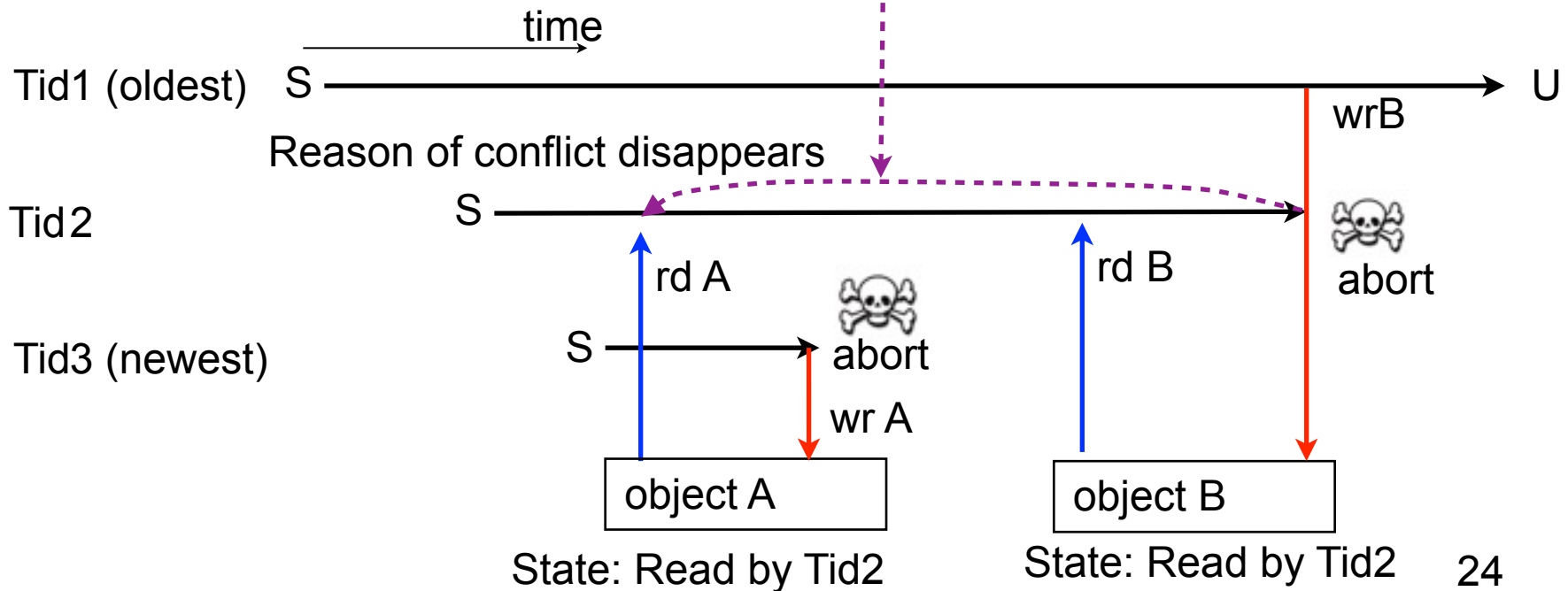
Notation)
 S: Started
 A: Aborted
 U: Uncommitted
 (Speculatively running)



If $\text{time_difference}(\text{Tid3}, \text{Tid2}) > \text{Tout}$
 Then
 Leaves both alive and decides winner at commit time.
 Else
 Abort transaction with newer Tid

Issues - False abort/Status piggyback

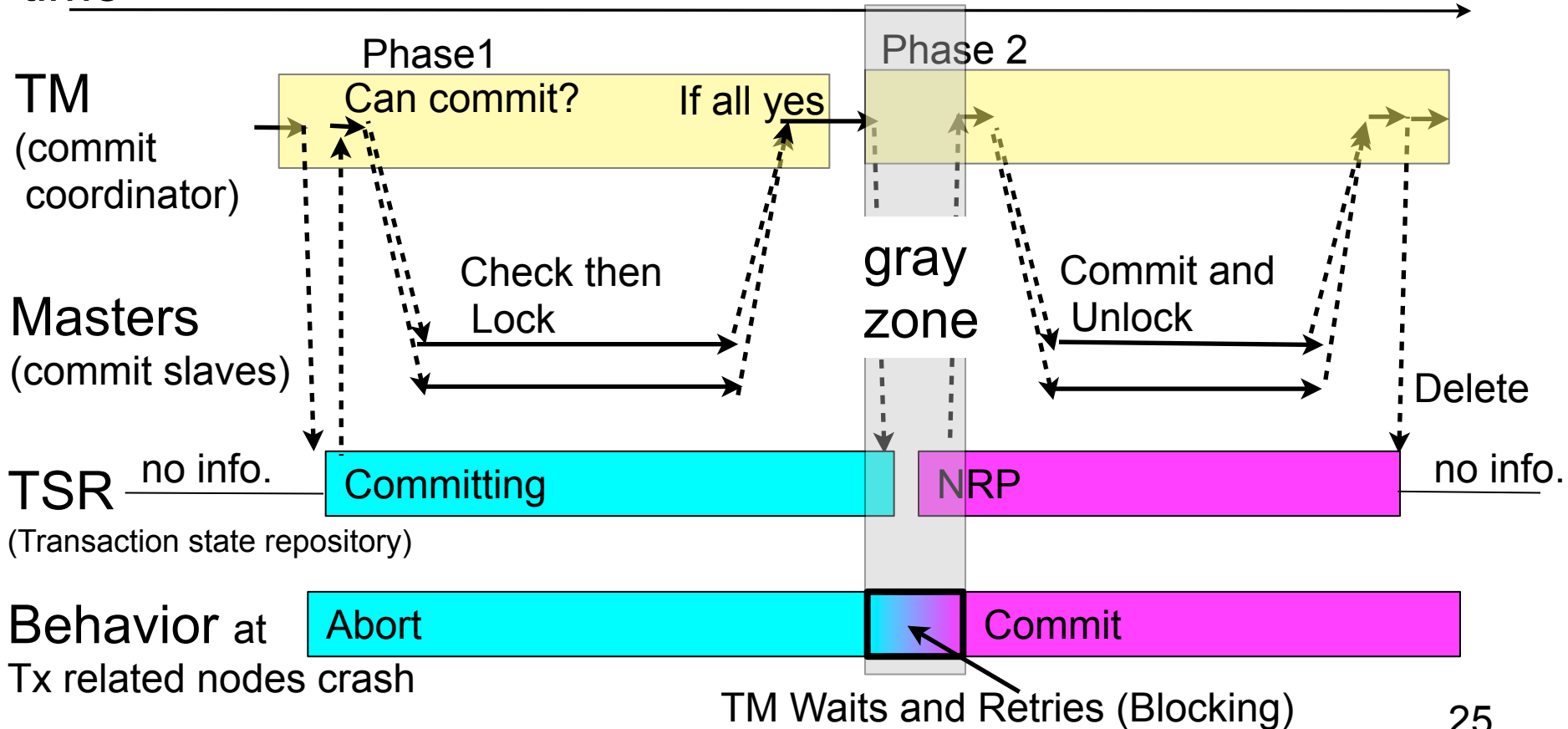
- False Abort: the conflict which aborted Tid3 disappears when Tid2 is aborted later.
 - Chain reaction of false abort may occur
 - Leave it because provability of false abort is small.
- Abort notified as status return (piggyback).
 - Tid2 is not aborted by Tid1-write, but by some request in the future (Needs callback to optimize)



3. Commit - Two phase commit

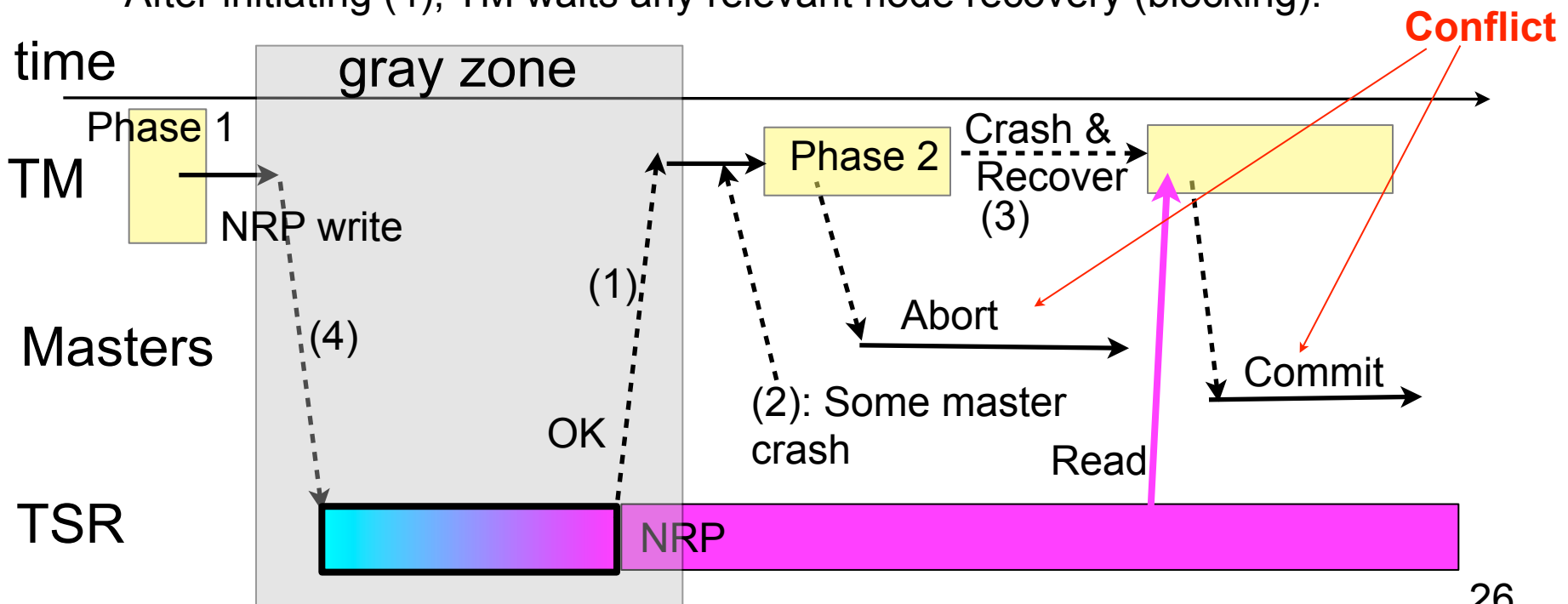
- TM coordinates commit operation
- Save durable state in TSR
 - Committing: unlock object by abort (optimization)
 - NRP: no-return-point for durable transition to commit time

time



3. Commit - Racing conditions

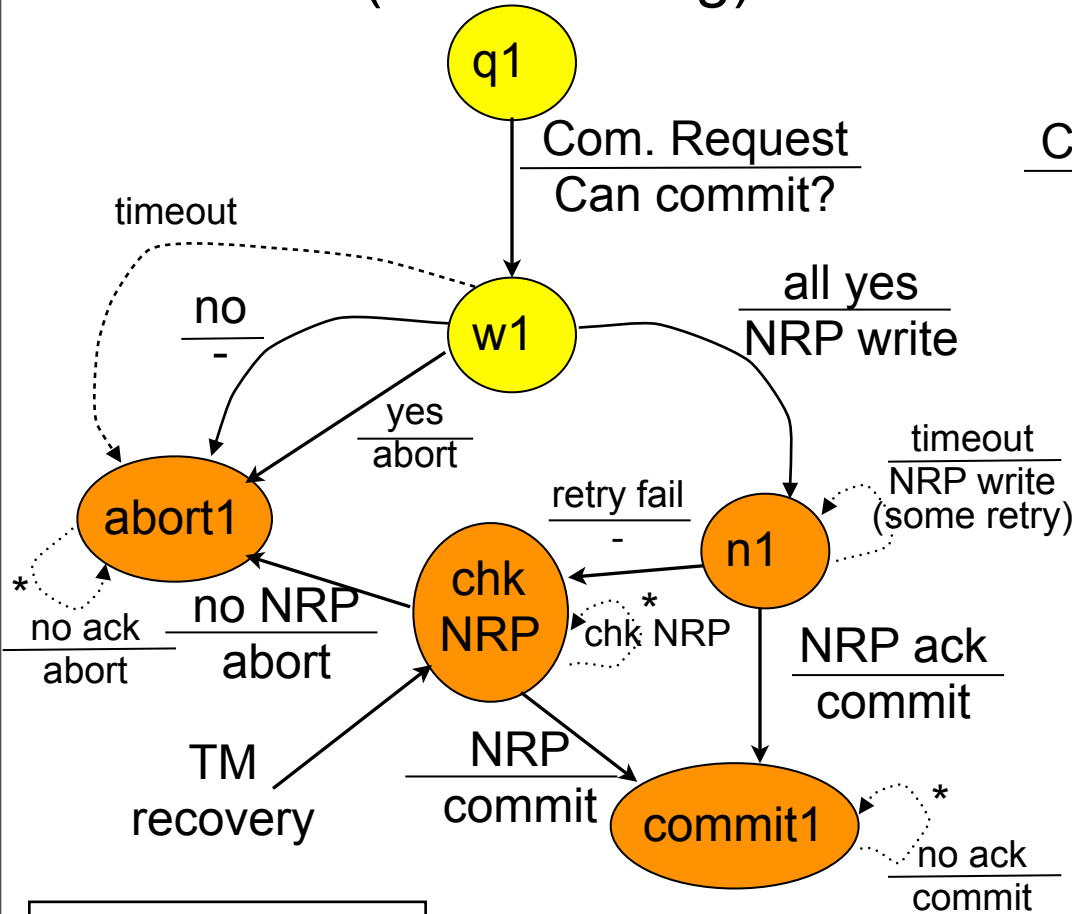
- Racing condition: Note that abort and commit are unilateral
 - After NRP is written, TM start aborting in Phase2 due to (1) 'OK' loss or (2) relevant node crash
 - Then TM crashes. The recovered TM reads NRP then starts commit.
 - (1) cannot be distinguished from (4) lost NRP req
- Solution
 - NRP is idempotent: TM retries (4) and waits (1)
 - If TM failed retry, TM reads TSR after enough timeout to decide behavior.
 - After initiating (4), TM waits any relevant node recovery (blocking).



3. Commit - State Machine with Failure

Yellow is non-blocking, **Orange** is blocking.

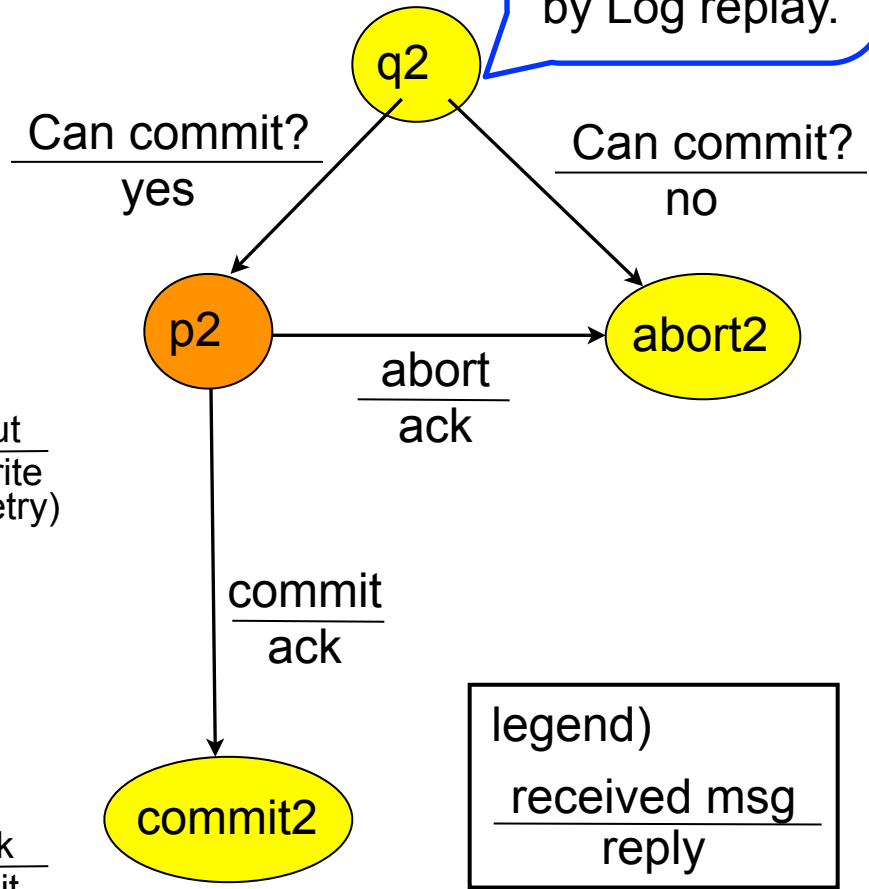
TM (coordinating)



* : client handles retry error

Ref: A Formal Model of Crash Recovery in a Distributed System, Dale Skeen and Michael Stonebraker, 1983

Masters



legend)
received msg
reply

4. Crash Recovery - Clean up

- TM crash
 - Completes commit/abort transactions for the TM
 - Commits transactions whose NRPs are found in TSR
 - Otherwise aborts transactions which belong to the TMid
 - Fast cleanup required to prevent other clients' blockage by accessing locked objects
- Server crash
 - Reconstruct hash and object status in memory from log
- TSR crash
 - Recover commit status of transactions

5. Implementation Alternatives

- TM - item 1 seems simplest and good for performance.
- ✓ 1. In client library such as crt0.
 - Pros) Application (Tire2, Tire3) needs to be recovered to continue web service anyway
 - Cons) Need client recovery mechanism by coordinator
- 2. In a master
 - Need TM locator
 - Cons) Extra access latency and network traffic by additional hop for data access
- 3. In a separate process/thread in a client node
 - Need recovery mechanism
 - Cons) Extra latency by process communication and dispatch
- TSR
 - In a master with defining a table and save transaction state as a normal object.

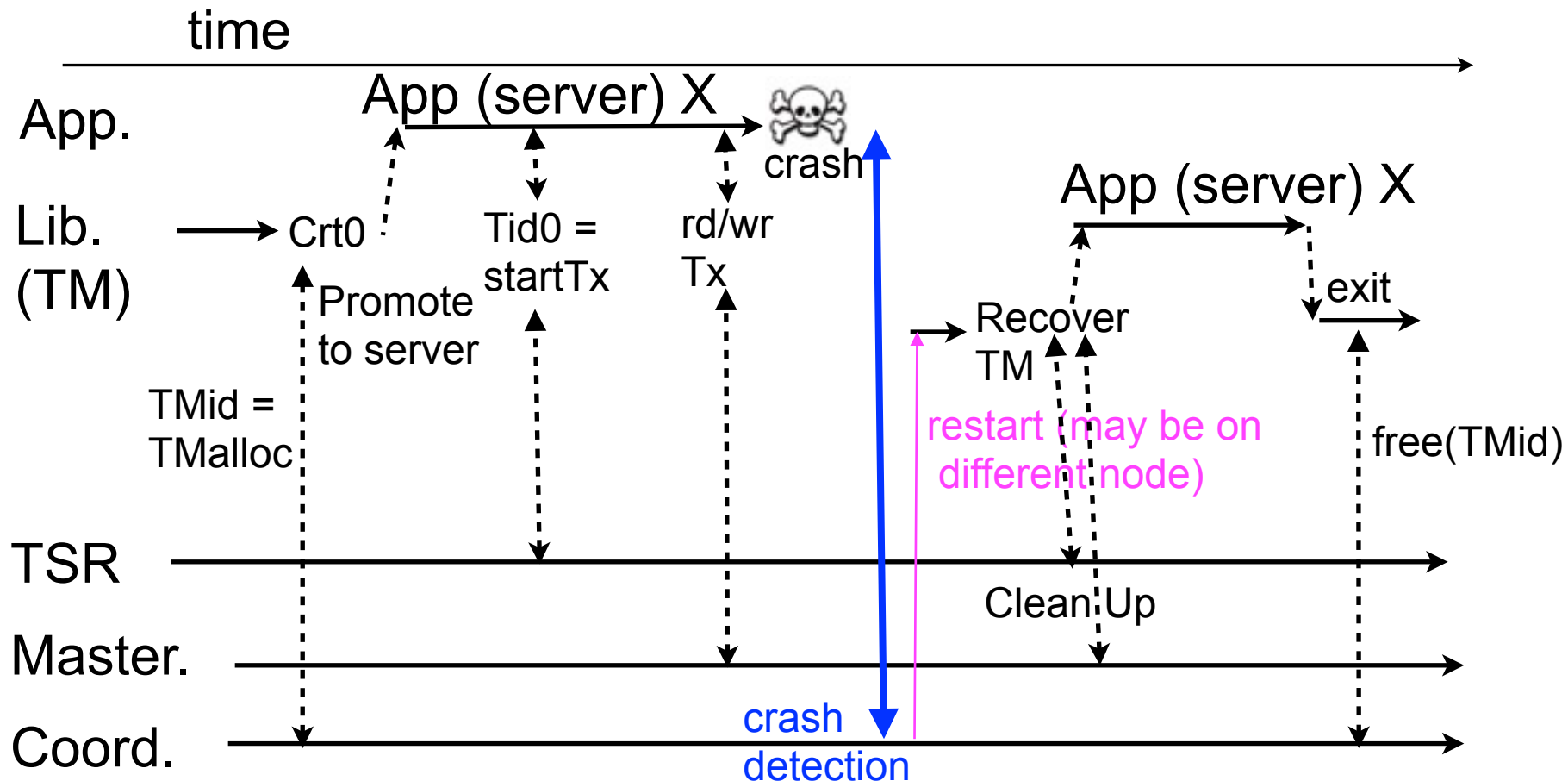
5. Implementation Proposal

- Implement TM in **client library**
 - Coordinator detects client failure and restarts
 - Naming issue: 'Once the liveness check/recovery is managed by coordinator, it should be called 'server' - not a client anymore'.
- TMid given by coordinator
- TM generates Tid = [TMid, TM's local time]
- TSR as a specific table
 - List of status: (Key, Value) = (TMid, list_of{ (Tid, TransactionStatus) })
 - Can find both a Tid and all the Tids with TMid.
 - Simple enough since one commit operation is underway for a TM normally.

[a, b] denotes concatenation

5. Implement TM in Client Library

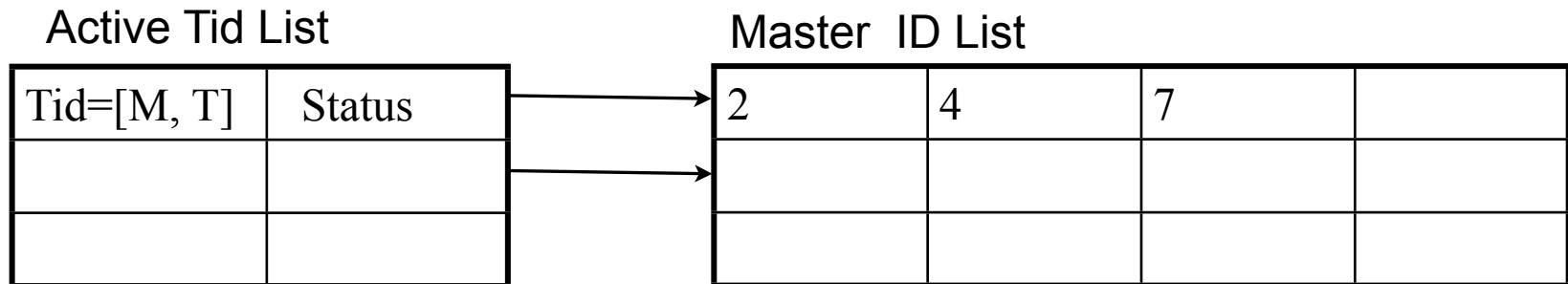
- Crt0 contacts coordinator to get TMid and register application info. for recovery.
- User can modify transaction algorithm by modifying library.



31

5. TM Data Structure

- All data in memory which consist:
 - Status: Speculative/Committing/NRP/Aborting
 - Master IDs: masters accessed by each transaction
 - After TM crash, states for Tids recovered from TSR:
 - State: NRP/Else
 - Finalize Commit if NRP / Abort otherwise
 - by broadcasting a request for a TMid to all masters
 - // TM crash probability is small and request size
 - // is small. (no optimization so far)



5. TM Control

Status for each Tid: Speculative/Committing/NRP/Aborting

To Be Done.

Master Data Structure

Simple example

- Volatile data forgot by crash: hash, array -- any improvement using log structure? memory management?

To Be Done.

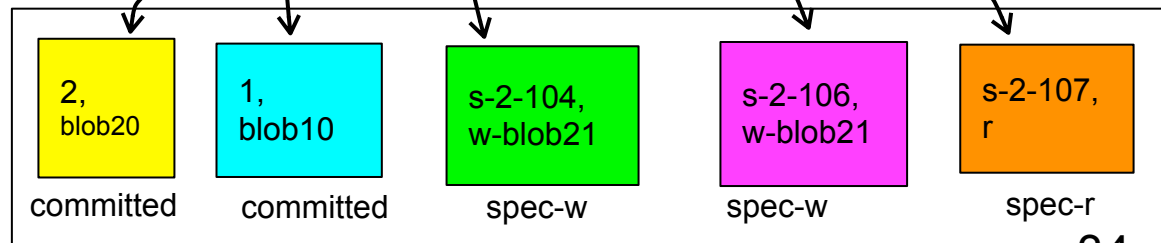
Tid Hash

T1 = [1, 60 ms]
T2 = [1, 85 ms]
T5 = [3, 85 ms]

Key hash

2
1
12

Log



Master Control

To Be Done.

TSR Data Structure

- TSR as a specific table:
 (Key, Value) = (TMid, list_of{ (Tid, TransactionStatus) })
- Distribute access by a hash of the Key
- Durable and available

