Distributed Systems (5DV147)

Transactions

Fall 2013

Transactions

Motivation

Objects a, b, c

Transfer 100 from a to b
Transfer 200 from c to b

```
a.withdraw(100);
b.deposit(100);
c.withdraw(200);
b.deposit(200);
```

Something can go wrong in the middle
….

- Transactions are indivisible units that either …
  - … complete successfully (changes recorded on permanent storage)
  - … or have no effect at all
  - These under crash-failures and when multiple transactions operate on same objects (require concurrency control)

Operations

openTransaction() -> trans;

starts a new transaction and delivers a unique TID *trans*. This identifier will be used in the other operations in the transaction.

closeTransaction(trans) -> (commit, abort);

ends a transaction: a *commit* return value indicates that the transaction has committed; an *abort* return value indicates that it has aborted.

abortTransaction(trans);

aborts the transaction.

Successful	Aborted by client	Aborted by server	
openTransaction	openTransaction		openTransaction
operation	operation		operation
operation	operation		operation
:	:	server aborts transaction →	:
operation	operation		operation ERROR reported to client
closeTransaction	abortTransaction		

ACID Properties

Atomicity: “all or nothing”

Consistency: transactions take system from one consistent state to another consistent state

Isolation: transactions do not interfere with each other

Durability: committed results of transactions are permanent

Nested transactions

- Tree-structured
- Sub-transactions at one level may execute concurrently
- Sub-transactions may provisionally commit or abort independently
    - parent may decide whether to abort or not
- Provisional commit is not a proper commit!

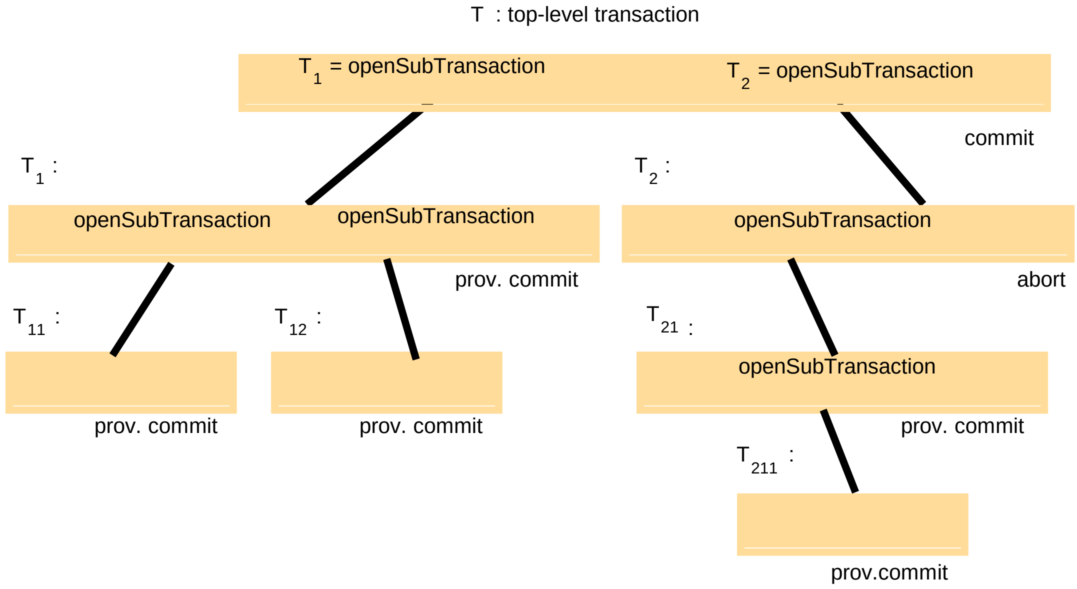


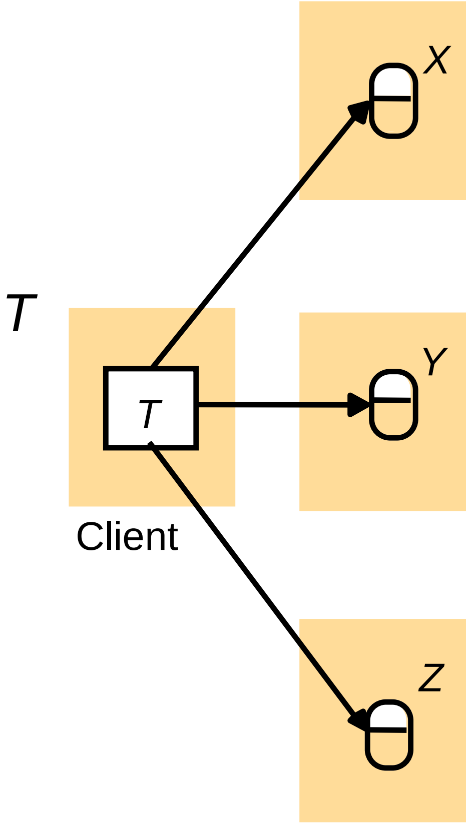
Figure adapted from Instructor's Guide for Coulouris, Dollimore, Kindberg and Blair, Distributed Systems: Concepts and Design Edn. 5 © Pearson Education 2012 – based on **Figure 16.13**

Rules for committing nested transactions

1. A transaction may commit/abort once all children transactions have completed
2. Sub-transactions make independent choices whether to provisionally commit or abort – abort is final
3. When a parent aborts, all sub-transactions abort
4. When a sub-transaction aborts, the parent may decide what to do
5. If the top-level transaction commits, all sub-transactions that have provisionally committed may commit as well

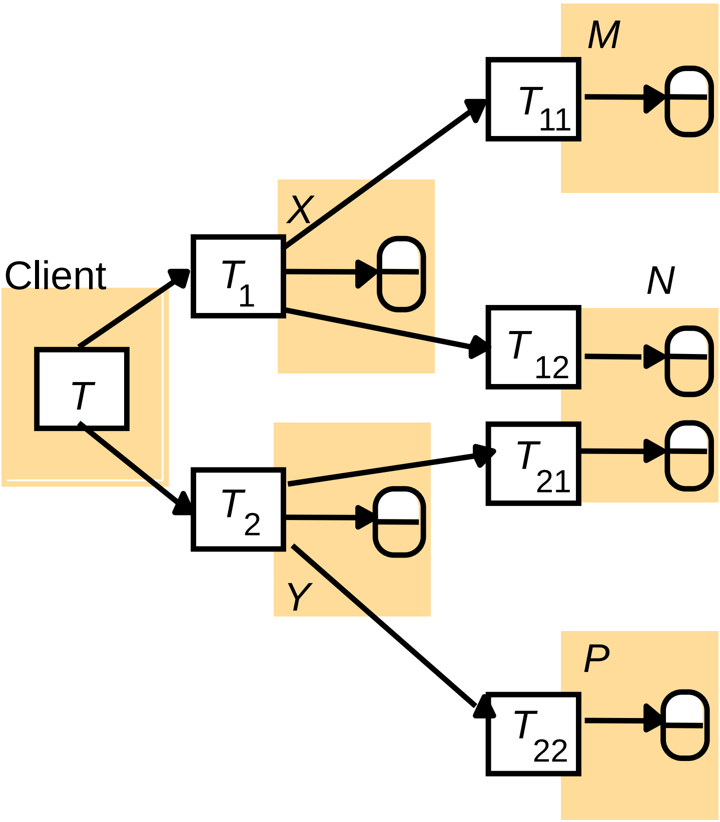
Flat and nested distributed transactions

- Distributed transaction:
  - Transactions dealing with objects managed by different processes
- Allows for even better performance
  - At the price of increased complexity
- Transaction coordinators and object servers
  - Participants in the transaction



Flat transactions

- Requests are made to more than one server
- Access to servers is sequential
- A transaction can only wait for one object that is locked at a time



Nested transactions

- Sub-transactions can be opened to any depth
- Sub-transactions at the same level can run concurrently
- If sub-transactions run on different servers, they can run concurrently

Problems with concurrent transactions

- Transactions are carried out concurrently for higher performance
  - Otherwise, painfully slow
- Two common problems that appear if performance is not handled correctly
  - Lost update
  - Inconsistent retrieval
- Solution
  - Serial equivalence (conflicting operations)

Lost update

T_1*: *A=read(x), write(x, A*10)

T_2*: *B=read(x), write(x, B*10)

If not properly isolated, we could get the following interleaving:

(T_1) A=read(x)
(T_2) B=read(x)

original value of x

*(T_1) write(x, A*10)*
*(T_2) write(x, B*10)*

Executing ***T_1*** and ***T_2*** should have increased x by ten times twice, but we lost one of the updates

(T_1) A=read(x)
*(T_1) write(x, A*10)*
(T_2) B=read(x)
*(T_2) write(x, B*10)*

Inconsistent retrieval

T_1: withdraw(x, 10), deposit(y, 10)

T_2: sum all accounts

<u>Improper interleaving:</u>

(T_1) withdraw(x, 10)
(T_2) sum+=read(x)
(T_2) sum+=read(y)
...
(T_1) deposit(y, 10)

Read concurrent with update transaction

The sum is incorrect, because it doesn't account for the 10 that are 'in transit' – neither in x nor in y – the retrieval is inconsistent

(T_1) withdraw(x, 10)
(T_1) deposit(y, 10)
(T_2) sum+=read(x)
(T_2) sum+=read(y)
...

How to work around this problems

- Serial equivalence
  - Interleaved operations produce same effect as if transactions have been performed one at a time
    - Not *actually* one transaction at a time
- Conflicting operations
  - Two operations are in conflict if the result depends on the order of execution
    - Read – Read → No conflict
    - Read – Write (or Write – Read) → Conflict!
    - Write – Write →Conflict!

Problems when aborting transactions: Dirty reads

T_1 reads a value that T_2 wrote, then commits and later, T_2 aborts

- The value is "dirty", since the update never happened
  - T_1 has committed, so it cannot be undone
- Fix –let T_1 wait until T_2 commits/aborts!

  But if T_2 aborts, we must abort T_1

  ...and so on: others may depend on T_1

  …cascading aborts

Better rule:

Transactions are only allowed to read objects that *committed* transactions have written

Premature writes

- Use "Before images" to recover from bad writes

`Let` *x = 50* `initially`

T_1: write(x, 10); T_2: write(x, 20)

Let T_1 execute before T_2

What happens if T_2 commits but T_1 aborts?

What happens if T_1 aborts and then T_2 aborts?

Order of commit/abort matters!

- If before images are used, delay writes to objects until other, earlier, transactions that write to the same object have committed/aborted
- Systems that avoid both dirty reads and premature writes are "strict"
  - Delay read(s) and write(s)
  - Highly desirable!
  - Tentative versions (local to each transaction)

Concurrency control protocols

Concurrency control

- Serialize access to objects
- Three protocols
  - Locks
  - Optimistic concurrency control
  - Timestamp ordering

Locks

- Need an object? Get a lock for it!
  - Read or write locks, or both (exclusive)
- Two-phase locking
  - Accumulate locks gradually, then release locks gradually
- Strict two-phase locking
  - Accumulate locks gradually, keep them all until completion

Enables “strict” systems

- Granularity and tradeoffs

Transaction *T* :	**Transaction *U* :**
balance = b.getBalance()	*balance = b.getBalance()*
*b.setBalance(bal*1.1)*	*b.setBalance(bal*1.1)*
a.withdraw(bal/10)	*c.withdraw(bal/10)*

Operations	Locks	Operations	Locks
openTransaction			
bal = b.getBalance()	lock *B*		
*b.setBalance(bal*1.1)*		*openTransaction*	
a.withdraw(bal/10)	lock *A*	*bal = b.getBalance()*	waits for *T*'s lock on *B*
closeTransaction	unlock *A* , *B*	● ● ●	
			lock *B*
		*b.setBalance(bal*1.1)*	
		c.withdraw(bal/10)	lock *C*
		closeTransaction	unlock *B* , *C*

Figure adapted from Instructor's Guide for Coulouris, Dollimore, Kindberg and Blair, Distributed Systems: Concepts and Design Edn. 5 © Pearson Education 2012 – based on **Figure 16.14**

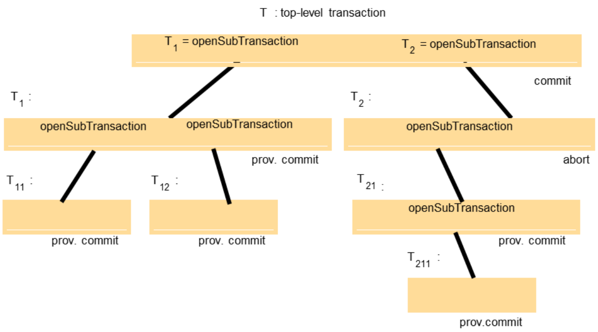
Sharing locks

- Read locks can be shared
- Promote read lock to write lock if no other transactions require a lock
- Requesting a write lock when there are already read locks, or a read lock when there is already a write lock?
  - Wait until lock is available

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

Lock compatibility

Locks and nested transactions



Isolation

- From other sets of nested transactions
- From other transactions in own set

Rules:

- Parents do not run concurrently with children
- Children can temporarily acquire locks from ancestors
- Parent inherits locks when child transactions commit
  - Locks are discarded if child aborts
- Sub-transactions at each level are treated as flat transactions

There are also rules for acquiring and releasing locks

Big problem: Deadlocks

- Typical deadlock:

  Transaction A waits for B,
  transaction B waits for A
- Deadlocks may arise in long chains
- Conceptually, construct a *wait-for graph*
  - Directed edge between nodes if one waits for the other
  - Cycles indicate deadlocks

    Abort transaction(s) as needed

Handling deadlock

- Deadlock prevention
  - Acquire all locks from the beginning
    Bad performance, not always possible
- Deadlock detection
  - As soon as a lock is requested, check if a deadlock will occur
    Bad performance: avoid checking always
  - Must include algorithm for determining which transaction to abort
- Lock timeouts
  - Locks invulnerable for a certain time, then they are vulnerable
  - Leads to unnecessary aborts
    - Long-running transactions
    - Overloaded system
  - How to decide useful timeout value?

Locking drawbacks

- Overhead (even on read-only transactions)
  - Necessary only in the worst case
- Deadlock
  - Prevention reduces concurrency severely
  - Timeouts or detection
- Reduced concurrency in general
  - Locks need to be maintained until transactions end

Enter optimistic concurrency control

Optimistic Concurrency Control

Assumes that conflicts are rare

- Probability of multiple accesses to same object is low
- Only need to worry about real conflicts

Transaction phases:

Working

- Transaction works with tentative data (*read* and *write* sets)

Validation (Upon completion)

- Check if transaction may commit or abort
- Conflict resolution

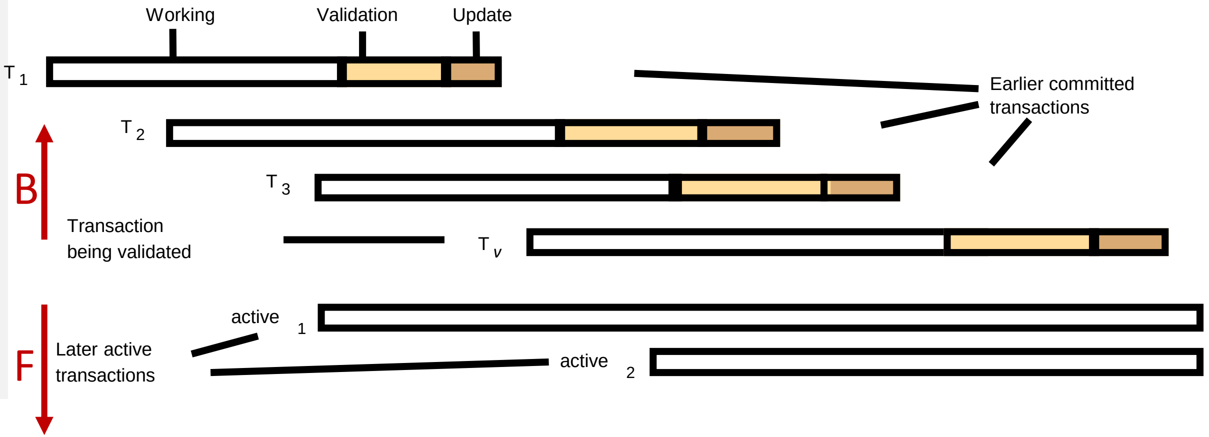
Update

- Write tentative data from committed transactions to permanent storage

Validation

- Use conflict rules from earlier!
  - On overlapping transactions
- Validate one transaction at a time against others
- Transactions are numbered (not to be confused with IDs) as they enter the validation phase
- Only a single transaction at a time in update phase
- Backward or Forward validation

T_v	T_i	Rule
write	read	T_i must not read objects written by T_v
read	write	T_v must not read objects written by T_i
write	write	T_i must not read objects written by T_v and T_i must not read objects written by T_v



Backward validation

- Check *read* set against *write* set of transactions that:
  - were active at the same time as the transaction currently being validated; and
  - have already committed
- Transactions with only *write* set need not be checked
- If overlap is found, then current transaction must be aborted!

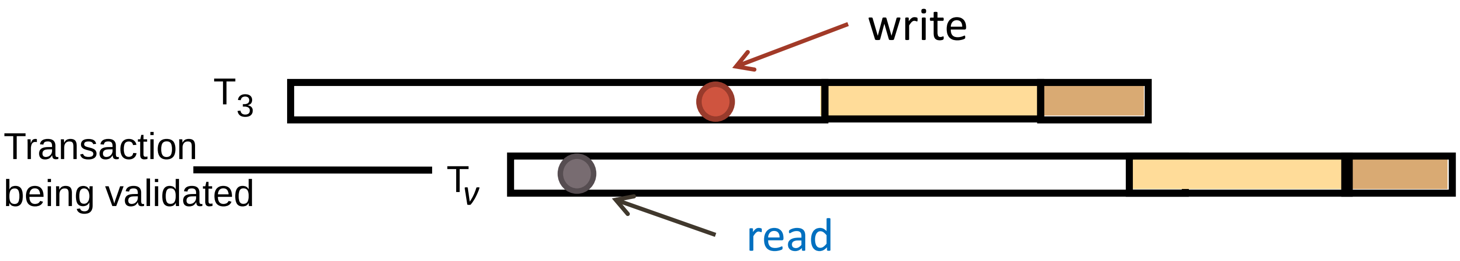
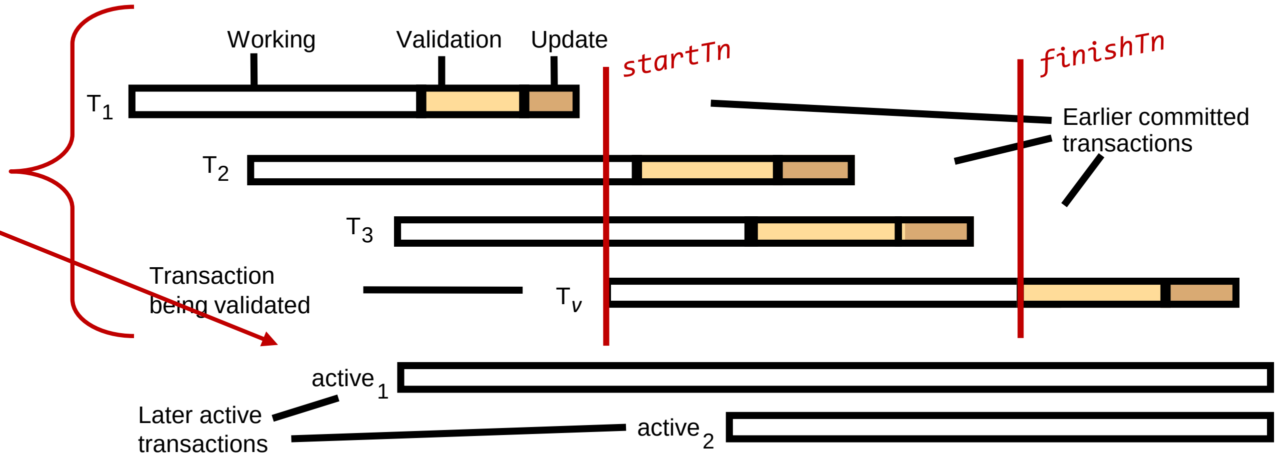


Figure adapted from Instructor's Guide for Coulouris, Dollimore, Kindberg and Blair, Distributed Systems: Concepts and Design Edn. 5 © Pearson Education 2012 – based on **Figure 16.28**

Backward validation - example

write sets must be saved until active finishes




```
Backward validation of transaction Tv
        boolean valid = true;
        for (int Ti  = startTn+1; Ti <= finishTn; Ti++){
                if (read set of Tv intersects write set of Ti) valid = false;
        }
```

Forward validation

- Check *write* set against *read* set of transactions that are currently active
  - Note that read sets of active transactions may change during validation
- Transactions with only *write* set need not be checked
- If overlap is found, we can choose which transaction(s) to abort
  - Wait until conflicting transactions have finished
  - Abort conflicting active transactions
  - Abort transactions being validated

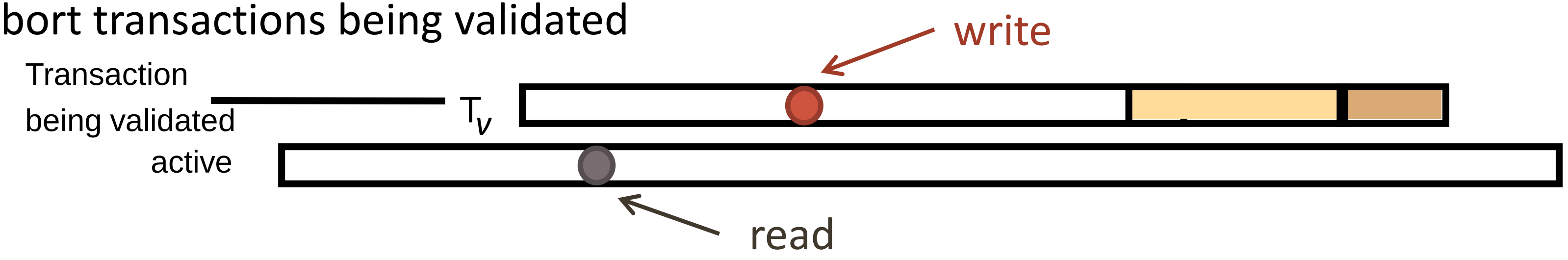
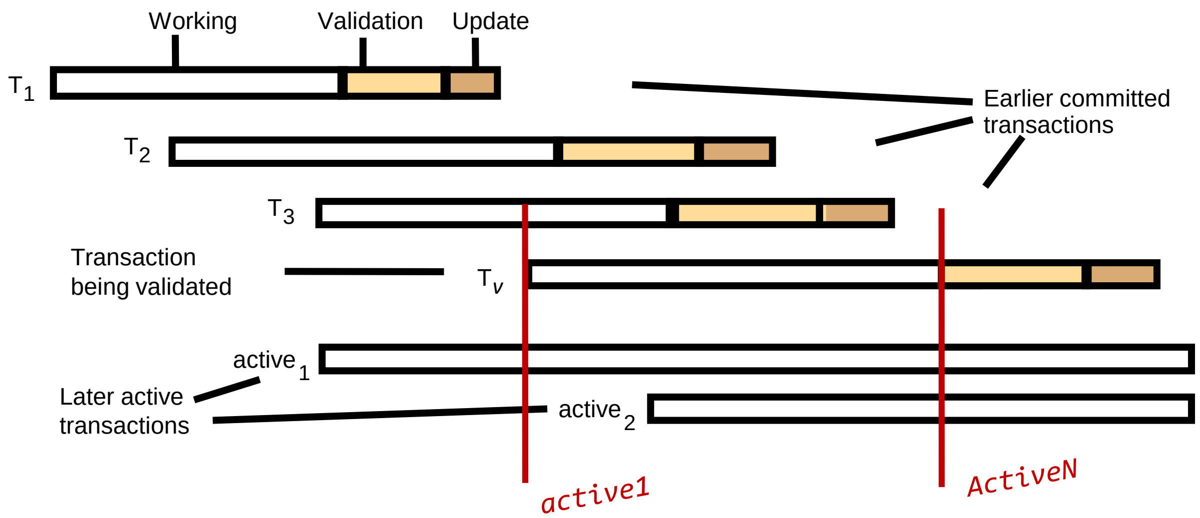


Figure adapted from Instructor's Guide for Coulouris, Dollimore, Kindberg and Blair, Distributed Systems: Concepts and Design Edn. 5 © Pearson Education 2012 – based on **Figure 18.2**

Forward validation - example




```
Forward validation of transaction Tv
        boolean valid = true;
        for (int Tid = active1; Tid <= activeN; Tid++){
                if (write set of Tv intersects read set of Tid) valid = false;
        }
```

Comparison of optimistic concurrency control

- Size of *read/write* sets
  - *Read* sets are usually bigger
  - Forward compares against “growing” *read* sets
- Choice of transaction to abort
  - Backward a single choice, Forward three choices
  - Linked to starvation
- Overhead
  - Backward requires storing old *write* sets
  - Forward may need to re-run each time the *read* set for any active transaction changes and must allow for checking new valid transactions

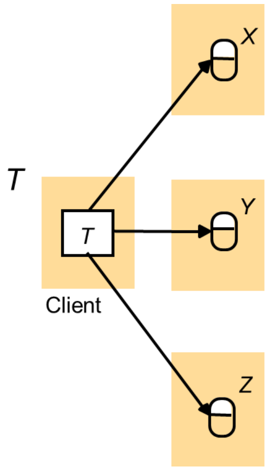
Comparison of concurrency control schemes

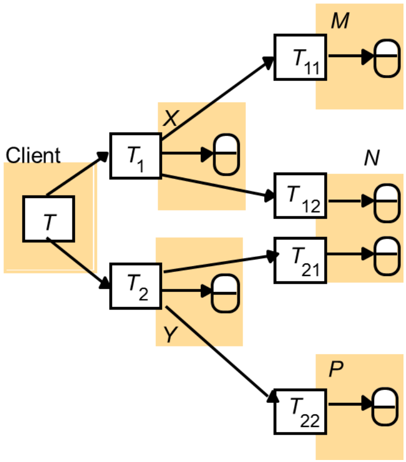
- Pessimistic CC (two-phase locking)
  - Transactions need to wait for locks ...and yet, can still be aborted
  - Large overhead (avoided in new systems)
- For systems with many CC-related issues
  - Pessimistic will give a more stable quality of service
  - Optimistic will abort a large number of transactions and requires substantial work

Two-phase commit

Atomic commit

- Distributed transaction
  - Transactions dealing with objects managed by different servers
- All servers commit or all abort
  - … at the same time
  - in spite of (crash) failures and asynchronous systems





Problem of ensuring atomicity relies on ensuring that all participants vote and reach the same decision

Two-phase commit protocol

Phase 1: Coordinator collects votes

“*Abort*”, any participant can abort its part of the transaction

“*Prepared to commit*”, save updates to permanent storage to survive crashes (May not change vote to “*abort*”)

Phase 2: Participants carry out the joint decision

Protocol can fail due to servers crashing or network partition

- Log actions into permanent storage

Algorithm

Phase 1 (voting)

1. Coordinator sends "*canCommit?*" to each participant
2. Participants answer "*yes*" or "*no*"
    - "Yes": update saved to permanent storage
    - "No": abort immediately

Phase 2 (completion)

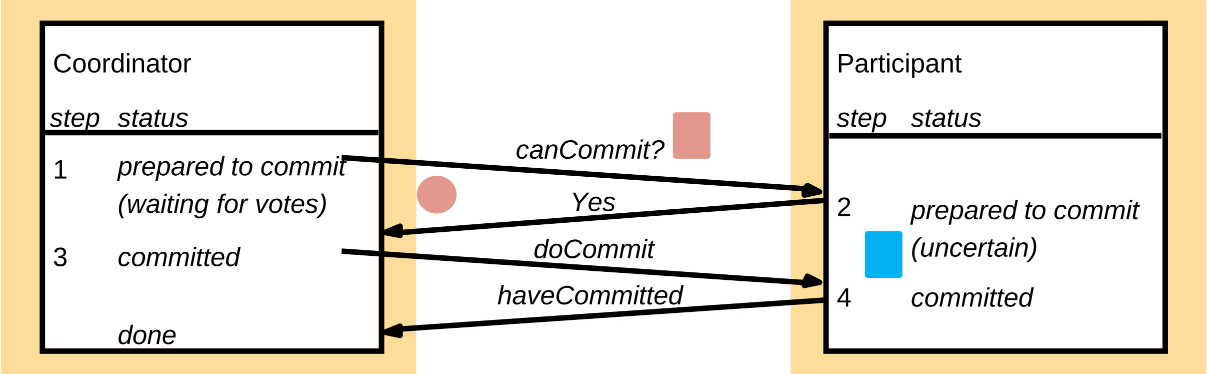
3. Coordinator collects votes (including own)
    - No failures and all "*yes*"? Send "*doCommit*" to each participant, otherwise, send "*doAbort*"
4. Confirm commit via "*haveCommitted*"

Note: Participants are in "uncertain" state until they receive "doCommit" or "doAbort", and may act accordingly (send "*getDecision*" message to coordinator)

Timeout actions

If coordinator fails:

- Participants are *"uncertain"*
  - If some have received an answer (or they can figure it out themselves), they can coordinate themselves
- Participants can request status (send *"getDecision"* message to coordinator)
- If participant has not received "canCommit?" and waits too long, it may abort



If participant fails:

- No reply to *"canCommit?"* in time?

  Coordinator can abort

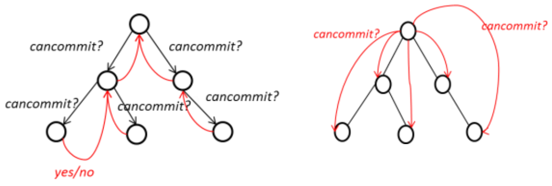
  Crash after *"canCommit?"*

  Use permanent storage to get up to speed

Two-phase commit protocol for nested transactions

- Sub-transactions "*provisional commit*"
  - Nothing written to permanent storage

    Ancestor could still abort!
  - If they crash, the replacement **cannot** commit
- Status information is passed upward in tree
  - List of provisionally committed sub-transactions eventually reach top level
- Hierarchical or flat voting phase



Summary (1)

- Transactions – specify sequence of operations that are atomic in presence of concurrent transactions and server crashes
- ACID properties
- Problems with transactions – lost updates, inconsistent retrievals
- Serial equivalence
  - Conflicting operations – read-read, read-write, write-read

Summary (2)

- Aborted transactions – dirty reads, premature writes
- Nested transactions – allow additional concurrency, can work in parallel, commit or abort independently
- Concurrency control protocols – locks and optimistic concurrency control
- Locks – (strict) two-phase locking, shared locks, nested transactions
- Deadlocks – how to handle them
- Optimistic concurrency control – backward and forward validation
- Two-phase commit

Next Lecture

Peer-to-peer