Proposal of Transaction on RAMCloud

rev0.61

06 Oct. 2013

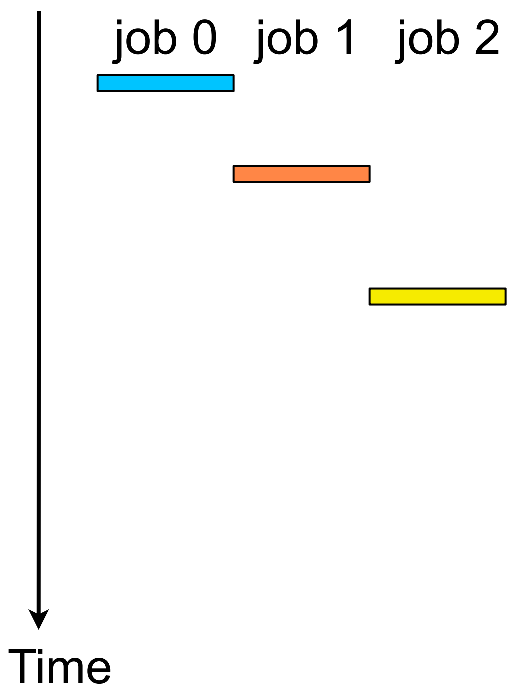
Satoshi Matsushita

1

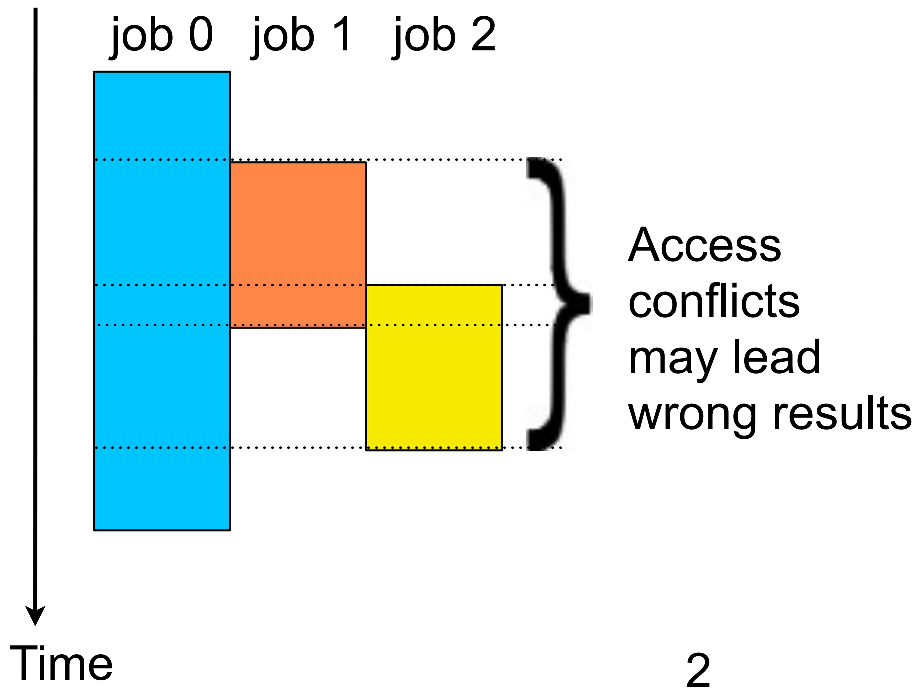
Solution

- Resolve resource access conflicts in parallel execution
- Requirement)
  - ACID: (*atomicity*, *consistency*, *isolation*, *durability*)
  - CAP Theorem: (Can relax partition tolerance) - discuss later

Ideal Parallel Execution)



Reality)

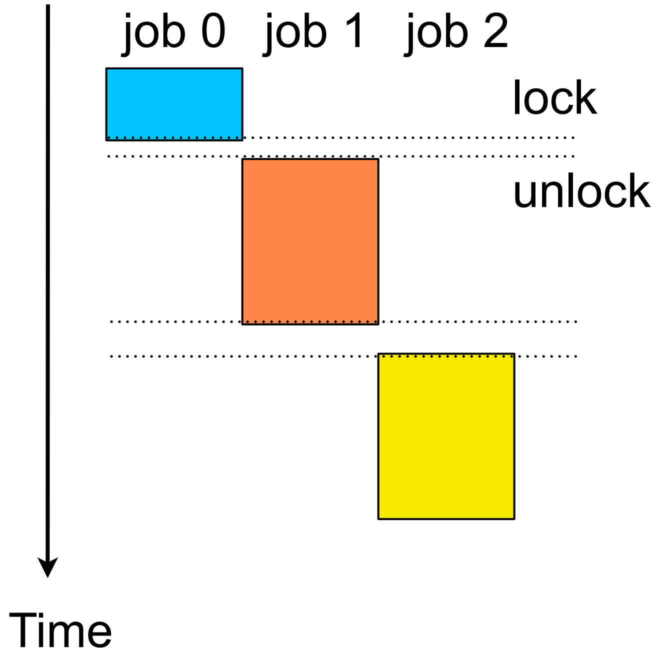


2

S. Matsushita, 10/16/2013, rev. 0.61

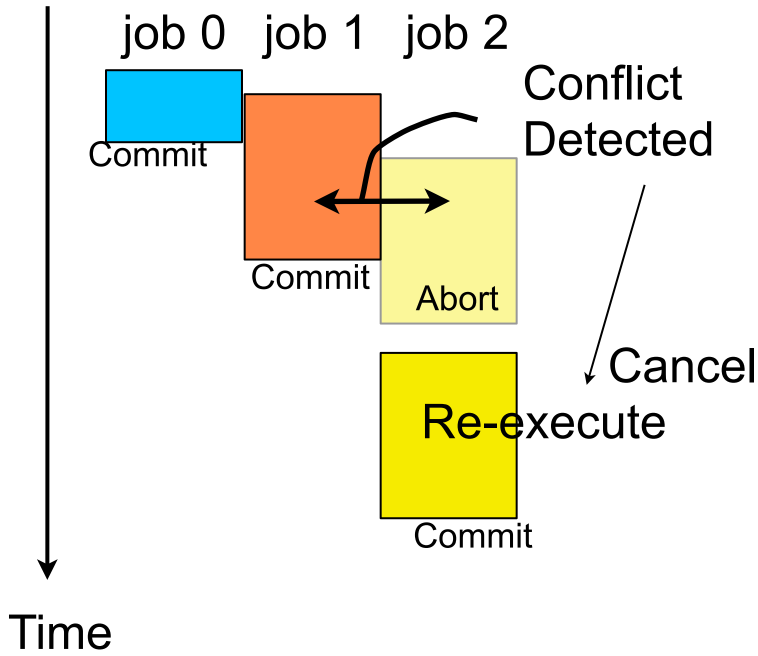
Solution

Pessimistic Lock)



Text

Optimistic Lock)



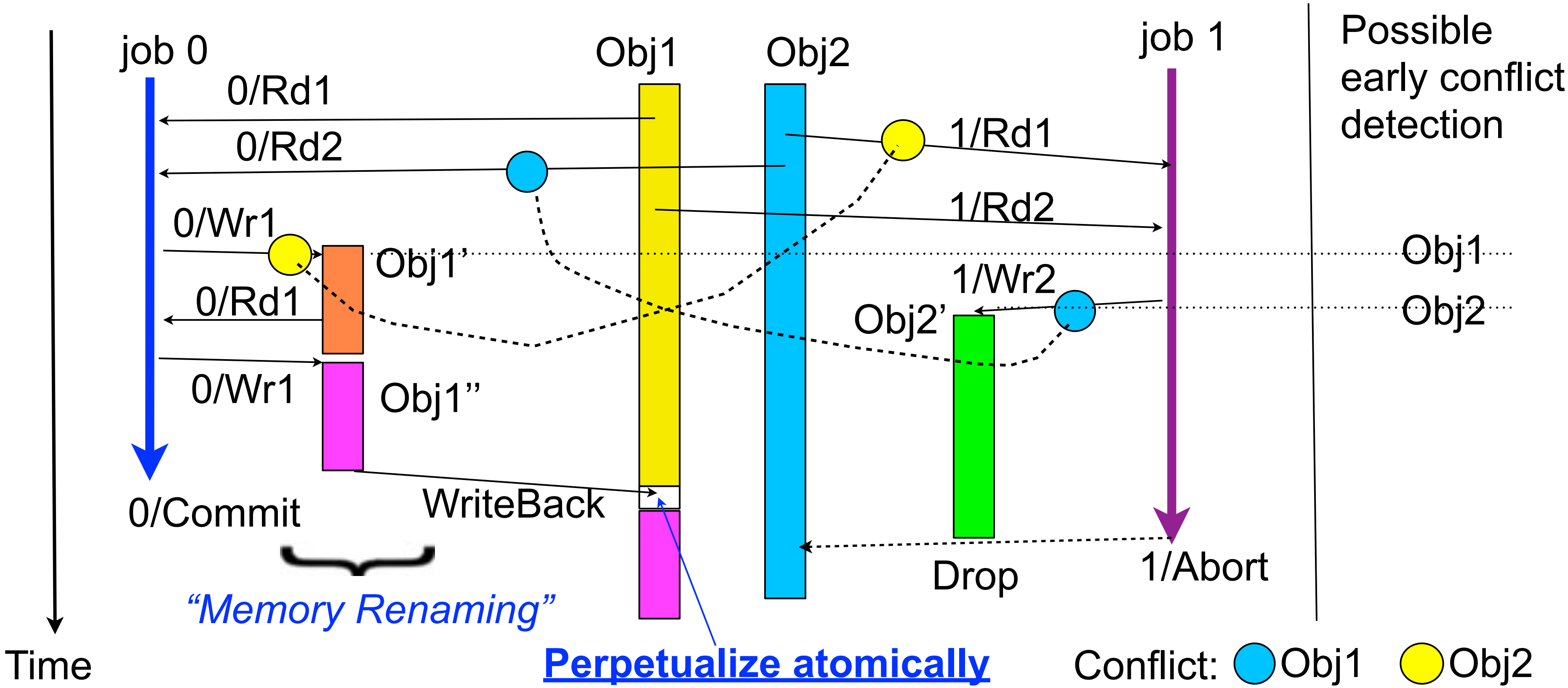
Problem)
- Lower parallelism with giant locks
- Dead lock prone with fine locks
- Need releasing lock with node crash

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

3

Optimistic Lock: General Solution

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



4

S. Matsushita, 10/16/2013, rev. 0.61

Design Assumptions

- Transaction life varies between short to long
  - Try early detection of conflict with avoiding live lock
- Small probability of conflicts
  - Use optimistic lock based design
  - Otherwise use pessimistic lock at user level
- Small number of server nodes involved in a transaction
  - Small probability of node failure during a transaction
  - Faster crash recovery around 1 sec
  - Can yield to blocking algorithm to prevent corner cases

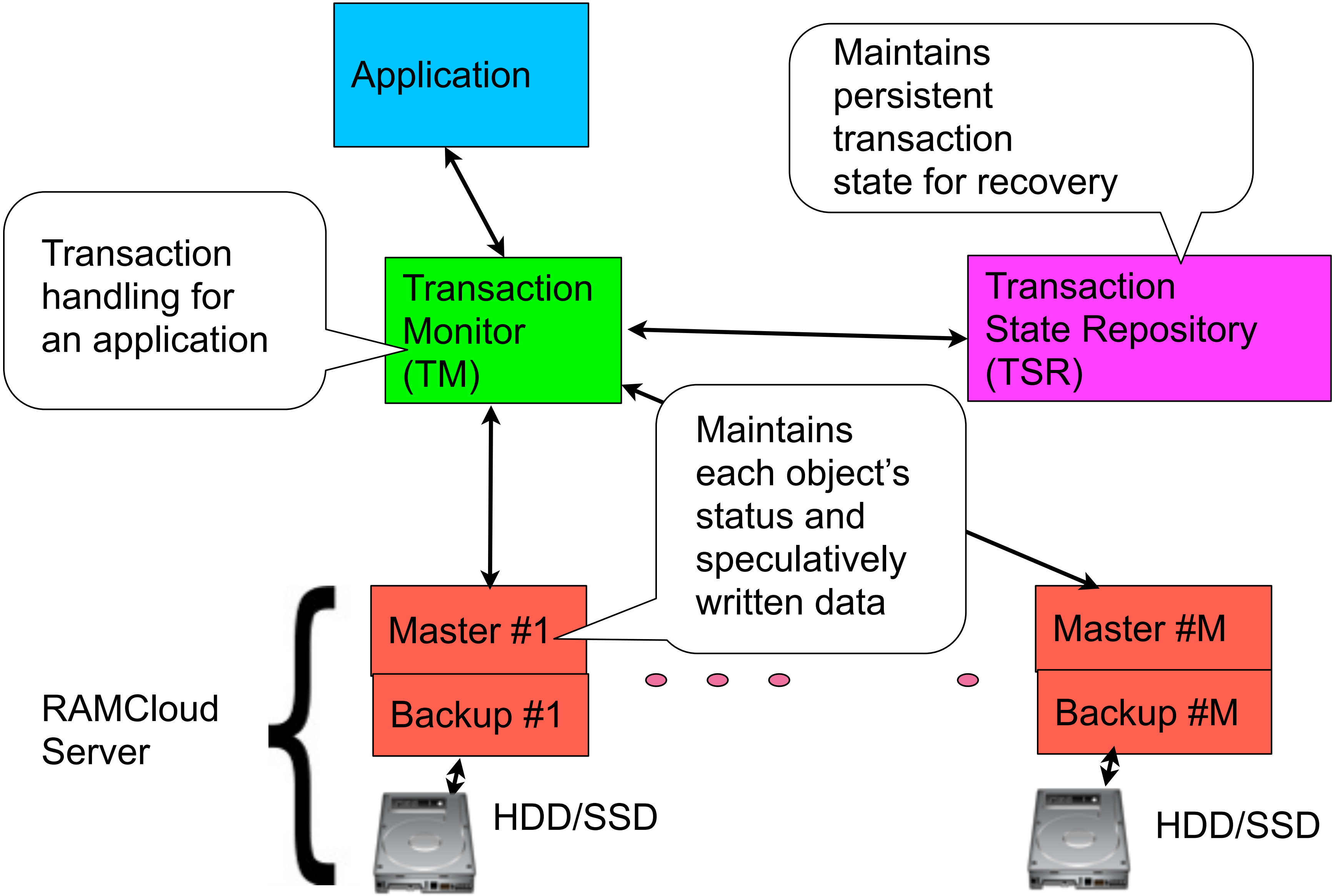
- First implement and tune hot-spot with real data

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

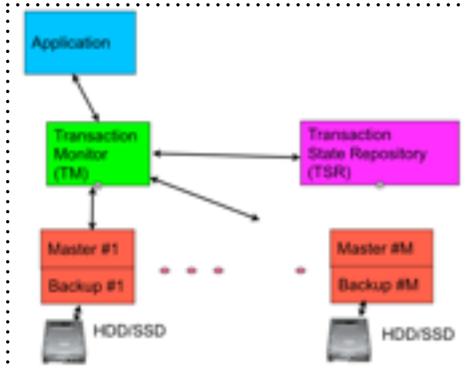
S. Matsushita, 10/16/2013, rev. 0.61

Components



S. Matsushita, 10/16/2013, rev. 0.61

Components - Functions

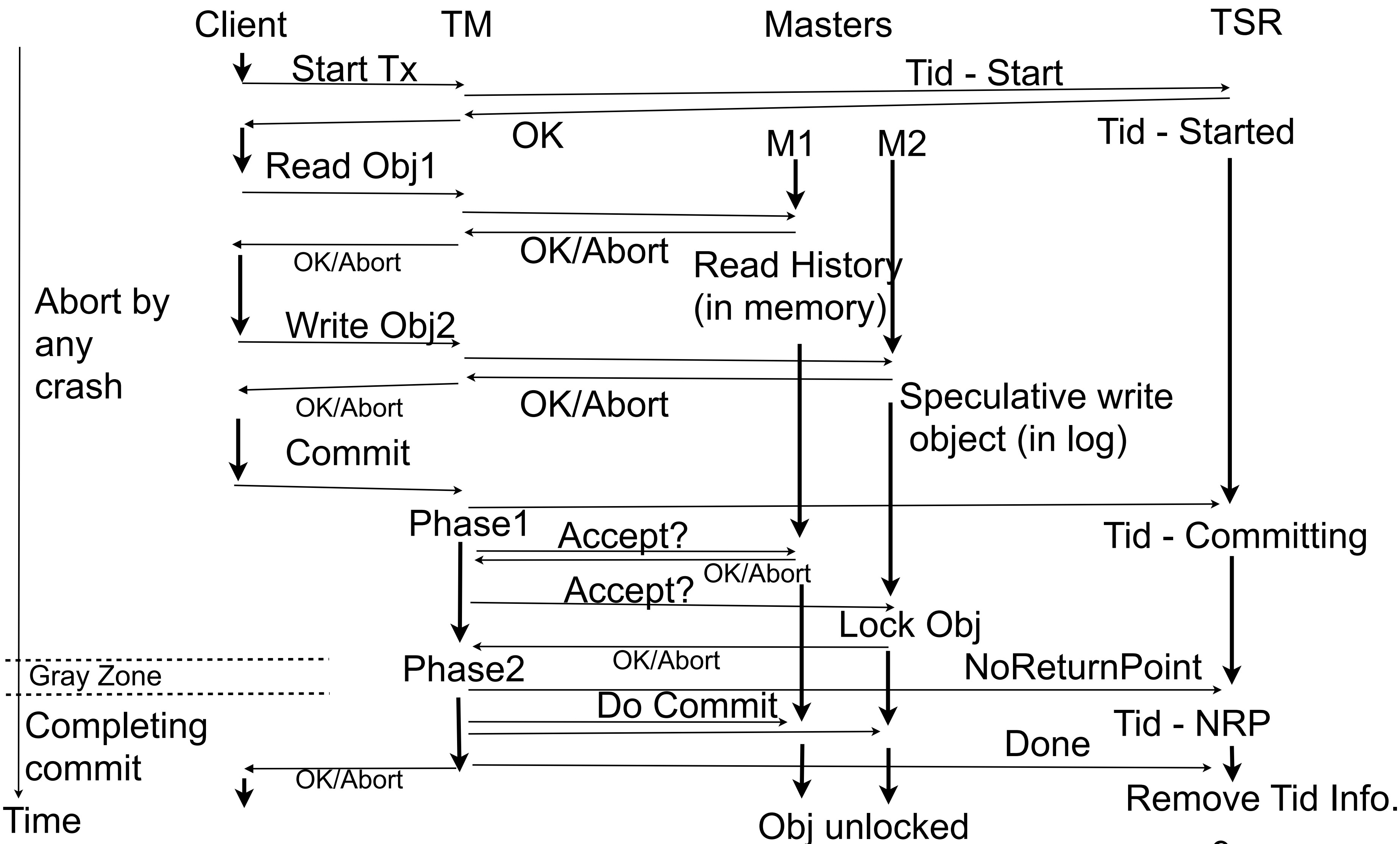


- If client application is restarted <u>immediately</u> (by coordinator, etc), TM can be implemented in client library.

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 (**<u>resource unlock</u>**)	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



S. Matsushita, 10/16/2013, rev. 0.61

9

Detailed discussion: outline

1. Client API
2. Conflict Management
   i. Resolution at object access with transaction priority
   ii. TMid/Tid for unique global transaction order
   iii. Timeout to avoid deadlock
3. Commit - transition from non-blocking to blocking
   (Gray zone solution)
4. 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
5. Data structure of entities
6. Optimization
   i. Callback instead of piggyback
   ii. Separate key/state and data for objects in log

1. Client API

- 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-...(),
    We can make tx_read, tx_write by default using tid=0 for non transactional operation.
- Commit Transaction
  - tx_commit(tid, &state);
  - tx_abort(tid, &state);
- Status
  - tx_status(tid, &state);   // return current transaction state

S. Matsushita, 10/16/2013, rev. 0.61

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

S. Matsushita, 10/16/2013, rev. 0.61

Tid, TMid

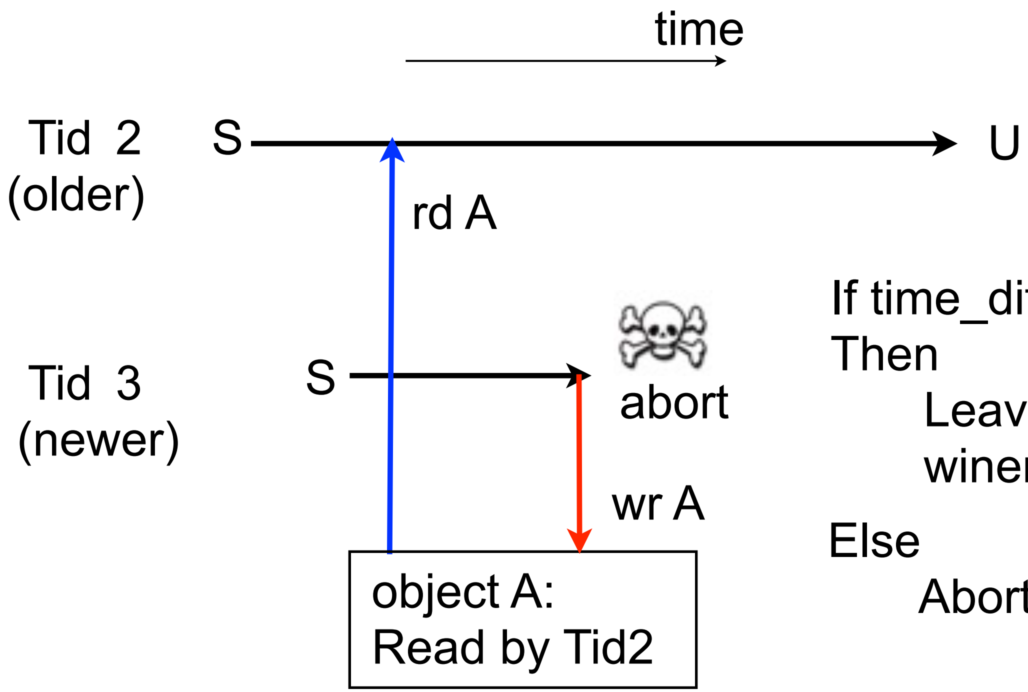
- TMid is given by coordinator at TM startup
- Tid
  - Define Tid = (TMid, TM-localtime) at a transaction generation
  - Compare TMid only when local time is the same
  - Preciseness is not needed, because Tid is just a priority to decide winner transaction at object access time.

S. Matsushita, 10/16/2013, rev. 0.61

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)

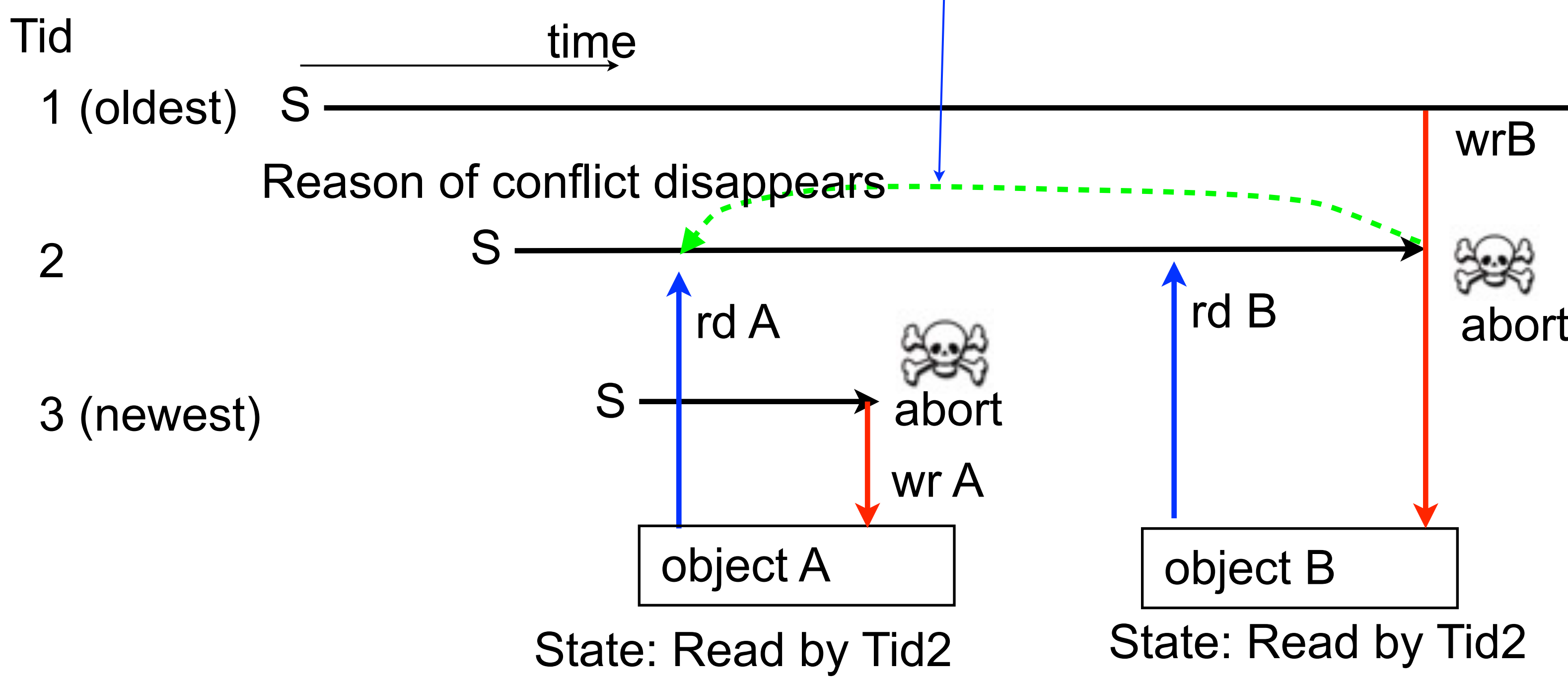


If time_difference(Tid3, Tid2) > Tout
Then
Leaves both alive and decides winer at commit time.

Else
Abort transaction with newer Tid

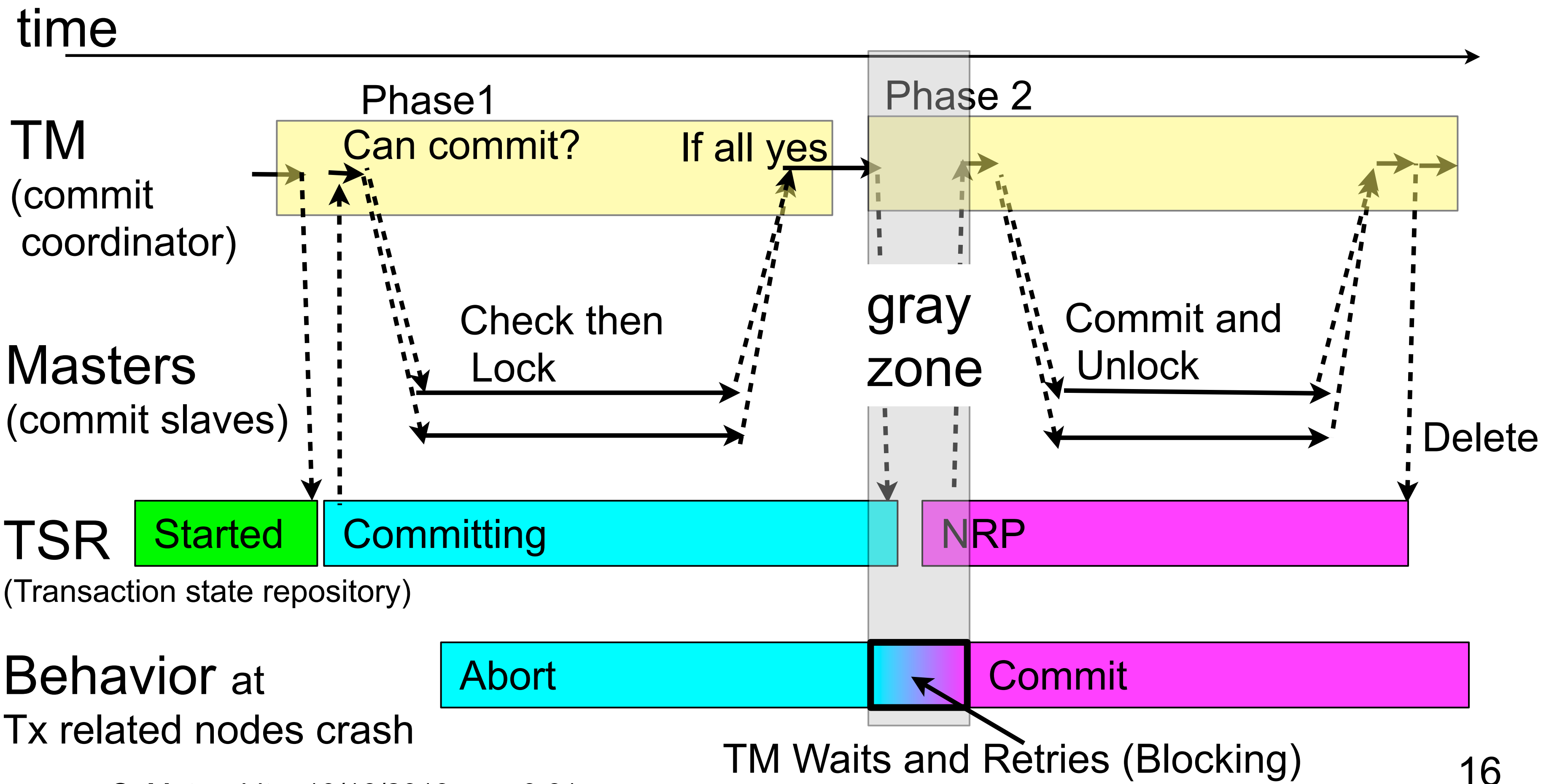
Issues - False abort/Status piggyback

- 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

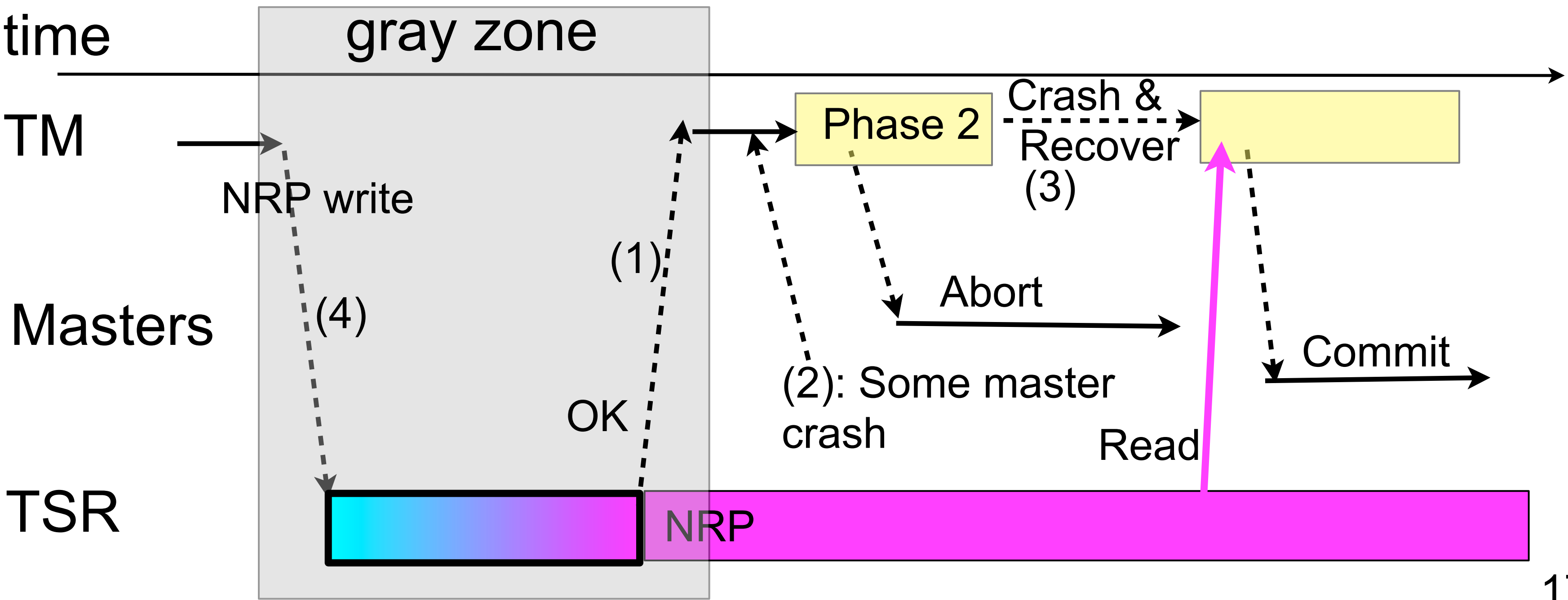
- TM coordinates commit operation
- Save durable state in TSR
  - Committing: unlock object by abort (optimization)
  - NRP: no-return-point for durable transition to commit



S. Matsushita, 10/16/2013, rev. 0.61

3. Commit - Racing conditions

- Racing condition:
  - After NRP is written, TM start aborting in Phase2 due to (1) lost ‘OK’ or (2) relevant node crash
  - Then TM crashes and recovered TM read NRP and start 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 decides behavior.
  - After initiating (4), TM stop aborting Tx by relevant node crash.



S. Matsushita, 10/16/2013, rev. 0.61

Crash Recovery - Clean up

- TM crash
  - completes commit/abort
    - Commits transaction if NRP is found. Otherwise abort transaction.
    - Fast restart required because other clients are blocked by accessing the locked objects
- Server crash
  - Reconstruct hash and object status in memory from log
- TSR crash
  - Recover status of transactions

S. Matsushita, 10/16/2013, rev. 0.61

Implementation of entities

- TM - item 1 seems simplest and good for performance.
  1. In client library such as crt0.
     - Need client recovery mechanism by coordinator
  2. In a master
     - Need a location decision and lookup by coordinator
     - Extra traffic and latency given because all the access for the transaction goes to the master first.
  3. In a separate agent in server node
     - Need another recovery mechanism.
- TSR
  - In a master with defining a table and save transaction state as a normal object.