Parallel and Distributed Transaction Processing

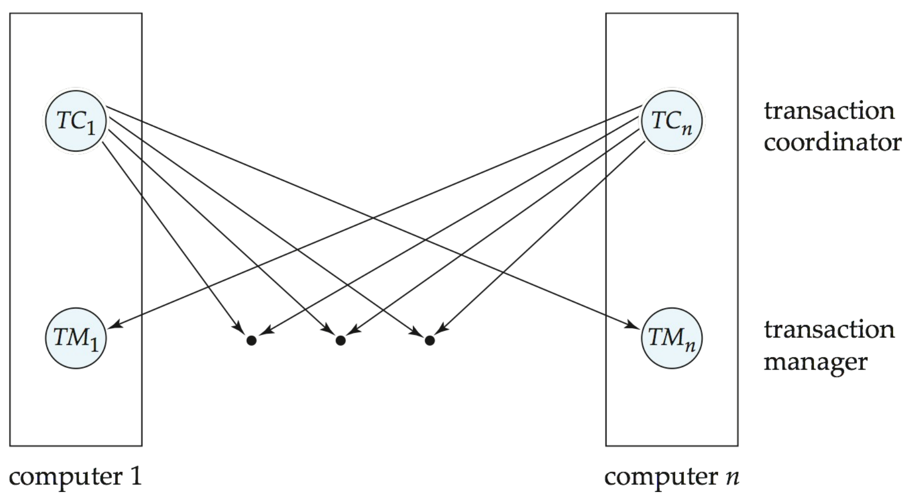
- **Distributed Transactions**
- Commit Protocol
- Concurrency Control in Distributed Databases
- Deadlock Handling

Distributed Transactions

- Transaction may access data at several sites.
  - **Local transactions**
    - Access/update data at only one database
  - **Global transactions**
    - Access/update data at more than one database
- Key issue: how to ensure ACID properties for transactions in a system with global transactions spanning multiple database
- Each site has a local **transaction manager** who manages the execution of those transactions that access data stored in a local site:
  - Maintaining a log for recovery purposes.
  - Coordinating the execution and commit/abort of the transactions executing at that site.

Distributed Transactions (cont.)

- Each site has a **transaction coordinator** who coordinates the execution of the various transactions (both local and global) initiated at that site:
  - Starting the execution of transactions that originate at the site.
  - Distributing *subtransactions* at appropriate sites for execution.
  - Coordinating the termination of each transaction that originates at the site
    - transaction must be committed at all sites or aborted at all sites(to ensure ***atomicity***).



Parallel and Distributed Transaction Processing

- Distributed Transactions
- **Commit Protocol**
- Concurrency Control in Distributed Databases
- Deadlock Handling

Commit Protocol

- The transaction coordinator must execute a **commit protocol** 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.

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 transaction coordinator *after* the last step of the transaction has been reached.
  - All the sites at which the transaction has executed inform the transaction coordinator that it has completed.
- The protocol involves all the local sites (participants) 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

- C_i asks all participants to ***prepare*** to commit transaction T.
  - C_i adds the record <**prepare** T> to the log and forces log to stable storage.
  - C_i sends **prepare** T messages to all sites at which T executed.

- Upon receiving message, transaction manager at that site determines if it can commit the transaction.

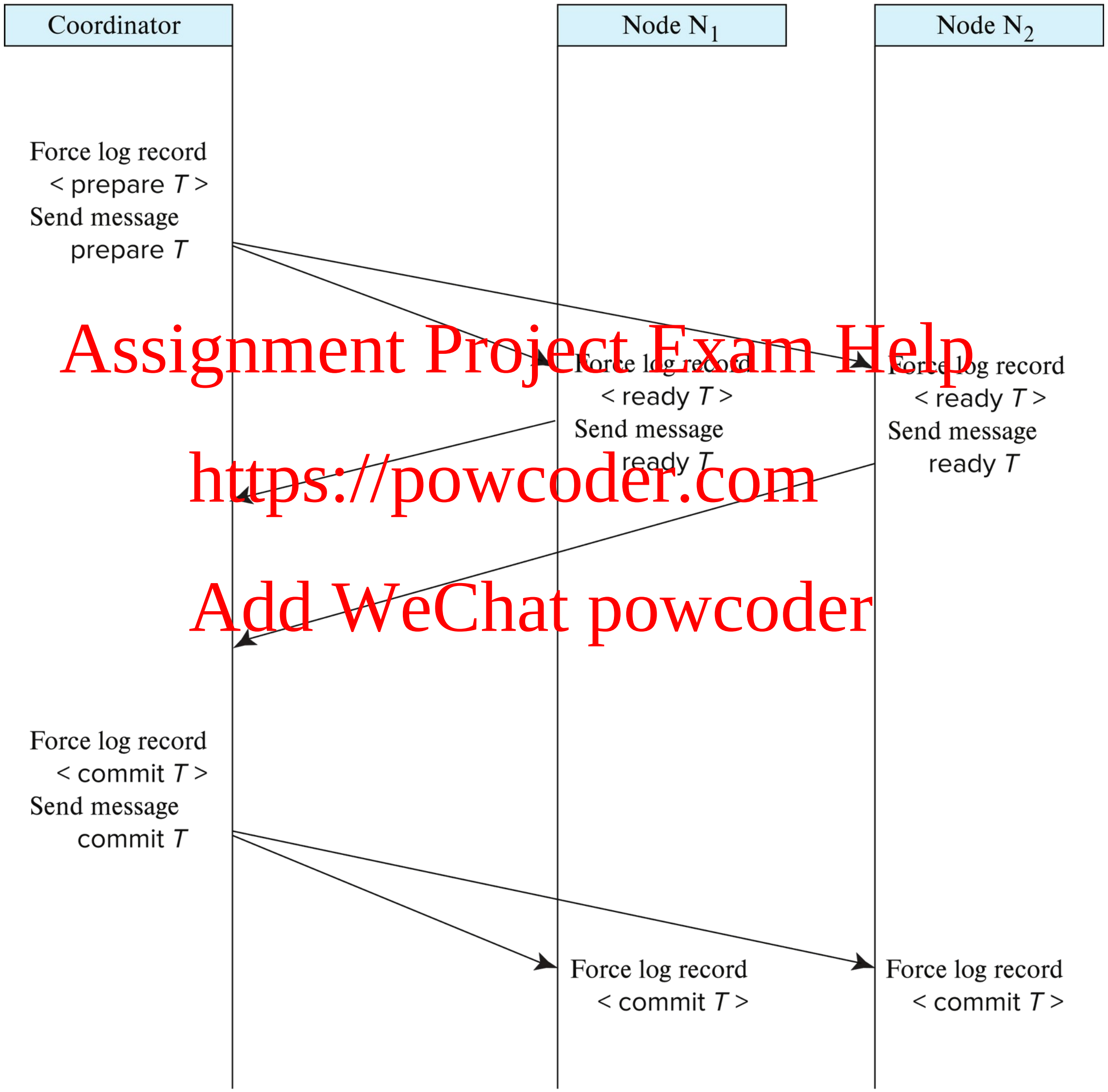
  - if no,

    - adds a record <**no** T> to the log
    - sends **abort** T message to C_i
  - if yes,
    - adds the record <**ready** T> to the log
    - forces the log (with all log records for T) to stable storage
      - to keep its promise, even if the site crashes after sending **ready** T message
    - sends **ready** T message to C_i

Phase 2: Recording the Decision

- *T* can be committed when C_i received a **ready** *T* message from ***all*** participants within a pre-specified interval of time, otherwise, *T* must be aborted.
- C_i adds a decision record, <**commit** *T*> or <**abort** *T*>, to the log and forces record onto stable storage. Once the record is forced onto stable storage, it is irrevocable (even if failures occur).
- C_i sends a message to each participant informing it of the decision (commit or abort).
- Participants record the message in the log.

Two-Phase Commit Protocol



System Failure Modes

- Failures to centralized systems:
  - software errors, hardware errors, disk crashes
- Failures unique to distributed systems:
  - **Failure of a site**.
  - Loss or corruption of messages
    - Handled by network transmission control protocols such as TCP-IP.
  - Failure of a communication link
    - Handled by network protocols, by routing messages via alternative links.
  - **Network partition**
    - A network is said to be **partitioned** when it has been split into two or more subsystems that lack any connection between them.
- Network partitioning and site failures are generally indistinguishable.

Handling of Failures - Site Failure

When site S_k fails and then 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 the coordinator C_i or other sites to determine the fate of *T*.
  - If *T* committed, **redo** (*T*)
  - If *T* aborted, **undo** (*T*)
- 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 C_i fails while the commit protocol for *T* is executing, then participants must decide on *T*'s fate:

1. If an active site contains a <**commit** *T*> record in its log, then *T* must be committed.

2. If an active site contains an <**abort** *T*> record in its log, then *T* must be aborted.

3. If some active participant does not contain a <**ready** *T*> record in its log, then the failed C_i cannot have decided to commit *T*. Can therefore abort *T*.

4. 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*> of <**commit** *T*>). In this case active sites must wait for C_i to recover, to find decision.

Handling of Failures-Coordinator Failure (Cont.)

- **Blocking problem**: *T* is blocked pending the recovery of site C_i.
  - *T* may hold system resources and other transactions may be forced to wait for the blocked *T*.
  - Data items may be unavailable not only on the failed site (C_i), but on active sites as well.

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, however, normal transaction processing cannot begin until all in-doubt transactions have been committed or rolled back.
- The recovering site must determine the commit-abort status of such transactions by contacting other sites; this can slow and potentially block recovery.
- Solution: 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).
  - After performing local recovery, 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.

Parallel and Distributed Transaction Processing

- Distributed Transactions
- Commit Protocol
- **Concurrency Control in Distributed Databases**
- Deadlock Handling

Concurrency Control

- Modify centralized concurrency control schemes for use in distributed environment.
  - Consider locking protocols here.
- Main issue: how can lock conflicts be detected in a distributed database with replicated data?
- We assume that each site participates in the execution of a commit protocol to ensure global transaction atomicity.
- We assume all replicas of any item are updated.

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 the lock manager determines whether the lock can be granted immediately.

  - If yes, the 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.
- Writes must be performed on ***all*** replicas of a data item
- Advantages of scheme:
  - Simple implementation
  - Simple deadlock handling
    - Centralized deadlock-handling algorithms can be applied directly.
- Disadvantages of scheme:
  - Bottleneck: lock manager site becomes a bottleneck.
  - Vulnerability: system is vulnerable to lock manager site failure.

Distributed Lock Manager

- In this approach, functionality of locking is implemented by lock manager at ***each*** site.
  - Lock managers control access to local data items.
  - Locking is performed separately on each site accessed by transaction.
- Advantage:
  - Work is distributed and can be made robust to failures.
- Disadvantage:
  - Possibility of a global deadlock without local deadlock at any single site.
  - Lock managers cooperate for deadlock detection (to be discussed).

Distributed Lock Manager (cont.)

- If the data item is not replicated, like single-lock-manager approach.
- If the data item is replicated, several variants of this approach
  - Primary copy
  - Majority protocol
  - Biased protocol
  - Quorum consensus

Primary Copy

- Choose one replica as **primary copy** for each data item.
  - Node containing primary replica is called **primary node**.
  - Concurrency control decisions made at the primary copy only.
- When a transaction needs to lock a data item *Q*, it requests a lock at the primary node of *Q*.
- Benefit
  - Simple implementation: concurrency control for replicated data to be handled like that for unreplicated data.
- Drawback
  - primary copy failure results in loss of lock information and non-availability of data item, even if other replicas are available.

Majority Protocol

- If data item *Q* is replicated in *n* different nodes, then a lock-request message must be sent to **more than one-half** of the *n* nodes in which Q is stored.
- Lock is successfully acquired on the data item only if lock obtained at a majority of replicas.

- Benefit
  - Resilient to node failures, processing can continue as long as at least a majority of replicas are accessible.

- Drawback

  - Higher cost due to multiple messages: requires 2(n/2 + 1) messages for handling lock requests, and (n/2 + 1) messages for handling unlock requests.
  - Possibility of deadlock even when locking single item, e.g., each of 3 transactions may have locks on 1/3rd of the replicas of a data.

Biased Protocol

- The difference from the majority protocol is that requests for shared locks are given more favorable treatment 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*** node that contains a replica of *Q*.
- **Exclusive locks**. When 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.
- Disadvantages
  - Additional overhead on writes.
  - Potential for deadlock (same as the majority protocol).

Quorum Consensus Protocol

- A generalization of both majority and biased protocols.
- Each node is assigned a weight.
  - Let S be the total weight of all nodes at which the item resides.
- Choose two values **read quorum** Q_r and **write quorum** Q_w for each item such that $Q_r + Q_w > S$ **and** $2 * Q_w > S$.
- To execute a read operation, enough replicas must be locked that their total weight is at least Q_r.
- To execute a write operation, enough replicas must be locked so that their total weight is at least Q_w.
- Benefits: can choose Q_r and Q_w to tune relative overheads on reads and writes
  - With a small read quorum, reads need to obtain fewer locks.
  - If higher weights are given to some (more fail-safe) nodes, fewer nodes need to be accessed for acquiring locks.

Parallel and Distributed Transaction Processing

- Distributed Transactions
- Commit Protocol
- Concurrency Control in Distributed Databases
- **Deadlock Handling**

Deadlock Handling

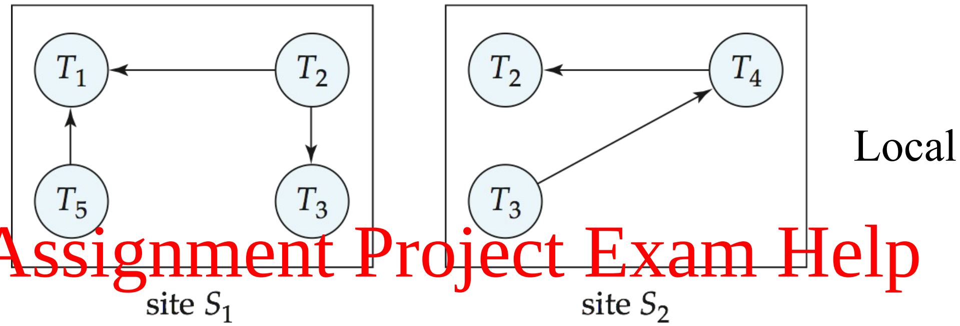
- *Reminder*: Deadlocks can be detected by the **wait-for graph**.
- Common techniques for maintaining the wait-for graph in a distributed system require that each site keeps a local wait-for graph.

- The nodes correspond to all transactions (local or nonlocal) that are currently either holding or requesting any of the items local to that site.

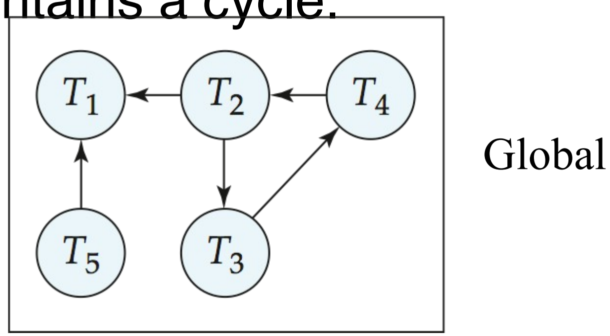
  - When a transaction T_i on site S_1 needs a resource in S_2, it sends a request message to S_2.
  - If the resource is held by T_j, the system inserts an edge $T_i \rightarrow T_j$ in the local wait-for graph of S_2.

Deadlock Handling (cont.)

- Example: T_2 and T_3 below have requested items at both sites.



- If any local wait-for graph has a cycle, deadlock has occurred.
- However, no cycles in any of the local wait-for cycles does not mean that there are no deadlocks.
- Example: Each wait-for graph of S_1 and S_2 above is acyclic, a deadlock exists in the system because the union of the local wait-for graphs contains a cycle.



Centralized Approach

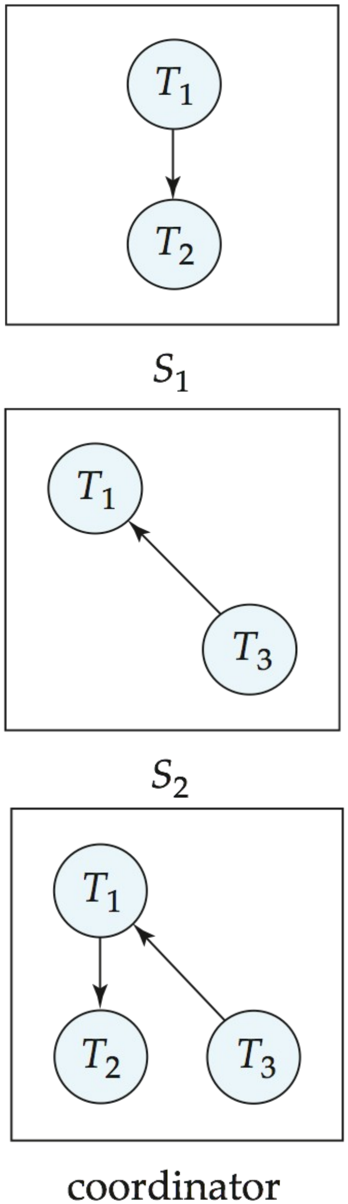
- A ***global wait-for graph*** is constructed and maintained in a *single* site: the ***deadlock-detection coordinator***.
  - *Real graph*: Real but unknown state of the system at any instance in time (due to communication delay).
  - *Constructed graph*: Approximation generated by the coordinator 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.

False Cycles

- Suppose that starting from the state shown in figure.
  1. T_2 releases resources at site S_1
     - resulting in a message remove $T_1 \rightarrow T_2$ from the Transaction Manager at S_1 to the coordinator
  2. Then T_2 requests a resource held by T_3 at 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
  - this can happen due to network delays
- 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.



Unnecessary Rollbacks

- Unnecessary rollbacks can result from false cycles in the global wait-for graph; however, likelihood of false cycles is low.
- Unnecessary rollbacks may also result when deadlock has indeed occurred and a victim has been picked, and meanwhile one of the transactions was aborted for reasons unrelated to the deadlock.
  - Example: Site S_1 decides to abort T_2.

    - At the same time, the coordinator has discovered a cycle in the global wait-for graph and has picked T_3 as a victim.

    - Both T_2 and T_3 are now rolled back, although only T_2 needed to be rolled back.