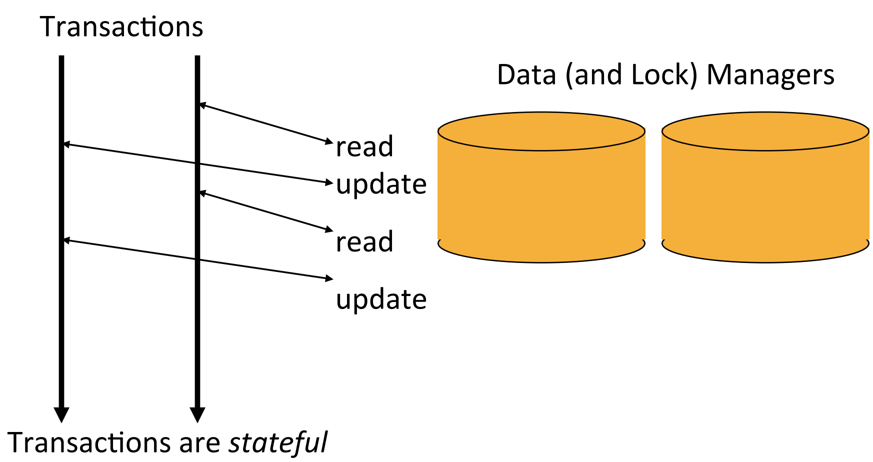
# Transactions

Dominic Duggan
Stevens Institute of Technology

1

# Transaction and Data Managers




2

## Atomicity

- Either a transaction happens completely, or it does not happen at all
- If a transaction happens, it appears to happen as a single indivisible action



## Consistency

- If the system has certain invariants, then if they held before a transaction, they hold after the completion of that transaction
  - Law of conservation of money
- That doesn't mean that the invariants need to be true throughout the transaction

## Isolation (Serializability)

- Final result looks as though all transactions ran sequentially in some (system-dependent) order
  - That is, they appear to have run serially, rather than in parallel



## Durability

- Once a transaction is successfully committed, regardless of what happens afterward, its results become permanent
- Specifically, no failure after the commit can undo the results or cause them to be lost

## IMPLEMENTATION

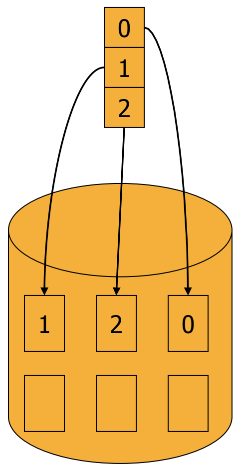
7

## Two Implementation Methods

- Private Workspace
  - Shadow paging
- Writeahead Log
  - Log of updates, including original values

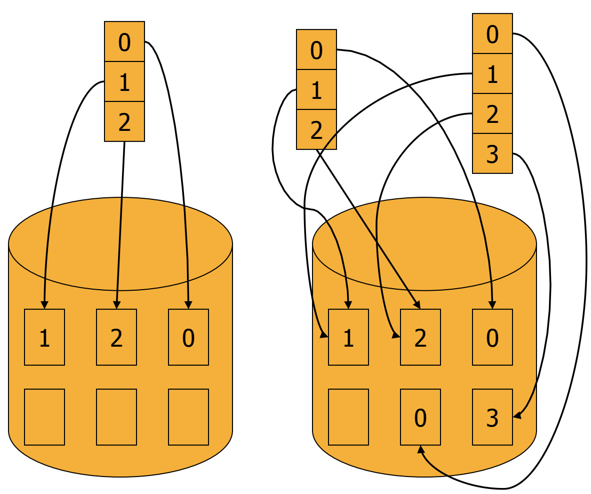
8

## Private Workspace

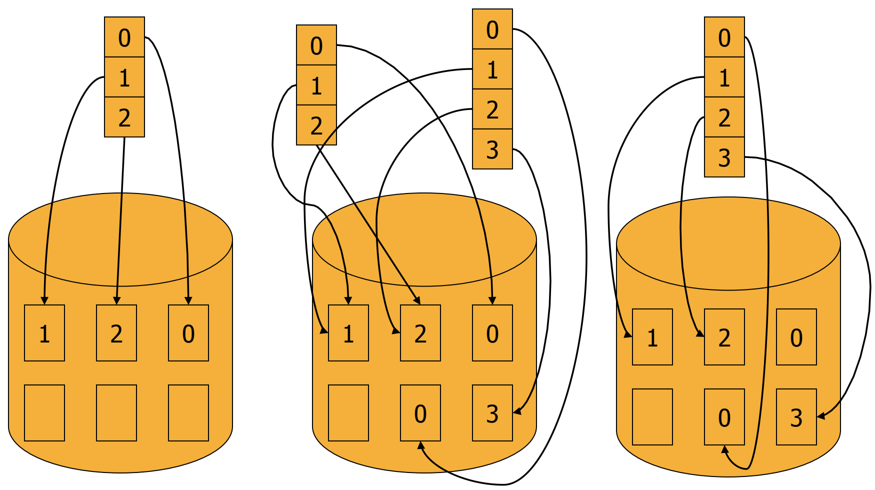





## Private Workspace



## Private Workspace






## Writeahead Log

- DB modified in place
- Log record:
  - transaction identifier
  - file and block that are being changed
  - old and new values of the block
- DB only changed after log record has been successfully written
  - File update may happen some time after log written
  - File/database access slower than log append

## Writeahead Log Example

```
x = 0;
y = 0;
BEGIN_TRANSACTION;
 (1) x = x + 1;
 (2) y = y + 2;
 (3) x = y * y;
END_TRANSACTION;
```

```
(1) [x = 0/1]

(2) [x = 0/1]
    [y = 0/2]

(3) [x = 0/1]
    [y = 0/2]
    [x = 1/4]
```

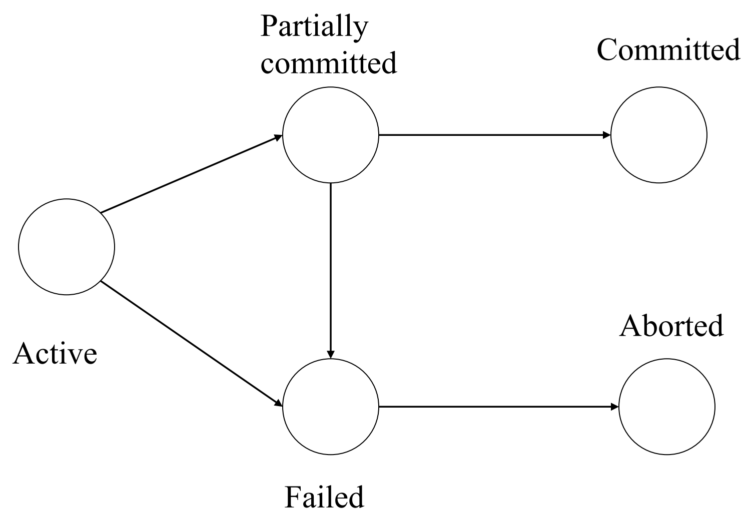
13

## Writeahead Log

- Success: commit record is written to the log
- Abort: roll back any changes

14

# Transaction State






# SERIALIZABILITY AND RECOVERABILITY

## Concurrent Transactions

- Transactions may execute in parallel
  - processor utilization
  - improved average response time
- Schedules
  - Interleaving of update operations of transactions
  - Updates for transaction X must occur in schedule in the same order as in X
- Serial Schedule
  - Ensures isolation of transactions
  - Ensures consistency if each transaction preserves consistency

17

## Serial Schedule

- $T_1$
  read A
  A := A - 50
  write A
  read B
  B := B + 50
  write B

- $T_2$
  read A
  temp := A / 10
  A := A - temp
  write A
  read B
  B := B + temp
  write B

18

# Conflicting Operations

**Read-Write Conflict**

- Suppose initial value of A is \$100
- Schedule 1:
```
Read A
Write 150 to A
```
- Schedule 2:
```
Write 150 to A
Read A
```

**Write-Write Conflict**

- Suppose initial value of A is \$100
- Schedule 1:
```
Write 150 to A
Write 200 to A
```
- Schedule 2:
```
Write 200 to A
Write 150 to A
```

19

# Conflict Serializability

- Operations O1 and O2 conflict if:
  - O1 ≡ read A, O2 ≡ write A
  - O1 ≡ write A, O2 ≡ read A
  - O1 ≡ write A, O2 ≡ write A
- Schedules S and S' are conflict equivalent if S' can be obtained from S by swapping non-conflicting operations
- S is serializable if S is equivalent to a serial schedule

20

## Non-Serial but Serializable Schedule

- $T_1$
  read A
  A := A - 50
  write A

  read B
  B := B + 50
  write B

- $T_2$
  read A
  temp := A / 10
  A := A - temp
  write A

  read B
  B := B + temp
  write B

21

## Non-Serial but Serializable Schedule

- $T_1$
  read A
  A := A - 50
  write A

  read B
  B := B + 50
  write B

- $T_2$
  read A
  temp := A / 10
  A := A - temp

  write A
  read B
  B := B + temp
  write B

22

## Non-Serial but Serializable Schedule

- $T_1$
  read A
  A := A - 50
  write A

  read B
  B := B + 50
  write B

- $T_2$
  read A

  temp := A / 10
  A := A - temp
  write A
  read B
  B := B + temp
  write B



## Non-Serial but Serializable Schedule

- $T_1$
  read A
  A := A - 50
  write A
  read B
  B := B + 50
  write B

- $T_2$
  read A
  temp := A / 10
  A := A - temp
  write A
  read B
  B := B + temp
  write B

# Non-Serializable Schedule

- $T_1$
  read A
  A := A - 50

  write A
  read B
  B := B + 50
  write B

- $T_2$
  read A
  temp := A / 10
  A := A - temp
  write A
  read B

  B := B + temp
  write B



# Non-Serializable Schedule

- $T_1$
  read A
  A := A - 50

  write A
  read B
  B := B + 50

  write B

- $T_2$
  read A
  temp := A / 10
  A := A - temp
  write A

  read B

  B := B + temp
  write B

## Recoverability

- Successful transaction ends with commit operation
- A schedule is **recoverable** if, whenever $T_i$ reads data item written by $T_j$, commit of $T_i$ follows commit of $T_j$



## Recoverability

- The following schedule is not recoverable:

| $T_1$ | $T_2$ |
|---|---|
| write A | |
| | read A |
| | commit |
| read B | |
| commit | |

## Cascading Rollbacks

- The following schedule is recoverable:

| $T_1$ | $T_2$ | $T_3$ |
|---|---|---|
| read A | | |
| write A | | |
| | read A | |
| | write A | |
| | | read A |
| commit | | |
| | commit | |
| | | commit |



## Cascading Rollbacks

- The following schedule is cascadeless:

| $T_1$ | $T_2$ | $T_3$ |
|---|---|---|
| read A | | |
| write A | | |
| commit | | |
| | read A | |
| | write A | |
| | commit | |
| | | read A |
| | | commit |

## CONCURRENCY CONTROL

31

# Implementing Isolation

- Schedules must be
  - serializable
  - recoverable
  - preferably cascadeless
- One-at-a-time execution inefficient
  - Transactions can use the CPU(s) while other transactions are blocked waiting for I/O operations to complete
- Concurrency control

32

# Scheduling

- Two operations conflict if they operate on the same data item and at least one of them is a write
  - read-write conflict
  - write-write conflict
- Two read operations can never conflict
- Concurrency control schemes are classified by how they synchronize read and write operations (locking, ordering via timestamps, …)

33

# Two Scheduling Approaches

- Pessimistic - if something can go wrong, it will
  - Operations explicitly synchronized during execution
- Optimistic - in general, nothing will go wrong
  - Synchronization happens at the end of the transaction

34

## Lost Updates

- $T_1$
  read A
  A := A - 50

  write A

- $T_2$
  read A
  A := A + 100

  write A

35

## Locking for Isolation

- $T_1$
  lock-X A
  read A
  A := A - 50
  write A
  unlock A

- $T_2$
  lock-X A
  read A
  A := A + 100
  write A
  unlock A

36

## Examples of lock coverage

- We could have one lock per object
- … or one lock for the whole database
- … or one lock for a category of objects
  - Tree
  - Table, row, column
- All transactions must use the same rules!
- “Write” locks for updates

37

## Lock-Based Concurrency Control

- Two modes of locking:
  - shared (read-only)
  - exclusive (read-write)
- Lock acquisition
  - transaction blocks if lock not available
  - update shared lock to exclusive
  - downgrade exclusive lock to shared
- Possibility of **deadlock**

38

# Lock-Based Concurrency Control

- Lock Compatibility

| Request by $T_i$ \ Held by $T_j$ | S | X |
|---|---|---|
| S | true | false |
| X | false | false |

- Lock manager implemented using readers-writers algorithm



# Locking for Isolation

- $T_1$

  lock-X A; read A
  A := A - 50
  write A; unlock A

  lock-X B; read B
  B := B + 50
  write B; unlock B

- $T_2$

  lock-X B; read B
  B := B - 100
  write B; unlock B

  lock-X A; read A
  A := A + 100
  write A; unlock A

## Locking for Serializability

- $T_1$
  lock-X A;  lock-X B
  read A;  A := A - 50
  write A
  read B;  B := B + 50
  write B
  unlock A; unlock B

- $T_2$
  lock-X B;  lock-X A
  read B;  B := B - 100
  write B
  read A;  A := A + 100
  write A
  unlock B; unlock A



## Locking for Serializability

- $T_1$
  lock-X A; read A
  A := A - 50
  write A;
  lock-X B; read B
  B := B + 50
  write B;
  unlock A; unlock B

- $T_2$
  lock-X B; read B
  B := B - 100
  write B;
  Lock-X A; read A
  A := A + 100
  write A;
  unlock B; unlock A

# Two-Phase Locking Protocol (2PL)

- Protocol:
  - Phase 1 (Growing)
    - transaction may obtain or upgrade locks
    - transaction may not release or downgrade locks
  - Phase 2 (Shrinking)
    - transaction may release or downgrade locks
    - transaction may not obtain or upgrade locks
- 2PL ensures serializability
  - Prevent cycles in temporal dependencies
- 2PL increases chance of deadlock



# Locking for Recoverability

| • $T_1$ | • $T_2$ |
| --- | --- |
| lock-X A; read A | |
| A := A – 50; write A | |
| lock-X B; read B | |
| B := B + 50; write B | |
| unlock A; unlock B | |
| | lock-X B; read B |
| | B := B – 100; write B |
| | lock-X A; read A |
| | A := A + 50; write A |
| | unlock B; unlock A |
| commit | |
| | commit |

## Locking for Recoverability

- T1
```
lock-X A
read A;  A := A - 50
write A;
lock-X B
read B;  B := B + 50
write B
commit; unlock A; unlock B
```
- T2
```
lock-X B
read B;  B := B - 100
write B
lock-X A
read A;  A := A + 50
write A
commit; unlock B; unlock A
```

45

## Variations on 2PL

- Strict 2PL
  - Transaction must hold all **exclusive** locks until it commits or aborts
  - Ensures cascadeless schedules
- Rigorous 2PL
  - Transaction must hold **all** locks until it commits or aborts
  - Ensures transactions can be serialized in the order in which they commit

46

## Pessimistic Timestamp-Based Concurrency Control

- Transactions get timestamps based on when they start running
- Database records have timestamps for reads, updates
- Abort transaction if:
  - Read future values (written by "future transactions")
  - Rewrite the past (written by "past transactions")

47

## Pessimistic Timestamp-Based Concurrency Control

- Each transaction assigned a timestamp $TS(T_i)$ when it starts
- If $TS(T_i) < TS(T_j)$ then $T_i$ is serialized to run before $TS(T_j)$
- Each data item has 2 timestamps:
  - W-timestamp(A): timestamp of transaction that did last write
  - R-timestamp(A): timestamp of transaction that did last read

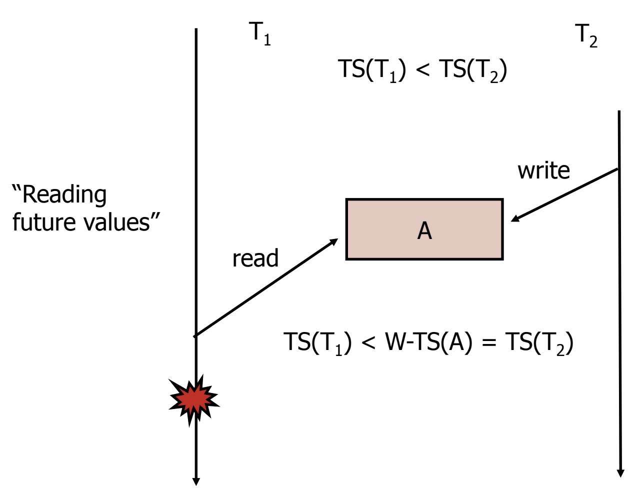
48

# Pessimistic Timestamp-Based Concurrency Control

- Suppose transaction $T_i$ executes "read A":
  - if $TS(T_i) < W\text{-timestamp}(A)$
    then read fails and $T_i$ is aborted/restarted
  - if $TS(T_i) > W\text{-timestamp}(A)$
    then read succeeds;
    $R\text{-timestamp}(A) := \max(R\text{-timestamp}(A), TS(T_i))$

49

# Pessimistic Timestamp-Based Concurrency Control



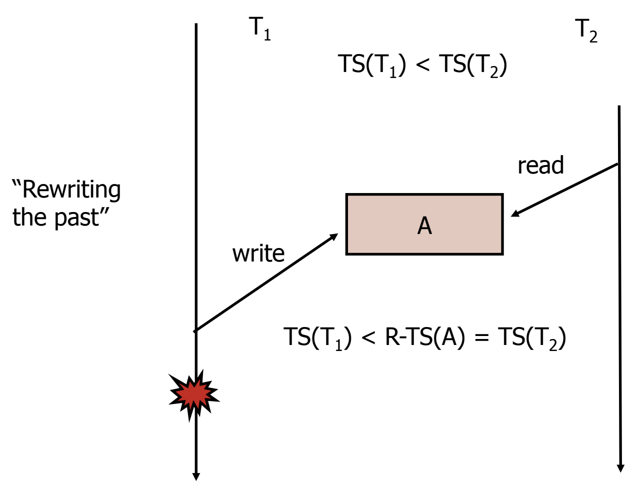

50

## Pessimistic Timestamp-Based Concurrency Control

- Suppose transaction Ti executes "write A":
  - if TS(Ti) < R-timestamp(A)
    then write fails and Ti is aborted/restarted
  - if TS(Ti) < W-timestamp(A)
    then write fails and Ti is aborted/restarted
  - (Thomas' Write rule: ignore this write;
    - may produce schedules that are not CS)
  - otherwise write succeeds;
    - W-timestamp(A) := TS(Ti)

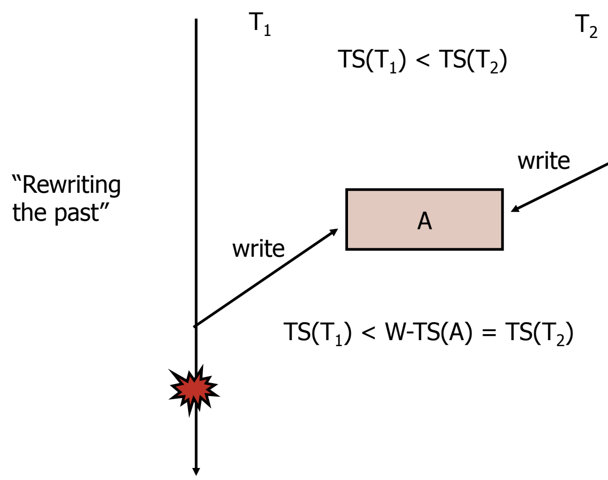
51

## Pessimistic Timestamp-Based Concurrency Control



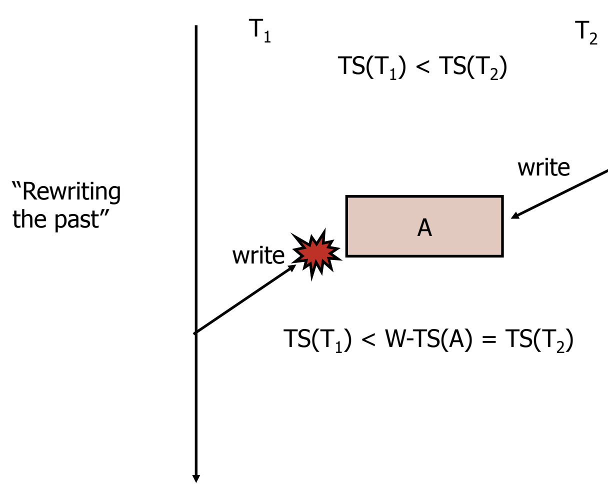

52

# Pessimistic Timestamp-Based Concurrency Control




53

# Pessimistic Timestamp-Based Concurrency Control




54

# Pessimistic Timestamp-Based Concurrency Control

| $T_1$ | $T_2$ | $T_3$ | $T_4$ | $T_5$ |
|---|---|---|---|---|
| | | | | rd X |
| | rd Y | | | |
| rd Y | | | | |
| | | wr Y | | |
| | | wr Z | | |
| | | | | rd Z |
| | rd Z | | | |
| | abort | | | |
| rd X | | | | |
| | | | wr Z | |
| | | | abort | |
| | | | | wr X |
| | | | | wr Z |



# Recoverability and Cascade Freedom

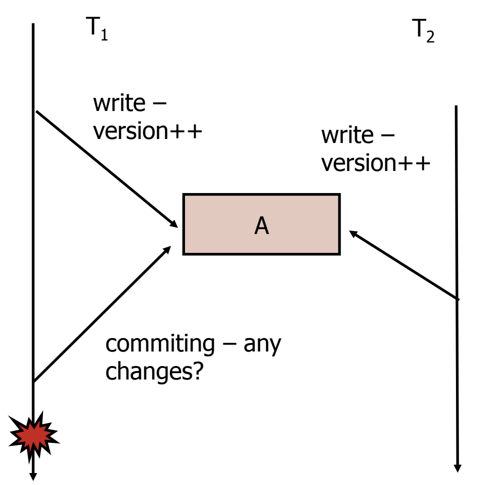
- Suppose $T_i$ reads data item A written by $T_j$
  - R-timestamp(A) = TS(Ti) > TS(Tj)
    = W-timestamp(A)
  - $T_i$ must delay commit until $T_j$ commits
  - Possibility of cascading rollbacks
- Delay all writes until end of transaction, and perform all writes as single atomic
  - Avoid cascading rollbacks

# Optimistic Version Ordering

- Execute operations without regard to conflicts, but keep track of timestamps
- When the time comes to commit, check to see if any data items used by the transaction have been changed since the transaction started
  - abort if something has been changed
  - commit otherwise
- Optimistic concurrency control allows maximum parallelism, but at a price…

57

# Optimistic Version Ordering




58

## Pros and cons of approaches

- Locking scheme works best when conflicts between transactions are common and transactions are short-running
- Timestamped scheme works best when conflicts are rare and transactions are long-running

59

**SERIALIZABILITY IN PRACTICE**

60

## Serializability & Practice

- Isolation levels
  - Read Uncommitted (rare)
  - Read Committed
  - Repeatable Read
  - Serializable (expensive option)

(usual default)

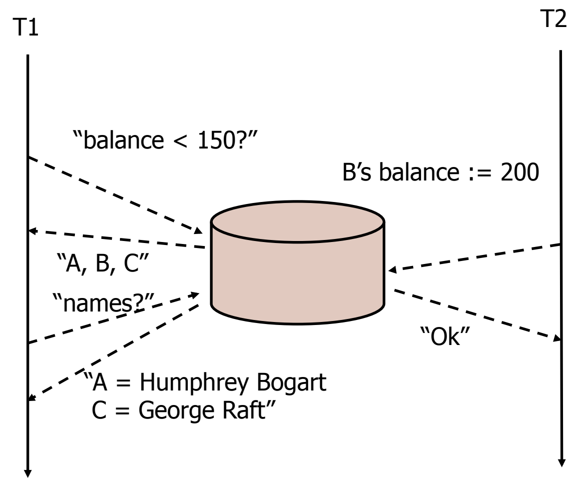
61

## Unrepeatable Read

- Example:
  - T1 reads account id and balance for all accounts with balance less than 150
  - T2 updates an account entity, changing its balance from 100 to 200
  - T1 reads the data again, including detailed information such as account name
  - An account which appears in the initial summary read, is not present in the second detailed read
- Only occurs when multiple reads in the same transaction—rare

62

# Unrepeatable Read



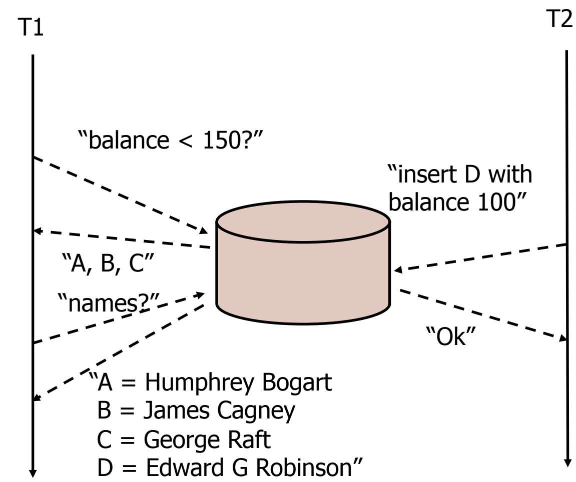

63

# Phantom Read

- Example
  - T1 reads account id and balance for all accounts with balance than 150
  - T2 adds a new account entity with a balance of 100
  - T1 reads the data again including detailed information such as account name
  - A new account now appears in the second detailed read which was not present in the initial summary read
- Similar to unrepeatable read, except that data is inserted rather than updated by the second transaction

64

# Phantom Read




65

# Isolation Levels

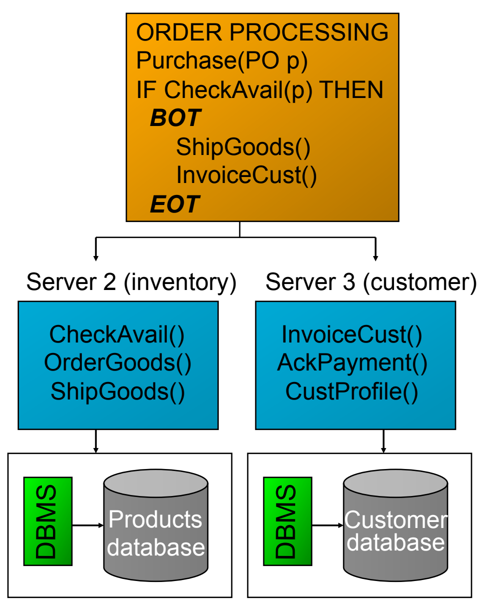
| Isolation Level | Dirty Read | Unrepeatable Read | Phantom Read |
| --- | --- | --- | --- |
| Read Uncommitted | YES | YES | YES |
| Read Committed | NO | YES | YES |
| Repeatable Read | NO | NO | YES |
| Serializable | NO | NO | NO |

66

## DISTRIBUTED TRANSACTIONS

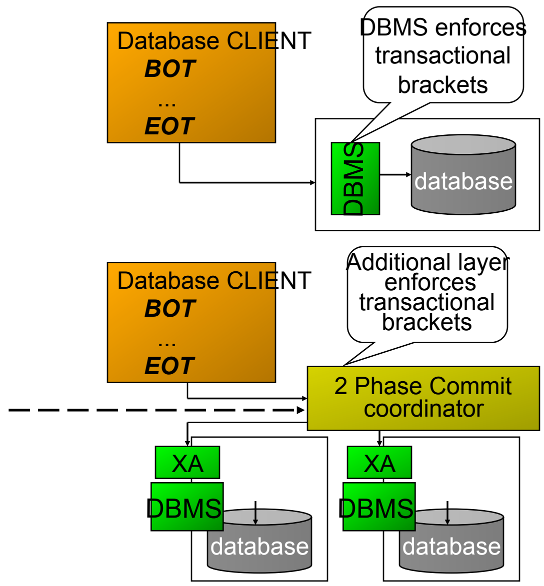
67

## Client, server, and databases




68

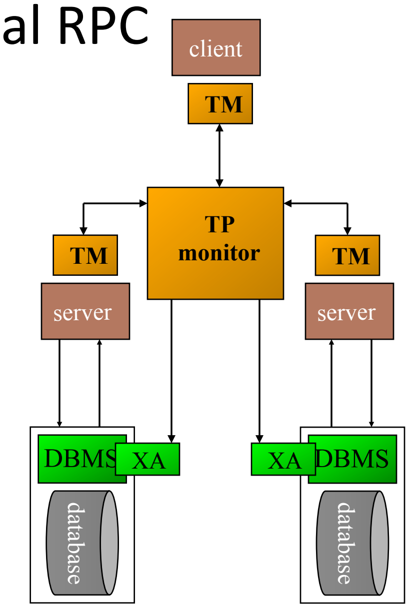
# Multiple Databases




69

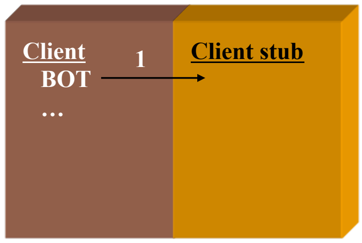
# Transactional RPC

- Transaction Managers
- Transaction Processing Monitor (TPM)
- XA API



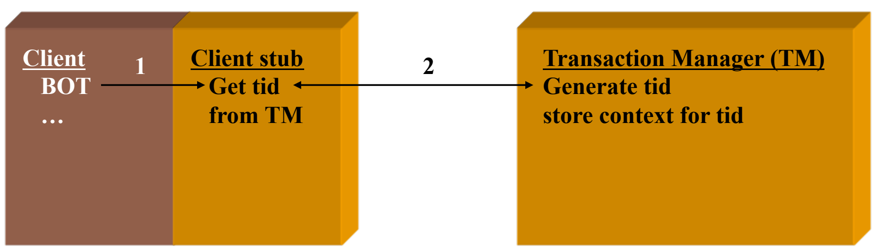

70

# Basic TRPC (making calls)

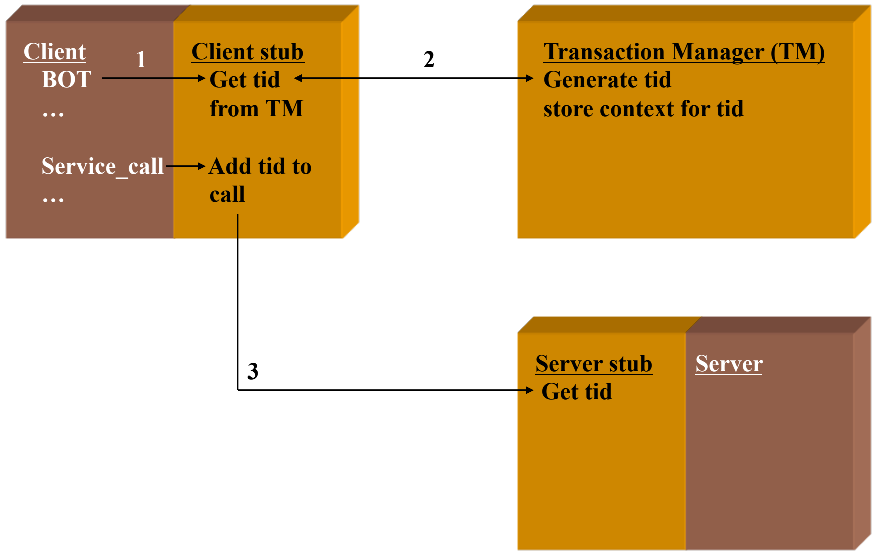





# Basic TRPC (making calls)

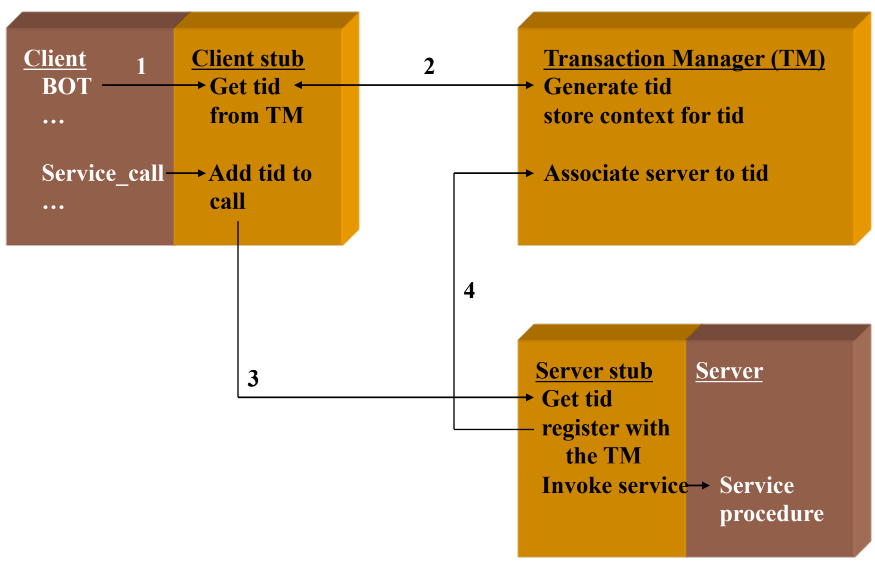


## Basic TRPC (making calls)



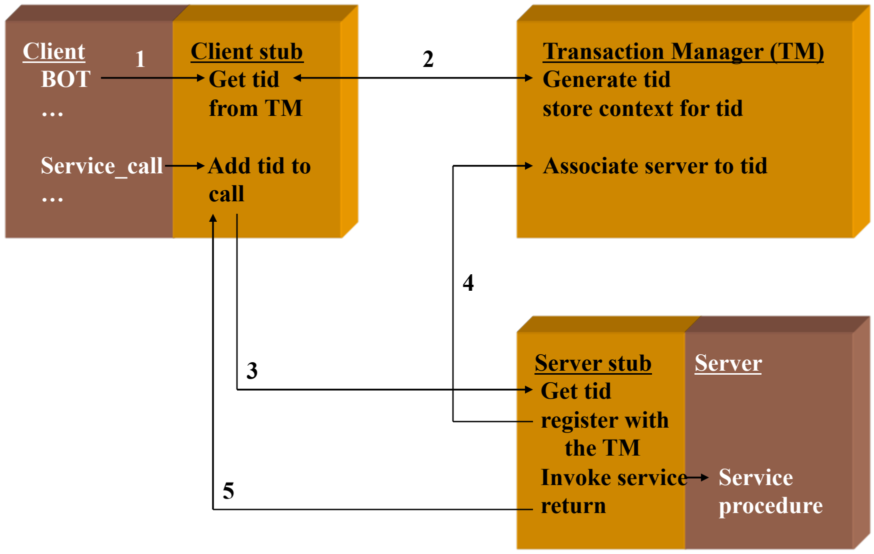

73

## Basic TRPC (making calls)



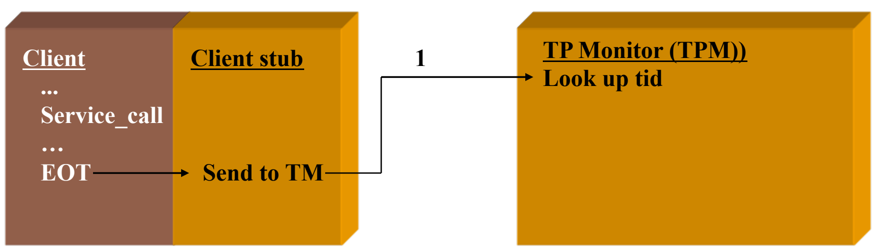

74

## Basic TRPC (making calls)



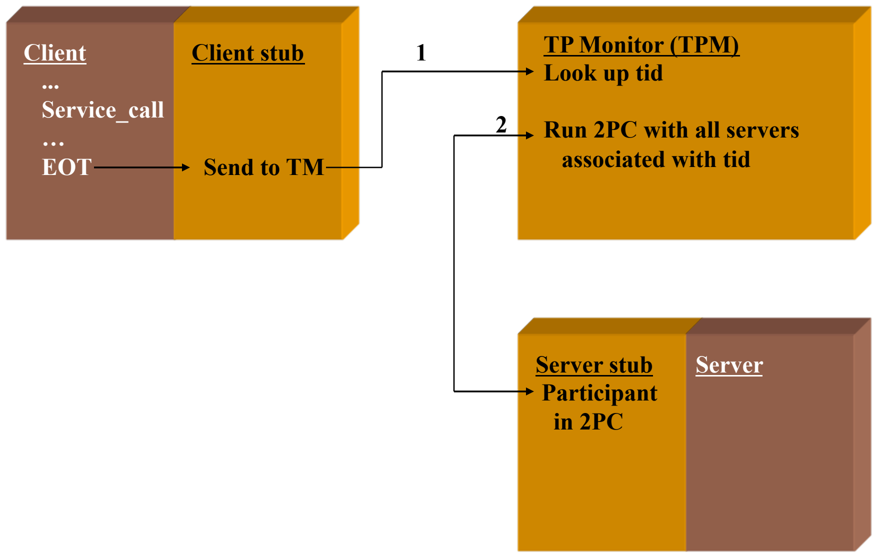

75

## Basic TRPC (committing calls)



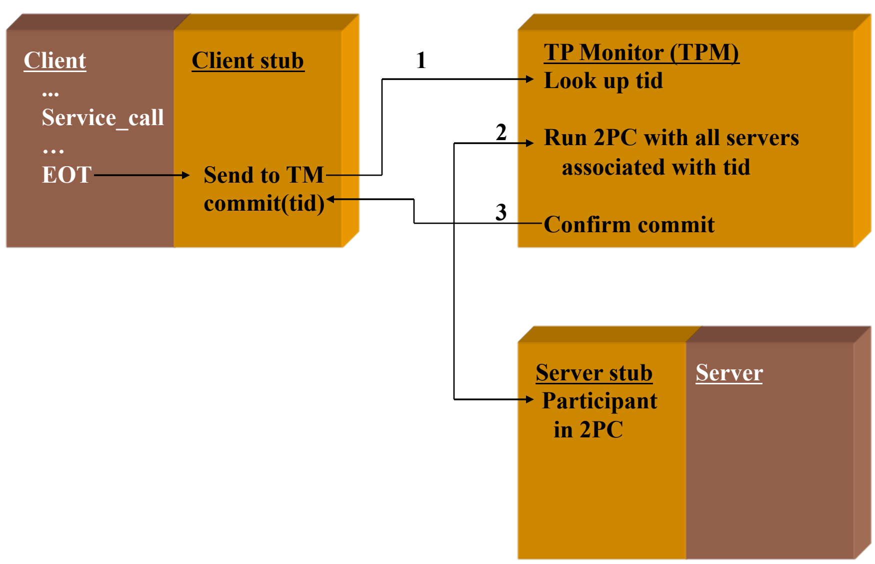

76

# Basic TRPC (committing calls)






# Basic TRPC (committing calls)



79

# EXTRAS

80

## LOCKING IN JAVA EE

81

## Two Scheduling Approaches

- Pessimistic - if something can go wrong, it will
  - Operations synchronized before they're carried out
  - Conflicts are never allowed to occur
- Optimistic - in general, nothing will go wrong
  - Synchronization happens at the end of the transaction
  - If conflict, the transaction is forced to abort

82

## Locking in Java EE

- EntityManager
  ```
  T x = em.find(T.class, LockModeType.LM);
  em.lock (x, LockModeType.LM)
  ```
- Query
  ```
  Query q;
  q.setLockMode(LockModeType.LM)
  ```
- Lock Modes
  - OPTIMISTIC
  - PESSIMISTIC_READ (no need to re-read at commit)
  - PESSIMISTIC_WRITE



## Optimistic Locking

- Designated version stamp
- Incremented on every database update

```
@Entity
public class FlowSheet {
  @Id @GeneratedValue
  private Long id;
  @Version
  private Integer version;
  ...
}
```

## Optimistic Locking

```
FlowSheet f1;           FlowSheet f2;
tx.begin();
em.persist(f1);
tx.commit();
// Version = 1          tx.begin();
                        f2 = em.find(…);
                        … update f2 …
                        tx.commit();
                        // Version = 2
```



## Optimistic Locking

```
FlowSheet f1;           FlowSheet f2;
tx.begin();
f1 = em.find(…);
… update f2 …
tx.commit();            tx.begin();
// Version = N          f2 = em.find(…);
                        … update f2 …
                        tx.commit();
                        // Version = N+1
```

# Optimistic Locking

```
FlowSheet f1;           FlowSheet f2;
tx.begin();
f1 = em.find(…);        tx.begin();
… update f1 …           f2 = em.find(…);
                        … update f2 …
                        tx.commit();
tx.commit();            // Version = N
// OptimisticLock
// Exception
```