

Advanced Distributed Systems

Karl M. Göschka
Karl.Goeschka@tuwien.ac.at

[http://www.infosys.tuwien.ac.at/teaching/courses/
AdvancedDistributedSystems/](http://www.infosys.tuwien.ac.at/teaching/courses/AdvancedDistributedSystems/)

1

Transactions

- Transaction model and concepts
- Serializability and recoverability
- Concurrency control
- Distributed commit: 2PC, 3PC
- Recovery

2

A client's banking transaction

In case of a crash,
money would vanish

Transaction T:
a.withdraw(100);
b.deposit(100);
c.withdraw(200);
b.deposit(200);

4

Goal of Transactions

- Indivisible set of operations on objects
 - all completed or no effect at all
 - other clients' transactions can not observe partial effects (prevent interference)
 - tolerate server crash and message omission
 - allow involvement of several servers
- Originate from business, then DBMS, now part of the middleware (e.g. CORBA OTS)
- Recoverable objects** needed (persistent memory)
- Basic synchronization (Programming language, OS, CPU) is required, but not sufficient

5

Synchronization Issues (1)

What's wrong?

```
public class Point { private float x, y;
public float x() { return x; }
public float y() { return y; }}
```

```
// in some other class:
public synchronized void print(Point p) {
    System.out.print("Coordinates are " + p.x() + " and " + p.y());
}
```

```
// alternatively with smaller protected part
public void print(Point p) { float safeX, safeY;
    synchronized(this) {safeX = p.x(); safeY = p.y();}
    System.out.print("Coordinates are " + safeX + " and " + safeY);
}
```

7

Synchronization Issues (2)

```
// in some other class:
// with small protected part
public void print(Point p) { float safeX, safeY;
    synchronized(p) {safeX = p.x(); safeY = p.y();}
    System.out.print("Coordinates are " + safeX + " and " + safeY);
}
```

oK!

```
public class Point { private float x, y;
public float x() { return x; }
public float y() { return y; }
public synchronized void setXandY(
    float newX, float newY){
    x = newX; y = newY;}
}
```

8

ACID properties

□ „All or nothing“:

- Failure **A**tomicity: Effects are atomic even if server crashes
- **D**urability: If completed successfully, all effects are saved in permanent storage

□ Isolation: Intermediate effects not be visible for others

Serially equivalent!

- **C**onsistency (data integrity): not in the *responsibility* of transaction handling, but
 - may have to be relaxed during transactions
 - can be compromised by weak isolation

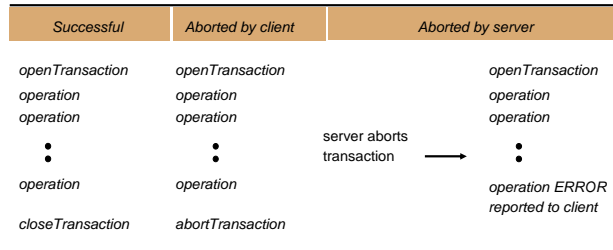
15

Transaction coordinator

- Cooperation between client, recoverable objects, and coordinator
- TID as
 - extra argument or
 - passed implicitly by middleware (e.g. CORBA transaction service)
- Completion: commit or abort
- Abort reasons:
 - Server abort: Nature of the transaction itself or conflicts or crashing
 - Deliberate client abort

16

Transaction life histories



18

Actions related to crashes

- Service action:
 - server crash: recovery, abort uncommitted transactions
 - client crash: expiry time
- Client action:
 - completion (re-do) or abandonment
 - consult the human user

19

Classification of Transactions

- Flat transactions
 - no commit/abort of partial results
 - long-running transactions
- Nested transactions
 - subtransactions are **atomic**, may run **concurrently**
 - subtransactions can **commit/abort independently**
 - permanence only for **top-level** transaction
- Distributed transactions
 - Data is distributed
 - Logically a flat transaction

20

Nested Transaction Rules

- Commit/abort only if child Tx have completed
- Subtransaction independent
 - abort (final)
 - commit (provisionally)
- Parent abort → all subTx abort
- SubTx abort → **parent can decide**
- Top-level Tx commit: All prov. comm. subTx can commit, too, if no ancestor has aborted

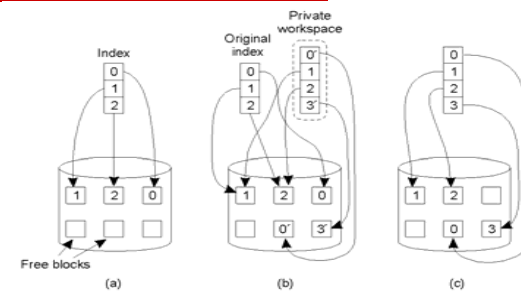
22

Implementation of Transactions

- Basic principles, file system example
- Private workspace
 - Read: Real file used (unless changed since Tx started)
 - Write: Copied to private workspace (copy index file → **granularity** „block“)
- Writeahead log
 - Files are modified in place
 - Before that: **Log record** (old/new)
 - Abort requires rollback

24

Private Workspace



- The file index and disk blocks for a three-block file
- The situation after a transaction has modified block 0 and appended block 3
- After committing

25

Writeahead Log

x = 0;	Log	Log	Log
y = 0;			
BEGIN_TRANSACTION;			
x = x + 1;	[x = 0/1]	[x = 0/1]	[x = 0/1]
y = y + 2		[y = 0/2]	[y = 0/2]
x = y * y;			[x = 1/4]
END_TRANSACTION;			
(a)	(b)	(c)	(d)

- a) A transaction
- b) – d) The log before each statement is executed

26

Transactions

- Transaction model and concepts
- Serializability and recoverability
- Concurrency control
- Distributed commit: 2PC, 3PC
- Recovery

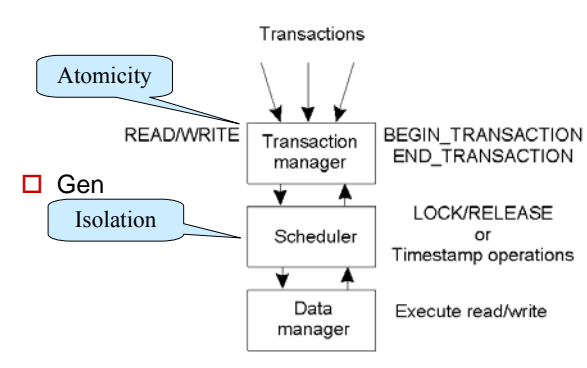
27

Concurrency Control

- Layering model
- Several well-known problems → Serial equivalence
 - lost update
 - inconsistent retrieval
- More problems in case of abort → Recoverability
 - dirty reads
 - premature writes
- Concurrency control protocols: can be achieved either by transactions **waiting** or by **restarting** after **conflicts** have been **detected**
 - locking
 - optimistic concurrency control
 - timestamp ordering

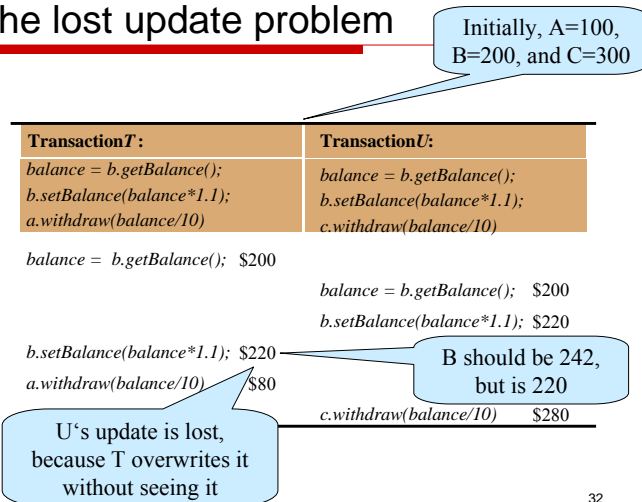
28

Concurrency control model(1)



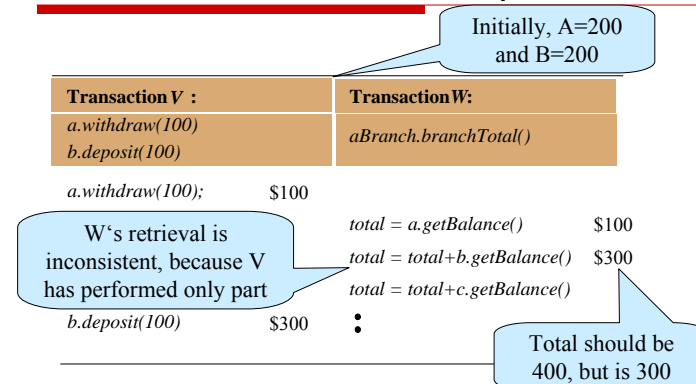
29

The lost update problem



32

The inconsistent retrievals problem



33

Serial equivalence

- An interleaving of the operations of concurrent transactions in which the combined **effect** is the same **as if** the transactions **had performed** one at the time in **some order** is a *serially equivalent* interleaving.
- Same effect:
 - all **read** op's return the same values
 - instance variables same **state** at the end
- Solves lost update and inconsistent retrieval

34

A serially equivalent interleaving T/U

TransactionT:	TransactionU:
<i>balance = b.getBalance()</i>	<i>balance = b.getBalance()</i>
<i>b.setBalance(balance*1.1)</i>	<i>b.setBalance(balance*1.1)</i>
<i>a.withdraw(balance/10)</i>	<i>c.withdraw(balance/10)</i>

<i>balance = b.getBalance()</i>	\$200	
<i>b.setBalance(balance*1.1)</i>	\$220	→
		<i>balance = b.getBalance()</i>
		\$220
<i>a.withdraw(balance/10)</i>	\$80	<i>b.setBalance(balance*1.1)</i>
		\$242
		<i>c.withdraw(balance/10)</i>
		\$278

35

A serially equivalent interleaving V/W

TransactionV:	TransactionW:
<i>a.withdraw(100);</i>	<i>aBranch.branchTotal()</i>
<i>b.deposit(100)</i>	

<i>a.withdraw(100);</i>	\$100	
<i>b.deposit(100)</i>	\$300	→
		<i>total = a.getBalance()</i>
		\$100
		<i>total = total+b.getBalance()</i>
		\$400
		<i>total = total+c.getBalance()</i>
		...

36

Conflicting operations

- Combined effect depends on the order
- Simplification: read and write
 - read: effect = result returned
 - write: effect = value set by write
- For two transactions to be **serially equivalent**, it is **necessary and sufficient** that **all pairs of conflicting** operations of the two transactions be executed in the **same order** at all of the objects they both access
- → this is „conflict serializability“ – there are other models as well, but harder to implement

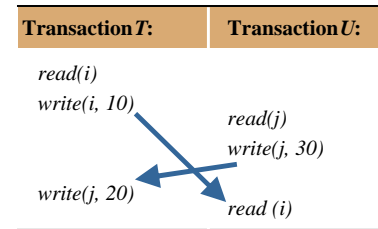
37

Read and write operation conflict rules

Operations of different transactions	Conflict	Reason
read read	No	Because the effect of a pair of read operations does not depend on the order in which they are executed
read write	Yes	Because the effect of a read and a write operation depends on the order of their execution
write write	Yes	Because the effect of a pair of write operations depends on the order of their execution

38

Non-serially equivalent interleaving



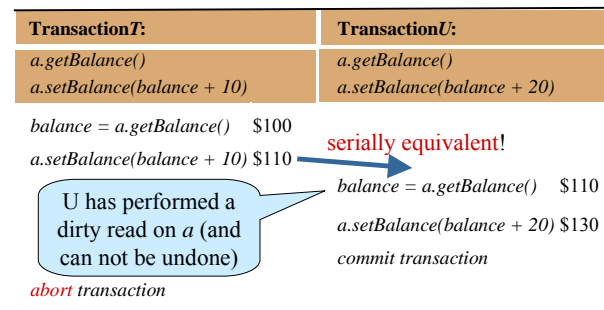
39

Recoverability from aborts

- A transaction may abort → Aborted transaction may not affect other concurrent transactions.
- Two problems that can occur even in the presence of serially equivalent executions:
 - dirty reads: transactions must not see the **uncommitted** state of other transactions
 - premature writes

40

A dirty read when transaction T aborts



41

Transactions

- Transaction model and concepts
- Serializability and recoverability
- Concurrency control
- Distributed commit: 2PC, 3PC
- Recovery

46

Concurrency Control

- Layering model
- Two well-known problems → Serial equivalence
 - lost update
 - inconsistent retrieval
- Recoverability
 - dirty reads
 - premature writes
- Concurrency control protocols: can be achieved either by transactions **waiting** or by **restarting** after **conflicts** have been **detected**
 - locking
 - optimistic concurrency control
 - timestamp ordering

47

Locking

- serializing access to the objects
- most practical systems use locking
- transactions wait or share the lock (read)
- deadlock may occur

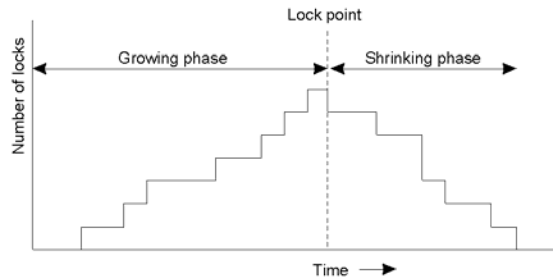
48

T and U with exclusive locks

TransactionT:		TransactionU:	
<i>balance = b.getBalance()</i>		<i>balance = b.getBalance()</i>	
<i>b.setBalance(bal*1.1)</i>		<i>b.setBalance(bal*1.1)</i>	
<i>a.withdraw(bal/10)</i>		<i>c.withdraw(bal/10)</i>	
Operations	Locks	Operations	Locks
<i>openTransaction</i>		<i>openTransaction</i>	
<i>bal = b.getBalance()</i>	lock B	<i>bal = b.getBalance()</i>	waits for T's lock on B
<i>b.setBalance(bal*1.1)</i>		...	lock B
<i>a.withdraw(bal/10)</i>	lock A	<i>b.setBalance(bal*1.1)</i>	
<i>closeTransaction</i>	unlock A, B	<i>c.withdraw(bal/10)</i>	lock C
		<i>closeTransaction</i>	unlock B, C

49

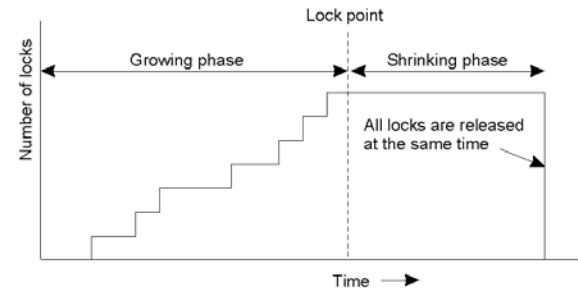
Two-Phase Locking (1)



- Two-phase locking: A transaction is not allowed any new lock after it has released a lock → to ensure **serial equivalence**

50

Two-Phase Locking (2)



- **Strict** Two-phase locking: A transaction holds all locks till completion to avoid dirty reads or premature writes → to ensure **strict execution**

51

Concurrency control protocol

- Granularity: site, class, object, data item
- Basic operations (read and write) are atomic
- Cope with conflicts on objects (read/write)
- Simple, exclusive lock too restrictive
- → many readers/single writer
- → **read** (shared) locks and **write** locks

52

Concurrency control protocol

- client's request **never rejected**, but may have to wait:
 - If T has already performed **read** on object, U **must not write** until T completes
 - If T has already performed **write** on object, U **must not read nor write** until T completes
- Vice versa:
 - A request for a **write** lock is **delayed** by an existing **read or write** lock
 - A request for a **read** lock is **delayed** by an existing **write** lock only

53

Lock compatibility on particular object

For one object		Lock requested	
		read	write
Lock already set	none	OK	OK
	read	OK	wait
	write	wait	wait

- ❑ Inconsistent retrievals:
 - first: read locks delay subsequent update
 - second: request for read locks delayed until update completed
- ❑ Lost updates: later transactions delay their reads, until earlier ones (writes) have completed.

54

Lock promotion

- ❑ Tx reads object and later wants to write
- ❑ → read lock is promoted to write lock
- ❑ only if no other read lock (else delayed)
- ❑ it is not safe to demote a lock (i.e. write lock to read lock)

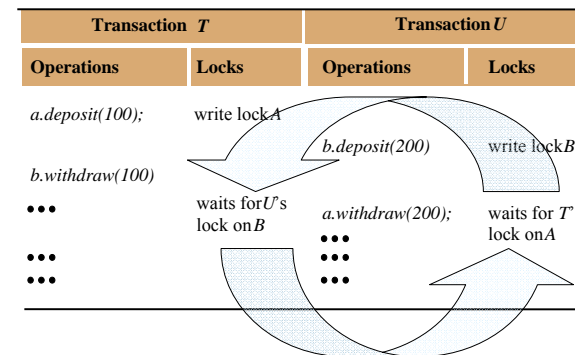
55

Use of locks in strict 2-phase locking

1. When an operation accesses an object within a transaction:
 - (a) If the object is not already locked, it is locked and the operation proceeds.
 - (b) If the object has a conflicting lock set by another transaction, the transaction must wait until it is unlocked.
 - (c) If the object has a non-conflicting lock set by another transaction, the lock is shared and the operation proceeds.
 - (d) If the object has already been locked in the same transaction, the lock will be promoted if necessary and the operation proceeds. (Where promotion is prevented by a conflicting lock, rule (b) is used.)
2. When a transaction is committed or aborted, the server unlocks all objects it locked for the transaction (**strictness ensured**).

56

Deadlock with write locks



60

Deadlocks (1)

- ❑ the use of locks can lead to deadlock
- ❑ finer or mixed **granularity** (hierarchic locking schemes) can avoid conflicts and deadlock
- ❑ although each transaction can only wait for one object at a time, it can be involved in several cycles
- ❑ Solution: e.g. **abort** one transaction from the cycl

64

Deadlocks (2)

- ❑ Deadlock prevention:
 - Request all locks in advance (interactive?)
 - Request locks in particular order
- ❑ Upgrade locks:
 - read lock to be promoted to write lock (conflicts with any other upgrade lock or write lock, but not with read locks)
- ❑ Deadlock detection:
 - detect cycles (overhead)
 - abort **one** transaction (not simple to chose!)
- ❑ **Timeouts** (lock vulnerability): commonly used

65

Resolution of the deadlock

Transaction T		Transaction U	
Operations	Locks	Operations	Locks
<i>a.deposit(100);</i>	write lock A		
		<i>b.deposit(200)</i>	write lock B
<i>b.withdraw(100)</i>			
...	waits for U's lock on B	<i>a.withdraw(200);</i>	waits for T's lock on A
	(timeout elapses)	...	
T's lock on A becomes vulnerable,	unlock A, abort T	...	
		<i>a.withdraw(200);</i>	write locks A, B unlock A, B

U breaks the lock on A of T

66

Timestamp Ordering (1)

- ❑ more pessimistic approach
- ❑ server records most recent time of reading and writing for **each object**
- ❑ **each transaction** has unique timestamp (TO)
- ❑ for **each operation**, the timestamp of the transaction is compared with that of the object:
 - → continue
 - → delay (wait)
 - → reject (abort)

71

Timestamp Ordering (2)

- Basic timestamp ordering:
 - A Tx's **request to write** an object is **valid** only if that object was **last read and written** by **earlier** transactions
 - A Tx's **request to read** an object is **valid** only if that object was **last written** by an **earlier** transaction
- Refinement: tentative versions per Tx, commitment order maintained, Tx's may have to **wait** for earlier Tx's to complete their writes
- No deadlock

72

Timestamp Ordering - Example

- write in tentative versions
- object has a write timestamp (committed) and a set of **tentative versions**, each of which has a write timestamp
- object has a set of read timestamps represented by maximum (latest) one
- write: new version, TS from Tx
- read: directed to version with maximum write TS < Tx; Tx added to the set of read
- commit: tentative versions → objects (including timestamps)

73

Conflicts for timestamp ordering

Rule	T_c	T_i	
1.	write	read	T_c must not write an object that has been read by any T_i where $T_i > T_c$ this requires that $T_c \geq$ the maximum read timestamp of the object.
2.	write	write	T_c must not write an object that has been written by any T_i where $T_i > T_c$ this requires that $T_c >$ write timestamp of the committed object.
3.	read	write	T_c must not read an object that has been written by any T_i where $T_i > T_c$ this requires that $T_c >$ write timestamp of the committed object.

later

- Key idea: Each request by current transaction T_c is checked to see whether it conforms to the conflict rules

75

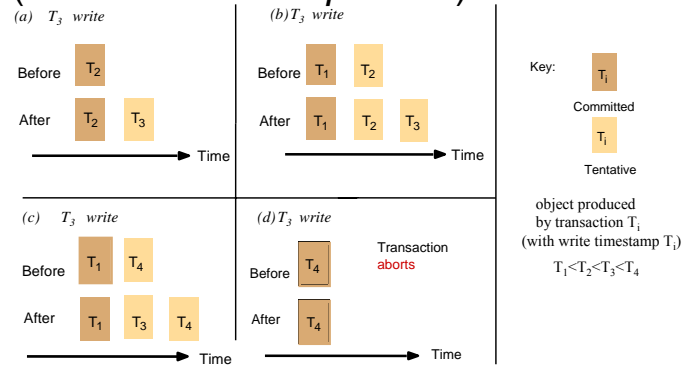
Timestamp ordering write rule

if ($T_c \geq$ **maximum read** timestamp on D &&
 $T_c >$ **write timestamp on committed** version of D)
 perform write operation on tentative version of D with write timestamp T_c
 else /* **write is too late** */
Abort transaction T_c

- Rule 1+2: Rule for accepting writes requested by T_c on object D → accept or abort
- Tentative version with timestamp T_c addressed or created

76

Write operations and timestamps (no read timestamps here)



77

Timestamp ordering read rule

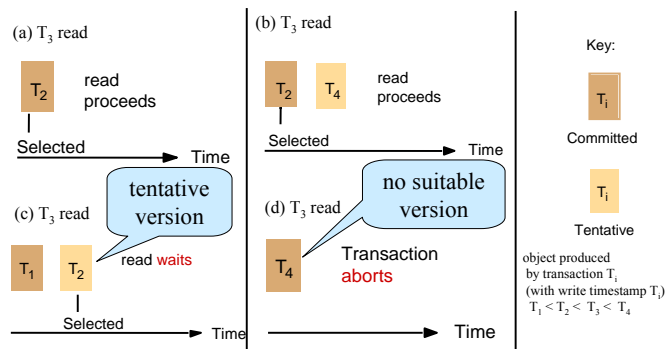
```

if ( $T_c >$  write timestamp on committed version of  $D$ ) {
  let  $D_{\text{selected}}$  be the version of  $D$  with the maximum write timestamp  $\leq T_c$ 
  if ( $D_{\text{selected}}$  is committed)
    perform read operation on the version  $D_{\text{selected}}$ 
  else
    Wait until the transaction that made version  $D_{\text{selected}}$  commits or aborts
    then reapply the read rule /* prevents dirty read */
} else
  Abort transaction  $T_c$  /* read is too late, later Tx has already written */
  
```

- Rule 3: Rule for accepting reads requested by T_c on object $D \rightarrow$ accept, wait, or reject

78

Read operations and timestamps



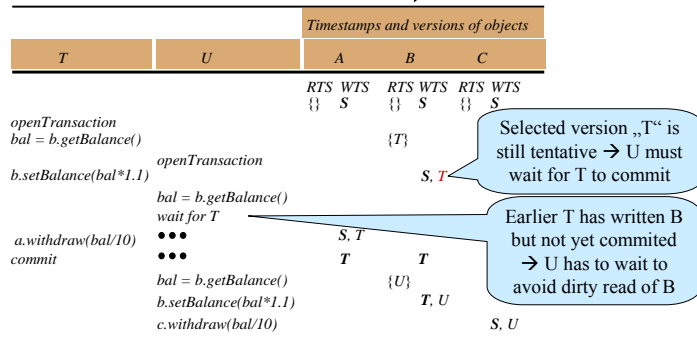
79

Timestamp Ordering - Discussion

- Commit is **always possible**, because all operations are checked immediately
- Committed versions must be created in timestamp order \rightarrow commit may have to wait (already persistent, **no client wait**)
- Here, ordering is strict:
 - Read rule delays read until all previous writes are committed/aborted
 - Commit in order ensures that write is delayed until all previous writes are committed/aborted

80

Timestamps in transaction $S < T < U$



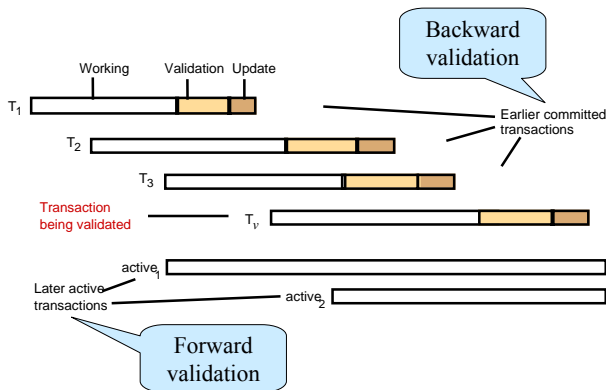
81

Optimistic Concurrency Control

- Drawbacks of locking:
 - Overhead (even for read) although necessary only in the worst case
 - Deadlock (detection/timeout)
 - Strict locking reduces concurrency
- Transaction proceeds until commit („optimistic“)
- then server checks, if conflicting operations have been performed \rightarrow abort
- fits well to „private workspace“ implementation

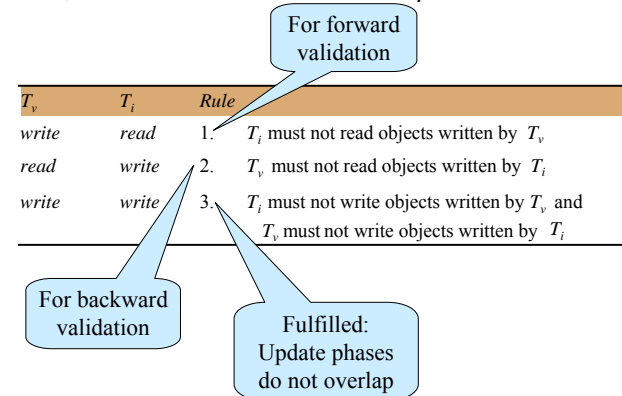
83

Validation of transactions



84

Serializability of transaction T with respect to transaction T_i



85

Transactions

- Transaction model and concepts
- Serializability and recoverability
- Concurrency control
- Distributed commit: 2PC, 3PC
- Recovery

88

Two-Phase Commit (2PC)

Voting phase

- Coordinator sends **vote-request**
- Participants return **vote-commit** or **vote-abort**

Commit phase

- Coordinator sends **global-commit** or (if only one participant aborted) **global-abort**
- Participants **that voted for commit** wait for **final decision** to commit or abort

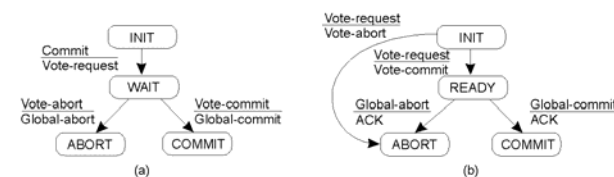
90

Distributed Commit

- remember atomic multicast
- often established by means of a coordinator
- one-phase commit: local commit may not always be possible
- two-phase commit: most common, but can not efficiently handle the failure of the coordinator
- three-phase commit: more robust

89

2PC: Finite state machines



- The finite state machine for the coordinator.
- The finite state machine for a participant.

91

2PC: Problems with failures

- Blocking states may wait for crashed processes
→ timeout mechanisms:
 - Participant in INIT: local abort, send „vote-abort“ to coordinator
 - Coordinator in WAIT: send „global-abort“ to all participants
 - Participant in READY: → can not simply decide!
 - block until coordinator recovers
 - contact another participant Q → next slide

92

2PC: Participant in READY

State of Q	Action by P
COMMIT	Make transition to COMMIT
ABORT	Make transition to ABORT
INIT	Make transition to ABORT
READY	Contact another participant

- Actions taken by a participant *P* when residing in state *READY* and having contacted another participant *Q*.
- If **all other participants** „READY“ → no decision, coordinator's vote needed, blocking (rare case) → „**blocking**“ commit protocol

93

2PC: Crash and recovery

- Participant in INIT: recover to local ABORT and send „vote-abort“ to coordinator
- Participant in COMMIT/ABORT: recover to same state and retransmit decision to coord.
- Participant in READY: When recovering it **can not decide** → contact other participants
- Coordinator in WAIT: retransmit „vote-request“ after recovery
- Coordinator in COMMIT/ABORT: recover decision and retransmit „global-commit/abort“

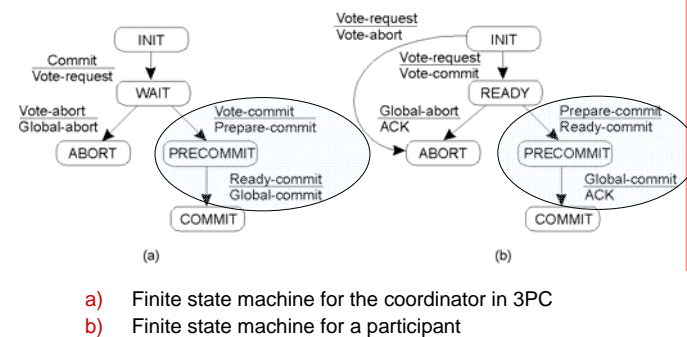
94

2PC: Avoid blocking

- Each participant multicasts received messages to all others
- Three-phase commit protocol
 - not applied often in practice
 - two conditions:
 1. No single state from which it is possible to make a transition to either COMMIT or ABORT
 2. No state, in which it is **not possible** to make a **final decision**, and from which a **transition to COMMIT** can be made
 - these two conditions are sufficient and necessary for a **non-blocking commit protocol**

98

Three-Phase Commit (3PC)



99

3PC: Differences

- Phase 1: Participants are in INIT, ABORT, and READY
- Phase 2: All participants are
 - either in READY + ABORT (2A)
 - or in READY + PRECOMMIT (2B)
- Phase 3: All participants are in PRECOMMIT and COMMIT
- **Coordinator blocked** in PRECOMMIT: Participant crashed, but voted commit → **global-commit** oK (Participant will later recover accordingly and commit)

101

3PC: Differences

- If a participant is **READY**, then others are
 - INIT, ABORT, READY (phase 1) or
 - ABORT, READY (phase 2A) or
 - PRECOMMIT, READY (phase 2B)
- If a participant is **PRECOMMIT**, then others are
 - PRECOMMIT and READY (phase 2B) or
 - PRECOMMIT and COMMIT (phase 3)
- → no participant can be in state INIT, when another participant is in PRECOMMIT

102

3PC: Crash and recovery

- Participant blocked in **PRECOMMIT**: Coordinator crashed, contact other participants:
 - if COMMIT → commit
 - if majority of participants → commit
 - precommitted/committed participant may become **new coordinator** to finish the protocol (the other participants and the coordinator will recover accordingly)
 - Majority: against different decisions in partitions (majority voting)

103

3PC: Crash and recovery

- Participant blocked in **READY**: Coordinator crashed, contact other participants:
 - if INIT, ABORT → abort
 - if **each** is READY (and **majority**) → abort!
→ Crashed participants may recover to INIT or ABORT, so only abort is safe (READY and PRECOMMIT can still abort!)
 - if **one** is PRECOMMIT (and **majority**) → commit!
→ Crashed participants may recover to COMMIT, so only commit is safe (READY and PRECOMMIT can commit as well!)

104

Transactions

- Transaction model and concepts
- Serializability and recoverability
- Concurrency control
- Distributed commit: 2PC, 3PC
- Recovery

105

Recovery

- Backward recovery:
 - previously correct state
 - checkpoints: record and restore states
 - generally applicable
 - costly
 - no failure avoidance
 - some operations are irreversible (e.g. ATM)
- Forward recovery:
 - bring the system in correct new state
 - error has to be known in advance

106

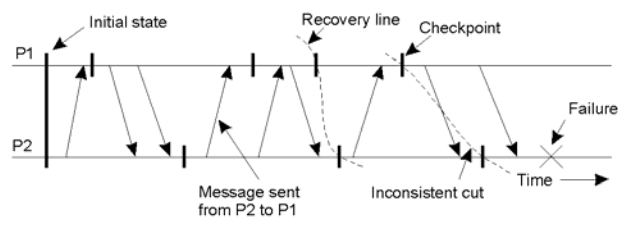
Checkpointing

- Record a consistent global state (distributed snapshot)
- Recovery line: most recent consistent cut
- Construct consistent global state from local states:
 - Independent checkpointing: Find a recovery line → domino effect
 - Coordinated checkpointing: synchronized

108

Checkpointing

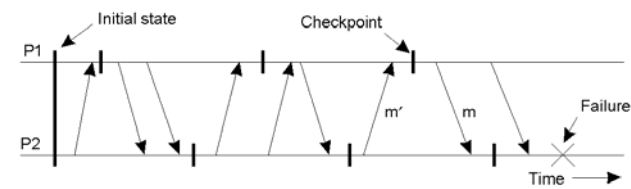
- A recovery line.



109

Independent Checkpointing

- The **domino effect**: all cuts are inconsistent → falls back to the initial state!



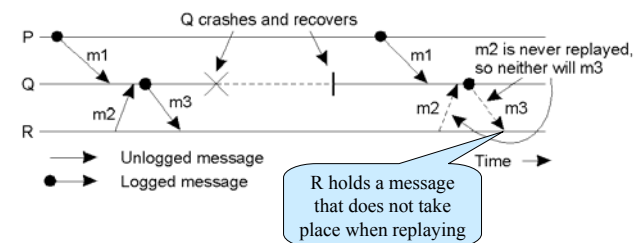
110

Message logging

- Checkpointing is expensive
- Idea: Replay transmission of messages
- Piecewise deterministic model (nondeterministic: e.g. receipt of message)
- When to log?
- How to deal with **orphan processes** (survives crash of other process, but is inconsistent with the crashed process after recovery)
 - pessimistic logging protocols (preferred way)
 - optimistic logging protocols

111

Message Logging



- Incorrect replay of messages after recovery, leading to an **orphan process**.

112

Summary

- Atomic operations and thread communication
- Flat, nested, and distributed transactions
- Private workspace and writeahead log
- Atomicity, Durability, and Isolation
- Lost update and inconsistent retrieval → serial equivalence
- Dirty read and premature write → strict
- Locking: 2PL, strict 2PL, deadlock
- Timestamp ordering and optimistic CC
- Distributed Commit: 2PC, 3PC
- Recovery: Checkpointing, message logging

113