

# Distributed Databases

Part 2

Advanced Topics in Database Management (INFSCI 1022)

Distributed Databases (TELCOM 2326)

**Textbook: Database System Concepts - 5<sup>th</sup> Edition, 2005**

**Vladimir Zadorozhny, GIST, University of Pittsburgh**

**Database System Concepts, 5th Ed.**

## 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.
- We assume all replicas of any item are updated
  - Will see how to relax this in case of site failures later

## 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.
- Writes must be performed on all replicas of a data item
- 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.

## Distributed Lock Manager

- In this approach, functionality of locking is implemented by lock managers at each site
  - Lock managers control access to local data items
    - ▶ But special protocols may be used for replicas
- Advantage: work is distributed and can be made robust to failures
- Disadvantage: deadlock detection is more complicated
  - Lock managers cooperate for deadlock detection
    - ▶ More on this later
- Several variants of this approach
  - Primary copy
  - Majority protocol
  - Biased protocol
  - Quorum consensus

## Primary Copy

- Choose one replica of data item to be the **primary copy**.
  - Site containing the replica is called the **primary site** for that data item
  - Different data items can have different primary sites
- When a transaction needs to lock a data item  $Q$ , it requests a lock at the primary site of  $Q$ .
  - Implicitly gets lock on all replicas of the data item
- Benefit
  - Concurrency control for replicated data handled similarly to unreplicated data - simple implementation.
- Drawback
  - If the primary site of  $Q$  fails,  $Q$  is inaccessible even though other sites containing a replica may be accessible.

## 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.

## Majority Protocol (Cont.)

- In case of replicated data
  - 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.
- Benefit
  - Can be used even when some sites are unavailable
    - details on how handle writes in the presence of site failure later
- Drawback
  - 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.

## 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 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

## Quorum Consensus Protocol

- A generalization of both majority and biased protocols
- Each site is assigned a weight.
  - Let  $S$  be the total of all site weights
- Choose two values **read quorum**  $Q_r$  and **write quorum**  $Q_w$ 
  - Such that  $Q_r + Q_w > S$  and  $2 * Q_w > S$
  - Quorums can be chosen (and  $S$  computed) separately for each item
- Each read must lock enough replicas that the sum of the site weights is  $\geq Q_r$
- Each write must lock enough replicas that the sum of the site weights is  $\geq Q_w$
- For now we assume all replicas are written
  - Extensions to allow some sites to be unavailable described later

## Timestamp-Based Protocols

- Each transaction is issued a timestamp when it enters the system. If an old transaction  $T_i$  has time-stamp  $TS(T_i)$ , a new transaction  $T_j$  is assigned time-stamp  $TS(T_j)$  such that  $TS(T_i) < TS(T_j)$ .
- The protocol manages concurrent execution such that the time-stamps determine the serializability order.
- In order to assure such behavior, the protocol maintains for each data  $Q$  two timestamp values:
  - **W-timestamp**( $Q$ ) is the largest time-stamp of any transaction that executed **write**( $Q$ ) successfully.
  - **R-timestamp**( $Q$ ) is the largest time-stamp of any transaction that executed **read**( $Q$ ) successfully.

## Timestamp-Based Protocols (Cont.)

- The timestamp ordering protocol ensures that any conflicting **read** and **write** operations are executed in timestamp order.
- Suppose a transaction  $T_i$  issues a **read**( $Q$ )
  1. If  $TS(T_i) \leq \mathbf{W-timestamp}(Q)$ , then  $T_i$  needs to read a value of  $Q$  that was already overwritten. Hence, the **read** operation is rejected, and  $T_i$  is rolled back.
  2. If  $TS(T_i) \geq \mathbf{W-timestamp}(Q)$ , then the **read** operation is executed, and **R-timestamp**( $Q$ ) is set to the maximum of **R-timestamp**( $Q$ ) and  $TS(T_i)$ .

## Timestamp-Based Protocols (Cont.)

- Suppose that transaction  $T_i$  issues **write**( $Q$ ).
- If  $TS(T_i) < R\text{-timestamp}(Q)$ , then the value of  $Q$  that  $T_i$  is producing was needed previously, and the system assumed that that value would never be produced. Hence, the **write** operation is rejected, and  $T_i$  is rolled back.
- If  $TS(T_i) < W\text{-timestamp}(Q)$ , then  $T_i$  is attempting to write an obsolete value of  $Q$ . Hence, this **write** operation is rejected, and  $T_i$  is rolled back.
- Otherwise, the **write** operation is executed, and  $W\text{-timestamp}(Q)$  is set to  $TS(T_i)$ .

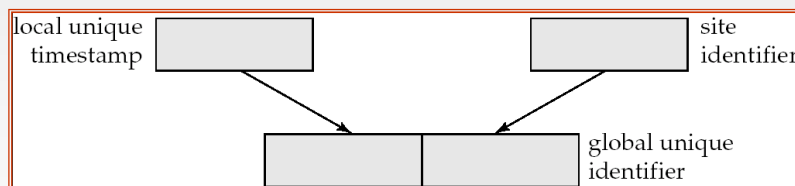
## Example Use of the Protocol

A partial schedule for several data items for transactions with timestamps 1, 2, 3, 4, 5

$T_1$	$T_2$	$T_3$	$T_4$	$T_5$
read( $Y$ )	read( $Y$ )			read( $X$ )
		write( $Y$ ) write( $Z$ )		read( $Z$ )
read( $X$ )	write( $X$ ) abort	write( $Z$ ) abort		write( $Y$ ) write( $Z$ )

## Timestamping

- Timestamp based concurrency-control protocols can be used in distributed systems
- Each transaction must be given a unique timestamp
- Main problem: how to generate a timestamp in a distributed fashion
  - 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 identifier.



## Timestamping (Cont.)

- A site with a slow clock will assign smaller timestamps
  - Still logically correct: serializability not affected
  - But: “disadvantages” transactions
- To fix this problem
  - 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 request is received from a transaction  $T_i$  with timestamp  $\langle x, y \rangle$  and  $x$  is greater than the current value of  $LC_i$ .
  - In this case, site  $S_i$  advances its logical clock to the value  $x + 1$ .

## Replication with Weak Consistency

- Many commercial databases support replication of data with weak degrees of consistency (i.e., without a guarantee of serializability)
- E.g.: **master-slave replication**: updates are performed at a single “master” site, and propagated to “slave” sites.
  - Propagation is not part of the update transaction: its is decoupled
    - May be immediately after transaction commits
    - May be periodic
  - Data may only be read at slave sites, not updated
    - No need to obtain locks at any remote site
  - Particularly useful for distributing information
    - E.g. from central office to branch-office
  - Also useful for running read-only queries offline from the main database

## Replication with Weak Consistency (Cont.)

- Replicas should see a **transaction-consistent snapshot** of the database
  - That is, a state of the database reflecting all effects of all transactions up to some point in the serialization order, and no effects of any later transactions.
- E.g. Oracle provides a **create snapshot** statement to create a snapshot of a relation or a set of relations at a remote site
  - snapshot refresh either by re-computation or by incremental update
  - Automatic refresh (continuous or periodic) or manual refresh

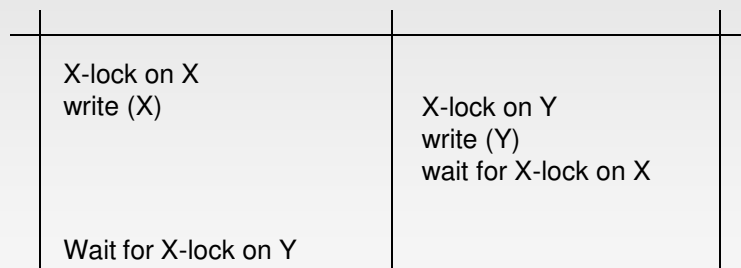
## Multimaster and Lazy Replication

- With multimaster replication (also called update-anywhere replication) updates are permitted at any replica, and are automatically propagated to all replicas
  - Basic model in distributed databases, where transactions are unaware of the details of replication, and database system propagates updates as part of the same transaction
    - Coupled with 2 phase commit
- Many systems support **lazy propagation** where updates are transmitted after transaction commits
  - Allows updates to occur even if some sites are disconnected from the network, but at the cost of consistency

## Deadlock Handling

Consider the following two transactions and history, with item X and transaction  $T_1$  at site 1, and item Y and transaction  $T_2$  at site 2:

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

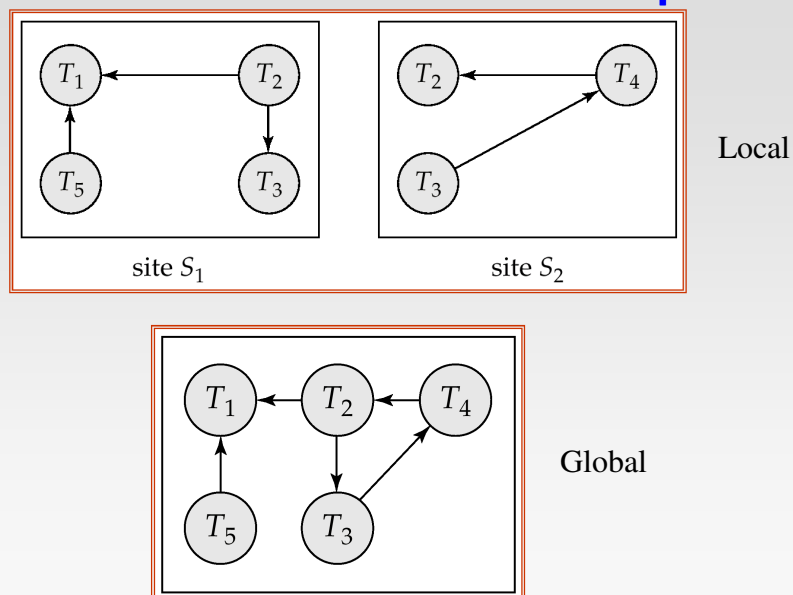


Result: deadlock which cannot be detected locally at either site

## Centralized Approach

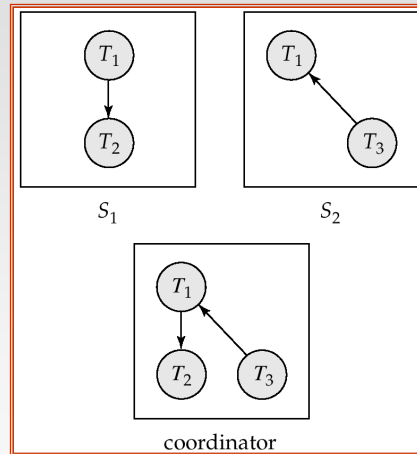
- A global wait-for graph is constructed and maintained in a *single* site; the deadlock-detection coordinator
  - *Real graph*: Real, but unknown, state of the system.
  - *Constructed graph*: Approximation generated by the controller during the execution of its algorithm .
- the global wait-for graph 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.

## Local and Global Wait-For Graphs



## Example Wait-For Graph for False Cycles

Initial state:



Database System Concepts - 5<sup>th</sup> Edition

V.Zadorozhny, INFSCI2711, TELCOM2326

## False Cycles (Cont.)

- Suppose that starting from the 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)
  2. And then  $T_2$  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
  - 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.
- False cycles cannot occur if two-phase locking is used.

Database System Concepts - 5<sup>th</sup> Edition

V.Zadorozhny, INFSCI2711, TELCOM2326

## Unnecessary Rollbacks

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

## Availability

- High availability: time for which system is not fully usable should be extremely low (e.g. 99.99% availability)
- Robustness: ability of system to function spite of failures of components
- Failures are more likely in large distributed systems
- To be robust, a distributed system must
  - Detect failures
  - Reconfigure the system so computation may continue
  - Recovery/reintegration when a site or link is repaired
- Failure detection: distinguishing link failure from site failure is hard
  - (partial) solution: have multiple links, multiple link failure is likely a site failure

## Reconfiguration

- Reconfiguration:
  - Abort all transactions that were active at a failed site
    - Making them wait could interfere with other transactions since they may hold locks on other sites
    - However, in case only some replicas of a data item failed, it may be possible to continue transactions that had accessed data at a failed site (more on this later)
  - If replicated data items were at failed site, update system catalog to remove them from the list of replicas.
    - This should be reversed when failed site recovers, but additional care needs to be taken to bring values up to date
  - If a failed site was a central server for some subsystem, an **election** must be held to determine the new server
    - E.g. name server, concurrency coordinator, global deadlock detector

## Reconfiguration (Cont.)

- Since network partition may not be distinguishable from site failure, the following situations must be avoided
  - Two or more central servers elected in distinct partitions
  - More than one partition updates a replicated data item
- Updates must be able to continue even if some sites are down
- Solution: majority based approach
  - Alternative of “read one write all available” is tantalizing but causes problems

## Majority-Based Approach

- The majority protocol for distributed concurrency control can be modified to work even if some sites are unavailable
  - Each replica of each item has a **version number** which is updated when the replica is updated, as outlined below
  - A lock request is sent to at least  $\frac{1}{2}$  the sites at which item replicas are stored and operation continues only when a lock is obtained on a majority of the sites
  - Read operations look at all replicas locked, and read the value from the replica with largest version number
    - ▶ May write this value and version number back to replicas with lower version numbers (no need to obtain locks on all replicas for this task)

## Majority-Based Approach

- Majority protocol (Cont.)
  - Write operations
    - ▶ find highest version number like reads, and set new version number to old highest version + 1
    - ▶ Writes are then performed on all locked replicas and version number on these replicas is set to new version number
  - Failures (network and site) cause no problems as long as
    - ▶ Sites at commit contain a majority of replicas of any updated data items
    - ▶ During reads a majority of replicas are available to find version numbers
    - ▶ Subject to above, 2 phase commit can be used to update replicas
  - Note: reads are guaranteed to see latest version of data item
  - Reintegration is trivial: nothing needs to be done
- Quorum consensus algorithm can be similarly extended

## Read One Write All (Available)

- Biased protocol is a special case of quorum consensus
  - Allows reads to read any one replica but updates require all replicas to be available at commit time (called **read one write all**)
- Read one write all available (ignoring failed sites) is attractive, but incorrect
  - If failed link may come back up, without a disconnected site ever being aware that it was disconnected
  - The site then has old values, and a read from that site would return an incorrect value
  - If site was aware of failure reintegration could have been performed, but no way to guarantee this
  - With network partitioning, sites in each partition may update same item concurrently
    - believing sites in other partitions have all failed

## Site Reintegration

- When failed site recovers, it must catch up with all updates that it missed while it was down
  - Problem: updates may be happening to items whose replica is stored at the site while the site is recovering
  - Solution 1: halt all updates on system while reintegrating a site
    - Unacceptable disruption
  - Solution 2: lock all replicas of all data items at the site, update to latest version, then release locks
    - Other solutions with better concurrency also available

## 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 fails executes 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,  $S_j$ , then waits for any of these processes to answer within  $T$ .
- If no response within  $T$ , assume that all sites with number 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.