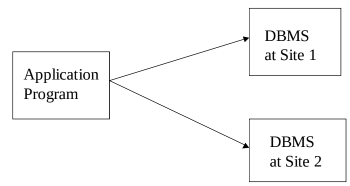
Implementing Distributed Transactions

Chapter 24

Distributed Transaction

- A distributed transaction accesses resource managers distributed across a network
- When resource managers are DBMSs we refer to the system as a *distributed database system*



Distributed Database Systems

- Each local DBMS might export
  - stored procedures, or
  - an SQL interface.
- In either case, operations at each site are grouped together as a subtransaction and the site is referred to as a *cohort* of the distributed transaction
  - Each subtransaction is treated as a transaction at its site
- *Coordinator* module (part of TP monitor) supports ACID properties of distributed transaction
  - Transaction manager acts as coordinator

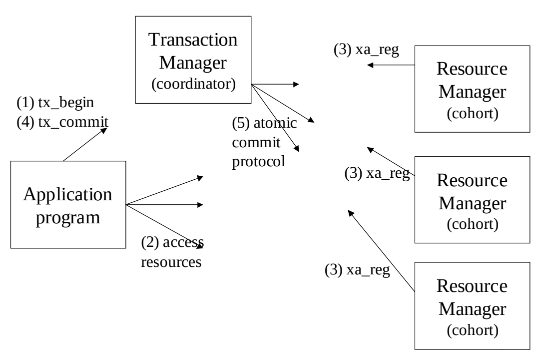
ACID Properties

- Each local DBMS
  - supports ACID properties locally for each subtransaction
    - Just like any other transaction that executes there
  - eliminates local deadlocks
- The additional issues are:
  - *Global atomicity*: all cohorts must abort or all commit
  - *Global deadlocks*: there must be no deadlocks involving multiple sites
  - *Global serialization*: distributed transaction must be globally serializable

Global Atomicity

- All subtransactions of a distributed transaction must commit or all must abort
- An *atomic commit protocol*, initiated by a *coordinator* (*e.g.*, the transaction manager), ensures this.
  - Coordinator polls *cohorts* to determine if they are all willing to commit
- Protocol is supported in the *xa* interface between a transaction manager and a resource manager

Atomic Commit Protocol



Cohort Abort

- Why might a cohort abort?
  - Deferred evaluation of integrity constraints
  - Validation failure (optimistic control)
  - Deadlock
  - Crash of cohort site
  - Failure prevents communication with cohort site

7

Atomic Commit Protocol

- Most commonly used atomic commit protocol is the *two-phase commit protocol*
- Implemented as an exchange of messages between the coordinator and the cohorts
- Guarantees global atomicity of the transaction even if failures should occur while the protocol is executing

8

Two-Phase Commit – The Transaction Record

- During the execution of the transaction, before the two-phase commit protocol begins:
  - When the application calls tx_begin to start the transaction, the coordinator creates a *transaction record* for the transaction in volatile memory
  - Each time a resource manager calls xa_reg to join the transaction as a cohort, the coordinator appends the cohort's identity to the transaction record

9

Two-Phase Commit -- Phase 1

- When application invokes tx_commit, coordinator sends *prepare* message to all cohorts
- *prepare message* (coordinator to cohort) :
  - If cohort wants to abort at any time prior to or on receipt of the message, it aborts and releases locks
  - If cohort wants to commit, it moves all update records to mass store by *forcing* a *prepare record* to its log
    - Guarantees that cohort will be able to commit (despite crashes) if coordinator decides commit (since update records are durable)
    - Cohort enters *prepared* state
  - Cohort sends a *vote message* ("ready" or "aborting"). It
    - cannot change its mind
    - retains all locks if vote is "ready"
    - enters *uncertain period* (it cannot foretell final outcome)

10

Two-Phase Commit -- Phase 1

- *vote message* (cohort to coordinator): Cohort indicates it is "ready" to commit or is "aborting"
  - Coordinator records vote in transaction record
  - If any votes are "aborting", coordinator decides abort and deletes transaction record
  - If all are "ready", coordinator decides commit, forces *commit record* (containing transaction record) to its log (end of phase 1)
    - Transaction committed when commit record is durable
    - Since all cohorts are in prepared state, transaction can be committed despite any failures
  - Coordinator sends *commit* or *abort message* to all cohorts

11

Two-Phase Commit -- Phase 2

- *Commit* or *abort message* (coordinator to cohort):
  - If *commit message*
    - cohort commits locally by forcing a *commit record* to its log
    - cohort sends *done message* to coordinator
  - If *abort message*, it aborts
  - In either case, locks are released and uncertain period ends
- *done message* (cohort to coordinator):
  - When coordinator receives a *done message* from each cohort, it writes a *complete record* to its log and deletes transaction record from volatile store

12

Two-Phase Commit (commit case)

Application | Coordinator | Cohort

- tx_commit → - send *prepare msg* to cohorts in trans. rec. → - force prepare rec. to cohort log
- *phase 1*
- - record vote in trans. rec. ← - send *vote msg*
- - *if all vote ready,* force commit rec. to coord. log
- - send *commit msg* → -force commit rec. to cohort log
- *uncertain period*
- *phase 2*
- - release locks
- - *when all done msgs rec'd,* write complete rec. to log ← - send *done msg*
- - delete trans. rec.
- resume ← - return status
- *xa interface*

Two-Phase Commit (abort case)

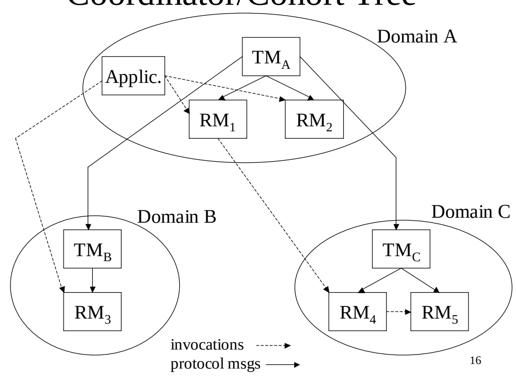
Application | Coordinator | Cohort

- tx_commit → - send *prepare msg* to cohorts in trans. rec. → - force prepare rec. to cohort log
- *phase 1*
- - record vote in trans.rec. ← - send *vote msg*
- - *if any vote abort,* delete transaction rec.
- *uncertain period*
- - send *abort msg* → - local abort
- resume ← - return status
- - release locks
- *xa interface*

Distributing the Coordinator

- A transaction manager controls resource managers in its *domain*
- When a cohort in domain A invokes a resource manager, RM_B, in domain B, the local transaction manager, TM_A, and remote transaction manager, TM_B, are notified
  - TM_B is a cohort of TM_A and a coordinator of RM_B
- A coordinator/cohort tree results

Coordinator/Cohort Tree



Distributing the Coordinator

- The two-phase commit protocol progresses down and up the tree in each phase
  - When TM_B gets a *prepare msg* from TM_A it sends a *prepare msg* to each child and waits
  - If each child votes ready, TM_B sends a *ready msg* to TM_A
    - if not it sends an *abort msg*

Failures and Two-Phase Commit

- A participant recognizes two failure situations.
  - *Timeout* : No response to a message. Execute a timeout protocol
  - *Crash :* On recovery, execute a restart protocol
- If a cohort cannot complete the protocol until some failure is repaired, it is said to be *blocked*
  - Blocking can impact performance at the cohort site since locks cannot be released

Timeout Protocol

- Cohort times out waiting for *prepare message*
  - Abort the subtransaction
    - Since the (distributed) transaction cannot commit unless cohort votes to commit, atomicity is preserved
- Coordinator times out waiting for *vote message*
  - Abort the transaction
    - Since coordinator controls decision, it can force all cohorts to abort, preserving atomicity

19

Timeout Protocol

- Cohort (in prepared state) times out waiting for *commit/abort message*
  - Cohort is *blocked* since it does not know coordinator's decision
    - Coordinator might have decided commit or abort
    - Cohort cannot unilaterally decide since its decision might be contrary to coordinator's decision, violating atomicity
    - Locks cannot be released
  - Cohort requests status from coordinator; remains blocked
- Coordinator times out waiting for *done message*
  - Requests done message from delinquent cohort

20

Restart Protocol - Cohort

- On restart cohort finds in its log
  - begin_transaction record, but no prepare record:
    - Abort (transaction cannot have committed because cohort has not voted)
  - prepare record, but no commit record (cohort crashed in its uncertain period)
    - Does not know if transaction committed or aborted
    - Locks items mentioned in update records before restarting system
    - Requests status from coordinator and *blocks* until it receives an answer
  - commit record
    - Recover transaction to committed state using log

21

Restart Protocol - Coordinator

- On restart:
  - Search log and restore to volatile memory the transaction record of each transaction for which there is a commit record, but no complete record
    - Commit record contains transaction record
- On receiving a request from a cohort for transaction status:
  - If transaction record exists in volatile memory, reply based on information in transaction record
  - If no transaction record exists in volatile memory, reply abort
    - Referred to as **presumed abort property**

22

Presumed Abort Property

- If, when a cohort asks for the status of a transaction, there is no transaction record in coordinator's volatile storage, either
  - The coordinator had aborted the transaction and deleted the transaction record
  - The coordinator had crashed and restarted and did not find the commit record in its log because
    - It was in Phase 1 of the protocol and had not yet made a decision, or
    - It had previously aborted the transaction
  - or …

23

Presumed Abort Property

  - The coordinator had crashed and restarted and found a complete record for the transaction in its log
  - The coordinator had committed the transaction, received done messages from all cohorts and hence deleted the transaction record from volatile memory
- The last two possibilities cannot occur
  - In both cases, the cohort has sent a done message and hence would not request status
- Therefore, coordinator can respond abort

24

Heuristic Commit

- What does a cohort do when in the blocked state and the coordinator does not respond to a request for status?
  - Wait until the coordinator is restarted
  - Give up, make a unilateral decision, and attach a fancy name to the situation.
    - Always abort
    - Always commit
    - Always commit certain types of transactions and always abort others
  - Resolve the potential loss of atomicity outside the system
    - Call on the phone or send email

25

Variants/Optimizations

- Read/only subtransactions need not participate in the protocol as cohorts
  - As soon as such a transaction receives the prepare message, it can give up its locks and exit the protocol.
- Transfer of coordination

26

Transfer of Coordination

- Sometimes it is not appropriate for the coordinator (in the initiator's domain) to coordinate the commit
  - Perhaps the initiator's domain is a convenience store and the bank does not trust it to perform the commit
- Ability to coordinate the commit can be transferred to another domain
    - Linear commit
    - Two-phase commit without a prepared state

27

Linear Commit

- Variation of two-phase commit that involves transfer of coordination
- Used in a number of Internet commerce protocols
- Cohorts are assumed to be connected in a linear chain

28

Linear Commit Protocol

- When leftmost cohort, *A*, is ready to commit, it goes into a prepared state and sends a *vote message* ("ready") to the cohort to its right, *B* (requesting *B* to act as coordinator).
- After receiving the *vote message*, if *B* is ready to commit, it also goes into a prepared state and sends a *vote message* ("ready") to the cohort to its right, *C* (requesting *C* to act as coordinator)
- And so on ...

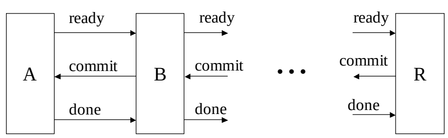
29

Linear Commit Protocol

- When *vote message* reaches the rightmost cohort, *R*, if *R* is ready to commit, it commits the entire transaction (acting as coordinator) and sends a *commit message* to the cohort on its left
- The *commit message* propagates down the chain until it reaches *A*
- When *A* receives the *commit message* it sends a *done message* to *B*, and that also propagates

30

Linear Commit



31

Linear Commit Protocol

- Requires fewer messages than conventional two-phase commit. For *n* cohorts,
  - – Linear commit requires 3(*n* - 1)
  - – Two-phase commit requires 4*n* messages
- But two-phase commit requires only 4 message times (messages are sent in parallel) while linear commit requires 3(*n* - 1) times (messages are sent serially)

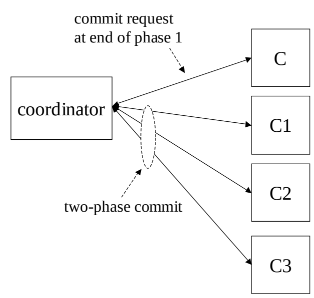
32

Two-Phase Commit Without a Prepared State

- Assume exactly one cohort, *C*, does not support a prepared state.
- Coordinator performs Phase 1 of two-phase commit protocol with all other cohorts
- If they all agree to commit, coordinator requests that *C* commit its subtransaction (in effect, requesting *C* to decide the transaction's outcome)
- *C* responds commit/abort, and the coordinator sends a *commit/abort* message to all other sites

33

Two-Phase Commit Without a Prepared State



34

Global Deadlock

- With distributed transaction:
  - – A deadlock might not be detectable at any one site
    - Subtransaction T_{1A} of T_1 at site A might wait for subtransaction T_{2A} of T_2, while at site B, T_{2B} waits for T_{1B}
  - – Since concurrent execution within a transaction is possible, a transaction might progress at some site even though deadlocked
    - T_{2A} and T_{1B} can continue to execute for a period of time

35

Global Deadlock

- Global deadlock cannot always be resolved by aborting and restarting a single subtransaction, since data might have been communicated between cohorts
  - – T_{2A}'s computation might depend on data received from T_{2B}. Restarting T_{2B} without restarting T_{2A} will not in general work.

36

Global Deadlock Detection

- Global deadlock detection is generally a simple extension of local deadlock detection
  - Check for a cycle when a cohort waits
    - If a cohort of T_1 is waiting for a cohort of T_2, coordinator of T_1 sends probe message to coordinator of T_2
    - If a cohort of T_2 is waiting for a cohort of T_3, coordinator of T_2 relays the probe to coordinator of T_3
    - If probe returns to coordinator of T_1 a deadlock exists
  - Abort a distributed transaction if the wait time of one of its cohorts exceeds some threshold

37

Global Deadlock Prevention

- Global deadlock prevention - use timestamps
  - For example an older transaction never waits for a younger one. The younger one is aborted.

38

Global Isolation

- If subtransactions at different sites run at different isolation levels, the isolation between concurrent distributed transactions cannot easily be characterized.
- Suppose all subtransactions run at SERIALIZABLE. Are distributed transactions as a whole serializable?
  - *Not necessarily*
    - T_{1A} and T_{2A} might conflict at site A, with T_{1A} preceding T_{2A}
    - T_{1B} and T_{2B} might conflict at site B, with T_{2B} preceding T_{1B}.

39

Two-Phase Locking and Two-Phase Commit

- **Theorem**: If all sites use a strict two-phase locking protocol and the transaction manager uses a two-phase commit protocol, transactions are globally serializable in commit order.
  - *Argument*: Suppose previous situation occurred.
    - At site A
      - T_{2A} cannot commit until T_{1A} releases locks (2Φ locking)
      - T_{1A} does not release locks until T_1 commits (2Φ commit)
      - Hence (if both commit) T_1 commits before T_2
    - At site B
      - Similarly (if both commit) T_2 commits before T_1 .
    - => Contradiction (transactions deadlock in this case)

40

When Global Atomicity Cannot Always be Guaranteed

- A site might refuse to participate
  - Concerned about blocking
  - Charges for its services
- A site might not be able to participate
  - Does not support prepared state
- Middleware used by client might not support two-phase commit
  - For example, ODBC
- Heuristic commit

41

Spectrum of Commit Protocols

- Two-phase commit
- One-phase commit
  - When all subtransactions have completed, coordinator sends a commit message to each one
  - Some might commit and some might abort
- Zero-phase commit
  - When each subtransaction has completed, it immediately commits or aborts and informs coordinator
- Autocommit
  - When each database operation completes, it commits

42

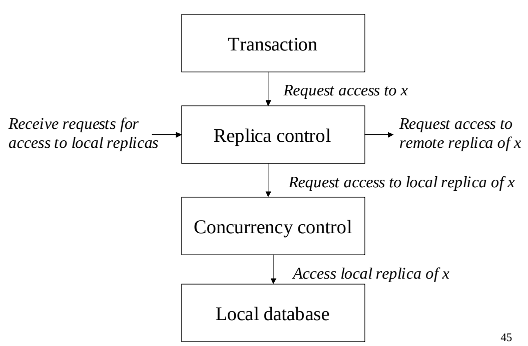
Data Replication

- **Advantages**
  - Improves *availability*: data can be accessed even though some site has failed
  - Can improve performance: a transaction can access the closest (perhaps local) replica
- **Disadvantages**
  - More storage
  - Increases system complexity
    - *Mutual consistency* of replicas must be maintained
    - Access by concurrent transactions to different replicas can lead to incorrect results

Application Supported Replication

- Application creates replicas
  - If X_1 and X_2 are replicas of the same item, each transaction enforces the global constraint $X_1 = X_2$
  - Distributed DBMS is unaware that X_1 and X_2 are replicas
  - When accessing an item, a transaction must specify which replica it wants

System Supported Replication



Replica Control

- Hides replication from transaction
- Knows location of all replicas
- Translates transaction's request to access an item into request to access a particular replica(s)
- Maintains some form of *mutual consistency*:
  - *Strong*: all replicas always have the same value
    - In every committed version of the database
  - *Weak*: all replicas eventually have the same value
  - *Quorum*: a quorum of replicas have the same value

Read One/Write All Replica Control

- Satisfies a transaction's read request using the nearest replica
- Causes a transaction's write request to update all replicas
  - Synchronous case: immediately (before transaction commits)
  - Asynchronous case: eventually
- Performance benefits result if reads occur substantially more often the writes

Synchronous-Update Read One/ Write All Replica Control

- Read request locks and reads the most local replica
- Write request locks and updates all replicas
  - Maintains strong mutual consistency
- Atomic commit protocol guarantees that all sites commit and makes new values durable
- Schedules are serializable
- **Problems**: Writing:
  - Has poor performance
  - Is prone to deadlock
  - Requires 100% availability

Generalizing Read One/Write All

- **Problem**: With read one/write all, availability is worse for writers since all replicas have to be accessible
- **Goal**: A replica control in which an item is available for all operations even though some replicas are inaccessible
- This implies:
  – Mutual consistency is not maintained
  – Value of an item must be reconstructed by replica control when it is accessed

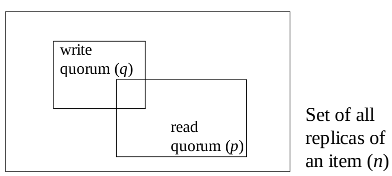
49

Quorum Consensus Replica Control

- Replica control dynamically selects and locks a read (or write) quorum of replicas when a read (or write) request is made
  – Read operation reads only replicas in the read quorum
  – Write operation writes only replicas in the write quorum
  – If p = |read quorum|, q = |write quorum| and n = |replica set| then algorithm requires
    $p+q > n$ (read/write conflict)
    $q > n/2$ (write/write conflict)
- Guarantees that all conflicts between operations of concurrent transactions will be detected at some site and one transaction will be forced to wait.
  – Serializability is maintained

50

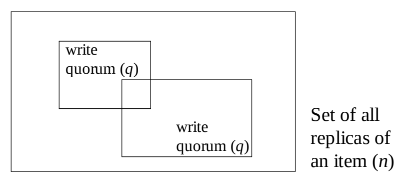
Quorum Consensus Replica Control



- $p+q > n$ (read/write conflict)
- An intersection between any read and any write quorum is guaranteed

51

Quorum Consensus Replica Control



- $q > n/2$ (read/write conflict)
- An intersection between any two write quorums is guaranteed

52

Mutual Consistency

- **Problem**: Algorithm does not maintain mutual consistency; thus reads of replicas in a read quorum might return different values
- **Solution**: Assign a timestamp to each transaction, T, when it commits; clocks are synchronized between sites so that timestamps correspond to commit order
  – T writes: replica control associates T's timestamp with all replicas in its write quorum
  – T reads: replica control returns value of replica in read quorum with largest timestamp. Since read and write quorums overlap, T gets most recent write
  – Schedules are serializable

53

Quorum Consensus Replica Control

- Allows a tradeoff among operations on availability and cost
  – A small quorum implies the corresponding operation is more available and can be performed more efficiently but ...
  – The smaller one quorum is, the larger the other

54

Failures

- Algorithm can continue to function even though some sites are inaccessible
- No special steps required to recover a site after a failure occurs
  - Replica will have an old timestamp and hence its value will not be used
  - Replica's value will be made current the next time the site is included in a write quorum

55

Asynchronous-Update Read One/Write All Replica Control

- **Problem**: Synchronous-update is slow since all replicas (or a quorum of replicas) must be updated before transaction commits
- **Solution**: With asynchronous-update only some (usually one) replica is updated as part of transaction. Updates propagate after transaction commits but…
  - only weak mutual consistency is maintained
  - serializability is not guaranteed

56

Asynchronous-Update Read One/Write All Replica Control

- Weak mutual consistency can result in non-serializable schedules

new

T_1: $w(x_A)$ $w(y_B)$ *commit*

T_2: $r(x_C)$ $r(y_B)$ *commit*

T_{rep_upd}: $w(x_C)$ $w(x_B)$ . . .

old

- Alternate forms of asynchronous-update replication vary the degree of synchronization between replicas; none support serializability

57

Primary Copy Replica Control

- One copy of each item is designated *primary*; the other copies are *secondary*
  - A transaction (locks and) reads the nearest copy
  - A transaction (locks and) writes the primary copy
  - After a transaction commits, updates it has made to primary copies are propagated to secondary copies (asynchronous)
- Writes of all transactions are serializable, reads are not

58

Primary Copy Replica Control

old

T_1: $w(x_{pri})$ $w(y_{pri})$ *commit*

T_2: $r(x_{pri})$ $r(y_B)$ *commit*

T_{rep_upd}: $w(x_C)$ $w(x_B)$ $w(y_C)$ $w(y_B)$

new

- The schedule is not serializable

59

Primary Copy Mutual Consistency

- Updates of an item are propagated by
  - A single (distributed) propagation transaction
  - Multiple propagation transactions
  - Periodic broadcast
- Weak mutual consistency is guaranteed if the sequence of updates made to the primary copy of an item (by all transactions) is applied to each secondary copy of the item (in the same order).

60

Example Where Asynchronous Update is OK

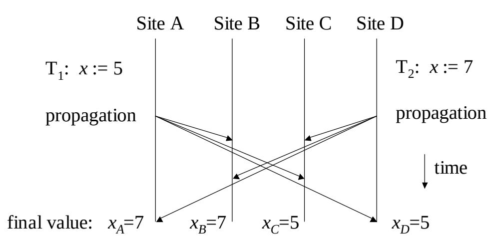
- Internet Grocer keeps replicated information about customers at two sites
  - Central (primary) site where customers place orders
  - Warehouse (secondary) site from which deliveries are made
- With synchronous update, order transactions are distributed and become a bottleneck
- With asynchronous update, order transaction updates the central site immediately; update is propagated to the warehouse site later.
  - Provides faster response time to customer
  - Warehouse site does not need data immediately

Variations on Propagation

- A secondary site might declare a view of the primary, so that only the relevant part of the item is transmitted
  - Good for low bandwidth connections
- With a *pull* strategy (in contrast to a *push* strategy) a secondary site requests that its view be updated
  - Good for sites that are not continuously connected, *e.g*. laptops of business travelers

Group Replication (asynchronous)

- A transaction can (lock and) update *any* replica.
- **Problem**: Does not support weak mutual consistency.



Group Replication - Conflicts

- *Conflict*: Updates are performed concurrently to the same item at different sites.
- **Problem**: If a replica takes as its value the contents of the last update message, weak mutual consistency is not maintained
- **Solution**: Associate unique timestamp with each update and each replica. Replica takes timestamp of most recent update that has been applied to it.
  - Update discarded if its timestamp is less than timestamp of replica
  - Weak mutual consistency is supported

Conflict Resolution

- No conflict resolution strategy yields serializable schedules
  - *e.g.*, timestamp algorithm allows lost update
- Conflict resolution strategies:
  - Most recent update wins
  - Update coming from highest priority site wins
  - User provides conflict resolution strategy
  - Notify the user

Procedural Replication

- **Problem**: Communication costs of previous propagation strategies are high if many items are updated
  - *Ex*: How do you propagate quarterly posting of interest to duplicate bank records?
- **Solution**: Replicate stored procedure at replica sites. Invoke the procedure at each site to do the propagation

Summary of Distributed Transactions

- *The good news*: If transactions run at SERIALIZABLE, all sites use two-phase commit for termination and synchronous update replication, then distributed transactions are globally atomic and serializable.
- *The bad news*: To improve performance
  – applications often do not use SERIALIZABLE
  – DBMSs might not participate in two-phase commit
  – replication is generally asynchronous update
- Hence, consistent transactions might yield incorrect results

67