

19: Distributed Coordination

Last Modified:
12/4/2002 9:23:18 PM

-1

Last Time

- We talked about the potential benefits of distributed systems
- We also talked about some of the reasons they can be so difficult to build
- Today we are going to tackle some of these problems!

-2

Recall

- Distributed systems
 - Components can fail (not fail-stop)
 - Network partitions can occur in which each portion of the distributed system thinks they are the only ones alive
 - Don't have a shared clock
 - Can't rely on hardware primitives like test-and-set for mutual exclusion
 - ...

-3

Distributed Coordination

- To tackle this complexity we are going to build distributed algorithms for:
 - Event Ordering
 - Mutual Exclusion
 - Atomicity
 - Deadlock Handling
 - Election Algorithms
 - Reaching Agreement

-4

Event Ordering

- Problem: distributed systems do not share a clock
 - Many coordination problems would be simplified if they did ("first one wins")
- Distributed systems do have some sense of time
 - Events in a single process happen in order
 - Messages between processes must be sent before they can be received
 - How helpful is this?

-5

Happens-before

- Define a *Happens-before* relation (denoted by \rightarrow).
 - 1) If A and B are events in the same process, and A was executed before B , then $A \rightarrow B$.
 - 2) If A is the event of sending a message by one process and B is the event of receiving that message by another process, then $A \rightarrow B$.
 - 3) If $A \rightarrow B$ and $B \rightarrow C$ then $A \rightarrow C$.

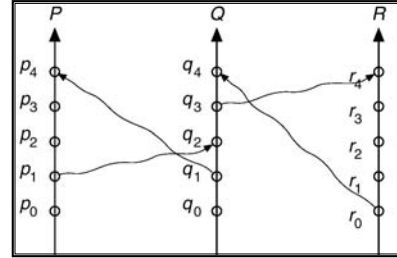
-6

Total ordering?

- Happens-before gives a partial ordering of events
- We still do not have a total ordering of events

-7

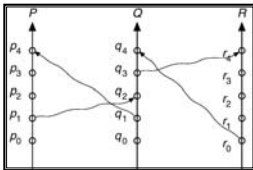
Partial Ordering



$P_i \rightarrow P_{i+1}; Q_i \rightarrow Q_{i+1}; R_i \rightarrow R_{i+1}$ $R_0 \rightarrow Q_4; Q_3 \rightarrow R_4; Q_1 \rightarrow P_4; P_1 \rightarrow Q_2$

-8

Total Ordering?



$P_0, P_1, Q_0, Q_1, Q_2, P_2, P_3, P_4, Q_3, R_0, Q_4, R_1, R_2, R_3, R_4$

$P_0, Q_0, Q_1, P_1, Q_2, P_2, P_3, P_4, Q_3, R_0, Q_4, R_1, R_2, R_3, R_4$

$P_0, Q_0, P_1, Q_1, Q_2, P_2, P_3, P_4, Q_3, R_0, Q_4, R_1, R_2, R_3, R_4$

-9

Timestamps

- Assume each process has a local logical clock that ticks once per event and that the processes are numbered
 - Clocks tick once per event (including message send)
 - When send a message, send your clock value
 - When receive a message, set your clock to MAX(your clock, timestamp of message + 1)
 - Thus sending comes before receiving
 - Only visibility into actions at other nodes happens during communication, communicate synchronizes the clocks
 - If the timestamps of two events A and B are the same, then use the process identity numbers to break ties.
- This gives a total ordering!

-10

Distributed Mutual Exclusion (DME)

- Problem: We can no longer rely on just an atomic test and set operation on a single machine to build mutual exclusion primitives
- Requirement
 - If P_i is executing in its critical section, then no other process P_j is executing in its critical section.

-11

Solution

- We present three algorithms to ensure the mutual exclusion execution of processes in their critical sections.
 - Centralized Distributed Mutual Exclusion (CDME)
 - Fully Distributed Mutual Exclusion (DDME)
 - Token passing

-12

CDME: Centralized Approach

- One of the processes in the system is chosen to coordinate the entry to the critical section.
 - A process that wants to enter its critical section sends a *request* message to the coordinator.
 - The coordinator decides which process can enter the critical section next, and it sends that process a *reply* message.
 - When the process receives a *reply* message from the coordinator, it enters its critical section.
 - After exiting its critical section, the process sends a *release* message to the coordinator and proceeds with its execution.
- 3 messages per critical section entry

-13

Problems of CDME

- Electing the master process? Hardcoded?
- Single point of failure? Electing a new master process?
- Distributed Election algorithms later...

-14

DDME: Fully Distributed Approach

- When process P_i wants to enter its critical section, it generates a new timestamp, TS , and sends the message $request(P_i, TS)$ to all other processes in the system.
- When process P_j receives a *request* message, it may reply immediately or it may defer sending a reply back.
- When process P_i receives a *reply* message from all other processes in the system, it can enter its critical section.
- After exiting its critical section, the process sends *reply* messages to all its deferred requests.

-15

DDME: Fully Distributed Approach (Cont.)

- The decision whether process P_j replies immediately to a $request(P_i, TS)$ message or defers its reply is based on three factors:
 - If P_j is in its critical section, then it defers its reply to P_i .
 - If P_j does *not* want to enter its critical section, then it sends a *reply* immediately to P_i .
 - If P_j wants to enter its critical section but has not yet entered it, then it compares its own request timestamp with the timestamp TS .
 - If its own request timestamp is greater than TS , then it sends a *reply* immediately to P_i (P_i asked first).
 - Otherwise, the reply is deferred.

-16

Problems of DDME

- Requires complete trust that other processes will play fair
 - Easy to cheat just by delaying the reply!
- The processes need to know the identity of all other processes in the system
 - Makes the dynamic addition and removal of processes more complex.
- If one of the processes fails, then the entire scheme collapses.
 - Dealt with by continuously monitoring the state of all the processes in the system.
- Constantly bothering people who don't care
 - Can I enter my critical section? Can I?

-17

Token Passing

- Circulate a token among processes in the system
- Possession of the token entitles the holder to enter the critical section
- Organize processes in system into a logical ring
 - Pass token around the ring
 - When you get it, enter critical section if need to then pass it on when you are done (or just pass it on if don't need it)

-18

Problems of Token Passing

- If machines with token fails, how to regenerate a new token?
- A lot like electing a new coordinator
- If process fails, need to repair the break in the logical ring

-19

Compare: Number of Messages?

- CDME: 3 messages per critical section entry
- DDME: The number of messages per critical-section entry is $2 \times (n - 1)$
 - Request/reply for everyone but myself
- Token passing: Between 0 and n messages
 - Might luck out and ask for token while I have it or when the person right before me has it
 - Might need to wait for token to visit everyone else first

-20

Compare : Starvation

- CDME : Freedom from starvation is ensured if coordinator uses FIFO
- DDME: Freedom from starvation is ensured, since entry to the critical section is scheduled according to the timestamp ordering. The timestamp ordering ensures that processes are served in a first-come, first served order.
- Token Passing: Freedom from starvation if ring is unidirectional
- Caveats
 - network reliable (I.e. machines not "starved" by inability to communicate)
 - If machines fail they are restarted or taken out of consideration (I.e. machines not "starved" by nonresponse of coordinator or another participant)
 - Processes play by the rules

-21

Why ever do DDME?

- Harder
- More messages
- Bothers more people
- Coordinator just as bothered

-22

Atomicity

- Recall: Atomicity = either all the operations associated with a program unit are executed to completion, or none are performed.
- In a distributed system may have multiple copies of the data , replicas are good for reliability/availability
- PROBLEM: How do we atomically update all of the copies?

-23

Replica Consistency Problem

- Imagine we have multiple bank servers and a client desiring to update their bank account
 - How can we do this?
- Allow a client to update any server then have server propagate update to other servers
 - Simple and wrong!
 - Simultaneous and conflicting updates can occur at different servers?
- Have client send update to all servers
 - Same problem - race condition - which of the conflicting update will reach each server first

-24

Two-phase commit

- Algorithm for providing atomic updates in a distributed system
- Give the servers (or replicas) a chance to say no and if any server says no, client aborts the operation

-25

Framework

- Goal: Update all replicas atomically
 - Either everyone commits or everyone aborts
 - No inconsistencies even in face of failures
 - Caveat: Assume no byzantine failures (servers stop when they fail - do not continue and generate bad data)
- Definitions
 - Coordinator: Software entity that shepherds the process (client in our example could be one of the servers)
 - Ready to commit: side effects of update safely stored non-volatilely (recall: write ahead logging)
 - Even if crash, once say I am ready to commit then when recover will find evidence and continue with commit protocol

-26

Two Phase Commit: Phase 1

- Coordinator send a PREPARE message to each replica
- Coordinator waits for all replicas to reply with a vote
- Each participant send vote
 - Votes PREPARED if ready to commit and locks data items being updated
 - Votes NO if unable to get a lock or unable to ensure ready to commit

-27

Two Phase Commit: Phase 2

- If coordinator receives PREPARED vote from all replicas then it may decide to commit or abort
- Coordinator send its decision to all participants
- If participant receives COMMIT decision then commit changes resulting from update
- If participant received ABORT decision then discard changes resulting from update
- Participant replies DONE
- When Coordinator received DONE from all participants then can delete record of outcome

-28

Performance

- In absence of failure, 2PC makes a total of 2 (1.5?) round trips of messages before decision is made
 - Prepare
 - Vote NO or PREPARE
 - Commit/abort
 - Done (but done just for bookkeeping, does not affect response time)

-29

Failure Handling in 2PC - Replica Failure

- The log contains a <commit T > record. In this case, the site executes **redo**(T).
- The log contains an <abort T > record. In this case, the site executes **undo**(T).
- The log contains a <ready T > record; consult C_i . If C_i is down, site sends **query-status** T message to the other sites.
- The log contains no control records concerning T . In this case, the site executes **undo**(T).

-30

Failure Handling in 2PC - Coordinator C_j Failure

- If an active site contains a <commit T > record in its log, the 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 the record <ready T > in its log then the failed coordinator C_j cannot have decided to commit T . Rather than wait for C_j to recover, it is preferable to abort T .
- All active sites have a <ready T > record in their logs, but no additional control records. In this case we must wait for the coordinator to recover.
 - Blocking problem - T is blocked pending the recovery of site S_i .

-31

Failure Handling

- Failure detected with timeouts
- If participant times out before getting a PREPARE can abort
- If coordinator times out waiting for a vote can abort
- If a participant times out waiting for a decision it is blocked!
 - Wait for Coordinator to recover?
 - Punt to some other resolution protocol
- If a coordinator times out waiting for done, keep record of outcome
- other sites may have a replica.

-32

Deadlock Handling

- Recall our discussion of deadlock in the single node case
- Same problem can occur in distributed system
- Worse? Because harder to do manual detection and recovery
 - Can't just note single machine is slow/hung and and reboot
- How can we deal with deadlock in a distributed system?

-33

Global Ordering

- Resource-ordering deadlock-prevention - define a *global* ordering among the system resources.
 - Assign a unique number to all system resources.
 - A process may request a resource with unique number i only if it is not holding a resource with a unique number greater than i .
- Simple to implement; requires little overhead but how easy is it to establish a global ordering?
 - We had this same issue in the single node case. This is a good approach when you can make it work.

-34

Extend the Banker's Algorithm

- Recall the Banker's algorithm
 - Avoids deadlock by not committing resources unless there is a guaranteed way to complete all
- Banker's algorithm is a distributed system?
- Designate one of the processes in the system as the process that maintains the information necessary to carry out the Banker's algorithm.
 - Straight-forward extension of single node case but
 - Banker is bottleneck
 - Messages on each resource acquire/release
 - Same as in single node case: sounds good but pretty expensive!

-35

Other choices?

- What about borrowing from how databases deal with deadlock?

-36

Recall: Timestamp-Based Protocols

- Method for selecting the order among conflicting transactions
- Associate with each transaction a number which is the timestamp or clock value when the transaction begins executing
- Associate with each data item the largest timestamp of any transaction that wrote the item and another the largest timestamp of a transaction reading the item

-37

Timestamp-Ordering

- If timestamp of transaction wanting to read data < write timestamp on the data then it would have needed to read a value already overwritten so abort the reading transaction
- If timestamp of transaction wanting to read data < read timestamp on the data then the last read would be invalid but it is committed so abort the writing transaction
- Ability to abort/rollback is crucial!

-38

Timestamped Deadlock-Prevention Scheme for DS

- Each process P_i is assigned a unique timestamp (or priority)
 - Timestamps are used to decide whether a process P_i should wait for a process P_j ; otherwise P_i is rolled back.
 - The scheme prevents deadlocks. For every edge $P_i \rightarrow P_j$ in the wait-for graph, P_i has a higher priority than P_j . Thus a cycle cannot exist.
 - Ability to abort/rollback is crucial

-39

Variations

- Wait-Die
 - Non-preemptive
- Wound-wait
 - Preemptive
- Both prevent deadlock by avoiding cycles in the wait-for graph

-40

Wait-Die Scheme

- Nonpreemptive
- If P_I requests a resource currently held by P_J , P_I is allowed to wait only if it has a smaller timestamp than P_J (P_I is older than P_J). Otherwise, P_I is rolled back (dies).
 - Example: Suppose that processes P_1 , P_2 , and P_3 have timestamps 1, 2, and 3 respectively.
 - if P_1 request a resource held by P_2 , then P_1 will wait.
 - If P_3 requests a resource held by P_2 , then P_3 will be rolled back.

-41

Wound-Wait Scheme

- Preemptive technique
- If P_I requests a resource currently held by P_J , P_I is allowed to wait only if it has a larger timestamp than does P_J (P_I is younger than P_J). Otherwise P_J is rolled back (P_J is wounded by P_I).
 - Example: Suppose that processes P_1 , P_2 , and P_3 have timestamps 1, 2, and 3 respectively.
 - If P_1 requests a resource held by P_2 , then the resource will be preempted from P_2 and P_2 will be rolled back.
 - If P_3 requests a resource held by P_2 , then P_3 will wait.

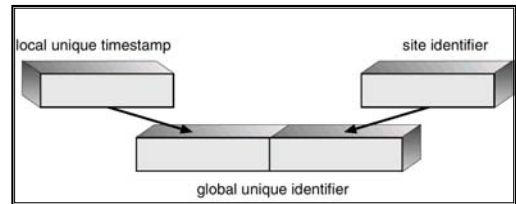
-42

Summary

	Holder has lower timestamp	Holder has higher timestamp
Wait-Die	Requester waits	Requester dies
Wound-Wait	Holder dies (Requester wounds holder)	Requester waits

-43

Unique Timestamps in Distributed Environment



Use site identifier as least significant to ensure that the global timestamps generated at one site not always bigger

-44

Avoiding Starvation

- Both are a priority based scheme and so subject to starvation
- Avoid starvation if when rollback a process allow it to keep its timestamp
- Eventually it should be the highest priority process and will never be rolled back

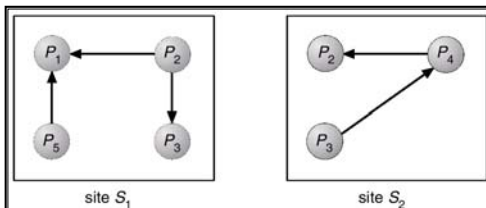
-45

Deadlock detection

- If instead of deadlock prevention, we could allow deadlocks to occur
- Manual detection and recovery is harder in a distributed system
 - Notice whole distributed system is slow/hung and reboot?
- But automatic detection would global knowledge to find cycles in the wait-for graph

-46

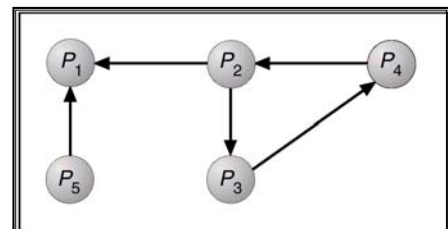
Two Local Wait-For Graphs



Local graphs have no cycles

-47

Global Wait-For Graph



Global graph has a cycle!

-48

Deadlock Detection - Centralized Approach

- Each site keeps a *local* wait-for graph. The nodes of the graph correspond to all the processes that are currently either holding or requesting any of the resources local to that site.
- A global wait-for graph is maintained in a *single* coordination process; this graph is the union of all local wait-for graphs.

-49

False Cycles

- Delay between state of local graph and when information is sent to coordinator and included in global graph
- Unnecessary rollbacks may occur as a result of *false cycles* that occur as a result of this communication latency
- Local graph snapshots may be taken at different points in time such that the union suggests a cycle that isn't really there

-50

Updating the Global Wait-for Graph

- When to send information from local wait-for graphs to be added into the global wait-for graph :
 1. Whenever a new edge is inserted or removed in one of the local wait-for graphs (implies communication with coordinator on every resource acquire/release!)
 2. Periodically, when a number of changes have occurred in a wait-for graph (at least this can batch info sent to coordinator)
 3. Whenever the coordinator needs to invoke the cycle-detection algorithm..

-51

A Fully Distributed Option?

- We could add one additional node P_{ex} to each local wait-for graph.
 - If a local wait-for graph contains a cycle that does not involve node P_{ex} , then the system is in a deadlock state.
 - A cycle involving P_{ex} implies the possibility of a deadlock.
- To ascertain whether a deadlock does really exist:
 - Send description of potential cycle to some other site
 - That site augments the graph with info from their local graph and either detects a deadlock or passes the augmented graph on to another site until all sites have contributed
 - Once been to all sites, have full global wait-for graph

-52

Election Algorithm

-53

Choosing a Coordinator

- In many of the distributed coordination algorithms, we've seen some machine is playing the role of a coordinator
 - Examples: Coordinators for Centralized Deadlock Detection or 2 phase commit
- How do we choose such a coordinator?
- Or elect a new one if the current fails?

-54

Election Algorithms

- GOAL: Determine where a new copy of the coordinator should be started/restarted.
- Formalize this as:
 - Assume that a unique priority number is associated with each active process in the system, and assume that the priority number of process P_i is i .
 - The coordinator is always the process with the largest priority number. When a coordinator fails, the algorithm must elect that active process with the largest priority number
- Two variants: bully and ring based on topology (ring for ring network topology, bully for everything else)

-55

Ring Algorithm

- Applicable to systems organized as a ring (logically or physically).
 - Assumes that the links are unidirectional, and that processes send their messages to their right neighbors.
- Each process maintains an *active list*, consisting of all the priority numbers of all active processes in the system when the algorithm ends.
- If process P_i detects a coordinator failure (timeout waiting for response), it creates a new active list that is initially empty.
 - It then sends a message *elect(i)* to its right neighbor, and adds the number i to its active list.

-56

Ring Algorithm (Cont.)

- If P_i receives a message *elect(j)* from the process on the left, it must respond in one of three ways:
 1. If this is the first (in some time) *elect* message it has seen or sent, P_i creates a new active list with the numbers i and j . It then sends the message *elect(i)*, followed by the message *elect(j)*.
 2. If the message does not contain P_i 's number then P_i adds j to its active list and forwards the message to the right
 3. If the message does contain P_i 's number, then P_i should have seen all previous messages and its active list should be full

-57

Recovery in Ring Algorithm

- Recovering process can send a message around the ring requesting to know who is the coordinator
- Coordinator will see message as it goes around ring and reply with its identity

-58

Bully Algorithm

- For network topologies other than ring
 - Must know all other processes in the system
- Process P_i sends a request that is not answered by the coordinator within a specified time \Rightarrow assume that the coordinator has failed
- P_i tries to elect itself as the new coordinator

-59

Bully Algorithm (Cont.)

- P_i sends an election message to every process with a higher priority number, P_i then waits for any of these processes to answer within T_1 .
 - If no response within T_1 , assume that all processes with numbers greater than i have failed; P_i elects itself the new coordinator.
 - If answer is received, P_i begins time interval T_2 , waiting to receive a message that a process with a higher priority number has been elected.
 - If no such message is received within T_2 , assume the process with a higher number has failed; P_i should restart the algorithm

-60

Bully Algorithm (Cont.)

- If P_i is not the coordinator, then, at any time during execution, P_i may receive one of the following two messages from process P_j
 - P_j is the new coordinator ($j > i$). P_i , in turn, records this information.
 - P_j started an election ($j < i$). P_i sends a response to P_j and begins its own election algorithms

-61

Recovery in Bully Algorithm

- After a failed process recovers, it immediately begins execution of the same algorithm.
- If there are no active processes with higher numbers, the recovered process forces all processes with lower number to let it become the coordinator process, even if there is a currently active coordinator with a lower number.

-62

Byzantine Generals Problem

- Deals with reaching agreement in the face of both faulty communications and untrustworthy peers
- Problem:
 - Divisions of an army each commanded by a general surrounding an enemy camp
 - Generals must reach agreement on whether to attack (a certain number must attack or defeat is certain)
 - Divisions are geographically separated such that they must communicate via messengers
 - Messengers may be caught and never reach the other side (lost messages)
 - Generals may be traitors (faulty/compromised processes)

-63

Problem 1: Losts Messengers/ Messages

- How can we deal with the fact that messages may be lost? (We saw this in TCP)
- Detect failures using a time-out scheme.
 - When send a message, specifies a time interval to wait for an acknowledgment
 - When receives a message, sends an acknowledgment
 - Acknowledgment can be lost too!
 - If receives the acknowledgment message within the specified time interval can conclude that message was received its message. If a time-out occurs, retransmit message and wait for another acknowledgment.
 - Continue until either receives an acknowledgment, or give up after some time?

-64

The Last Word?

- Suppose, the receiver needs to know that the sender has received its acknowledgment message, in order to decide on how to proceed
- Actually, in the presence of failure, it is not possible to accomplish this task
- It is not possible in a distributed environment for processes P_i and P_j to agree completely on their current respective states
- Always level of uncertainty about last message

-65

Traitors?

- Consider that generals can be traitors (processes can be faulty)
- What could traitors do?
 - Refuse to send any messages
 - Delay sending messages
 - Send incorrect messages
 - Send different messages to different generals

-66

Formalize Agreement

- Consider a system of n processes, of which no more than m are faulty.
- Devise an algorithm that allows each non-faulty P_i to construct a vector $X_i = (A_{i1}, A_{i2}, \dots, A_{in})$ such that:
 - Each process P_i has some private value of V_i .
 - If P_j is a nonfaulty process, then $A_{ij} = V_j$.
 - If P_i and P_j are both nonfaulty processes, then $X_i = X_j$.

-67

Solutions to Problem of Reaching Agreement

- Solutions share the following properties.
 - Assume reliable communication
 - Bound maximum number of traitors to m
 - A correct algorithm can be devised only if $n \geq 3 \times m + 1$.
 - The worst-case delay for reaching agreement is proportionate to $m + 1$ message-passing delays.

-68

Simplest Example

- An algorithm for the case where $m = 1$ and $n = 4$ ($\geq 3 \times m + 1$) requires $m + 1 = 2$ rounds of information exchange:
 - Each process sends its private value to the other 3 processes.
 - Each process sends the information it has obtained in the first round to all other processes.

-69

Simplest Example (con't)

- If a faulty process refuses to send messages, a nonfaulty process can choose an arbitrary value and pretend that that value was sent by that process.
- After the two rounds are completed, a nonfaulty process P_i can construct its vector $X_i = (A_{i1}, A_{i2}, A_{i3}, A_{i4})$ as follows:
 - $A_{ij} = V_j$.
 - For $j \neq i$, if at least two of the three values reported for process P_j agree, then the majority value is used to set the value of A_{ij} . Otherwise, a default value (*nil*) is used.

-70

Consider

- What if $n < 4$
 - If $n=3$ and there was one traitor then it could lie differently to the two non-traitors and they could not resolve the discrepancy by a majority vote
- What if only one round?
 - Second round used to check agreement
 - Used to see that there is no agreement on what a "traitor" said

-71

Outtakes

-72

- Ensuring atomicity in a distributed system requires a *transaction coordinator*, which is responsible for the following:
 - Starting the execution of the transaction.
 - Breaking the transaction into a number of subtransactions, and distribution these subtransactions to the appropriate sites for execution.
 - Coordinating the termination of the transaction, which may result in the transaction being committed at all sites or aborted at all sites.

-73

Two-Phase Commit Protocol (2PC)

- Assumes fail-stop model.
- Execution of the protocol is initiated by the coordinator after the last step of the transaction has been reached.
- When the protocol is initiated, the transaction may still be executing at some of the local sites.
- The protocol involves all the local sites at which the transaction executed.
- Example: Let T be a transaction initiated at site S_i and let the transaction coordinator at S_j be C_j .

-74

Phase 1: Obtaining a Decision

- C_j adds $\langle \text{prepare } T \rangle$ record to the log.
- C_j sends $\langle \text{prepare } T \rangle$ message to all sites.
- When a site receives a $\langle \text{prepare } T \rangle$ message, the transaction manager determines if it can commit the transaction.
 - If no: add $\langle \text{no } T \rangle$ record to the log and respond to C_j with $\langle \text{abort } T \rangle$.
 - If yes:
 - add $\langle \text{ready } T \rangle$ record to the log.
 - force all log records for T onto stable storage.
 - transaction manager sends $\langle \text{ready } T \rangle$ message to C_j .

-75

Phase 1 (Cont.)

- Coordinator collects responses
 - All respond "ready", decision is *commit*.
 - At least one response is "abort", decision is *abort*.
 - At least one participant fails to respond within time out period, decision is *abort*.

-76

Phase 2: Recording Decision in the Database

- Coordinator adds a decision record $\langle \text{abort } T \rangle$ or $\langle \text{commit } T \rangle$ to its 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.

-77

Concurrency Control

I cut this all together - too similar to mutual Exclusion - does it deserve a separate discussion

-78

Concurrency Control

- Modify the centralized concurrency schemes to accommodate the distribution of transactions.
- Transaction manager coordinates execution of transactions (or subtransactions) that access data at local sites.
- Local transaction only executes at that site.
- Global transaction executes at several sites.

-79

Locking Protocols

- Nonreplicated scheme - each site maintains a local lock manager which administers lock and unlock requests for those data items that are stored in that site.
 - Simple implementation involves two message transfers for handling lock requests, and one message transfer for handling unlock requests.
 - Deadlock handling is more complex.

-80

Single-Coordinator Approach

- A single lock manager resides in a single chosen site, all lock and unlock requests are made at that site.
 - Simple implementation
 - Simple deadlock handling
 - Possibility of bottleneck
 - Vulnerable to loss of concurrency controller if single site fails
- *Multiple-coordinator approach* distributes lock-manager function over several sites.

-81

Majority Protocol

- Avoids drawbacks of central control by dealing with replicated data in a decentralized manner.
- Must get ok from at least $n/2 + 1$ participants
- Deadlock-handling algorithms must be modified; possible for deadlock to occur in locking only one data item.
 - Example: two processes trying to lock each get 2 out of 4 processes to say ok - each need a third?

-82

Biased Protocol (OUTTAKE)

- Similar to majority protocol, but requests for shared locks prioritized over requests for exclusive locks.
- Less overhead on read operations than in majority protocol; but has additional overhead on writes.
- Like majority protocol, deadlock handling is complex.

-83

Primary Copy

- One of the sites at which a replica resides is designated as the primary site. Request to lock a data item is made at the primary site of that data item.
- Concurrency control for replicated data handled in a manner similar to that of unreplicated data.
- Simple implementation, but if primary site fails, the data item is unavailable, even though other sites may have a replica.

-84

Example Centralized Deadlock Detection Algorithm

-85

Detection Algorithm Based on Option 3

- Append unique identifiers (timestamps) to requests from different sites.
- When process P_i at site A , requests a resource from process P_j at site B , a request message with timestamp TS is sent.
- The edge $P_i \rightarrow P_j$ with the label TS is inserted in the local wait-for of A . The edge is inserted in the local wait-for graph of B only if B has received the request message and cannot immediately grant the requested resource.

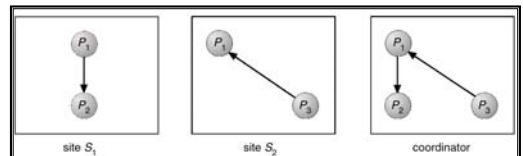
-86

The Algorithm

1. The controller sends an initiating message to each site in the system.
2. On receiving this message, a site sends its local wait-for graph to the coordinator.
3. When the controller has received a reply from each site, it constructs a graph as follows:
 - (a) The constructed graph contains a vertex for every process in the system.
 - (b) The graph has an edge $P_i \rightarrow P_j$ if and only if (1) there is an edge $P_i \rightarrow P_j$ in one of the wait-for graphs, or (2) an edge $P_i \rightarrow P_j$ with some label TS appears in more than one wait-for graph.
 If the constructed graph contains a cycle \Rightarrow deadlock.

-87

Local and Global Wait-For Graphs

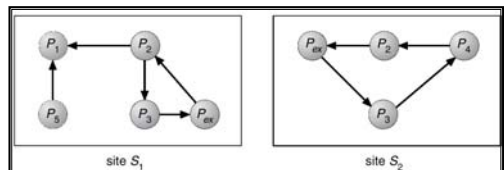


-88

Distributed Deadlock Detection

-89

Augmented Local Wait-For Graphs



-90

Augmented Local Wait-For Graph
in Site S_2

