

# Distributed Transaction Management

Anastassia Ailamaki  
<http://www.cs.cmu.edu/~natassa>

---

---

---

---

---

---

---

---

## Distributed Transactions

### Two Issues:

- **Develop an atomic commit protocol**
  - a cooperative procedure used by a set of servers involved in a distributed transaction
  - enable the servers to reach a joint decision as to whether a transaction can be committed or aborted
- **Deal with distributed Deadlock**
  - each member of a group of transactions is waiting for some other member to release a lock.

2

---

---

---

---

---

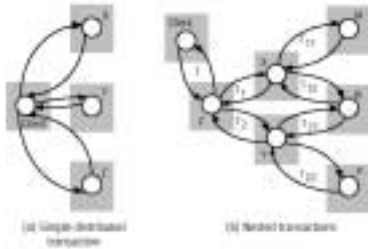
---

---

---

## Distributed Transactions

Distributed transactions:



3

---

---

---

---

---

---

---

---

## Atomic Commit Protocol

- **Transaction atomicity:** either all of its operations are carried out or none of them
- In a distributed environment, all the servers involved a transaction must **agree** on the final outcome of the transaction.
  - i.e., a transaction must either commit or abort all servers.
- Why do we need an atomic commitment protocol?
  - uncertainty of the servers' decisions on the transaction commit
  - the server's decision is affected by the concurrency control, server and network failure

4

---

---

---

---

---

---

---

---

## Two-Phase Commit Protocol

- **In the second phase:**
  - every server carries out joint decision
  - one server votes to abort => abort transaction
  - all servers vote to commit => commit transaction
- **The problem:** How to ensure that all of the servers vote + that they all reach the same decision.
- **Answer:** It is simple if no errors occur, but the protocol must work correctly even when server fails, messages are lost, etc.

5

---

---

---

---

---

---

---

---

## Two-Phase Commit Protocol

- Simplest and most widely used commit protocol
- **In the first phase:**
  - each server votes for the transaction to be committed or aborted
  - once a server has voted to commit a transaction, it is not allowed to abort it even if it fails and restarts in the interim
  - the server voting to commit must ensure that the updated data have been saved in the stable storage and enters the prepared state

6

---

---

---

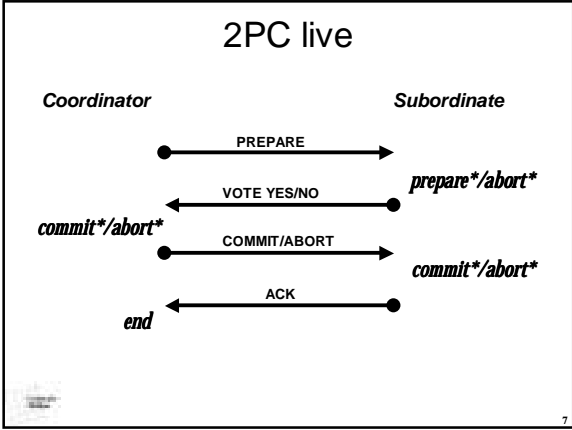
---

---

---

---

---




---

---

---

---

---

---

---

---

- ### 2PC principles of operation
- 4 types of messages:  
prepare, vote y/n, commit/abort, ack
  - 4 types of log records:  
prepare\*, commit\*, abort\*, end
  - Subordinates force-write log records – why?  
(never ask coordinator about that info)
  - Why are ACKs required?  
(to ensure everyone knows final outcome)

---

---

---

---

---

---

---

---

- ### Blocking
- There are various stages at which a server cannot progress its part of the protocol until it receives another message
  - Example: if a server has voted Yes and is waiting for the decision of the coordinator
    - the server is blocked until it gets the commit decision because it cannot decide unilaterally.
    - But, the data items held cannot be released for use by other transactions.
    - If the coordinator has failed, the server must wait for the decision until the coordinator recovers
  - Timeouts at the coordinator in the first phase may avoid the long waiting due to the long delay of server's response.

---

---

---

---

---

---

---

---

## Summary thus far

- Committing Transaction:
  - Subordinate
    - Writes 2 records (prepare\*, commit\*)
    - Sends 2 messages (YES vote and ACK)
  - Coordinator
    - Writes 2 records (commit\*, end)
    - Sends 2 msgs to each subord (prepare and commit)
- If everything goes well:
  - 3(N-1) messages.
  - The ACK messages are not counted since the protocol can function correctly without them

10

---

---

---

---

---

---

---

---

## 2PC and Failures

- Assumptions
  - recovery exists both sides
  - all failed nodes ultimately recover
- What happens if recovery finds node in
  - **prepared** state  
(periodically polls coordinator to find what happened)
  - transaction alive at crash, **no** log information  
(don't know, don't care, undo, write abort record)
  - **commit** or **abort** state  
(periodically send commit or abort to no-ack subords)

11

---

---

---

---

---

---

---

---

## 2PC and Failures (cont.)

- Coordinator notices subordinate failure.
  - If subordinate **has not sent vote**  
coordinator aborts transaction
  - If subordinate **has not sent ACK**  
coordinator hands Xtion over to recovery process
- Subordinate notices coordinator failure.
  - If subordinate **has not sent vote (not prepared)**  
subordinate *unilaterally* aborts transaction
  - If subordinate **is in prepared state**  
subordinate hands Xtion over to recovery process

12

---

---

---

---

---

---

---

---

## 2PC and Failures (cont.)

- Recovery receives inquiry from prepared subord
  - If **there is final information** about Xtion sends abort or commit accordingly
  - If **no information is found** about Xtion abort

13

---

---

---

---

---

---

---

---

## Hierarchical 2PC

- Hierarchical 2PC: simple extension
- How save messages/communication/blocking?
- “Presumed abort” and “Presumed Commit”

14

---

---

---

---

---

---

---

---

## Presumed Abort

- Obs.1: it is safe to “forget” a Xtion after deciding to abort
  - ⇒ Abort need not be \*, no ACKs are needed for aborts
  - ⇒ No end record after writing an abort record
  - ⇒ Subord failure => no need to do anything
- Obs.2: Partially read-only Xtion
  - ⇒ Subordinate only needs “forget” Xtion
  - ⇒ Messages: PREPARE, READ VOTE, COMMIT (only to writers)

15

---

---

---

---

---

---

---

---

## PA Protocol Overhead

- Completely read-only Xtion:
  - Noone writes log
  - Each nonleaf sends *prepare* to each subordinate
  - Each nonroot sends *READ vote*
- Partially read-only Xtion:
  - Each nonleaf
    - sends *prepare* and *commit* to update subords
    - sends *prepare* to read-only subords
    - Writes *prepare\**, *commit\**, end (if it updates) log records
    - If nonroot, sends *YES vote* and *ACK* to coordinate
  - Read leaf = read-only PA
  - Update leaf = subordinate in regular 2PC

16

---

---

---

---

---

---

---

---

## Presumed Commit

- Observation: Transactions usually commit ☺
- Cheaper to:
  - Require ACKs for Aborts
  - Eliminate ACKs for commits
- Force only abort\*, no information means commit!
  - Is this going to work?  
(No! Commit after crash after sending out "prepare!")
- Record subord names before prepared state
  - = Subordinates as in PA; coord writes collecting\*
  - = Read-only optimizations apply here

17

---

---

---

---

---

---

---

---

## PC Protocol Overhead

- Completely read-only Xtion:
  - Each nonleaf writes *collecting\** and commit records
  - Each nonleaf sends *prepare* to each subordinate
  - Each nonroot sends *READ vote*
- Partially read-only Xtion:
  - Root
    - sends *prepare* and *commit* to subords that sent YES vote
    - sends *prepare* to read-only subords
    - If root, *collecting\**, *commit\** log records
  - Each nonleaf, nonroot
    - sends *prepare* and *commit* to subords that sent YES vote
    - sends *prepare* to read-only subords
    - sends YES vote to coordinator if update subtree
    - writes *collecting\**, *prepared\**, *commit*
  - Read leaf = read-only PA
  - Update leaf = sends YES vote, writes *prepare\** and *commit*

18

---

---

---

---

---

---

---

---

## Deadlock with RW Locks

Transaction T		Transaction U	
Operations	Locks	Operations	Locks
Balance := A.Read()	read locks A	Balance := C.Read()	read locks C
A.Write(balance + 4)	write locks A	C.Write(balance - 5)	write locks C
***			
Balance := B.Read()	read locks B	Balance := B.Read()	shares read lock on B
B.Write(balance + 4)	waits for U	B.Write(balance + 2)	waits for T
***		***	

19

---

---

---

---

---

---

---

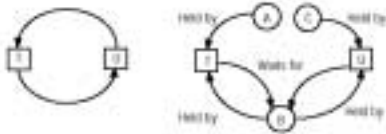
---

---

---

## Wait-For Graph

The wait-for graph for Figure 13.4.



20

---

---

---

---

---

---

---

---

---

---

## Distributed Deadlock

Holdings of transactions U, V and W.

U		V		W	
Deposit(B)	lock B	Deposit(B)	lock B		
Deposit(A)	lock A				
Withdraw(B)	wait			Deposit(C)	lock C
		Withdraw(C)	wait	Withdraw(A)	wait

21

---

---

---

---

---

---

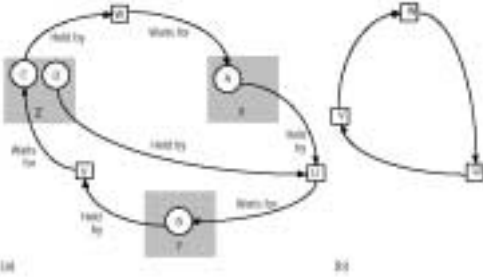
---

---

---

---

## Distributed Deadlock



22

---

---

---

---

---

---

---

---

## Distributed Deadlock

- How detect a distributed deadlock?  
(find global waits-for cycle)
- Solution: dedicated server
  - detects global deadlocks periodically
  - combine local wait-for graphs to check for cycles
- **disadvantages:** single point of failure, lack of fault tolerance and no ability to scale.
- How often to detect deadlocks?

23

---

---

---

---

---

---

---

---

## R\* Deadlock Detection

- DDs at every site gather deadlock information
- Potential Global Deadlock Cycles (PGDCs)
- Information is received/consumed once
  - To avoid redetection of the same deadlock
- PGDG is list of transactions
- Information travels at waits-for direction
  - Only half the involved sites receive it

24

---

---

---

---

---

---

---

---

## Resolving Deadlocks

- ❑ How choose victim?
- ❑ Solution: Cost measures
- ❑ Problem: transaction may not be in the site where deadlock is detected!
- ❑ Find who needs to be informed: too messy
  
- ❑ Solution: just choose local victim
  - ❑ Choose cheaper
  - ❑ Choose the one that will solve most deadlocks

25

---

---

---

---

---

---

---

---

## Deadlocks when Aborting

- ❑ Observation: Aborting transactions also lock!
- ❑ What if a cycle only consists of aborting transactions?

26

---

---

---

---

---

---

---

---

## Summary

- ❑ **Develop an atomic commit protocol**
  - ❑ 2PC preserves ACID properties
  - ❑ Pretty costly
  - ❑ PA, PC alternatives are harder to implement but save messages
  - ❑ Risk of failure is high in tall coord/subord trees
  
- ❑ **R\* deals with distributed Deadlock**

27

---

---

---

---

---

---

---

---