Distributed Systems (5DV147)

Transactions

Fall 2014

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)

Transactions

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

- recoverable objects

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		

Types of transactions

Flat transactions

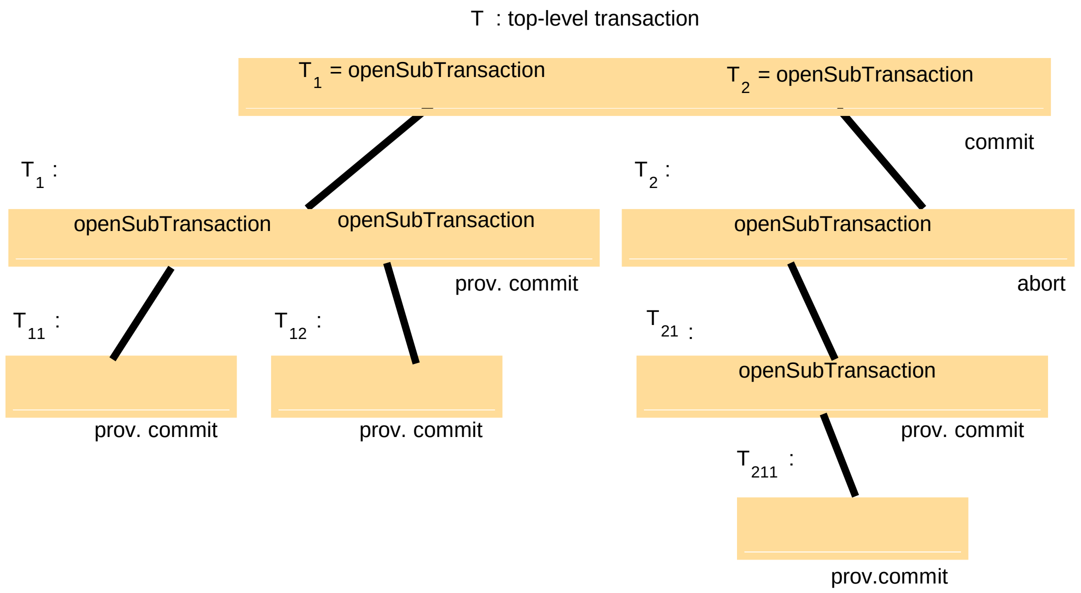
- We have seen those already:

  open-transaction() … commit()/abort()

- The entire transaction must commit or abort

Nested transactions

- ❑ Tree-structured
- ❑ Sub-transactions at one level may execute concurrently
  - ➢ Shared objects' accesses are serialized
- ❑ Sub-transactions may provisionally commit or abort independently
  - ➢ parent may decide whether to abort or not
  - ➢ Provisional commit is not a proper commit!

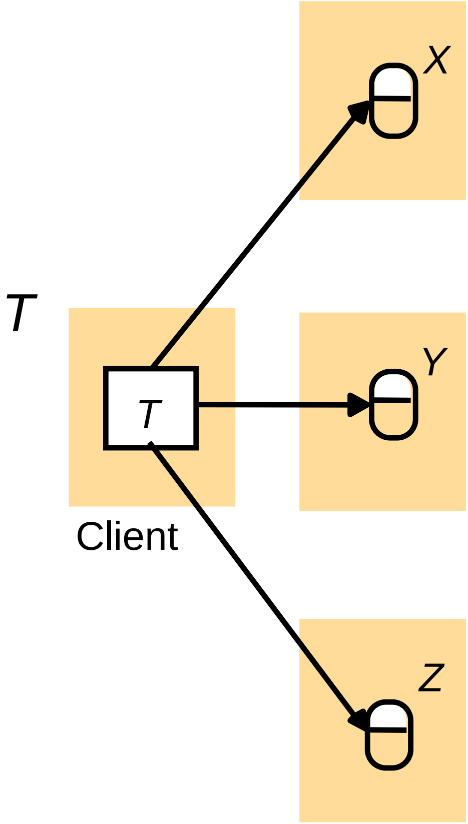


Rules for committing nested transactions

1. All children transactions need to complete before deciding on commit/abort the parent transaction
2. Sub-transactions independently *provisionally commit* or *abort* – abort is final
3. When parent aborts, all sub-transactions abort
4. When a sub-transaction aborts, parent decide what to do
5. If the top-level transaction commits, all sub-transactions that have provisionally committed may commit as well if none of their ancestors has aborted

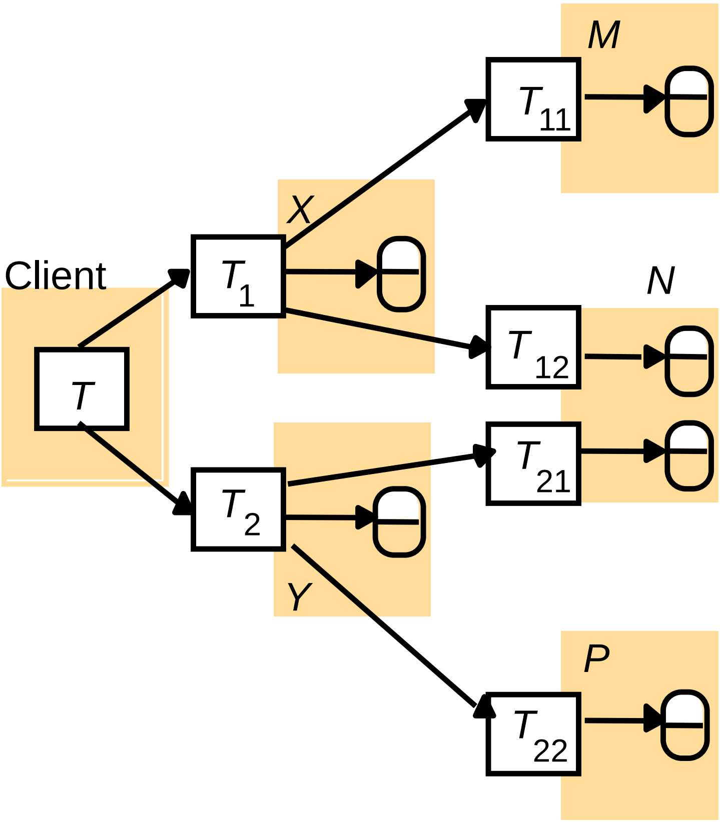
Flat and nested distributed transactions

- Distributed transaction:
  - Transactions accessing objects managed by more than one server (processes)
  - All servers need to commit or abort a transaction
- Allows for even better performance
  - At the price of increased complexity
- One coordinator and multiple participants



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 in parallel

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 17.1**

Example: Distributed flat transaction

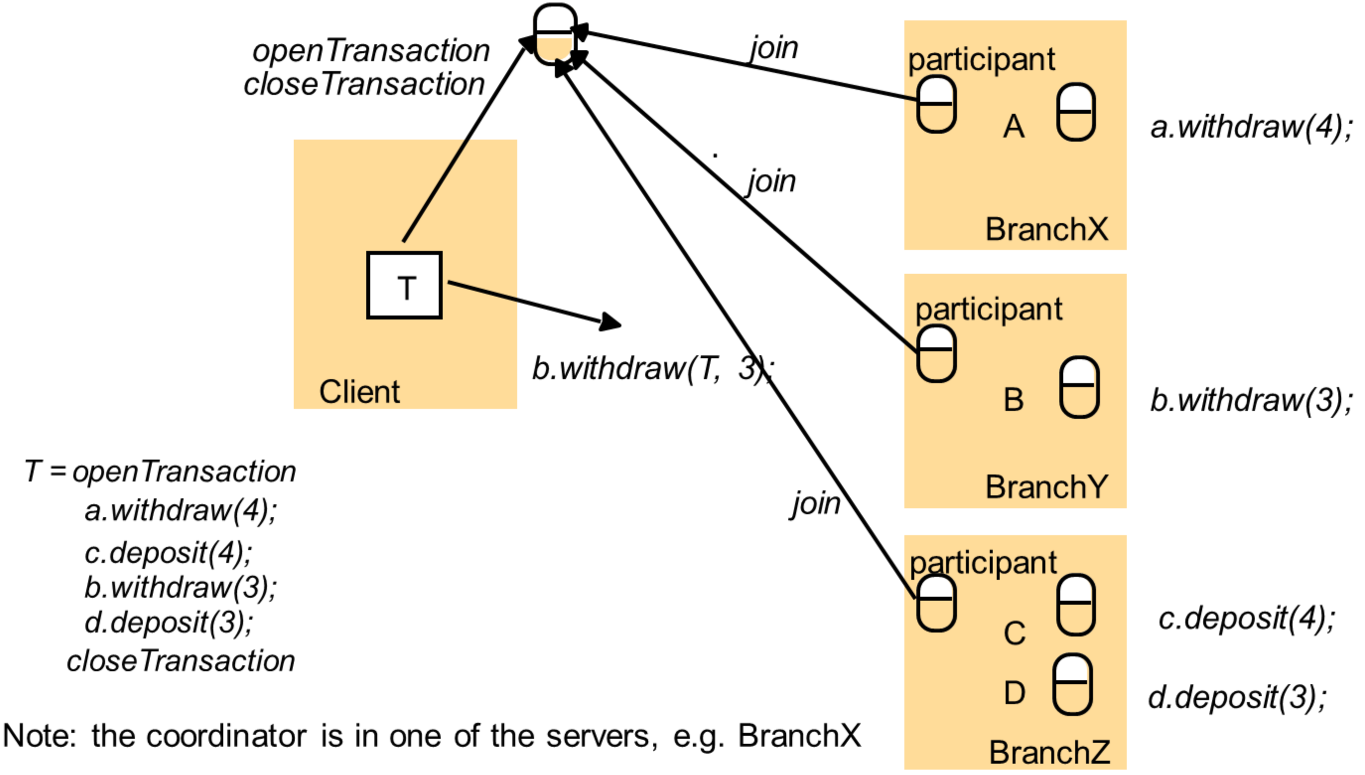


Figure adapted from Instructor's Guide for Coulouris, Dollimore, Kindberg and Blair, Distributed Systems: Concepts and Design Edn. 5 © Pearson Education 2012 – based on **Figure 17.3**

Concurrent transactions

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: manage conflicting operations and create schedules that ensure the consistency requirement

Lost update

$\boldsymbol{T_1}$: *A=read(x), write(x, A*10)*

$\boldsymbol{T_2}$: *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 $\boldsymbol{T_1}$ and $\boldsymbol{T_2}$ 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)*

Two transactions read the old value of the variable an then use that value to calculate a new value

Inconsistent retrieval

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

T_2: sum all accounts

Improper interleaving:

(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)
...

A *retrieval* transaction runs concurrent with an *update* transaction

How to work around these problems?

- Serial Equivalence
  - Interleaved operations produce same effect as if transactions have been performed one at a time

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

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

  - Does not mean to *actually* perform one transaction at a time, as this would lead to horrible performance

A better example

a.balance = 100
b.balance = 200
c.balance = 300

Transaction *T* :		**Transaction *U*:**	
balance = b.getBalance(); *b.setBalance(balance*1.1);* *a.withdraw(balance/10)*		*balance = b.getBalance();* *b.setBalance(balance*1.1);* *c.withdraw(balance/10)*	
balance = b.getBalance();	$200		
		balance = b.getBalance();	$200
		*b.setBalance(balance*1.1);*	$220
*b.setBalance(balance*1.1);*	$220		
a.withdraw(balance/10)	$80		
		c.withdraw(balance/10)	$280

Better interleaving

Transaction *T*:		**Transaction *U*:**	
balance = b.getBalance() *b.setBalance(balance*1.1)* *a.withdraw(balance/10)*		*balance = b.getBalance()* *b.setBalance(balance*1.1)* *c.withdraw(balance/10)*	
balance = b.getBalance()	$200		
*b.setBalance(balance*1.1)*	$220		
		balance = b.getBalance()	$220
		*b.setBalance(balance*1.1)*	$242
a.withdraw(balance/10)	$80		
		c.withdraw(balance/10)	$278

Conflicting operations

- Two operations are in conflict if the final result depends on the order of execution
  - Value set by a *write*
  - Result of a *read*

Read – Read → No conflict

Read – Write (or Write – Read) → Conflict!

Write – Write →Conflict!

Back to the example

Transaction *T* :	Transaction *U*:
balance = b.getBalance();	*balance = b.getBalance();*
*b.setBalance(balance*1.1);*	*b.setBalance(balance*1.1);*

T: (1) B.Read, (2) B.Write

U: (3) B.Read, (4) B.Write

Conflicts: (1,4), (2,3)

Interleave: 1, 3, 4, 2

Wrong

Transaction :		Transaction *U*:	
balance = b.getBalance();	$200		
		balance = b.getBalance();	$200
		*b.setBalance(balance*1.1);*	$220
*b.setBalance(balance*1.1);*	$220		
a.withdraw(balance/10)	$80		
		c.withdraw(balance/10)	$280

The problem is that the pairs of conflicting operations should be performed in the same order! e.g., [(1,4),(2,3)] or [(4,1), (3,2)]

Serializability

- 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
- Produce consistent schedules

Concurrency control protocols

- Ensure serially equivalent interleavings
- Maximize concurrency
  - Locks (wait for access)
  - Optimistic concurrency control (check for conflicts at the end)
  - Timestamp ordering (check to delay or reject)

Some more things to consider…

Problems when aborting transactions

- Results from transactions that commit must be recorded
- Results from transactions that abort should be forgotten
- Transactions can be aborted for whatever reason
  - Nature of transaction
  - Conflicts with another transaction
  - Crash of a process or computer
- Two common problems associated with aborted transactions
  - Dirty reads
  - Premature writes

Dirty reads

- T1 reads a value that T2 wrote, then commits and later, T2 aborts
  - The value is "dirty", since the update to it should not have happened
  - T1 has committed, so it cannot be undone

Transaction *T* :		**Transaction** *U* :	
a.getBalance() *a.setBalance(balance + 10)*		*a.getBalance()* *a.setBalance(balance + 20)*	
balance = a.getBalance()	\$100		
a.setBalance(balance + 10)	\$110		
		balance = a.getBalance()	\$110
		a.setBalance(balance + 20)	\$130
		commit transaction	
abort transaction			

Handling dirty reads

- New rule: let T1 wait until T2 commits/aborts!
  - But if T2 aborts, we must abort T1

    ...and so on: others may depend on T1

    ...cascading aborts
- Better rule:
  - Transactions are only allowed to read objects that *committed* transactions have written
  - Delay commits until after all transactions whose uncommitted state has been seen (delay reads for writes)

Premature writes

- Sometimes "Before images" are used when recovering from an aborted transaction

Let x = 50 initially

***T1:** write(x, 10); **T2:** write(x, 20)*

Let T1 execute before T2

What happens if either one aborts?

Order of commit/abort matters!

T2 aborts, T1 commits (x=10)

T2 commits, T1 aborts (x=50)

T2 aborts, T1 aborts (x=10)

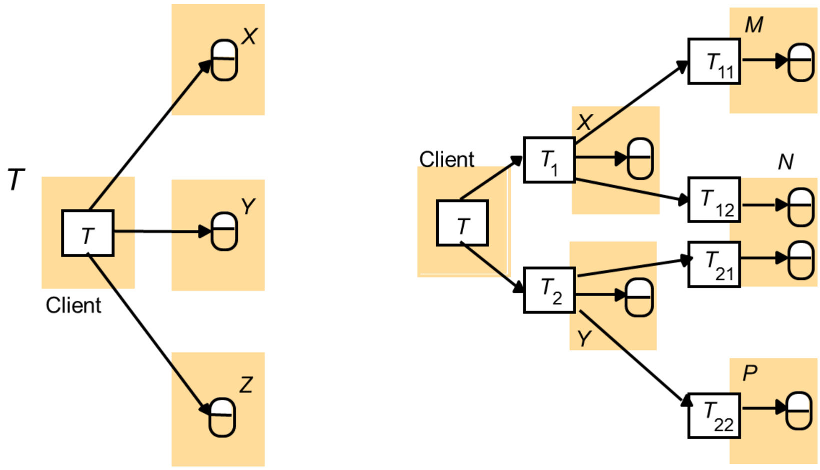
Handling premature writes

- 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 and writes
  - Highly desirable, enforce isolation
  - Tentative versions (local to each transaction)

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)

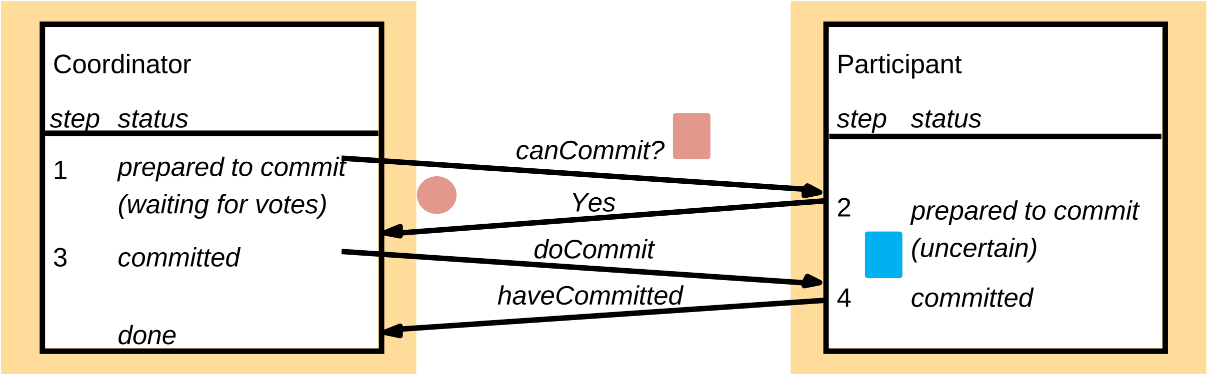
3. Coordinator collects votes (including own)
    - No failures and all “*yes*”? Send “*doCommit*” to each participant, otherwise, send “*doAbort*”
4. Confirm commit via “*haveCommitted*”

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*"
  - ➢ Participants can request status (send "*getDecision*" message to coordinator)
  - ➢ If some have received an answer (or they can figure it out themselves), they can coordinate themselves
- ❑ 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

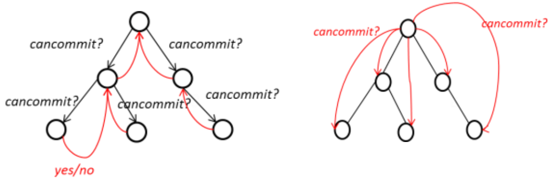
  Crash after "*canCommit*?"

  Use permanent storage to get up to speed

Two-phase commit protocol for nested transactions

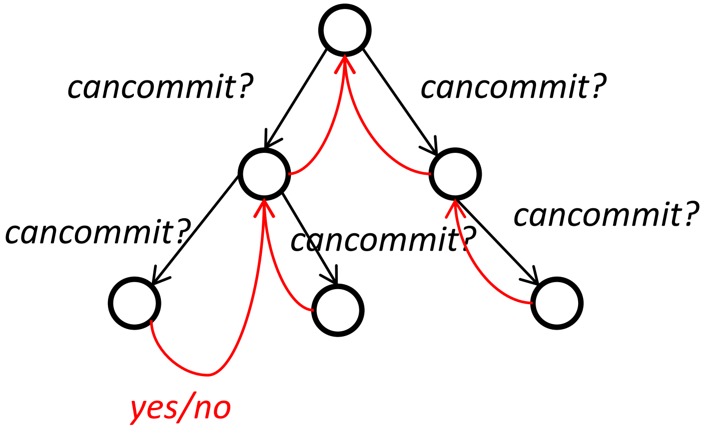
- Sub-transactions "*provisional commit*"
  - Nothing written to permanent storage

    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



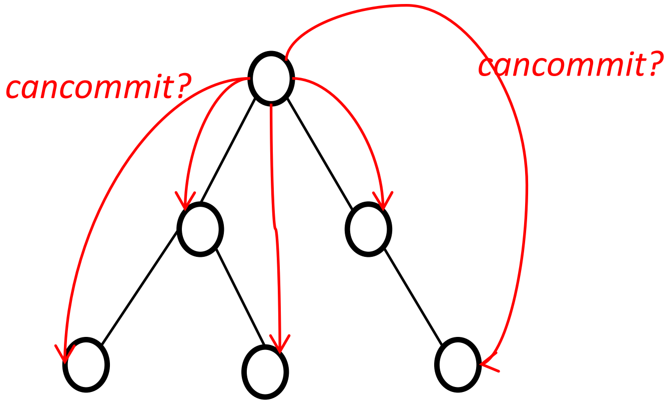
Hierarchic voting

- Responsibility to vote passed one level/generation at a time, through the tree



Flat voting

- The coordinator contacts participants directly
  - Sends: Transaction ID and the list of transactions that are reported as aborted
- Coordinators may manage more than one sub-transaction, and due to crashes, this information may be required
- Coordinators must check if managed sub-transactions have an aborted ancestor (from the aborted transactions list)



Summary

- Transactions – specify sequence of operations that are atomic in presence of server crashes
- ACID properties
- Types of transactions
  - Flat and nested transactions
  - Distributed—flat and nested– transactions
- Problems due to concurrency
  - Lost update
  - Inconsistent retrieval

- Serial equivalence (Serializabitily)
  - Conflicting operations – read-read, read-write, write-read
- Aborted transactions
  - Dirty reads, premature writes
- Atomic commit
- Two-phase commit

Next Lecture

Concurrency Control