Intro to Distributed Transactions

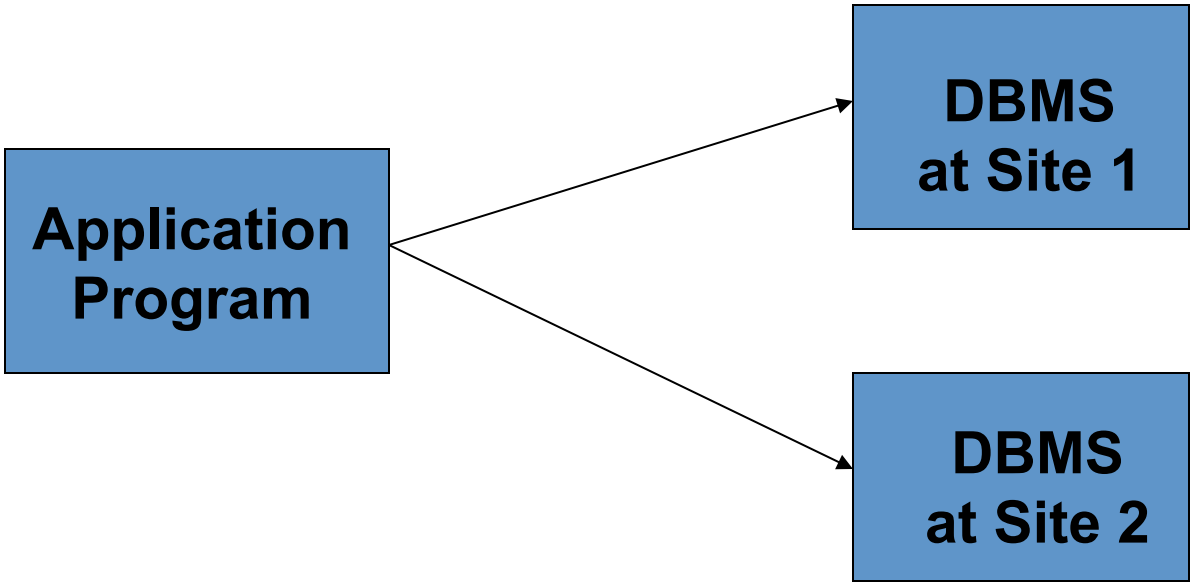
Alex Kalinin

Acknowledgements

- CSE515: Database Transaction Processing Systems (most of the slides)

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.
- 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

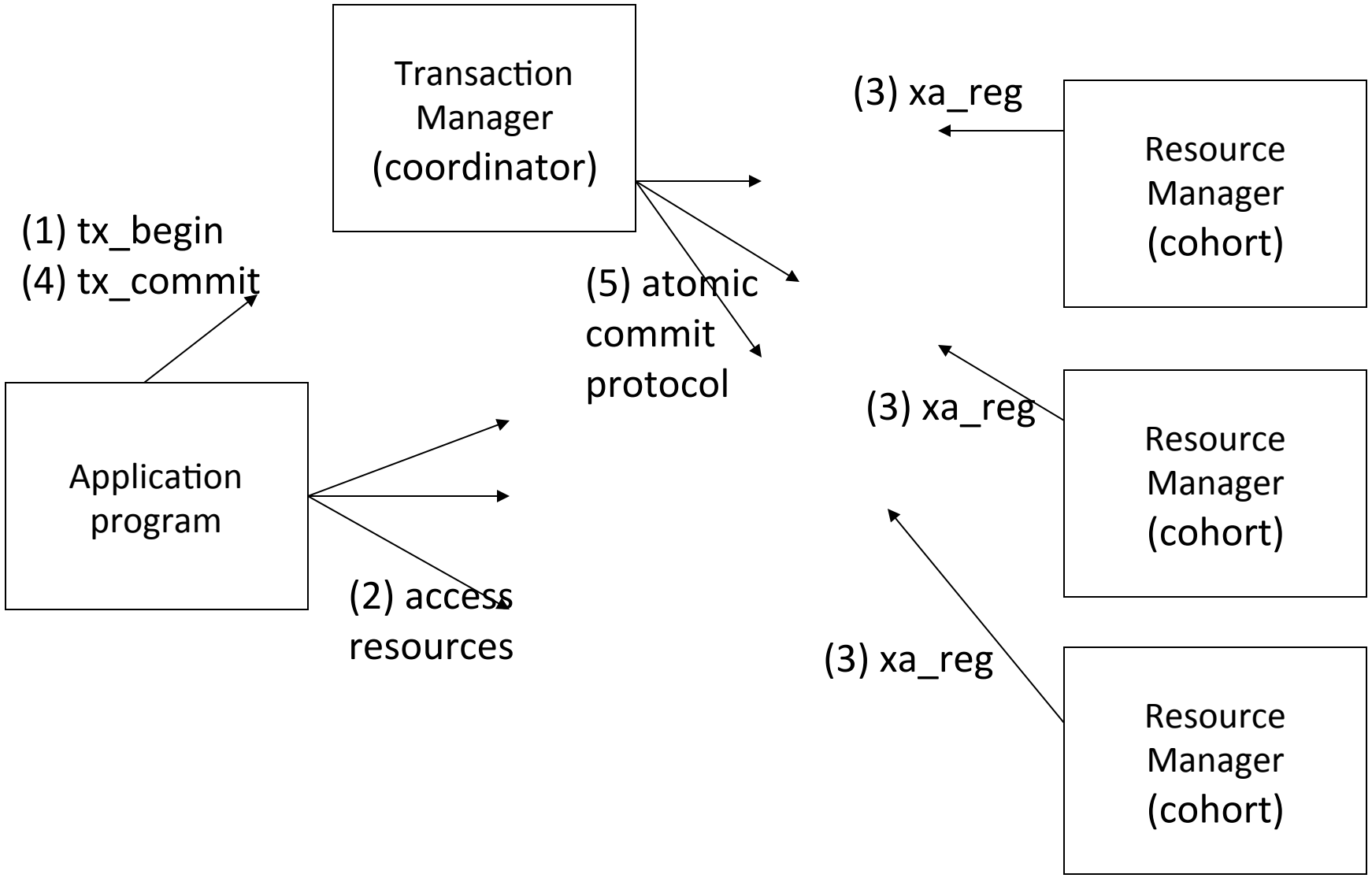
ACID Properties

- Each local DBMS:
  - Supports ACID 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

Atomic Commit Protocol

- Two-phase commit protocol: most commonly used atomic 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.

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

Two-Phase Commit -- Phase 1

- When application invokes tx_commit, coordinator
- Sends prepare message (coordin. to all cohorts) :
  - 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)

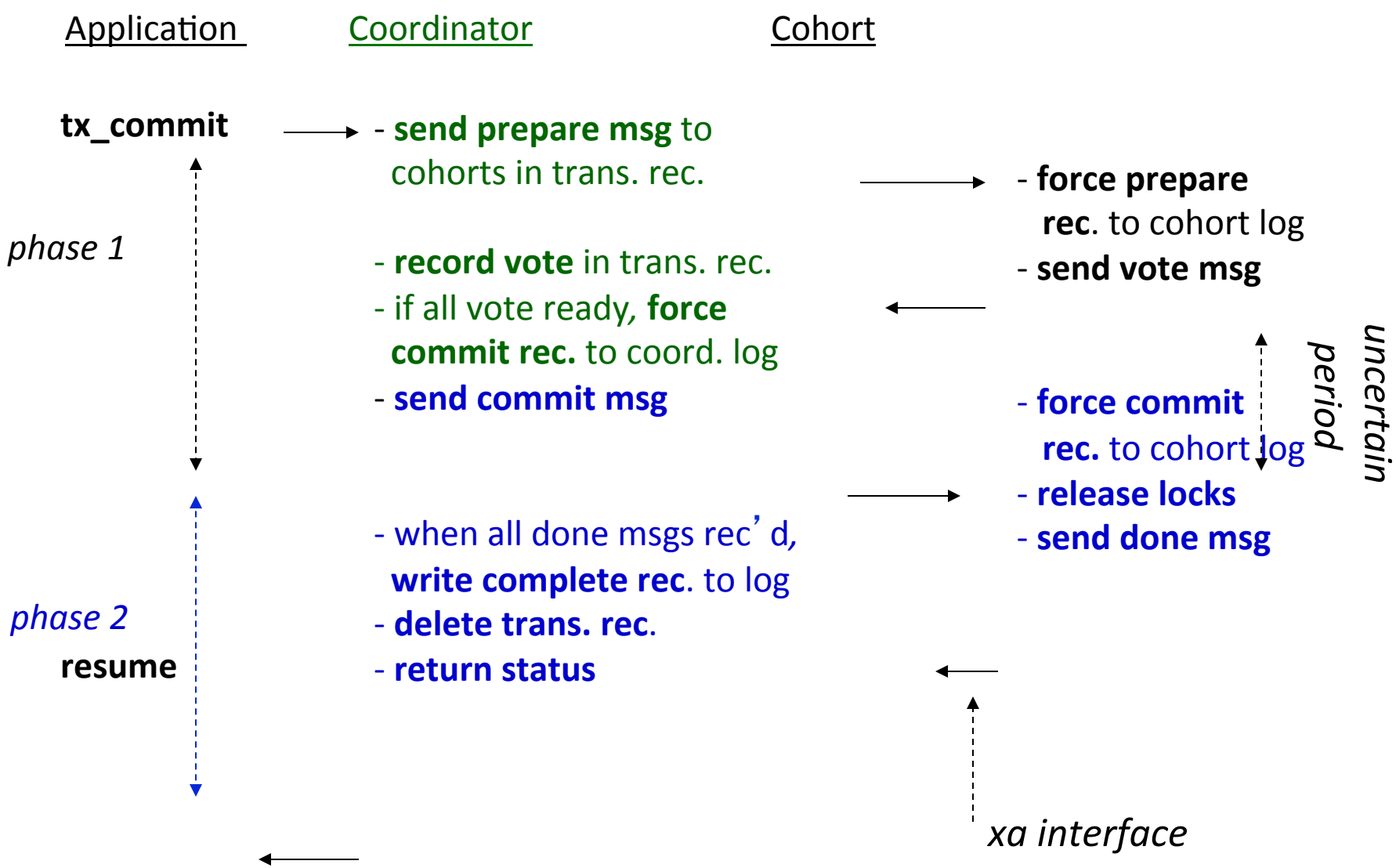
Two-Phase Commit -- Phase 1

- 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

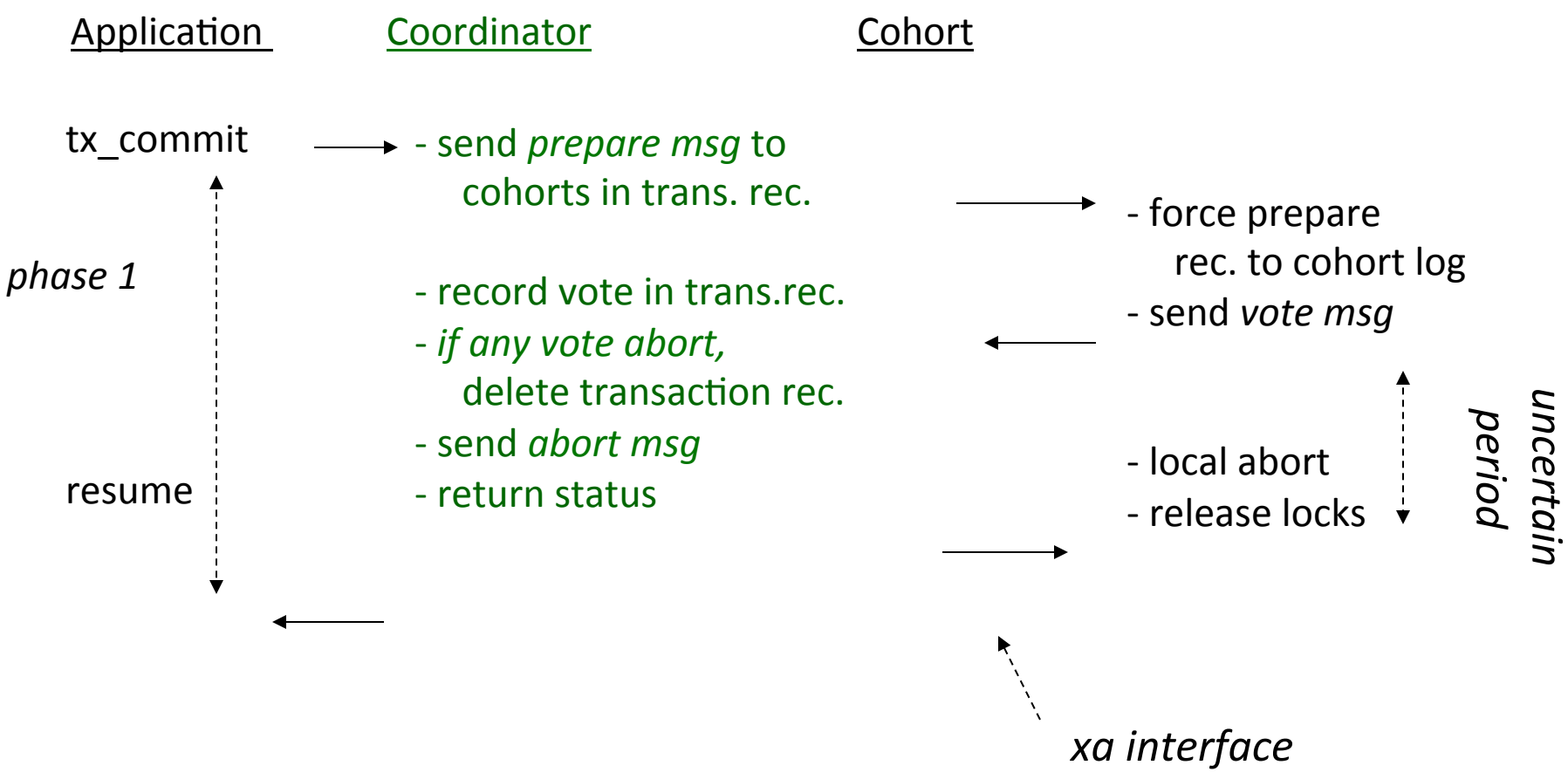
Two-Phase Commit -- Phase 2

- 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

Two-Phase Commit (commit case)



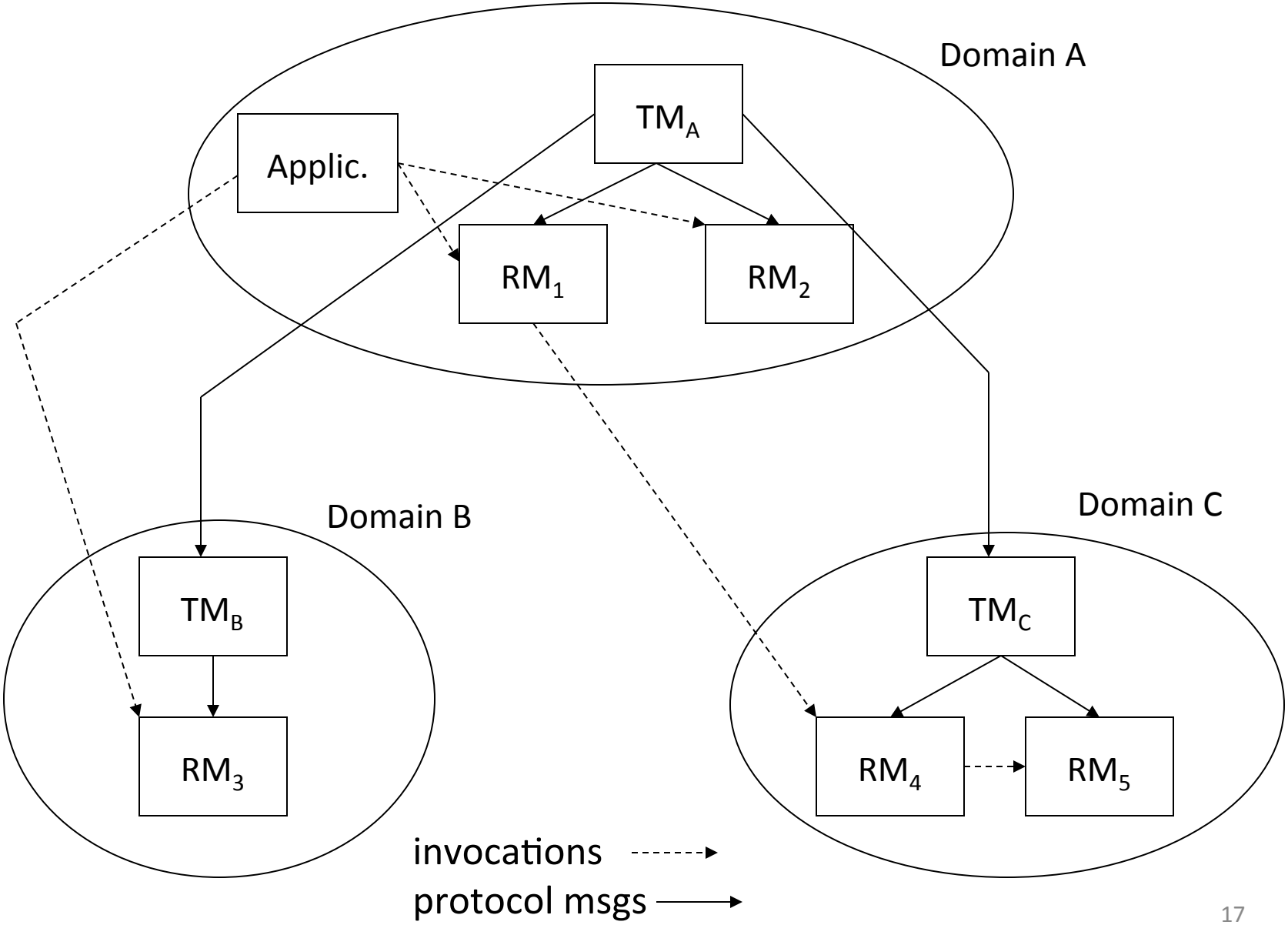
Two-Phase Commit (abort case)



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

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

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

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

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

Presumed Abort Property

- or
  - 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

Presumed Commit

- Acknowledge aborts, not commits
- Force-write abort records, not commits
- Coordinator force-writes a *collecting* record

- No information? Assume commit
- Useful when many subordinates update

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

Optimizations

- Optimize for:
  - Number of messages between the coordinator and cohorts
  - Number of writes to the log

Read-Only Optimization

- Read-only participants do not care about the outcome – no second phase.
- Send the READ vote
- Hierarchical case – send the READ only when you and your children send the READ

Last Agent

- Single remote partner (“last agent”) – high latency
- Collect votes from others, decide and send the result to the “last agent”

Unsolicited vote

- Ready to commit?
- Force-write the “prepare” record, send YES

- Reduces the number of messages at the first stage
- Useful when the network delays are high

Sharing the log

- LRM and TM share the log
- Less records are forced-written
- RM writes “prepared” record
- TM force-writes commit record

- Single log guarantees ordering of records

Group Commits

- Want to combine several force-writes
- Two choices:
  - Wait for a predefined number of transactions
  - Timeout occurs

Long Locks

- ACKs are send at commits
- Delay an ACK until the next transaction starts
- Coordinator waits longer to release the locks

- Reduces network traffic
- Useful when a density of transactions is big

Commit Acknowledgement

- Early: report commit as soon as the record is logged.
- Propagation is not finished!

- Late: report commit after getting all ACKs
- Better guarantees with heuristic decisions

Voting Reliable

- When sending YES, say if you are reliable
- If all YESs are reliable – early acknowledgement
- If not – late acknowledgement

Wait for Outcome

- Coordinator waits for subordinates
- Recovery is in progress? Huge delays
- Can report with “outcome pending”
- Application-dependent

Global Deadlock

- With distributed transaction:
  - A deadlock might not be detectable at any one site
    - Subtrans T_{1A} of T_1 at site A might wait for subtrans 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

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.

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

Global Deadlock Prevention

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

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}.

Two-Phase Locking & Two-Phase Commit

- Theorem: If
  - All sites use a strict two-phase locking protocol,
  - Trans Manager uses a two-phase commit protocol,

  Then
  - Trans are globally serializable in commit order.

Two-Phase Locking & Two-Phase Commit (Argument)

- Suppose previous situation occurred:

  - At site A

  * T2A cannot commit until T1A releases locks (2Φ locking)

  * T1A does not release locks until T1 commits (2Φ commit)

  Hence (if both commit) T1 commits before T2

  - At site B

  * Similarly (if both commit) T2 commits before T1,

- Contradiction (transactions deadlock in this case)

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