Proposal of Transaction on RAMCloud

rev0.64

25 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.

2

back

S. Matsushita, 10/25/2013, rev. 0.64

Characteristics of Transaction

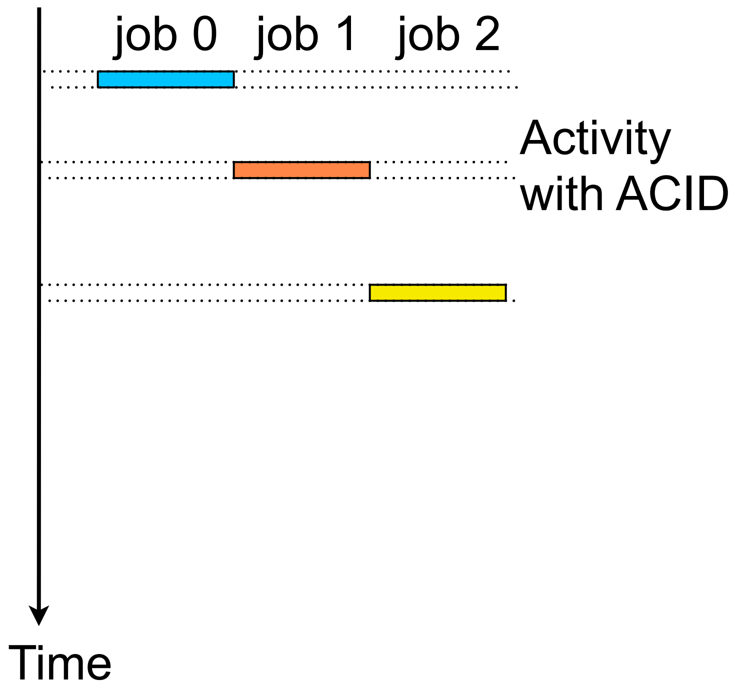
1. Duration varies from short to long: 0.1ms to 100ms
2. Very small chance of conflict to other transactions
3. Too many conflicts are data/control design issue

Example	Duration	Chance of Conflict
Analytic (Data analysis)	min. to hours	none after start
Ticket reservation, auction	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

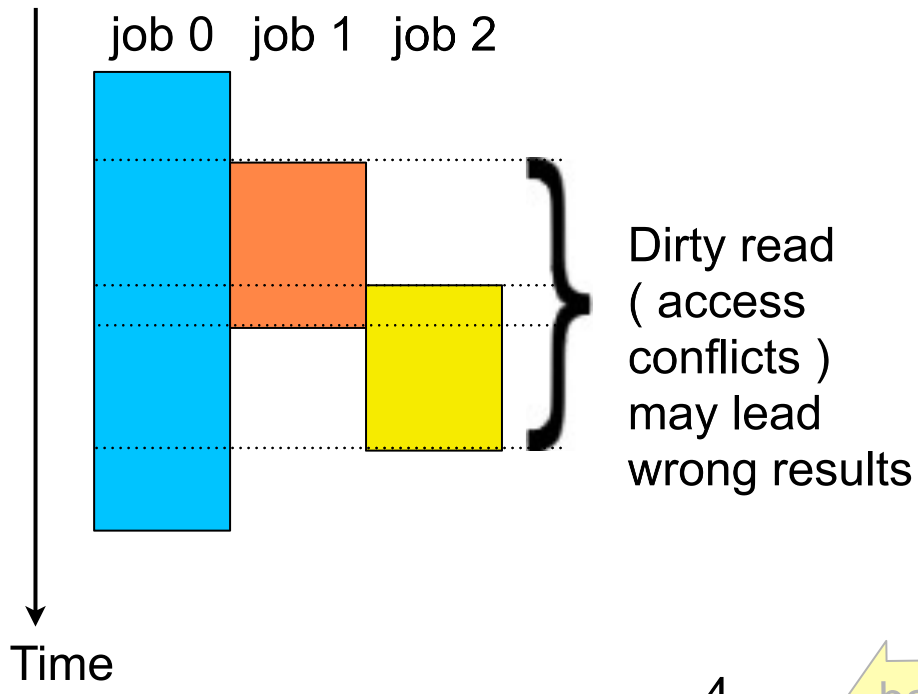
Issues in Parallel Execution

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

Ideal Parallel Execution)



Reality)

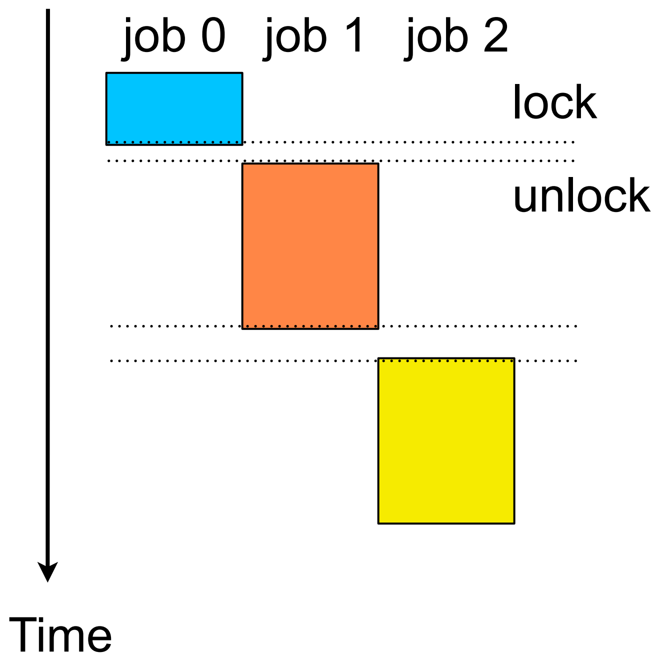


S. Matsushita, 10/25/2013, rev. 0.64

back

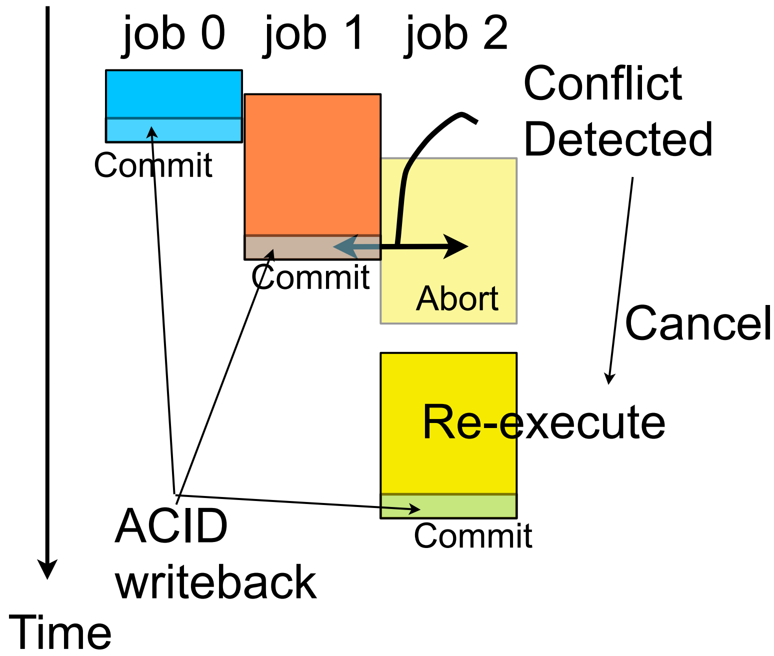
Conflict Solutions

Pessimistic Lock)



Text

Optimistic Lock)



Pros & Cons)
- Lower parallelism with giant locks
- Dead lock prone with fine locks
- Need releasing lock with node crash

Pros & Cons)
- Need conflict detection logic
- Lower Performance loss by frequent conflicts
- Alternatives in abort detection

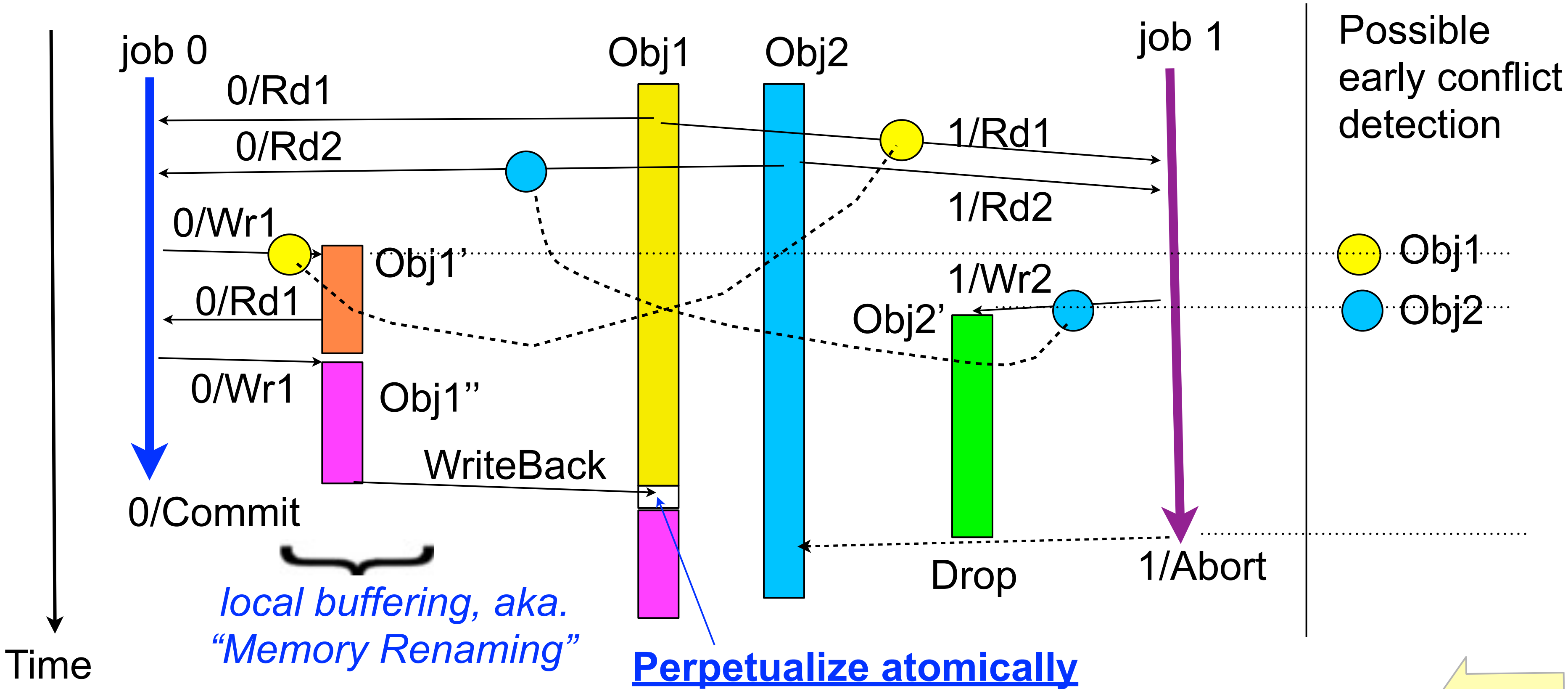
5

back

S. Matsushita, 10/25/2013, rev. 0.64

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



S. Matsushita, 10/25/2013, rev. 0.64

back

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**

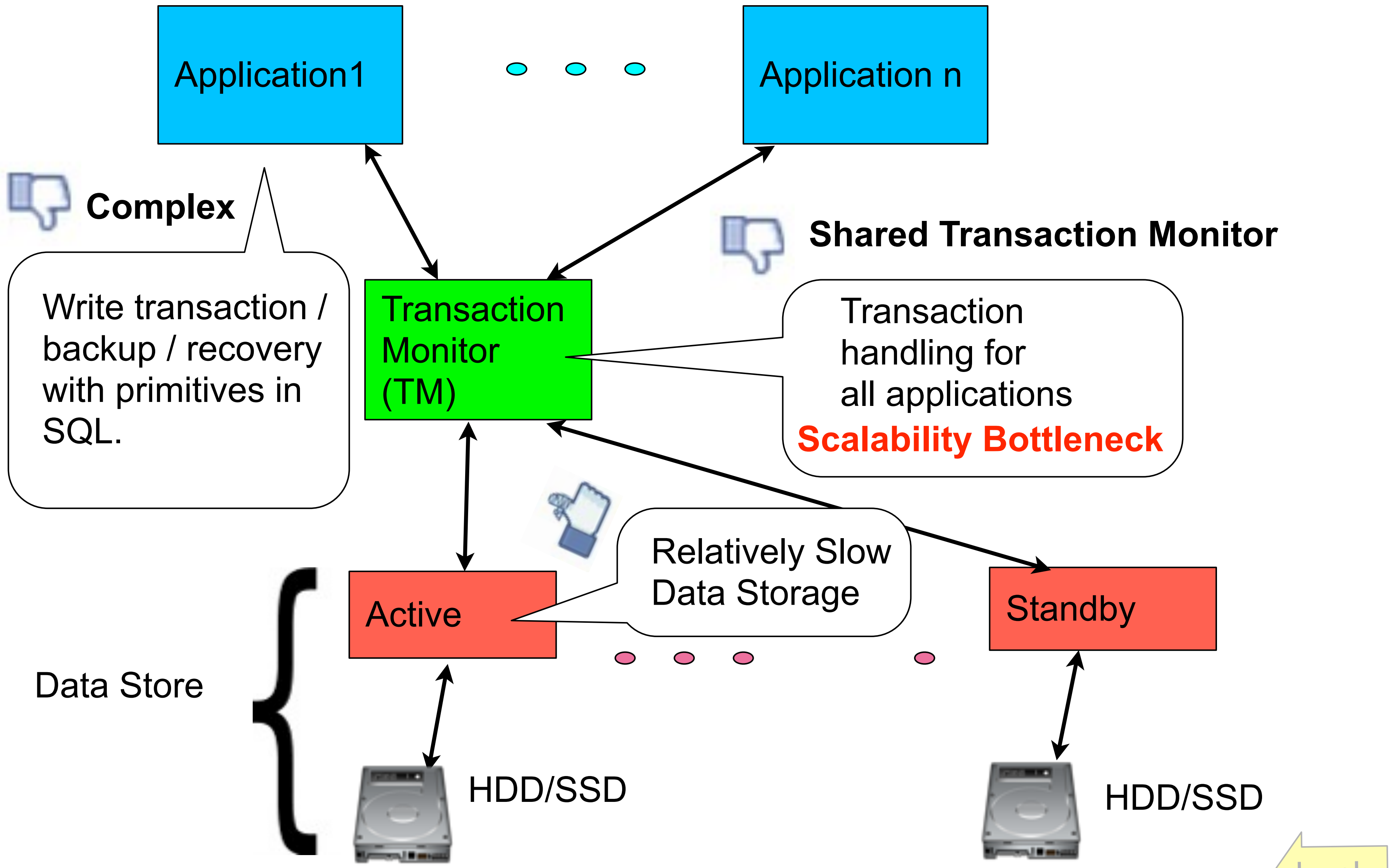
Note)

- 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

back

S. Matsushita, 10/25/2013, rev. 0.64

Traditional Transaction System



S. Matsushita, 10/25/2013, rev. 0.64

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

- 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

To Be Reviewed

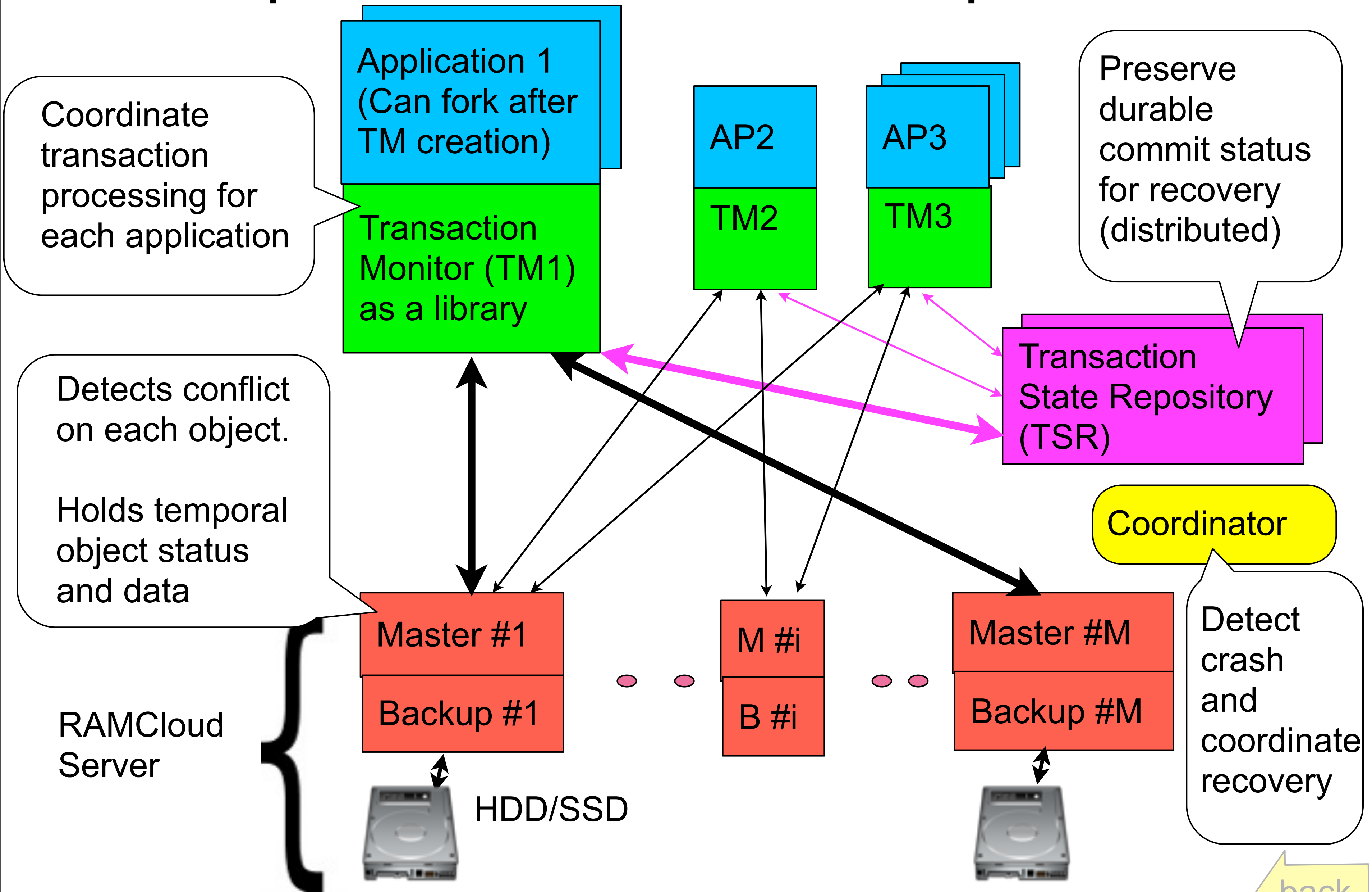
Ref: Sinfonia: A New Paradigm of r Building Scalable Distributed Systems, Macros K. Aguilera (HP Lab.), et. al. , SOSP, Oct. 07

Proposal: Key Idea

1. **Better scalability**
   - Distributed TM(Transaction Manager) with taking advantage of **distributed log** in masters.
   - **No replicated data** enables distributed conflict detection in masters
2. **Better performance** for wide duration range of transactions
   - Implementing TM in library for **low latency**
   - **Earier conflict detection** to prune shared information and reduce retry overhead
3. **Integrated and automated crash recovery**
   - Triggered by RAMCloud coordinator which is always available by consensus algorithm
4. **Natural** transaction API
   - **Dynamic**: enclose a part of native code with StartTx /{Commit,Abort} primitive - needs bad code safety
   - **Database schema in RAMCloud** with flexible and minimum set of primitives for smaller users' efforts
   - **Not exposing** internal structure: recovery, log data structure/checkpoint
     - Enable system side tuning adapting to new hardware / algorithm

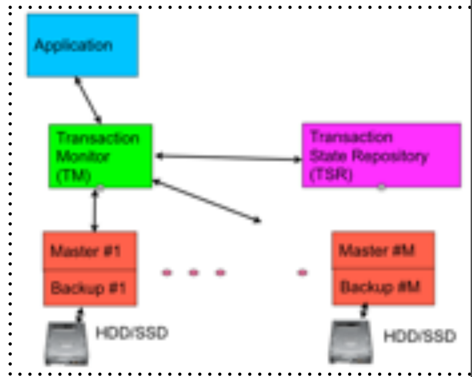
back

Proposal: Distributed Components



Components - Functions

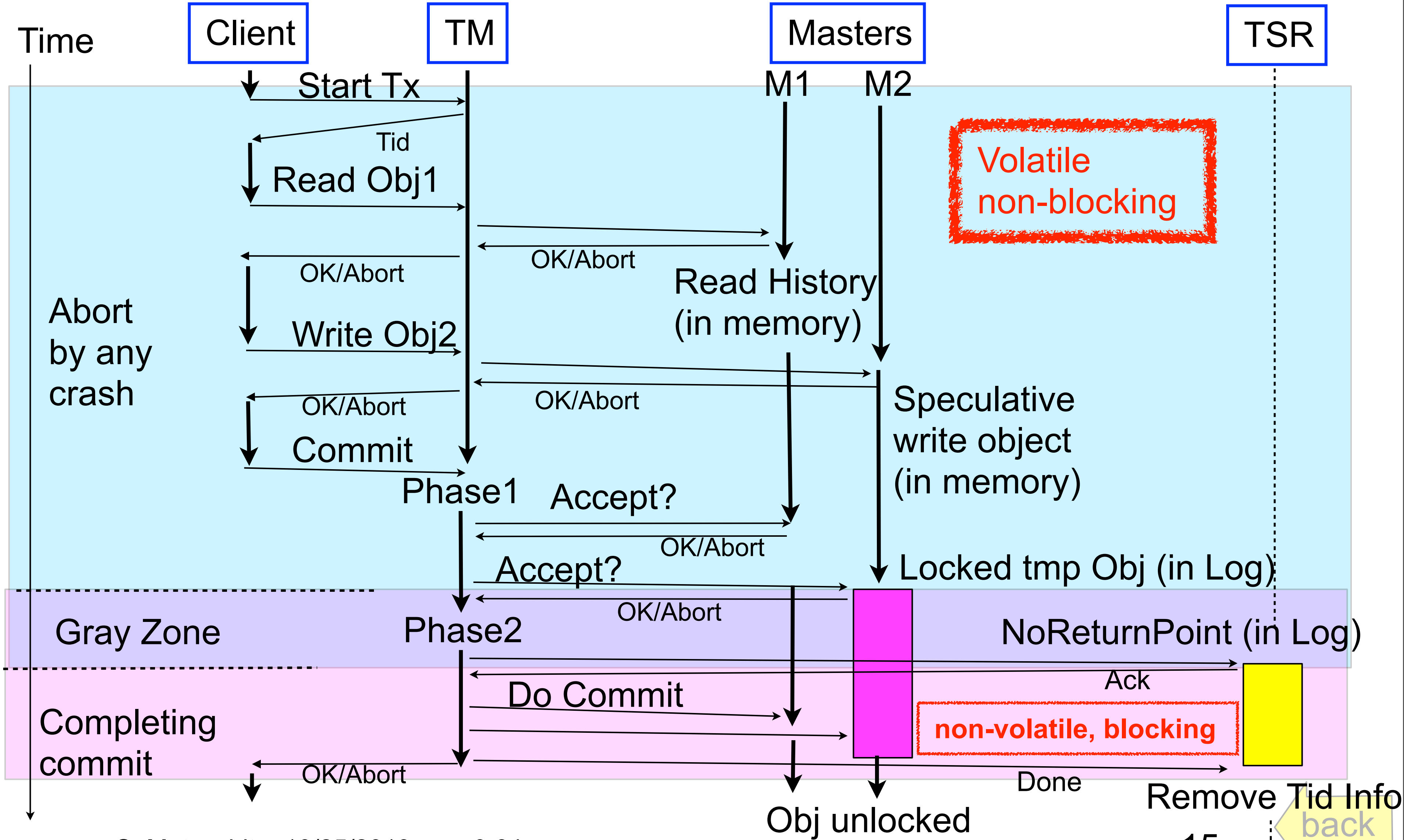
- An application needs to be immediately restarted by coordinator after crash to run **lock cleanup** in completion helper of commit/abort in the application library.



Functions	TM:Trans. Monitor	TSR:Trans. State Repo.	Master	Coordinator
Normal Op.	Generate unique Transaction ID. Coordinate transactional operation.	Maintain persistent status during commit operation.	Maintain objects' status and temporal data, and return appropriate data	Define TM identifier and watch entity crash.
At Recovery	Continue 2phase commit (**resource unlock**)	Provide transaction status to TM	Complete commit/abort triggered by TM	Start TM to run completion helper
Location	Client library	A table in masters	Master	Coordinator

Basic Flow: Life of a Transaction

Key words) Conflict detection at object access, 2 phase commit, volatile to non-volatile



back

S. Matsushita, 10/25/2013, rev. 0.64

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 - Minimum

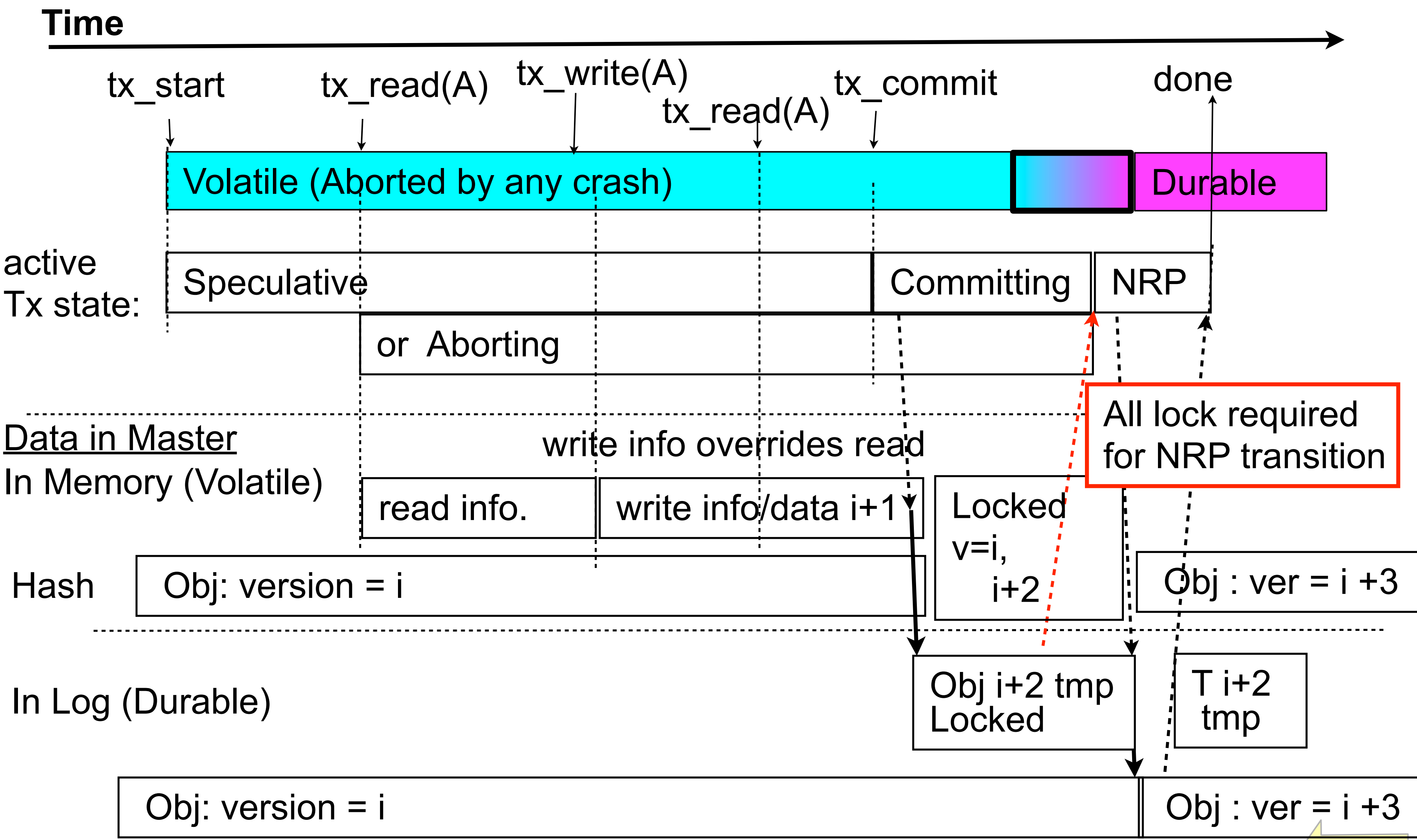
- Start Transaction
  - tx_start(&tid); // return new tid
- Object Access
  - tx_read(tid, tableId, key, &buf, &state...);
  - tx_write (tid, tableId, key, &buf, &state...);
  - tx_remove(), tx_multi-...(),
  - ▸ Can define tid=0 as non-transactional operation
  - ▸ Still need compare & swap for multi-threading app?
- Commit & Status of Transaction
  - tx_commit(tid, &state);
  - tx_abort(tid, &state);
  - tx_status(tid, &state);
- Issues) 1. Implicit TM startup? 2. thread ready?
  3. User's responsibility for multi-process client.

r0.64 Add

2. Transaction States

Detail

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



S. Matsushita, 10/25/2013, rev. 0.64

back

r0.64 Add

2. New Object Type

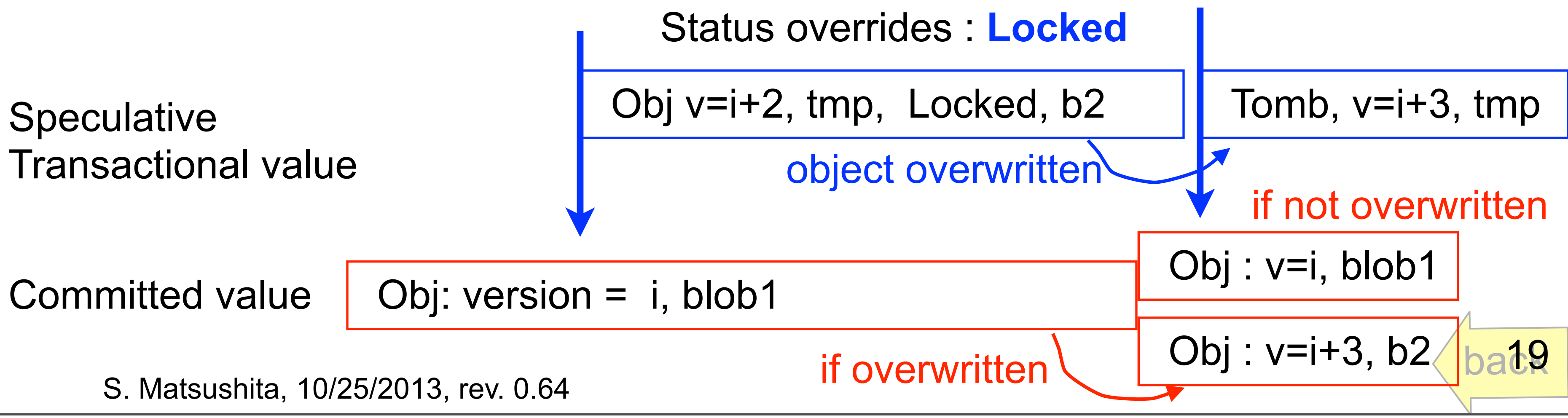
- How to declare locked object in log?

✓ 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

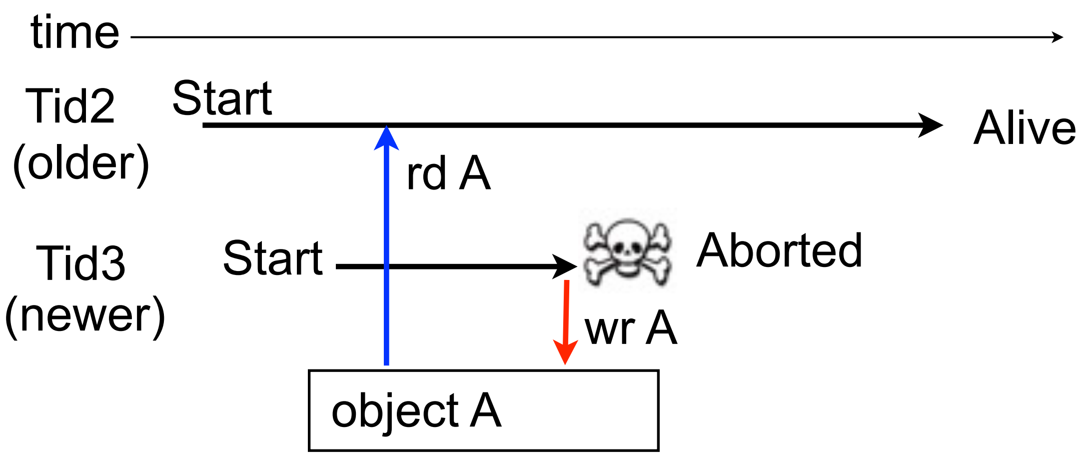
- Grouping object by group field
  - Compare version within the same group
  - Define temporal object as group=tmp
    - Object(key, ver, group=tmp, tid, status=locked, blob)
      Tomb(key, ver+1, group=tmp, tid)



S. Matsushita, 10/25/2013, rev. 0.64

3. Conflict management at object access

- Compares Tid in Master. **Abort newer Tid immediately**.



- **Timeout** to avoid deadlock by incorrect client code which prevents the completion of the oldest transaction.

If time_difference(Tid3, Tid2) > Timeout
Then
    Leaves both alive and decides winner at commit
Else
    Abort newer transaction at data access

back

S. Matsushita, 10/25/2013, rev. 0.64

2. Truth Table of Conflicts Management

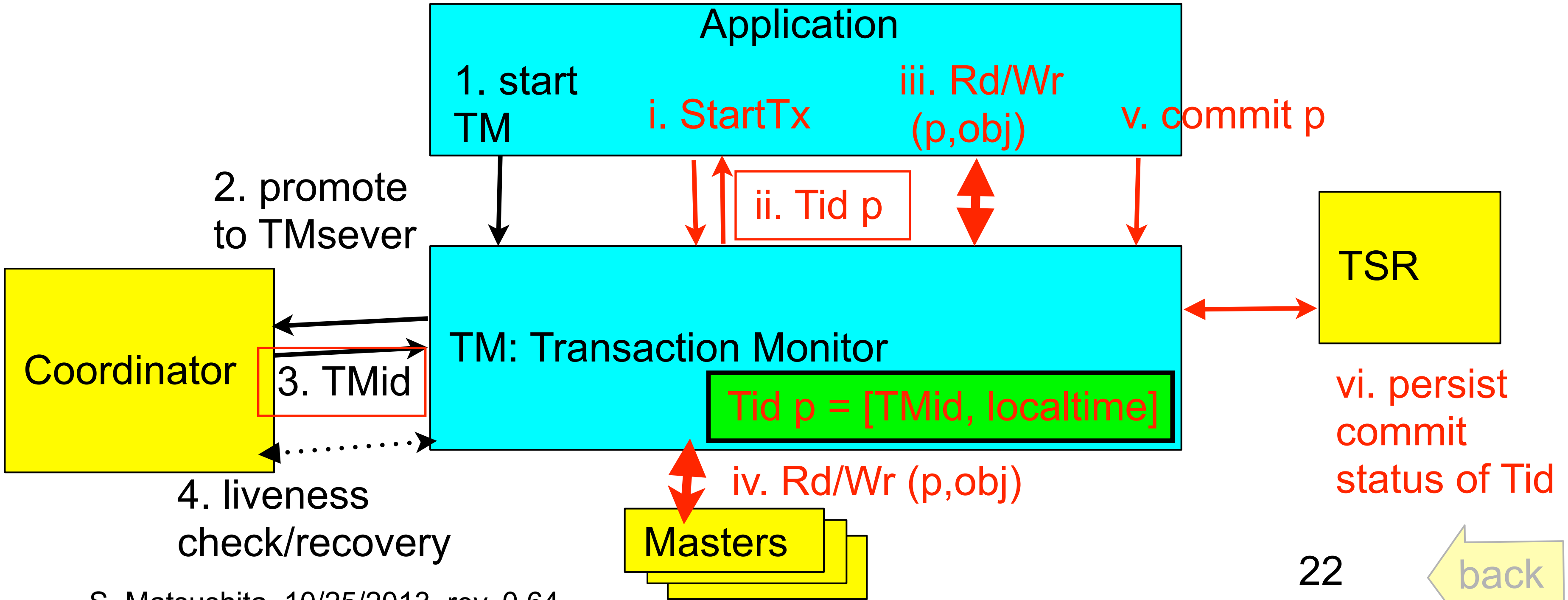
- Older transaction id wins at data access
- Provides only shared reads: can detect Read/Read conflict with dummy write: Rd (Obj1) with Wr(Dummy1)

Tid 1 (Older) < Tid 2 (Younger)

operation mode	Tid 1	Tid 2	winner
mode1	read	read	both
~~mode2~~	Not Supported	~~read~~	~~Tid 1~~
both modes	read	write	Tid 1
both modes	write	read	Tid 1
both modes	write	write	Tid 1

Tid, TMid

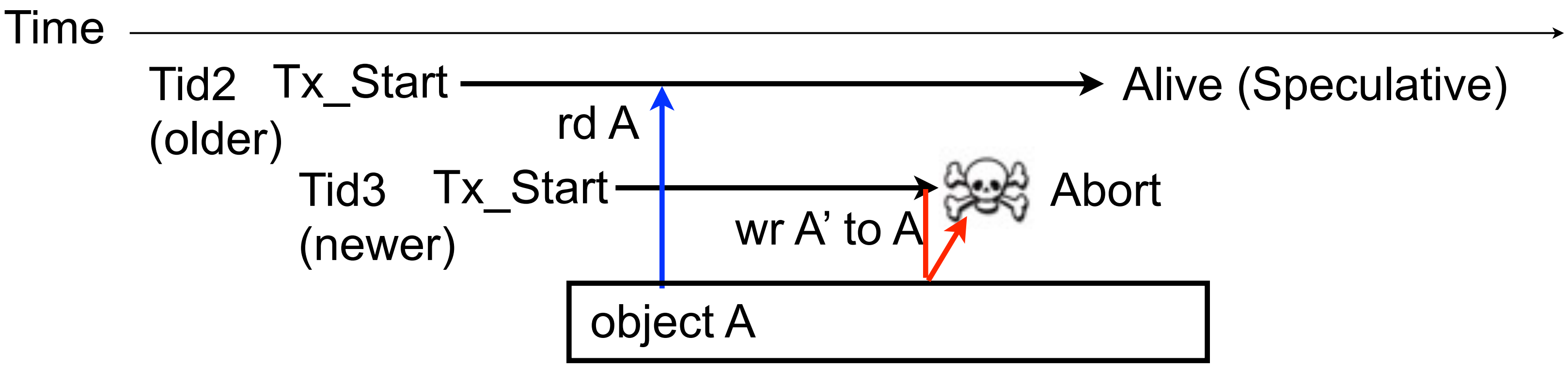
- TMid (TM identifier) is given by coordinator at TM startup
- Tid (Transaction identifier)
  - Define Tid = [TMid, TM-localtime] at the transaction creation // note: [a, b] = concatenation of ‘a’ and ‘b’
  - Compare TMid only when local time is the same
  - No need for timer accuracy because Tid is just a priority to decide a winner transaction at object access time.



S. Matsushita, 10/25/2013, rev. 0.64

Conflict management at object access

- Compares Tid in Master. Abort newer Tid immediately



- <u>Timeout</u> to avoid deadlock by non-terminating oldest transaction due to incorrect code or client crash, etc.

If time_difference(Tid3, Tid2) > Timeout
Then
    Leaves both alive and decides winer at commit
Else
    Abort transaction with newer Tid

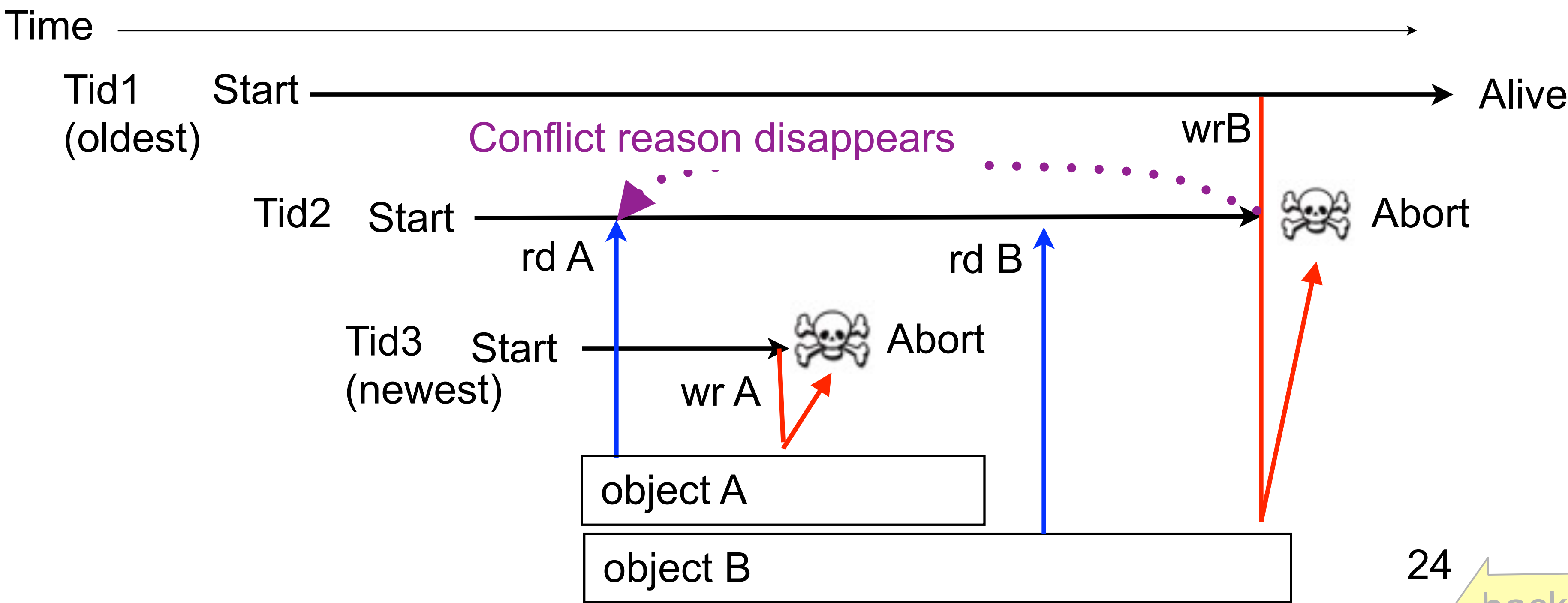
Note) time difference is directly calculated with Tid which contains localtime

S. Matsushita, 10/25/2013, rev. 0.64

back

Issues - False abort/Status piggyback

- False abort: the conflict which aborted Tid3 disappears by later Tid2 abort
  - May cause chain reaction of false abort
  - Leave it because provability of false abort is small
- Abort notified in status return (piggyback)
  - Tid2 is not aborted by wr B of Tid1, until status return of another future request (Needs callback to optimize)

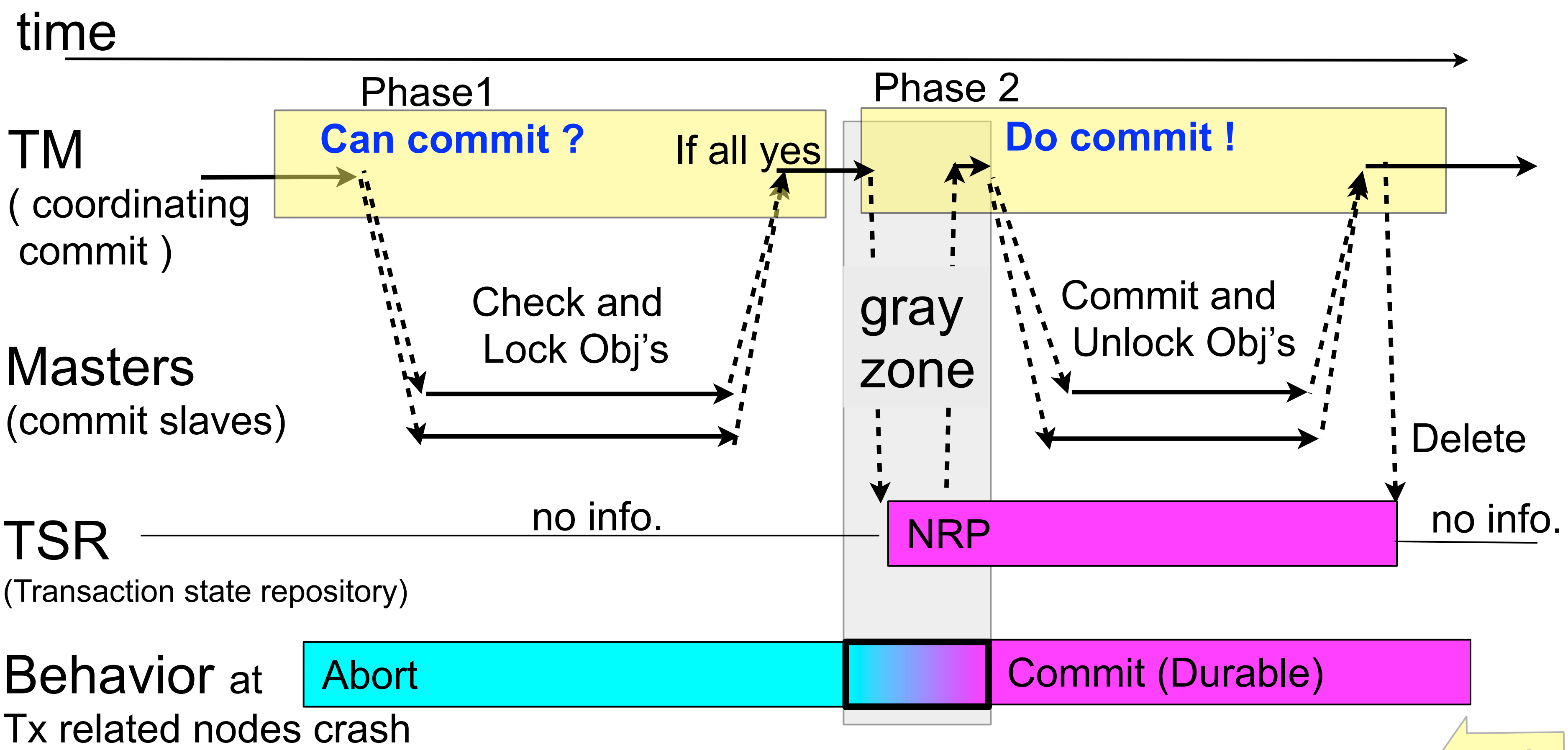


back

S. Matsushita, 10/25/2013, rev. 0.64

3. Commit - Two phase commit

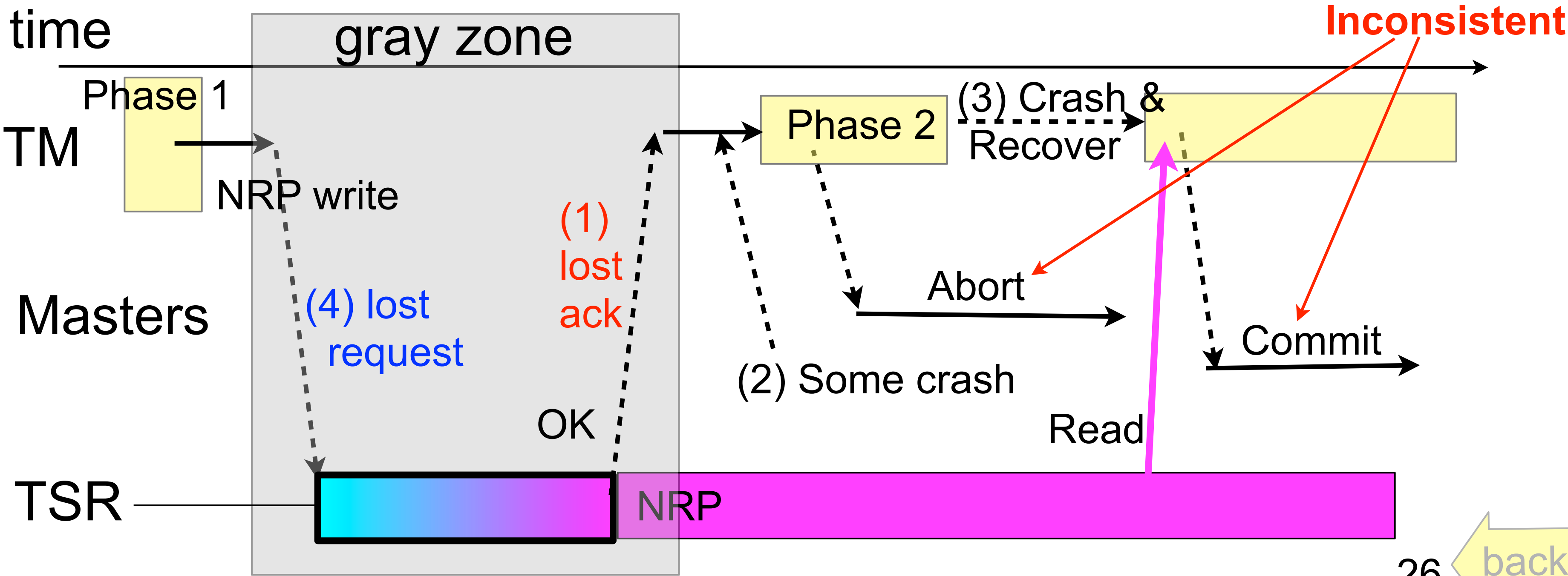
- TM coordinates commit operation
  - Need to unlock objects in masters in phase2
- Save durable status in TSR
  - NRP: no-return-point for durable transition to commit



S. Matsushita, 10/25/2013, rev. 0.64

3. Commit - Racing conditions

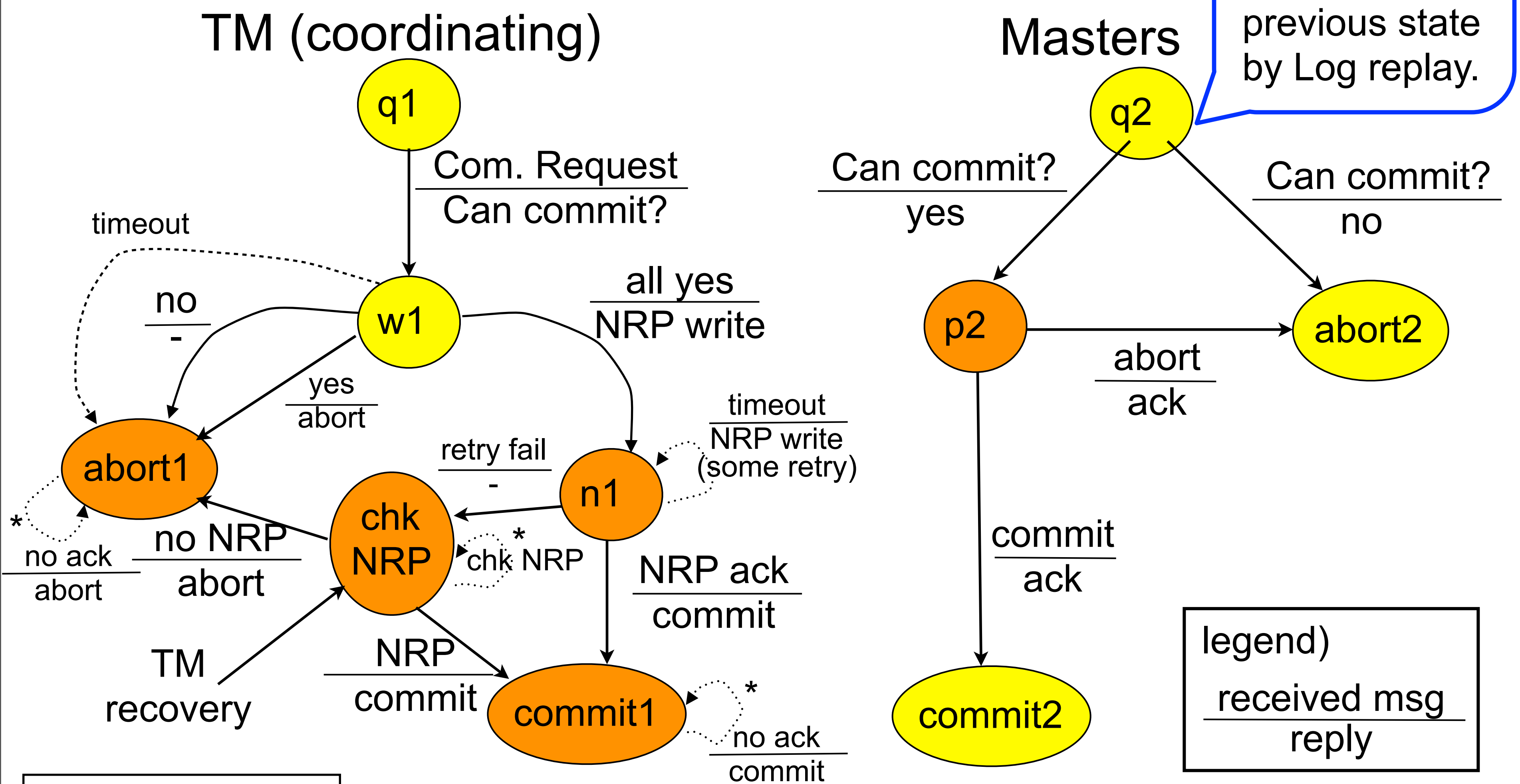
- Racing condition: Note that abort and commit are unilateral
  - Non-blocking behavior) After NRP is written, TM start aborting in Phase2 due to (1) lost ack or (2) Tx relevant node crash
  - Then (3) TM crashes, and recovered TM reads NRP then starts commit
  - Cannot distinguish (4) lost request: (no NRP written) from (1) : (NRP written)
- Solution: **Transferring to Blocking**
  - NRP is idempotent: TM retries (4) and waits (1)
  - TM reads TSR to decide behavior after retry error of (4)(1) or after TM crash(3)
  - Once (4) initiated, TM waits recovery of any relevant node crash (2)



S. Matsushita, 10/25/2013, rev. 0.64

3. Commit - State Machine with Failure

Yellow is non-blocking, Orange is blocking.



* : client handles retry error

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

S. Matsushita, 10/25/2013, rev. 0.64

27 back

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 when they access 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 (C/C++ startup runtime)
    - 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**

29

back

5. Implementation of TM/TSR

- 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, then TM locally generates Tid = [TMid, TM’s local time] at tx_start request.

[a, b] denotes concatenation

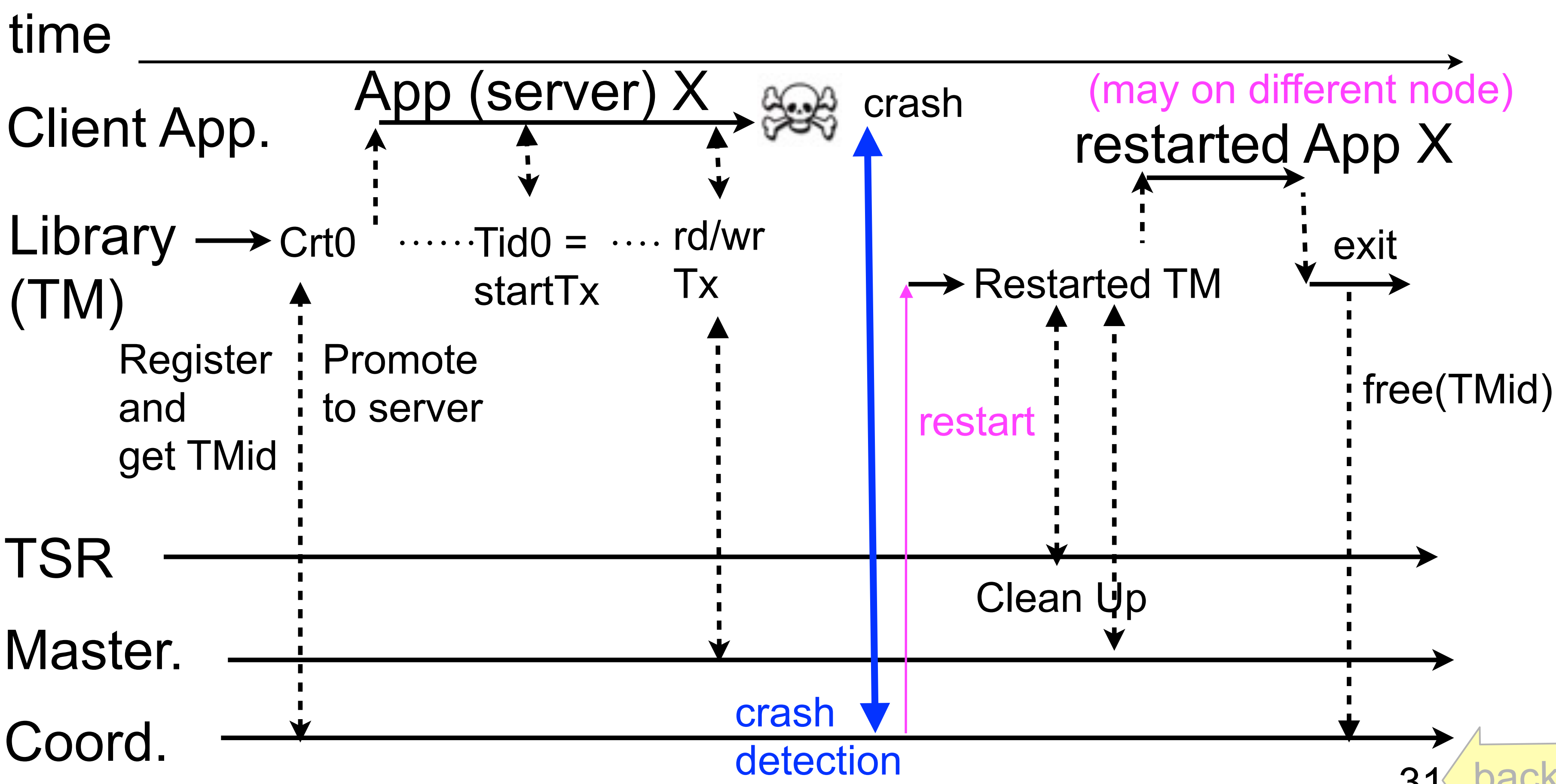
- TSR as a specific table
  - List of status: (Key, Value) = (TMid, list_of{ (Tid, TransactionStatus) })
    - Only one TransactionStatus so far: NRP
    - Can find both Tid’s status and Tids with given TMid
    - Simple enough since at most one commit-phase2 is ongoing for a TM in normal condition

back

S. Matsushita, 10/25/2013, rev. 0.64

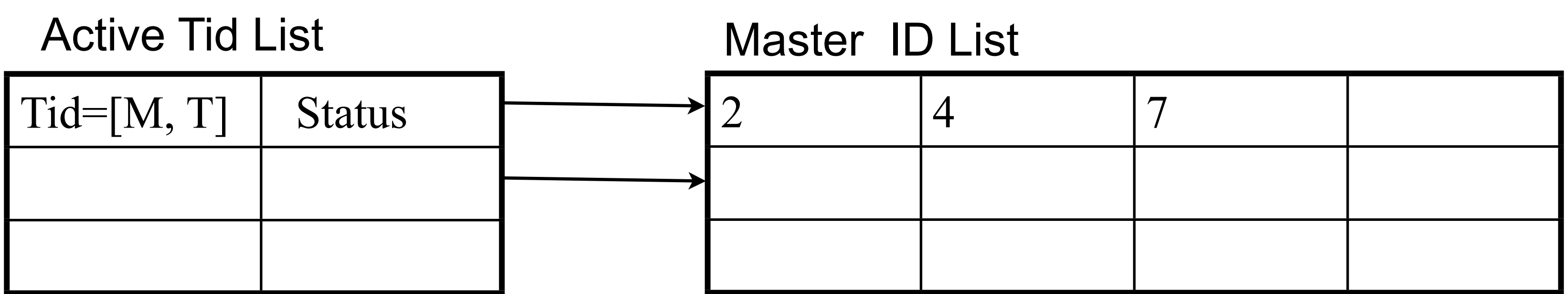
5. Implement TM in Client Library

- Crt0 contacts coordinator to get TMid and register application for its fail detection/recovery
- Application state is saved/recovered with ACID transaction
- User may modify transaction algorithm by modifying library



5. TM Data Structure and Crash Control

- All in-memory data:
  - Tid, ***Status***: Speculative/Committing/NRP/Aborting, etc
  - Master ID list: masters accessed by each transaction
- After TM crash
  - (Tid, Status) recovered from NRP in TSR
    - No need to recover data if no NRP found
  - Broadcast 'Commit' for NRP / Abort otherwise with TMid to complete commit/abort of Tx coordinated by the TM
    - No optimization needed)
      - Small TM crash probability
      - Small size of request payload



S. Matsushita, 10/25/2013, rev. 0.64

5. TM Control

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

To Be Done.

S. Matsushita, 10/25/2013, rev. 0.64

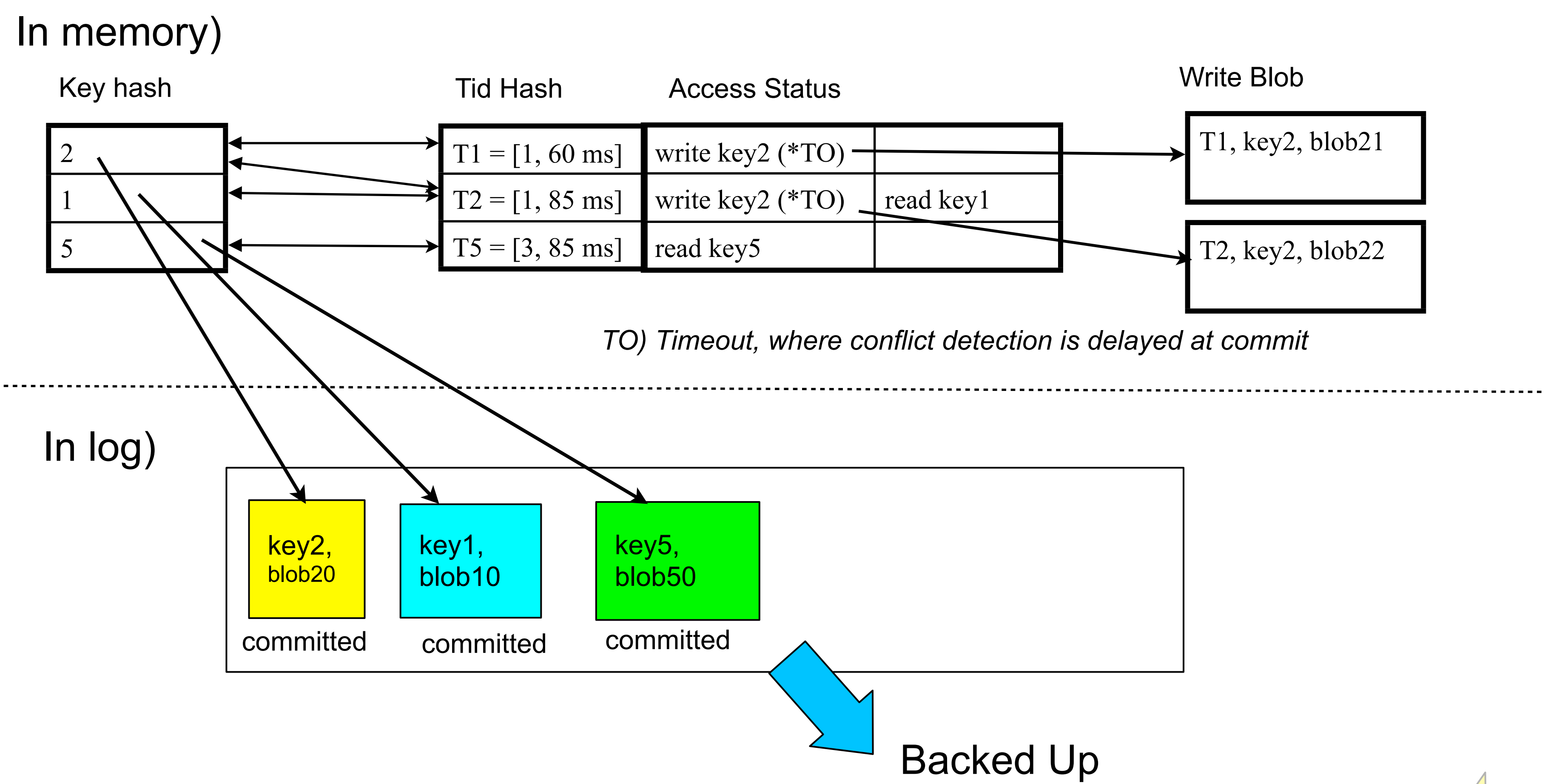
back

r0.64 Add

Master Data Structure

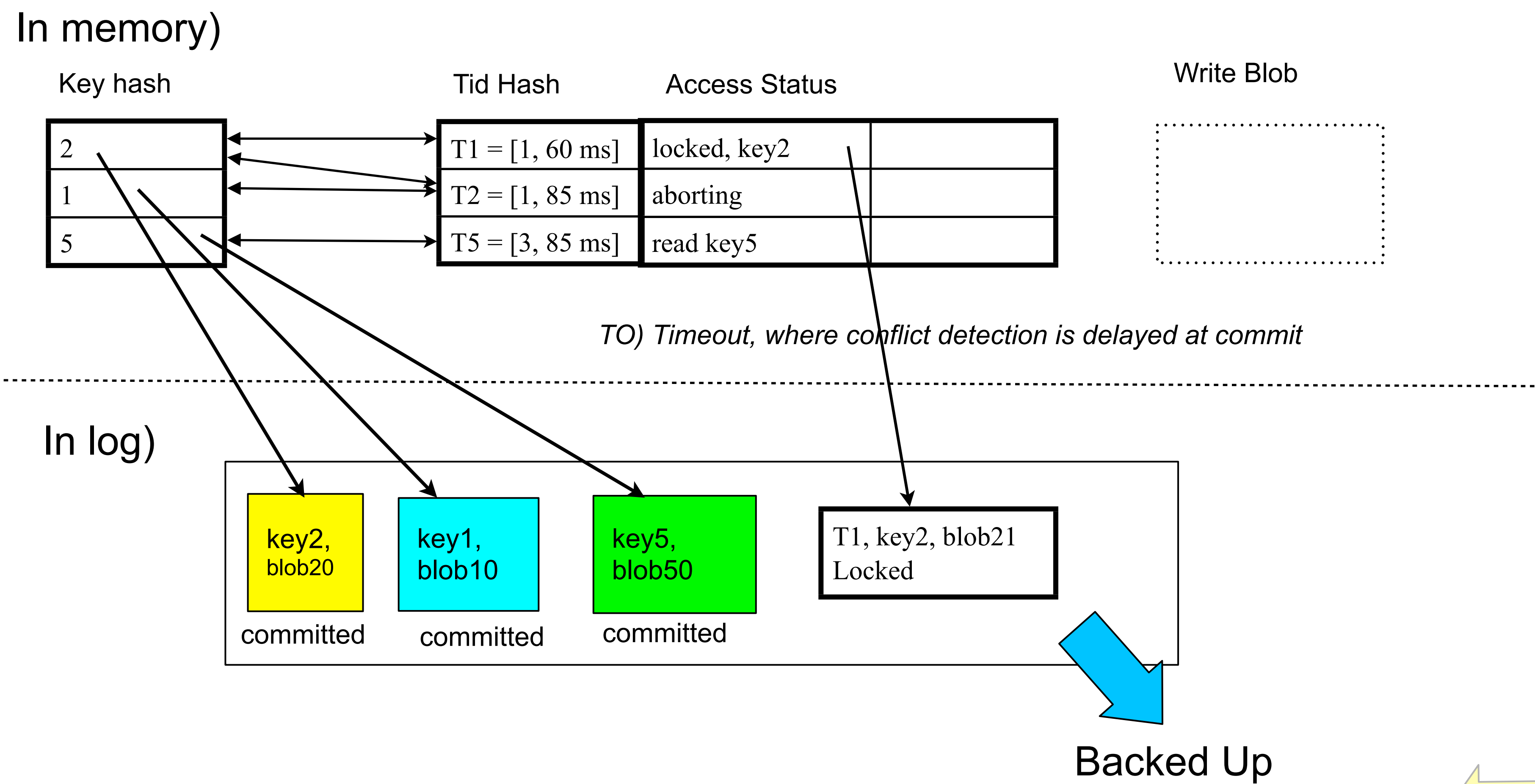
Simple example

- Volatile data lost by crash: hash, array -- any improvement with in-memory log?
- Non-volatile data: locked object, committed object in log



Master Data Structure: at Commit

Simple example



Master Control

To Be Done.

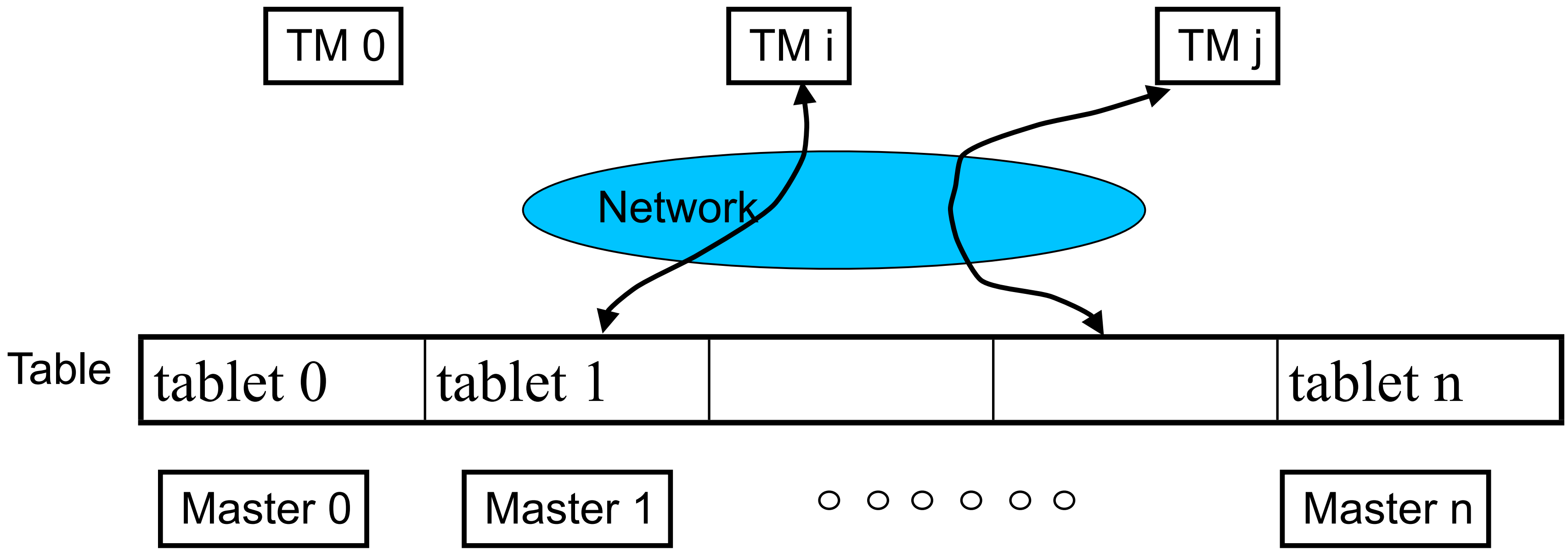
36

back

S. Matsushita, 10/25/2013, rev. 0.64

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



back

S. Matsushita, 10/25/2013, rev. 0.64