Chapter 18: Distributed Databases

- Distributed Data Storage
- Network Transparency
- Distributed Query Processing
- Distributed Transaction Model
- Commit Protocols
- Coordinator Selection
- Concurrency Control
- Deadlock Handling
- Multidatabase Systems

Distributed Database System

- Database is stored on several computers that communicate via media such as wide-area networks, telephone lines, or local area networks.
- Appears to user as a single system
- Processes complex queries
- Processing may be done at a site other than the initiator of the request
- Transaction management
- Optimization of queries provided automatically

Distributed Data Storage

Assume relational data model

- Replication: system maintains multiple copies of data, stored in different sites, for faster retrieval and fault tolerance.
- Fragmentation: relation is partitioned into several fragments stored in distinct sites.
- Replication and fragmentation: relation is partitioned into several fragments; system maintains several identical replicas of each such fragment.

Data Replication

- A relation or fragment of a relation is replicated if it is stored redundantly in two or more sites.
- Full replication of a relation is the case where the relation is stored at all sites.
- Fully redundant databases are those in which every site contains a copy of the entire database.

Data Replication (Cont.)

- Advantages of Replication
  - Availability: failure of a site containing relation *r* does not result in unavailability of *r* if replicas exist.
  - Parallelism: queries on *r* may be processed by several nodes in parallel.
  - Reduced data transfer: relation *r* is available locally at each site containing a replica of *r*.
- Disadvantages of Replication
  - Increased cost of updates: each replica of relation *r* must be updated.
  - Increased complexity of concurrency control: concurrent updates to distinct replicas may lead to inconsistent data unless special concurrency control mechanisms are implemented.

Data Fragmentation

- Division of relation *r* into *fragments* $r_1, r_2, ..., r_n$ which contain sufficient information to reconstruct relation *r*.
- Horizontal fragmentation: each tuple of *r* is assigned to one or more fragments.
- Vertical fragmentation: the schema for relation *r* is split into several smaller schemas.
  - All schemas must contain a common candidate key (or superkey) to ensure lossless join property.
  - A special attribute, the *tuple-id* attribute may be added to each schema to serve as a candidate key.
- Fragments may be successively fragmented to an arbitrary depth. Vertical and horizontal fragmentation can be mixed.
- Example: relation *account* with following schema

Account-schema = (*branch-name*, *account-number*, *balance*)

Horizontal Fragmentation of *account* Relation

branch-name	*account-number*	*balance*
Hillside	A-305	500
Hillside	A-226	336
Hillside	A-155	62

$account_1$

branch-name	*account-number*	*balance*
Valleyview	A-177	205
Valleyview	A-402	10000
Valleyview	A-408	1123
Valleyview	A-639	750

$account_2$

Vertical Fragmentation of *deposit* Relation

branch-name	*customer-name*	*tuple-id*
Hillside	Lowman	1
Hillside	Camp	2
Valleyview	Camp	3
Valleyview	Kahn	4
Hillside	Kahn	5
Valleyview	Kahn	6
Valleyview	Green	7

$deposit_1$

account-number	*balance*	*tuple-id*
A-305	500	1
A-226	336	2
A-177	205	3
A-402	10000	4
A-155	62	5
A-408	1123	6
A-639	750	7

$deposit_2$

Advantages of Fragmentation

- Horizontal:
  - allows parallel processing on a relation
  - allows a global table to be split so that tuples are located where they are most frequently accessed
- Vertical:
  - allows for further decomposition than can be achieved with normalization
  - tuple-id attribute allows efficient joining of vertical fragments
  - allows parallel processing on a relation
  - allows tuples to be split so that each part of the tuple is stored where it is most frequently accessed

Network Transparency

- Degree to which system users may remain unaware of the details of how and where the data items are stored in a distributed system

- Consider transparency issues in relation to:
  - Naming of data items
  - Replication of data items
  - Fragmentation of data items
  - Location of fragments and replicas

Naming of Data Items – Criteria

1. Every data item must have a system-wide unique name.
2. It should be possible to find the location of data items efficiently.
3. It should be possible to change the location of data items transparently.
4. Each site should be able to create new data items autonomously.

Centralized Scheme — Name Server

- Structure:
  - name server assigns all names
  - each site maintains a record of local data items
  - sites ask name server to locate non-local data items
- Advantages:
  - satisfies naming criteria 1-3
- Disadvantages:
  - does not satisfy naming criterion 4
  - name server is a potential performance bottleneck
  - name server is a single point of failure

Use of Aliases

- Alternative to centralized scheme: each site prefixes its own site identifier to any name that it generates, i.e., *site*17.*account*.
  - Fulfills having a unique identifier, and avoids problems associated with central control.
  - However, fails to achieve network transparency.
- Solution: Create a set of *aliases* for data items; Store the mapping of aliases to the real names at each site.
- The user can be unaware of the physical location of a data item, and is unaffected if the data item is moved from one site to another.

Use of Aliases (Cont.)

- Each replica and each fragment of a data item must have a unique name.
  - Use postscripts to determine those replicas that are replicas of the same data item, and those fragments that are fragments of the same data item.
  - fragments of same data item: ".*f*1", ".*f*2", . . . , ".*fn*"
  - replicas of same data item: ".*r*1", ".*r*2", . . . , ".*rn*"

*site*17.*account*.*f*3.*r*2

refers to replica 2 of fragment 3 of *account*; this item was generated by site 17.

Name-Translation Algorithm

if *name* appears in the alias table
  then *expression* := *map* (*name*)
  else *expression* := *name*;
function *map* (*n*)
if *n* appears in the replica table
  then *result* := name of replica of *n*;
if *n* appears in the fragment table
  then begin
      result := expression to construct fragment;
      for each n' **in** *result* **do begin**
        replace n' in *result* with *map* (n');
      end
    end
return *result*;

Example of Name-Translation Scheme

- A user at the Hillside branch, (site S_1), uses the alias *local-account* for the local fragment *account.f1* of the *account* relation.
- When this user references *local-account*, the query-processing subsystem looks up *local-account* in the alias table, and replaces *local-account* with $S1$.*account.f1*.
- If $S1$.*account.f1* is replicated, the system must consult the replica table in order to choose a replica.
- If this replica is fragmented, the system must examine the fragmentation table to find out how to reconstruct the relation.
- Usually only need to consult one or two tables, however, the algorithm can deal with any combination of successive replication and fragmentation of relations.

Transparency and Updates

- Must ensure that all replicas of a data item are updated and that all affected fragments are updated.
- Consider the *account* relation and the insertion of the tuple:

("Valleyview", A-733, 600)

- Horizontal fragmentation of *account*

$$account_1 = \sigma_{branch\text{-}name = \text{"Hillside"}}(account)$$
$$account_2 = \sigma_{branch\text{-}name = \text{"Valleyview"}}(account)$$

  - Predicate P_i is associated with the i^{th} fragment
  - Apply P_i to the tuple ("Valleyview", A-733, 600) to test whether that tuple must be inserted in the i^{th} fragment
  - Tuple inserted into $account_2$

Transparency and Updates (Cont.)

- Vertical fragmentation of *deposit* into $deposit_1$ and $deposit_2$
- The tuple (“Valleyview”, A-733, ‘Jones”, 600) must be split into two fragments:
  - one to be inserted into $deposit_1$
  - one to be inserted into $deposit_2$
- If *deposit* is replicated, the tuple (“Valleyview”, A-733, “Jones”600) must be inserted in all replicas
- Problem: If *deposit* is accessed concurrently it is possible that one replica will be updated earlier than another (see section on Concurrency Control).

Distributed Query Processing

- For centralized systems, the primary criterion for measuring the cost of a particular strategy is the number of disk accesses.
- In a distributed system, other issues must be taken into account:
  - The cost of data transmission over the network.
  - The potential gain in performance from having several sites process parts of the query in parallel.

Query Transformation

- Translating algebraic queries to queries on fragments.
  - It must be possible to construct relation *r* from its fragments
  - Replace relation *r* by the expression to construct relation *r* from its fragments
- Site selection for query processing.

Example Query

- Consider the horizontal fragmentation of the *account* relation into

$$account_1 = \sigma_{branch\text{-}name = \text{"Hillside"}}(account)$$
$$account_2 = \sigma_{branch\text{-}name = \text{"Valleyview"}}(account)$$

- The query $\sigma_{branch\text{-}name = \text{"Hillside"}}(account)$ becomes

$$\sigma_{branch\text{-}name = \text{"Hillside"}}(account_1 \cup account_2)$$

  which is optimized into

$$\sigma_{branch\text{-}name = \text{"Hillside"}}(account_1) \cup$$
$$\sigma_{branch\text{-}name = \text{"Hillside"}}(account_2)$$

Example Query (Cont.)

- Since $account_1$ has only tuples pertaining to the Hillside branch, we can eliminate the selection operation.
- Apply the definition of $account_2$ to obtain

$$\sigma_{branch\text{-}name = \text{“Hillside”}} (\sigma_{branch\text{-}name = \text{“Valleyview”}} (account))$$

- This expression is the empty set regardless of the contents of the *account* relation.
- Final strategy is for the Hillside site to return $account_1$ as the result of the query.

Simple Join Processing

Consider the following relational algebra expression in which the three relations are neither replicated nor fragmented

$$account \bowtie depositor \bowtie branch$$

- *account* is stored at site S_1
- *depositor* at S_2
- *branch* at S_3
- For a query issued at site S_I, the system needs to produce the result at site S_I.

Possible Query Processing Strategies

- Ship copies of all three relations to site S_I and choose a strategy for processing the entire query locally at site S_I.
- Ship a copy of the *account* relation to site S_2 and compute $temp_1 = account \bowtie depositor$ at S_2. Ship $temp_1$ from S_2 to S_3, and compute $temp_2 = temp_1 \bowtie branch$ at S_3. Ship the result $temp_2$ to S_I.
- Devise similar strategies, exchanging the roles of S_1, S_2, S_3.
- Must consider following factors:
  - amount of data being shipped
  - cost of transmitting a data block between sites
  - relative processing speed at each site

Semijoin Strategy

- Let r_1 be a relation with schema R_1 stored at site S_1
  Let r_2 be a relation with schema R_2 stored at site S_2
- Evaluate the expression $r_1 \bowtie r_2$, and obtain the result at S_1.
  1. Compute $temp1 \leftarrow \Pi_{R_1 \cap R_2}(r_1)$ at S_1.
  2. Ship $temp1$ from S_1 to S_2.
  3. Compute $temp2 \leftarrow r_2 \bowtie temp1$ at S_2.
  4. Ship $temp2$ from S_2 to S_1.
  5. Compute $r_1 \bowtie temp2$ at S_1. This is the result of $r_1 \bowtie r_2$.

Formal Definition

- The semijoin of r_1 with r_2, is denoted by:

$$r_1 \rhd\!< \; r_2$$

  it is defined by:

$$\Pi_{R_1} (r_1 \; \bowtie \; r_2)$$

- Thus, $r_1 \rhd\!< \; r_2$ selects those tuples of r_1 that contributed to $r_1 \; \bowtie \; r_2$.
- In step 3 above, $temp2 = r_2 \rhd\!< \; r_1$.
- For joins of several relations, the above strategy can be extended to a series of semijoin steps.

Join Strategies that Exploit Parallelism

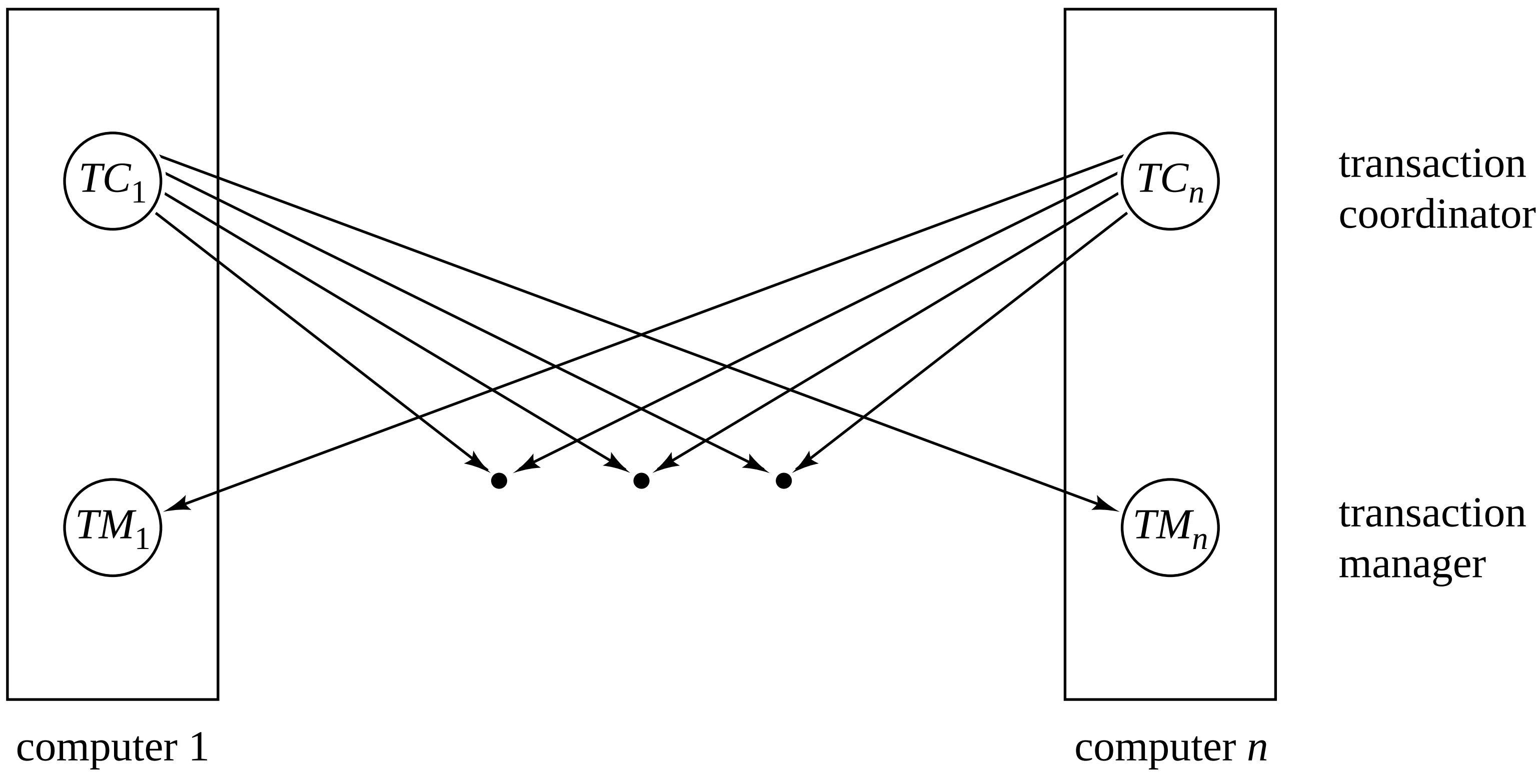
- Consider $r_1 \bowtie r_2 \bowtie r_3 \bowtie r_4$ where relation r_i is stored at site S_i. The result must be presented at site S_1.

- Pipelined-join strategy
  - r_1 is to S_2 and $r_1 \bowtie r_2$ is computed at S_2; simultaneously r_3 is shipped to S_4 and $r_3 \bowtie r_4$ is computed at S_4
  - S_2 ships tuples of $(r_1 \bowtie r_2)$ to S_1 as they are produced; S_4 ships tuples of $(r_3 \bowtie r_4)$ to S_1
  - Once tuples of $(r_1 \bowtie r_2)$ and $(r_3 \bowtie r_4)$ arrive at S_1, $(r_1 \bowtie r_2) \bowtie (r_3 \bowtie r_4)$ is computed in parallel with the computation of $(r_1 \bowtie r_2)$ at S_2 and the computation of $(r_3 \bowtie r_4)$ at S_4.

Distributed Transaction Model

- Transactions may access data at several sites
- Each site has a *local transaction manager* responsible for:
  - Maintaining a log for recovery purposes.
  - Participating in coordinating the concurrent execution of the transactions executing at that site.
- Each site has a *transaction coordinator*, which is responsible for:
  - Starting the execution of transactions that originate at the site.
  - Distributing subtransactions to appropriate sites for execution.
  - Coordinating the termination of each transaction that originates at the site, which may result in the transaction being committed at all sites or aborted at all sites.

Transaction System Architecture



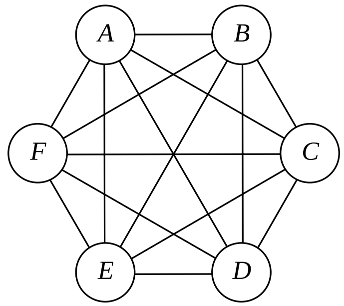
System Failure Modes

- Failures unique to distributed systems:
  - Failure of a site.
  - Loss of messages.
  - Failure of a communication link.
  - Network partition.
- The configurations of how sites are connected physically can be compared in terms of:
  - Installation cost.
  - Communication cost.
  - Availability.

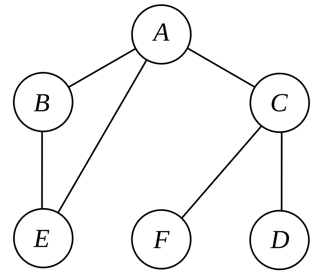
System Failure Modes (Cont.)

- Partially connected networks have direct links between some, but not all, pairs of sites.
  - Lower installation cost than fully connected network
  - Higher communication cost to *route* messages between two sites that are not directly connected

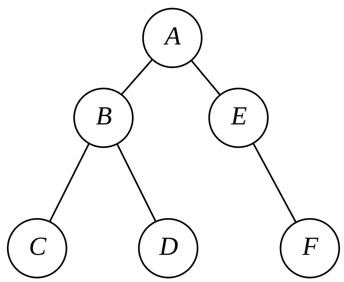
Network Topology



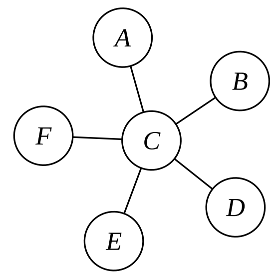
fully connected network



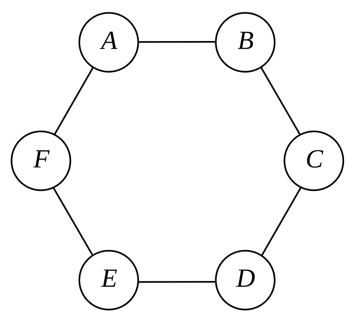
partially connected network



tree structured network



star network



ring network

Network Topology (Cont.)

- A *partitioned* system is split into two (or more) subsystems (*partitions*) that lack any connection.
- Tree-structured: low installation and communication costs; the failure of a single link can partition network
- Ring: At least two links must fail for partition to occur; communication cost is high
- Star:
  - the failure of a single link results in a network partition, but since one of the partitions has only a single site it can be treated as a single-site failure
  - low communication cost
  - failure of the central site results in every site in the system becoming disconnected

Robustness

- A robust system must:
  - Detect site or link failures
  - Reconfigure the system so that computation may continue.
  - Recover when a processor or link is repaired.
- Handling failure types:
  - Retransmit lost messages.
  - Unacknowledged retransmits indicate link failure; find alternative route for message.
  - Failure to find alternative route is a symptom of network partition.
- Network link failures and site failures are generally indistinguishable.

Procedure to Reconfigure System

- If replicated data is stored at the failed site, update the catalog so that queries do not reference the copy at the failed site.
- Transactions active at the failed site should be aborted.
- If the failed site is a central server for some subsystem, an *election* must be held to determine the new server.
- Reconfiguration scheme must work correctly in case of network partitioning; must avoid:
  - Electing two or more central servers in distinct partitions.
  - Updating replicated data item by more than one partition
- Represent recovery tasks as a series of transactions; concurrent control subsystem and transaction management subsystem may then be relied upon for proper reintegration.

Commit Protocols

- Commit protocols are used to ensure atomicity across sites
  - a transaction which executes at multiple sites must either be committed at all the sites, or aborted at all the sites.
  - not acceptable to have a transaction committed at one site and aborted at another
- The *two-phase commit* (2PC) protocol is widely used — will study this first
- The *three-phase commit* (3PC) protocol is more complicated and more expensive, but avoids some drawbacks of two-phase commit protocol.

Two-Phase Commit Protocol (2PC)

- Assumes *fail-stop* model – failed sites simply stop working, and do not cause any other harm, such as sending incorrect messages to other sites.
- Execution of the protocol is initiated by the coordinator after the last step of the transaction has been reached
- The protocol involves all the local sites at which the transaction executed
- Let T be a transaction initiated at site S_i, and let the transaction coordinator at S_i be C_i.

Phase 1: Obtaining a Decision

- Coordinator asks all participants to *prepare* to commit transaction T_i
  - C_i adds the record <**prepare** T> to the log and forces log to stable storage
  - sends **prepare** T message to all sites at which T executed
- Upon receiving message, transaction manager at site determines if it can commit the transaction
  - if not, add a record <**no** T> to the log and send **abort** T message to C_i
  - if the transaction can be committed, then:
    - ∗ add the record <**ready** T> to the log
    - ∗ force *all log records* for T to stable storage
    - ∗ send **ready** T message to C_i

Phase 2: Recording the Decision

- T can be committed if C_i received a **ready** T message from all the participating sites; otherwise T must be aborted
- Coordinator adds a decision record, $<$**commit** $T>$ or $<$**abort** $T>$, to the log and forces record onto stable storage
  Once that record reaches stable storage it is irrevocable (even if failures occur)
- Coordinator sends a message to each participant informing it of the decision (commit or abort)
- Participants take appropriate action locally

Handling of Failures – Site Failure

When site S_k recovers, it examines its log to determine the fate of transactions active at the time of the failure

- Log contains $<$**commit** $T>$ record: site executes **redo**(T).
- Log contains $<$**abort** $T>$ record: site executes **undo**(T).
- Log contains $<$**ready** $T>$ record: site must consult C_i to determine the fate of T.
  - if T committed, **redo**(T)
  - if T aborted, **undo**(T)
- The log contains no control records concerning T: implies that S_k failed before responding to the **prepare** T message from C_i
  - since the failure of S_k precludes the sending of such a response, C_i must abort T
  - S_k must execute **undo**(T)

Handling of Failures – Coordinator Failure

If coordinator fails while the commit protocol for T is executing, then participating sites must decide on T's fate:

- If an active site contains a $<$**commit** $T>$ record in its log, then T must be committed.
- If an active site contains an $<$**abort** $T>$ record in its log, then T must be aborted.
- If some active site does *not* contain a $<$**ready** $T>$ record in its log, then the failed coordinator C_i cannot have decided to commit T. Can therefore abort T.
- If none of the above cases holds, then all active sites must have a $<$**ready** $T>$ record in their logs, but no additional control records (such as $<$**abort** $T>$ or $<$**commit** $T>$). In this case active sites must wait for C_i to recover, to find decision.
- *Blocking* problem: active sites may have to wait for failed coordinator to recover.

Handling of Failures – Network Partition

- If the coordinator and all its participants remain in one partition, the failure has no effect on the commit protocol.
- If the coordinator and its participants belong to several partitions:
  - Sites that are not in the partition containing the coordinator think the coordinator has failed, and execute the protocol to deal with failure of the coordinator.
    - No harm results, but sites may still have to wait for decision from coordinator
  - The coordinator and the sites that are in the same partition as the coordinator think that the sites in the other partition have failed, and follow the usual commit protocol.
    - Again, no harm results

Recovery and Concurrency Control

- *In-doubt* transactions have a $<$**ready** $T>$, but neither a $<$**commit** $T>$, nor an $<$**abort** $T>$ log record.
- The recovering site must determine the commit–abort status of such transactions by contacting other sites; this can slow and potentially block recovery.
- Recovery algorithms can note lock information in the log.
  - Instead of $<$**ready** $T>$, write out $<$**ready** T, $L>$
    L = list of locks held by T when the log is written (read locks can be omitted).
  - For every in-doubt transaction T, all the locks noted in the $<$**ready** T, $L>$ log record are reacquired.
- After lock reacquisition, transaction processing can resume; the commit or rollback of in-doubt transactions is performed concurrently with the execution of new transactions.

Three Phase Commit (3PC)

- Assumptions:
  - No network partitioning
  - At any point, at least one site must be up.
  - At most K sites (participants as well as coordinator) can fail
- Phase 1: Obtaining Preliminary Decision: Identical to 2PC Phase 1.
  - Every site is ready to commit if instructed to do so
  - Under 2PC each site is obligated to wait for decision from coordinator
  - Under 3PC, knowledge of pre-commit decision can be used to commit despite coordinator failure

Phase 2: Recording the Preliminary Decision

- Coordinator adds a decision record (<**abort** *T*> or <**precommit** *T*>) in its log and forces record to stable storage
- Coordinator sends a message to each participant informing it of the decision
- Participant records decision in its log
  - If abort decision reached then participant aborts locally
  - If pre-commit decision reached then participant replies with <**acknowledge** *T*>

Phase 3: Recording Decision in the Database

Executed only if decision in phase 2 was to precommit

- Coordinator collects acknowledgments. It sends <**commit** *T*> message to the participants as soon as it receives *K* acknowledgments.
- Coordinator adds the record <**commit** *T*> in its log and forces record to stable storage.
- Coordinator sends a message to each participant to <**commit** *T*>.
- Participants take appropriate action locally

Handling Site Failure

Site Failure. Upon recovery, a participating site examines its log and does the following:

- Log contains <**commit** T> record: site executes **redo**(T).
- Log contains <**abort** T> record: site executes **undo**(T).
- Log contains <**ready** T> record, but no <**abort** T> or <**precommit** T> record: site consults C_i to determine the fate of T.
  - if C_i says T aborted, site executes **undo**(T) (and writes <**abort** T> record)
  - if C_i says T committed, site executes **redo**(T) (and writes <**commit** T> record)
  - if C_i says T precommitted, site resumes the protocol from receipt of **precommit** T message (thus recording <**precommit** T> in the log, and sending **acknowledge** T message sent to coordinator)

Handling Site Failure (Cont.)

- Log contains <**precommit** *T*> record, but no <**abort** *T*> or <**commit** *T*> record: site consults C_i to determine the fate of transaction *T*.
  - if C_i says *T* aborted, site executes **undo**(*T*)
  - if C_i says *T* committed, site executes **redo**(*T*)
  - if C_i says *T* still in precommit state, site resumes protocol at this point
- Log contains no <**ready** *T*> record for a transaction *T*: site executes **undo**(*T*) and writes <**abort** *T*> record.

Coordinator-Failure Protocol

1. The active participating sites select a new coordinator, C_{new}
2. C_{new} requests local status of T from each participating site
3. Each participating site, including C_{new}, determines the local status of T:
   - **Committed**. The log contains a $<$**commit** $T>$ record.
   - **Aborted**. The log contains an $<$**abort** $T>$ record.
   - **Ready**. The log contains a $<$**ready** $T>$ record but no $<$**abort** $T>$ or $<$**precommit** $T>$ record.
   - **Precommitted**. The log contains a $<$**precommit** $T>$ record but no $<$**abort** $T>$ or $<$**commit** $T>$ record.
   - **Not ready**. The log contains neither a $<$**ready** $T>$ nor an $<$**abort** $T>$ record.

   A site that failed and recovered must ignore any **precommit** record in its log when determining its status.
4. Each participating site sends its local status to C_{new}.

Coordinator Failure Protocol (Cont.)

5. C_{new} decides either to commit or abort T, or to restart the three-phase commit protocol:
   - Commit state for any one participant $\Rightarrow$ commit
   - Abort state for any one participant $\Rightarrow$ abort
   - Precommit state for any one participant and above 2 cases do not hold $\Rightarrow$

     A precommit message is sent to those participants in the uncertain state. Protocol is resumed from that point.
   - Uncertain state at all live participants $\Rightarrow$ abort

     Since at least $n - k$ sites are up, the fact that all participants are in an uncertain state means that the coordinator has not sent a <**commit** *T*> message, implying that no site has committed T.

Coordinator Selection

- Backup coordinators
  - site which maintains enough information locally to assume the role of coordinator if the actual coordinator fails
  - executes the same algorithms and maintains the same internal state information as the actual coordinator
  - allows fast recovery from coordinator failure, but involves overhead during normal processing
- Election algorithms
  - used to elect a new coordinator in case of failures
  - Example: Bully Algorithm—applicable to systems where every site can send a message to every other site

Bully Algorithm

- If site S_i sends a request that is not answered by the coordinator within a time interval T, assume that the coordinator has failed; S_i tries to elect itself as the new coordinator.
- S_i sends an election message to every site with a higher identification number, S_i then waits for any of these processes to answer within T.
- If no response within T, assume that all sites with numbers greater than i have failed; S_i elects itself the new coordinator.
- If answer is received, S_i begins time interval T', waiting to receive a message that a site with a higher identification number has been elected.

Bully Algorithm (Cont.)

- If no message is sent within T', assume the site with a higher number has failed; S_i restarts the algorithm.
- After a failed site recovers, it immediately begins execution of the same algorithm.
- If there are no active sites with higher numbers, the recovered site forces all processes with lower numbers to let it become the coordinator site, even if there is a currently active coordinator with a lower number.

Concurrency Control

- Modify concurrency control schemes for use in distributed environment.
- We assume that each site participates in the execution of a commit protocol to ensure global transaction atomicity.

Single-Lock-Manager Approach

- System maintains a *single* lock manager that resides in a *single* chosen site, say S_i.
- When a transaction needs to lock a data item, it sends a lock request to S_i and lock manager determines whether the lock can be granted immediately
  - If yes, lock manager sends a message to the site which initiated the request
  - If no, request is delayed until it can be granted, at which time a message is sent to the initiating site

Single-Lock-Manager Approach (Cont.)

- The transaction can read the data item from *any* one of the sites at which a replica of the data item resides.
- In the case of a write, all the sites where a replica of the data item resides must be involved in the writing.
- Advantages of scheme:
  - Simple implementation
  - Simple deadlock handling
- Disadvantages of scheme are:
  - Bottleneck: lock manager site becomes a bottleneck
  - Vulnerability: system is vulnerable to lock manager site failure

Majority Protocol

- Local lock manager at each site administers lock and unlock requests for data items stored at that site.
- When a transaction wishes to lock an unreplicated data item Q residing at site S_i, a message is sent to S_i's lock manager.
- If Q is locked in an incompatible mode, then the request is delayed until it can be granted.
- When the lock request can be granted, the lock manager sends a message back to the initiator indicating that the lock request has been granted.
- Advantage of simple implementation, however, since lock and unlock requests are no longer made at a single site, deadlock handling is more complex.

Majority Protocol (Cont.)

- In case of replicated data, majority protocol is more complicated to implement than the previous schemes
  - If Q is replicated at n sites, then a lock request message must be sent to more than half of the n sites in which Q is stored.
  - The transaction does not operate on Q until it has obtained a lock on a majority of the replicas of Q.
  - When writing the data item, transaction performs writes on all replicas.
- Requires $2(n/2 + 1)$ messages for handling lock requests, and $(n/2 + 1)$ messages for handling unlock requests.
- Potential for deadlock even with single item — e.g., each of 3 transactions may have locks on 1/3rd of the replicas of a data item

Biased Protocol

- Local lock manager at each site as in majority protocol, however, requests for shared locks are handled differently than requests for exclusive locks.
  - **Shared locks**. When a transaction needs to lock data item Q, it simply requests a lock on Q from the lock manager at one site containing a replica of Q.
  - **Exclusive locks**. When a transaction needs to lock data item Q, it requests a lock on Q from the lock manager at all sites containing a replica of Q.
- Advantage — imposes less overhead on **read** operations.
- Disadvantage — additional overhead on writes and complexity in handling deadlock.

Primary Copy

- Choose one replica to be the primary copy, which must reside in precisely one site (e.g., *primary site of Q*).
- When a transaction needs to lock a data item Q, it requests a lock at the primary site of Q.
- Concurrency control for replicated data handled similarly to unreplicated data—simple implementation.
- If the primary site of Q fails, Q is inaccessible even though other sites containing a replica may be accessible.

Timestamping

- Each site generates a unique local timestamp using either a logical counter or the local clock.
- Global unique timestamp is obtained by concatenating the unique local timestamp with the unique site identifier.

locally-unique timestamp	globally-unique site identifier

Timestamping (Cont.)

- A site with a slow clock will assign smaller timestamps $\rightarrow$ "disadvantages" transactions
- Define within each site S_i a *logical clock* (LC_i), which generates the unique local timestamp
- Require that S_i advance its logical clock whenever a transaction T_i with timestamp $<x,y>$ visits that site and x is greater than the current value of LC_i.
- In this case, site S_i advances its logical clock to the value $x + 1$.

Deadlock Handling

Consider the following two transactions and history:

T_1:	write(X)	T_2:	write(Y)
	write(Y)		write(X)

T_1	T_2
X-lock on X	
write(X)	
	X-lock on Y
	write(Y)
	wait for X-lock on X
wait for X-lock on Y	

deadlock

Centralized Approach

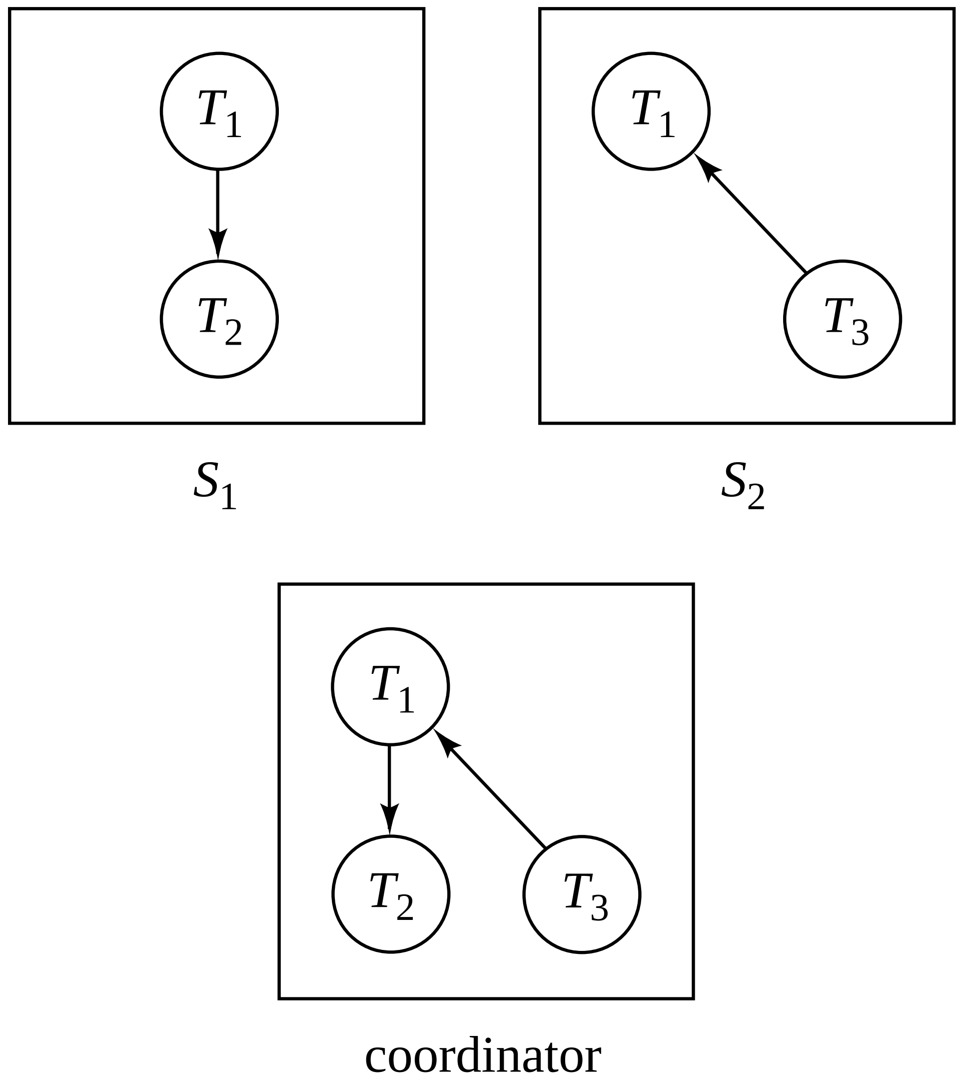
- A global wait-for graph is constructed and maintained in a *single* site: the deadlock-detection coordinator.
  - *Real* graph: Real, but unknown, state of the system.
  - *Constructed* graph: Approximation generated by the controller during the execution of its algorithm.
- The global wait-for graph can be constructed when:
  - a new edge is inserted in or removed from one of the local wait-for graphs.
  - a number of changes have occurred in a local wait-for graph.
  - the coordinator needs to invoke cycle-detection.
- If the coordinator finds a cycle, it selects a victim and notifies all sites. The sites roll back the victim transaction.

Centralized Approach (Cont.)

- Unnecessary rollbacks may result when a *deadlock* has indeed occurred and a victim has been picked, and meanwhile one of the transactions was aborted for reasons unrelated to the deadlock.

- Unnecessary rollbacks can result from *false cycles* in the global wait-for graph; however, likelihood of false cycles is low.

False Cycles



False Cycles (Cont.)

- Suppose that starting from state shown in figure,

  1. T_2 releases resources at S_1 (resulting in a message *remove* $T_1 \rightarrow T_2$ message from the Transaction Manager at site S_1 to the coordinator), and then
  2. T_2 requests a resource held by T_3, at site S_2 (resulting in a message **insert** $T_2 \rightarrow T_3$ from S_2 to the coordinator)

- Suppose further that the **insert** message reaches before the **delete** message. The coordinator would then find a *false cycle*

$$T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T_1$$

- The false cycle above never existed in reality.
- False cycles cannot occur if two-phase locking is used

Fully Distributed Approach

- Each site has local wait-for graph; system combines information in these graphs to detect deadlock
- Local Wait-for Graphs

Site 1	Site 2	Site 3
$T_1 \rightarrow T_2 \rightarrow T_3$	$T_3 \rightarrow T_4 \rightarrow T_5$	$T_5 \rightarrow T_1$

- Global Wait-for Graph

$$T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T_4 \rightarrow T_5$$

Fully Distributed Approach (Cont.)

- System model: a transaction runs at a single site, and makes requests to other sites for accessing non-local data.
- Each site maintains its own local wait-for graph in the normal fashion: there is an edge $T_i \rightarrow T_j$ if T_i is waiting on a lock held by T_j (note: T_i and T_j may be non-local).
- Additionally, arc $T_i \rightarrow T_{\text{ex}}$ exists in the graph at site S_k if
  (a) T_i is executing at site S_k, and is waiting for a reply to a request made on another site, or
  (b) T_i is non-local to site S_k, and a lock has been granted to T_i at S_k.
- Similarly, arc $T_{\text{ex}} \rightarrow T_i$ exists in the graph at site S_k if
  (a) T_i is non-local to site S_k, and is waiting on a lock for data at site S_k, or
  (b) T_i is local to site S_k, and has accessed data from an external site.

Fully Distributed Approach (Cont.)

- Centralized Deadlock Detection — all graph edges sent to central deadlock detector
- Distributed Deadlock Detection — "path pushing" algorithm
- Path pushing initiated when a site detects a local cycle involving T_{ex}, which indicates *possibility* of a deadlock.
- Suppose cycle at site S_i is

$$T_{ex} \rightarrow T_i \rightarrow T_j \rightarrow \ldots \rightarrow T_n \rightarrow T_{ex}$$

  and T_n is waiting for some transaction at site S_j. Then S_i passes on information about the cycle to S_j
- Optimization: S_i passes on information only if $i > n$.
- S_j updates it graph with new information and if it finds a cycle it repeats above process.

Fully Distributed Approach: Example

Site 1

EX(3) $\rightarrow$ T_1 $\rightarrow$ T_2 $\rightarrow$ T_3 $\rightarrow$ EX(2)

Site 2

EX(1) $\rightarrow$ T_3 $\rightarrow$ T_4 $\rightarrow$ T_5 $\rightarrow$ EX(3)

Site 3

EX(2) $\rightarrow$ T_5 $\rightarrow$ T_1 $\rightarrow$ EX(1)

EX(i): Indicates T_{ex}, plus wait is on/by a transaction at Site i

Fully Distributed Approach Example (Cont.)

- Site passes wait-for information along path in graph:
  - Let EX(*j*) $\rightarrow$ T_i $\rightarrow$ ... T_n $\rightarrow$ EX(*k*) be a path in the local wait-for graph at Site *m*
  - Site *m* “pushes” the path information to site *k* if $i > n$
- Example:
  - Site 1 does not pass information : $1 < 3$
  - Site 2 does not pass information : $3 < 5$
  - Site 3 passes (T_5, T_1) to Site 1 because:
    * $5 > 1$
    * T_1 is waiting for a data item at site 1

Fully Distributed Approach (Cont.)

- After the path EX(2) $\rightarrow T_5 \rightarrow T_1 \rightarrow$ EX(1) has been pushed to Site 1 we have:

Site 1
EX(2) $\rightarrow T_5 \rightarrow T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow$ EX(2)

Site 2
EX(1) $\rightarrow T_3 \rightarrow T_4 \rightarrow T_5 \rightarrow$ EX(3)

Site 3
EX(2) $\rightarrow T_5 \rightarrow T_1 \rightarrow$ EX(1)

Fully Distributed Approach (Cont.)

- After the push, only Site 1 has new edges. Site 1 passes (T_5, T_1, T_2, T_3) to site 2 since $5 > 3$ and T_3 is waiting for a data item at site 2
- The new state of the local wait-for graph:

Site 1

$EX(2) \rightarrow T_5 \rightarrow T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow EX(2)$

Site 2

$T_5 \rightarrow T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T_4$

Deadlock Detected

Site 3

$EX(2) \rightarrow T_5 \rightarrow T_1 \rightarrow EX(1)$

Multidatabase Systems

- Software layer on top of existing database systems required to manipulate information in heterogeneous database
- Data models may differ (hierarchical, relational, etc.)
- Transaction commit protocols may be incompatible
- Concurrency control may be based on different techniques (locking, timestamping, etc.)
- System-level details almost certainly are totally incompatible

Advantages

- Preservation of investment in existing
  - hardware
  - systems software
  - applications
- Local autonomy and administrative control
- Allows use of special-purpose DBMSs
- Step towards a unified homogeneous DBMS

Unified View of Data

- Agreement on a common data model
- Agreement on a common conceptual schema
- Agreement on a single representation of shared data (that may be stored in multiple DBMSs)
- Agreement on units of measure
- Willingness to accept limited function in global transactions

Transaction Management

- Local transactions are executed by each local DBMS, outside of the MDBS system control.
- Global transactions are executed under MDBS control.
- Local autonomy—local DBMSs cannot communicate directly to synchronize global transaction execution and the MDBS has no control over local transaction execution.
  - local concurrency control scheme needed to ensure that DBMS's schedule is serializable
  - in case of locking, DBMS must be able to guard against local deadlocks
  - need additional mechanisms to ensure global serializability

Two-Level Serializability

- DBMS ensures local serializability among its local transactions, including those that are part of a global transaction.
- The MDBS ensures serializability among global transactions alone — *ignoring the orderings induced by local transactions*.
- 2LSR does not ensure global serializability, however, it can fulfill requirements for *strong correctness*:
  1. Preserve consistency as specified by a given set of constraints
  2. Guarantee that the set of data items read by each transaction is consistent
- *Global-read protocol*: Global transactions can read, but not update, local data items; local transactions do not have access to global data. There are no consistency constraints between local and global data items.

Two-Level Serializability (Cont.)

- *Local-read protocol*: Local transactions have read access to global data; disallows all access to local data by global transactions.
  - A transaction has a *value dependency* if the value that it writes to a data item at one site depends on a value that it read for a data item on another site.
  - For strong correctness: No transaction may have a value dependency.
- *Global-read–write/local-read protocol*: Local transactions have read access to global data; global transactions may read and write all data;
  - No consistency constraints between local and global data items.
  - No transaction may have a value dependency.

Global Serializability

- Global 2PL—each local site uses a strict 2PL (locks are released at the end); locks set as a result of a global transaction are released only when that transaction reaches the end.
- Even if no information is available concerning the structure of the various local concurrency control schemes, a very restrictive protocol that ensures serializability is available.
  - Transaction-graph: a graph with vertices being global transaction names and site names.
    An undirected edge (T_i, S_k) exists if T_i is active at site S_k.
  - Global serializability is assured if transaction-graph contains no undirected cycles.

Ensuring Global Serializability

- Each site S_i has a special data item, called a *ticket*
- Every transaction T_j that runs at site S_k writes to the ticket at site S_i
- Ensures global transactions are serialized at each site, regardless of local concurrency control method, so long as the method guarantees local serializability
- Global transaction manager decides serial ordering of global transactions by controlling order in which tickets are accessed
- However, above protocol results in low concurrency between global transactions