DISTRIBUTED COORDINATION

Tightly coupled systems:

- Same clock, usually shared memory.
- Communication is via this shared memory.
- Multiprocessors.

Loosely coupled systems:

- Different clock.
- Use communication links.
- Distributed systems.

EVENT ORDERING

- "**Happening before**" vs. **concurrent**.

- Here **A --> B** means A occurred before B and thus could have caused B.

- Of the events in **<<< FIGURE 18.1 >>>,** which are happened-before and which are concurrent?

- Ordering is easy if the systems share a common clock (i.e., it’s in a centralized system.)

- With no common clock, each process keeps a logical clock.

- This Logical Clock can be simply a counter - it may have no relation to real time.

- Adjust the clock if messages are received with time higher than current time.

- We require that **LC(A) < LC(B)**, the time of transmission be less than the time of receipt for a message.

- So if on message receipt, **LC(A) >= LC(B)**,
  then set **LC(B) = LC(A) + 1.**

MUTUAL EXCLUSION / SYNCHRONIZATION

- Using Distributed Semaphores

- A particular processor can provide mutual exclusion, but it's much harder to do with a distributed system. (May need a distributed system when the network is not fully connected.)

CENTRALIZED APPROACH

- Choose one processor as coordinator who handles all requests.

- A process that wants to enter its critical section sends a request message to the coordinator.

- On getting a request, the coordinator doesn't answer until the critical section is empty (has been released by whoever is holding it).

- On getting a release, the coordinator answers the next outstanding request.

- If coordinator dies, elect a new one who recreates the request list by polling all systems to find out what resource each thinks it has.

- Requires three messages per critical section entry;

  request,
  reply,
  release.

- The method is free from starvation.

FULLY DISTRIBUTED APPROACH

- Approach due to Lamport

    a) The general mechanism is for a process **P[i]** to send a request (with ID and time stamp) to **all** other processes.
    b) When a process **P[j]** receives such a request, it may reply immediately or it may defer sending a reply back.
    c) When responses are received from all processes, then **P[i]** can enter its Critical Section.
    d) When **P[i]** exits its critical section, the process sends reply messages to all its deferred requests.

- The general rules for reply for processes receiving a request:

    1. If **P[j]** receives a request, and **P[j]** process is in its critical section, defer (hold off) the response to **P[i].**

    2. If **P[j]** receives a request,, and not in critical section, and doesn't want to get in, then reply immediately to **P[i].**

    3. If **P[j]** wants to enter its critical section but has not yet entered it, then it compares its own timestamp **TS[j]** with the timestamp **TS[i]** from **T[i].**

    4. If **TS[j] > TS[i],** then it sends a reply immediately to **P[i]. P[i]** asked first.

    5. Otherwise the reply is deferred until after **P[j]** finishes its critical section.

- The Fully Distributed Approach assures:

    a) Mutual exclusion

    b) Freedom from deadlock

    c) Freedom from starvation, 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.

    d) 2 X (n - 1) messages needed for each entry. This is the minimum number of required messages per critical-section entry when processes act independently and concurrently.

- Problems with the method include:

    a) Need to know identity of everyone in system.
    b) Fails if anyone dies - must continually monitor the state of all processes.
    c) Processes are always coming and going so it's hard to maintain current data.

TOKEN PASSING APPROACH

Tokens with rings

- Whoever holds the token can use the critical section. When done, pass on the token. Processes must be logically connected in a ring -- it may not be a physical ring.

- Advantages:

  No starvation if the ring is unidirectional.

  There are many messages passed per section entered if few users want to get in section.

  Only one message/entry if everyone wants to get in.

- OK if you can detect loss of token and regenerate via election or other means.

- If a process is lost, a new logical ring must be generated.

Tokens without rings (Chandy)

- A process can send a token to any other process.

- Each process maintains an ordered list of requests for a critical section.

- Process requiring entrance broadcasts message with ID and new count (current logical time).

- When using the token, store into it the time-of-request for the request just finished.

- If a process is holding token and not in critical section, send to first message received (if time maintained in token is later than that for a request in the list, it's an old message and can be discarded.) If no request, hang on to the token.

ATOMICITY

- Either ALL the operations associated with a program unit are executed to completion, or none are performed.

- Ensuring atomicity in a distributed system requires a **transaction coordinator**, which is responsible for the following:

    Starting the execution of a transaction.

    Breaking the transaction into a number of sub transactions, and distributing these sub transactions 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.

Two-Phase Commit Protocol (2PC)

- For atomicity to be ensured, all the sites in which a transaction T executes must agree on the final outcome of the execution. 2PC is one way of doing this.
- 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.
- Let T be a transaction initiated at site Si, and let the transaction coordinator at Si be Ci .

Phase 1: Obtaining a decision

- Ci adds <prepare T> record to the log.
- Ci sends <prepare T> message to all sites.
- When a site receives a <prepare T> message, the transaction manager determines if it can commit the transaction.

  If no: add <no T> record to the log and respond to Ci with <abort T >.

  If yes:
  * add <ready T > record to the log.
  * force all log records for T onto stable storage.
  * transaction manager sends <ready T > message to Ci .

- 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 timeout period, decision is abort.

Phase 2: Recording the decision in the database

- Coordinator adds a decision record (<abort T >or <commit T >) 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

Failure Handling in 2PC:

Failure of a participating Site:

- 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 Ci . If Ci 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).

Failure of the Coordinator Ci:

- 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 the record <ready T> in its log, then the failed coordinator Ci cannot have decided to commit T. Rather than wait for Ci 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 Si.

Concurrency Control

- Modify the centralized concurrency schemes to accommodate the distribution of transactions.

- Transaction manager coordinates execution of transactions (or sub transactions) that access data at local sites.

- Local transaction only executes at that site.

- Global transaction executes at several sites.

Locking Protocols

- Can use the two-phase locking protocol in a distributed environment by changing how the lock manager is implemented.

- **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.

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.

Majority protocol:

- Avoids drawbacks of central control by dealing with replicated data in a decentralized manner.

  More complicated to implement

  Deadlock-handling algorithms must be modified; possible for deadlock to occur in locking only one data item.

Biased protocol:

- 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.

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 for unreplicated data.
- Simple implementation, but if primary site fails, the data item is unavailable, even though other sites may have a replica.

Timestamping:

- Generate unique timestamps in distributed scheme:

  a) Each site generates a unique local timestamp.

  b) The global unique timestamp is obtained by concatenation of the unique local timestamp with the unique site identifier.

  c) Use a logical clock defined within each site to ensure the fair generation of timestamps.

- Timestamp-ordering scheme - combine the centralized concurrency control timestamp scheme with the (2PC) protocol to obtain a protocol that ensures serializability with no cascading rollbacks.

DEADLOCK PREVENTION

- To **prevent Deadlocks**, must stop one of the four conditions:

  Mutual exclusion,

  Hold and wait,

  No preemption,

  Circular wait.

- Possible Solutions Include:

  a) Global resource ordering (all resources are given unique numbers and a process can acquire them only in ascending order.) Simple to implement, low cost, but requires knowing all resources. Prevents a circular wait.

  b) Banker's algorithm with one process being banker (can be bottleneck.) Large number of messages is required so method is not very practical.

  c) Priorities based on unique numbers for each process has a problem with starvation.

  d) Priorities based on timestamps can be used to prevent circular waits. Each process is assigned a timestamp at its creation. Several variations are possible:

Non-preemptive	Requester waits for resource if older than current resource holder, else it's rolled back losing all its resources. The older a process gets, the longer it waits.
Preemptive	If the requester is older than the holder, then the holder is preempted (rolled back). If the requester is younger, then it waits. Fewer rollbacks here. When P(i) is preempted by P(j), it restarts and, being younger, ends up waiting for P(j).

  Keep timestamp if rolled back (don't reassign them) - prevents starvation since a preempted process will soon be the oldest.

  The preemption method has fewer rollbacks because in the non-preemptive method, a young process can be rolled back a number of times before it gets the resource.

DEADLOCK DETECTION

- The previous Prevention Techniques can unnecessarily preempt a resource. Can we do rollback only when a deadlock is detected??

- Wait for graphs - recall, with a single resource of a type, a cycle is a deadlock.

- Each site maintains a local wait-for-graph, with nodes being local or remote processes requesting LOCAL resources. **<<< FIGURE 7.7 >>>**

- To show no deadlock has occurred, show the union of graphs has no cycle.
  <<< FIGURE 18.3 >>> <- P2 is in both graphs
  <<< FIGURE 18.4 >>> <- Cycle formed.

CENTRALIZED

- In this method, the union is maintained in one process. If a global (centralized) graph has cycles, a deadlock has occurred.

- Construct graph incrementally (whenever an edge is added or removed), OR periodically (at some fixed time period), OR whenever checking for cycles (because there's some reason to fear deadlock).

- Can roll back unnecessarily due to false cycles {because information is obtained asynchronously (a delete may not be reported before an insert)} and because cycles are broken by terminated processes.

- Can avoid false cycles with timestamps that force synchronization.

FULLY DISTRIBUTED

- All controllers share equally in detecting deadlocks.

- See **<<< FIGURE 18.6 >>>**. At Site 1, P[ext] shows that P3 is waiting for some external process, and that some external process is waiting for P2 -- but beware, they may not be related external processes.

- Each site collects such a local graph and uses this algorithm:

  1. If a local site has a cycle, not including a P[ext] , there is a deadlock.

  2. If there's no cycle, then there's no deadlock.

  3. If a cycle includes a P[ext] , then there MAY be a deadlock. Each site waiting for a P[ext] sends its graph to the site of the P[ext] it's waiting for. That site combines the two local graphs and starts the algorithm again.

ELECTION ALGORITHMS

- Determining who should be coordinator
- Method depends on configuration

THE BULLY ALGORITHM

- Suppose P(i) sends a request to the coordinator which is not answered.
- We want the highest priority process to be the new coordinator.
- Steps to be followed:

  a) P(i) sends "I want to be elected" to all P(j) of higher priority.

  b) If no response, then P(i) has won the election.

  c) All living P(j) send "election" requests to **THEIR** higher priority P(k), and send "you lose" messages back to P(i).

  d) Finally only one process receives no response.

  e) That process sends "I am it" messages to all lower priority processes.

A RING ALGORITHM

- Used where there are unidirectional links. The algorithm uses an "active list" that is filled in upon a failure. Upon completion, this list contains priority numbers and the active processes in the system.

  a) Every site sends every other site its priority.

  b) If coordinator not responding, start active list with its ID on it and send messages that it is holding election.

  c) If this is first for receiver, create active list with received ID and its ID, send 2 messages, one for it and one for received (second message).

  d) If not first (and not same ID), add to active list and pass on.

  e) If receives message it sent, active list complete, and can name coordinator.

REACHING AGREEMENT BETWEEN PROCESSES

- The problem here is how to get agreement with an unreliable mechanism.

UNRELIABLE COMMUNICATIONS

- Can have faulty links - can use a timeout to detect this.

FAULTY PROCESSES

- Can have faulty processes generating bad messages.
- Cannot guarantee agreement.