Silberschatz, et al.
Topics based on Chapter 16

Distributed Coordination

Distributed Coordination

- Synchronization in a distributed environment
  - Event ordering
  - Mutual exclusion
  - Atomicity
  - Concurrency Control
  - Deadlock Handling
  - Election Algorithms
  - Reaching Agreement

Event Ordering

- Distributed systems do not have common memory or clock, as do centralized systems. Event ordering is a *partial* ordering in distributed systems.
- Extending the *happened-before* relation from partial ordering to total ordering

Event Ordering: happened-before

- *Happened-before* relation (denoted by $\rightarrow$)
  - If A and B are events in the same process, and A was executed before B, then $A \rightarrow B$
  - 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 $A \rightarrow B$
  - If $A \rightarrow B$ and $B \rightarrow C$ then $A \rightarrow C$
- Irreflexive partial ordering (an event cannot occur before itself)

Event Ordering: *happened-before*

- A and B not related by $\rightarrow$; A and B were executed *concurrently*
- Neither event can causally affect the other
- Example

  $p0 \rightarrow p1 \rightarrow p2 \rightarrow p3 \rightarrow p4 \rightarrow p5$

  $q0 \rightarrow q1 \rightarrow q2 \rightarrow q3 \rightarrow q4 \rightarrow q5$

  $p3 \rightarrow q3,\ q4 \rightarrow p2$
- Hence $p3 \rightarrow q4,\ q2 \rightarrow p4$
- Concurrent events include p0 and q0, p0 and q1, p5 and q4

Event Ordering *happened-before* implementation

- Defining happened-before without the use of (synchronized) physical clocks
  - Associate a timestamp with each system event. Require that for every pair of events A and B,if $A \rightarrow B$, then the timestamp of A is less than the timestamp of B.
  - Within each process P_i a logical clock,LC_i is associated. The logical clock can be implemented as a simple counter that is incremented between any two successive events executed within a process.
  - A process advances its logical clock when it receives a message whose timestamp is greater than the current value of its logical clock.
  - If the timestamps of two events A and B are the same, then the events are concurrent. We may use the process identity numbers to break ties and to create a total ordering.

Distributed Mutual Exclusion

- Assumptions
  - The system consists of n processes; each process P_i resides at a different processor.
  - Each process has a critical section that requires mutual exclusion.
- Requirement
  - If P_i is executing in its critical section, then no other process P_j is executing in its critical section.
- We present two algorithms to ensure the mutual exclusion execution of processes in their critical sections.

Distributed Mutual Exclusion: 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.
- This scheme requires three messages per critical-section entry:
  - request
  - reply
  - release

Distributed Mutual Exclusion: 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.

Distributed Mutual Exclusion: Fully Distributed Approach

- 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.

Distributed Mutual Exclusion:
Example of Fully Distributed Approach

- Processes P1, P2, P3, P4
- P1 requests critical section (timestamp=10)
- P2 requests critical section (TS=15)
- P2, P3, and P4 reply to P2 ; P1 defers reply to P2
- P1, P2, P3, and P4 replies to P1
- P1 enters critical section
- P3 requests critical section (TS=16)
- P3, and P4 reply to P3; P1 defers reply (still in critical section); P2 defers reply (its TS is smaller)
- P1 completes critical section, replies to P2 and P3
- P2 enters critical section, completes, replies to P3
- P3 enters critical section, completes

Distributed Mutual Exclusion: Fully Distributed Approach

- Desirable behaviors of the fully distributed approach
  - Freedom from deadlock is ensured.
  - 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.
  - The number of messages per critical-section entry is 2(n-1). This is the minimum number of required messages per critical-section entry when processes act independently and concurrently.

Distributed Mutual Exclusion: Fully Distributed Approach

- Some undesirable consequences of the approach
  - The processes need to know the identity of all other processes in the system, which makes the dynamic addition and removal of processes more complex.
  - If one of the processes fails, then the entire scheme collapses. This can be dealt with by continuously monitoring the state of all the processes in the system.
  - Processes that have not entered their critical section must pause frequently to assure other processes that they intend to enter the critical section. This protocol is therefore suited for small, stable sets of cooperating processes.

Distributed Mutual Exclusion: Token Passing

- Token: special kind of message passed around a logical ring
- Possession of token allows process to enter critical section
- Starvation-free if unidirectional ring
- Failure situations:
  - Token lost (election to generate a new token)
  - Process fails (new logical ring must be established)

Atomicity

- *Atomic transaction*: either all of the operations associated with a program unit are executed to completion, or none are performed.
- *Transaction coordinator*: ensures atomicity in a distributed system. Each site has its own transaction coordinator, which is responsible for:
  - Starting the execution of the 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.
- Each site maintains a log for recovery purposes.

Transaction Logs

- Write-ahead log – all updates are recorded on the log, which is kept in stable storage; log has following fields:
  - transaction name
  - data item name, old value, new value
- The log has a record of <T_i **starts**>, and either
  - < T_i **commits**> if the transactions commits, or
  - < T_i **aborts**> if the transaction aborts.

Transaction logs and log-based recovery

- Recovery algorithm uses two procedures:
  - **undo**(T_i) – restores value of all data updated by transaction T_i to the old values. It is invoked if the log contains record <T_i **starts**>, but not <T_i **commits**>.
  - **redo**(T_i) – sets value of all data updated by transaction T_i to the new values. It is invoked if the log contains both <T_i **starts**> and <T_i **commits**>.

Atomicity: Two-Phase Commit Protocol (2PC)

- Guarantees atomicity
- Initiated by 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
- Protocol proceeds in two phases
  - Phase one: voting on whether to commit the transaction
  - Phase two: carrying out the selected action
- Assumes “fail-stop” model
  - Hardware and software bugs bring the system to a halt but do not corrupt nonvolatile storage contents

2PC: Phase 1
Obtaining a decision

- T, a transaction initiated at site S_i; Transaction coordinator at S_i is C_i
- C_i adds <prepare T> record to the log.
- C_i 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 C_i 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 C_i.
- 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*

2PC: Phase 2
Recording 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.

2PC: Handling a Site 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.
  - Receiving sites consult log to see if T has executed there and if so respond with information about the outcome
  - If no response, **query-status** sent again after a wait
- The log contains no control records concerning T. In this case, the site executes **undo**(T) because it failed before responding to the <prepare T> message and hence the transaction was aborted by C_i

2PC: Handling coordinator failure

- If coordinator fails in midst of commit protocol for transaction T, the participating sites must decide whether to commit or abort the transaction
- In certain cases the sites cannot decide, and must wait for the coordinator to recover

2PC: Handling coordinator failure

- 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 C_i cannot have decided to commit T. Rather than wait for C_i 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.
  - Called the *blocking problem* – T is blocked pending the recovery of site S_i.

2PC: Network failure

- Effects of network failure on pending transactions results in same cases as site/coordinator failures
  - Network failure appears to sites as failure of other sites
  - If network partitioned, either all sites and coordinator are on the same side of the partition (no change) or they are on different sides (site/coordinator failure)

Concurrency Control in a Centralized Environment: two-phase locking protocol

- Chapter 6 defined a two-phase locking protocol in the context of concurrent atomic transactions
- Each transaction required to issue lock and unlock requests in two phases
  - **Growing phase:** A transaction may obtain locks, but may not release any lock.
  - **Shrinking phase:** A transaction may release locks, but may not obtain any new locks.
- Ensures *conflict serializability* but not freedom from deadlock
  - Conflict serializability: read/write operations in concurrently-executing transactions can be rearranged to have same effect as if the transactions were executed serially

Locking protocols: data access modes

- **Shared:** If T_i has obtained a shared-mode lock on data item Q,then T_i can read this item, but it cannot write Q.
- **Exclusive:** If T_i has obtained an exclusive-mode lock on data item Q,then T_i can both read and write Q.

Concurrency Control

- We can modify the centralized concurrency schemes presented earlier to accommodate a distributed environment (i.e., to accommodate the distribution of transactions).
- *Transaction manager* coordinates execution of transactions (or subtransactions) that access data at local sites.
  - Local transaction: executes only at that ise
  - Global transaction: executes at several sites
- Transaction manager maintains log for recovery purposes and participates in concurrency control schemes

Concurrency Control: 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 since lock/unlock decisions are not made at a single site (to be discussed).

Concurrency Control: Locking Protocols

- 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.

Concurrency Control: Locking Protocols

- Majority protocol (a multiple-coordinator approach)
  - Request to lock a data item that is replicated at n sites requires agreement from more than n/2 of the managers. Transaction cannot proceed until it has acquired locks on the majority of the replicas.
  - 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 (e.g., an even number of replicas and two requests, each with 50% of the locks)

Concurrency Control
Locking Protocols

- Biased protocol
  - Similar to majority protocol, but requests for shared locks prioritized over requests for exclusive locks.
  - A request for a **shared** lock requires agreement of one lock manager
  - A request for **exclusive** lock requires agreement from all lock managers
  - Less overhead on **read** operations than in majority protocol; but has additional overhead on **writes**.
  - Like majority protocol, deadlock handling is complex.

Concurrency Control
Locking Protocols

- 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.

Concurrency Control: Timestamping

- Generate unique timestamps in distributed scheme that can be used in deciding serialization order
  - Each site generates a unique local timestamp (perhaps at different clock rates).
  - The global unique timestamp is obtained by concatenation of the unique local timestamp with the unique site identifier.
  - Use a logical clock defined within each site to ensure the fair generation of timestamps (synchronize logical clock whenever site receives transaction with greater timestamp)

Deadlock Handling

- Recall the three approaches
  - Deadlock prevention
  - Deadlock avoidance
  - Deadlock detection

Deadlock Prevention

- Earlier techniques can be modified for use here, as well
- 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.
- Banker's algorithm – designate one of the processes in the system as the process that maintains the information necessary to carry out the Banker's algorithm.
  - Also implemented easily, but may require too much overhead.

Deadlock Prevention using timestamps

- Each process P_i is assigned a unique priority number.
- Priority numbers are used to decide whether a process P_i should wait for a process P_j. P_i can wait for P_j if P_i has a higher priority than 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.
- To avoid starvation, use timestamps (see following for two schemes)

Deadlock Prevention
Wait-Die Scheme

- Based on a nonpreemptive technique.
- 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 does 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 5, 10, and 15, respectively.
  - If P_1 requests 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.

Deadlock Prevention
Wound-Wait Scheme

- Based on a preemptive technique; counterpart to the wait-die system.
- 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 (j is wounded by P_i).
- Example: Suppose that processes P_1, P_2, and P_3 have timestamps 5, 10, and 15, 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.

Deadlock Prevention

- Wait-Die and Wound-Wait both avoid starvation (assuming new timestamp is not issued) because younger processes will have a larger timestamp
- Wait-die requires older process to wait for younger one. The older a process gets, the more it waits. In wound-wait, older does not wait for younger
- Fewer rollbacks in wound-wait (wait-die may repeat sequence multiple times)
- Both schemes may cause unnecessary rollbacks

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.
- There are three different options (points in time) when the wait-for graph may be constructed:
  1. Whenever a new edge is inserted or removed in one of the local wait-for graphs.
  2. Periodically, when a number of changes have occurred in a wait-for graph.
  3. Whenever the coordinator needs to invoke the cycle-detection algorithm.
- Unnecessary rollbacks may occur as a result of false cycles if, for example, actions are reordered as they propagate; text includes an algorithm based on option 3 that avoids this.

Deadlock Detection
Fully Distributed Approach

- All controllers share equally the responsibility for detecting deadlock.
- Every site constructs a wait-for graph that represents a part of the total graph.
- We add one additional node P_{ex} to each local wait-for graph.
  - Arc $P_i \rightarrow P_{ex}$ if P_i is waiting for a data item on another site being held by any process
  - Arc $P_{ex} \rightarrow P_i$ if there exists a process at another site that is waiting to acquire a resource currently held by P_i at this local site
- 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 exist, a distributed deadlock-detection algorithm must be invoked.

Election Algorithms

- We need these algorithms for a number of purposes
  - Determine where a new copy of the coordinator should be restarted
  - Determine who should regenerate a token after it has been lost
- Election algorithms assume that a unique priority number is associated with each active process in the system. Assume that the priority number of process P_i is i
- Assume a one-to-one correspondence between processes and sites (refer to both as processes)
- We are looking for the process with the largest priority number; this process will be elected
- Two algorithms: bully algorithm and ring algorithm

Election Algorithms
Bully Algorithm

- Applicable to systems where every process can send a message to every other process in the system.
- If process P_i sends a request that is not answered by the coordinator within a time interval T, assume that the coordinator has failed; P_i tries to elect itself as the new coordinator.
- 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.
- If no response within T, assume that all processes with numbers greater than i have failed; P_i elects itself the new coordinator.
- If answer is received, Pi begins time interval T', waiting to receive a message that a process with a higher priority number has been elected. If no message is sent within T', assume the process with a higher number has failed; P_i should restart the algorithm.

Election Algorithms
Bully Algorithm

- 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 algorithm, provided that P_i has not already initiated such an election.
- 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 numbers to let it become the coordinator process, even if there is a currently active coordinator with a lower number.

Election Algorithms
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, 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.

Election Algorithms
Ring Algorithm

- 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 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 $i \neq j$, then P_i adds j to its active list and forwards the message to its right neighbor.
  - 3. If $i = j$, then the active list for P_i now contains the numbers of all the active processes in the system.P_i can now determine the largest number in the active list to identify the new coordinator process.

Reaching Agreement

- Applications such as the ones just discussed involve a set of processes agreeing on a common “value”.
- One example: Byzantine generals problem
  - Generals deciding whether to attack at dawn
  - Communication between camp via runners
  - Runners may be captured or may be traitors
- Agreement may be made more difficult (or made impossible) due to
  - Faulty communication medium
  - Faulty processes
    - Processes that send garbled or incorrect messages to other processes
    - Processes that collaborate with each other in an attempt to defeat a scheme
  - See text for discussion of these issues