(selected topics from chapter 16 & 18)

Distributed Systems

Presented By: Dr. El-Sayed M. El-Alfy

Note: Most of the slides are compiled from the textbook and its complementary resources

Objectives/Outline

Objectives

- Provide a high-level overview of distributed systems
- Describe various methods for achieving mutual exclusion in a distributed system
- Present schemes for handling deadlocks in a distributed system
- Present algorithms used in case of coordinator failure

Outline

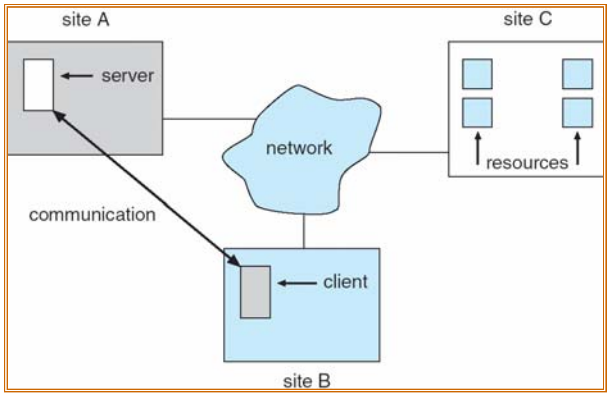
- Introduction
- Types of Distributed Operating Systems
- Event Ordering
- Mutual Exclusion
- Deadlock Handling
- Election Algorithms

16.1

Introduction

- Distributed system (DS) is a collection of loosely coupled processors that do not share memory or clock (i.e. each processor has its own memory and clock); communicate through a network
  - Processors are referred to as sites, nodes, computers, machines, hosts
  - Processors in DS are most likely heterogeneous (i.e. vary in size and function)
- Reasons for distributed systems
  1. Resource sharing
     - sharing and printing files at remote sites
     - processing information in a distributed database
     - using remote specialized hardware devices
  2. Computation speedup – load sharing
  3. Reliability – detect and recover from site failure, function transfer, reintegrate failed site
  4. Communication – message passing
- Require mechanisms for process synchronization & communication, dealing with deadlocks, handling failures not encountered in a centralized system

Introduction (cont.)



A general structure of a distributed system

16.2

Types of Distributed Operating Systems

- Network Operating Systems
  - Users are aware of multiplicity of machines. Access to resources of various machines is done explicitly by:
    - Remote logging into the appropriate remote machine (telnet, ssh)
    - Transferring data from remote machines to local machines, via FTP
- Distributed Operating Systems
  - Users access remote resources in the same way they access local resources (seamless manner)
  - Data migration – transfer data by transferring entire file, or transferring only those portions of the file necessary for the immediate task
  - Computation migration – transfer the computation, rather than the data, across the system
  - Process migration – execute an entire process, or parts of it, at different sites

---

Types of Distributed Operating Systems (cont.)

- Process Migration
  - Load balancing – distribute processes across network to even the workload
  - Computation speedup – subprocesses can run concurrently on different sites
  - Hardware preference – process execution may require specialized processor
  - Software preference – required software may be available at only a particular site
  - Data access – run process remotely, rather than transfer all data locally

---

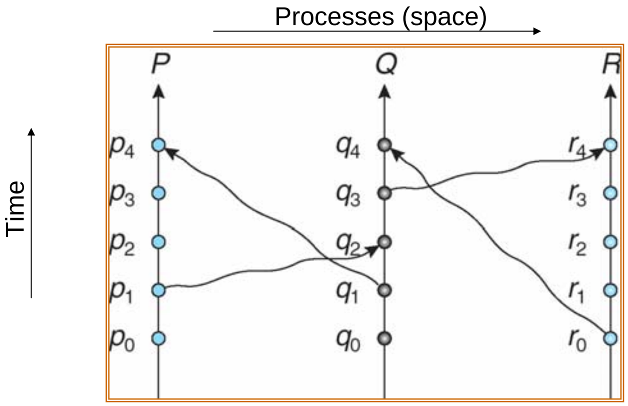
18.1

Event Ordering

- *Happened-before* relation (denoted by →)
  - If *A* and *B* are events in the same process, and *A* was executed before *B*, then *A* → *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* → *B*
  - If *A* → *B* and B → *C* then *A* → *C (transitive)*
- Irreflexive relation: since an event can not be happened-before itself
- If two events are not related by → relation, they are concurrent
- If A → B then A can affect B

Relative Time for Three Concurrent Processes

- Space-time diagram



Implementation of →

- Associate a timestamp with each system event
  - Require that for every pair of events A and B, if A → B, then the timestamp of A is less than the timestamp of B
- Within each process Pi a logical clock (LCi) is associated
  - The logical clock can be implemented as a simple counter that is incremented between any two successive events executed within a process
    - Logical clock is **monotonically increasing**
- 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

---

18.2

Mutual Exclusion (ME) in DS

- 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 three algorithms to ensure the mutual exclusion execution of processes in their critical sections

---

ME: Centralized Approach

- One of the processes in the system is chosen to coordinate the entry to the critical section (CS)
- A process that wants to enter its CS sends a request message to the coordinator
- The coordinator decides which process can enter the CS next, and its sends that process a reply message
- When the process receives a reply message from the coordinator, it enters its CS
- After exiting its CS, the process sends a release message to the coordinator and proceeds with its execution
- No starvation if the coordinator scheduling policy is fair (e.g. FCFS)
- Requires three messages per CS entry: request, reply, and release
- Upon failure of the coordinating process, a new process must be elected as a coordinator, poll all processes to construct request queue

---

ME: Fully Distributed Approach

- When process P_i wants to enter its CS, 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 CS
- After exiting its CS, the process sends reply messages to all its deferred requests

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 CS, then it defers its reply to P_i
  - If P_j does *not* want to enter its CS, then it sends a *reply* immediately to P_i
  - If P_j wants to enter its CS 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

Fully Distributed Approach (Cont.)

- Desirable Behavior
  - Mutual exclusion is obtained
  - Freedom from deadlock is ensured
  - Freedom from starvation is ensured, since entry to the CS is scheduled according to the timestamp ordering
    - Which ensures that processes are served in a FCFS order
  - The number of messages per CS entry is

    $$2 \times (n - 1)$$

    the minimum number of required messages per CS entry when processes act independently and concurrently

Fully Distributed Approach (Cont.)

- Three Undesirable Consequences
  - 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; if one process fails, all others are notified
  - Processes that have not entered their CS must pause frequently to assure other processes that they intend to enter the CS
- This protocol is therefore suited for small, stable sets of cooperating processes

ME: Token-Passing Approach

- Circulate a token among processes in system
  - Token is special type of message
  - Possession of token entitles holder to enter critical section
- Processes are logically organized in a ring structure
- Unidirectional ring guarantees freedom from starvation
- Number of messages per CS entry may vary
- Two types of failures
  - Lost token – election must be called
  - Failed processes – new logical ring established

18.5

Deadlock Prevention and Avoidance

- 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 for deadlock avoidance
  - designate one of the processes in the system as the process that maintains the information necessary to carry out the Banker's algorithm (banker)
  - also implemented easily, but may require too much overhead

---

New Time-stamped Deadlock-Prevention Techniques

- 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 (if it has a higher priority); otherwise P_i is rolled back (dies)
- 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
- Starvation is possible → use timestamp to avoid it
- Two complementary deadlock prevention using timestamps
  - Wait-Die Scheme
  - Wound-Wait Scheme

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 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

---

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 (P_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 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

---

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

---

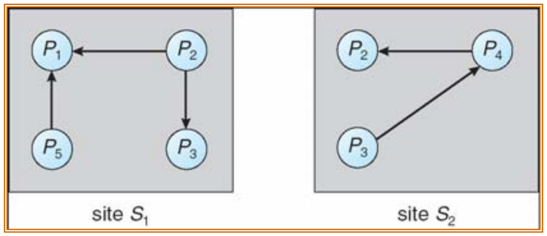
The Algorithm

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 Pi → Pj if and only if
     - (1) there is an edge Pi → Pj in one of the wait-for graphs, or
     - (2) an edge Pi → Pj with some label TS appears in more than one wait-for graph

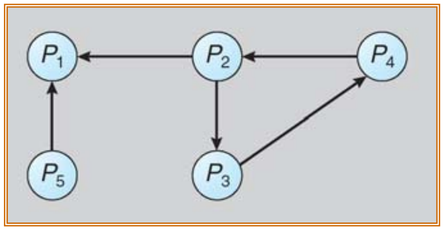
If the constructed graph contains a cycle ⇒ deadlock

Example

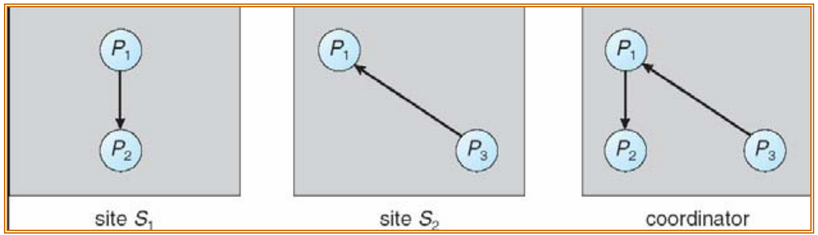
Two Local Wait-For Graphs



Global Wait-For Graph



False Cycles & Unnecessary Rollbacks

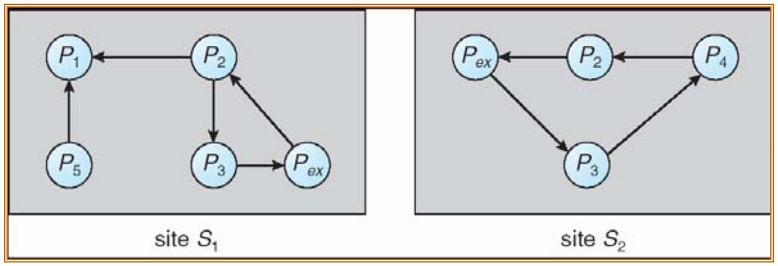


- Suppose p2 releases the resource it is holding at S1
- The edge p1 → p2 is removed from the local wait-for graph at S1
- Then P2 request a resource held by P3 at S2
- Edge p2 → p3 is added at S2
- If the add message is arrived before the delete at the coordinator, a cycle is detected (which is false)

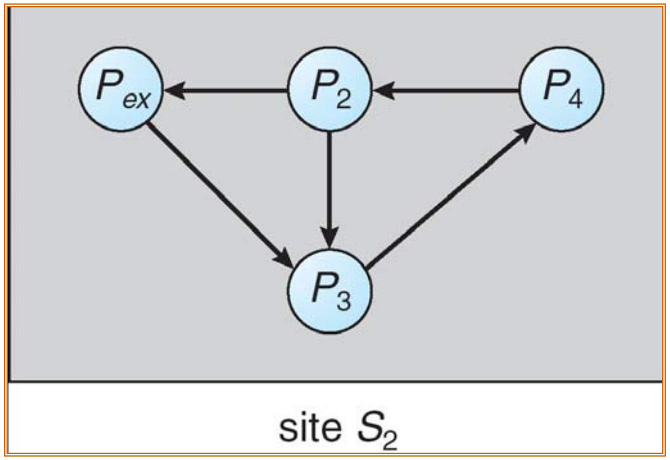
Fully Distributed Approach

- 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
- 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

Augmented Local Wait-For Graphs



Augmented Local Wait-For Graph in Site S2



18.6

Election Algorithms

- Determine where a new copy of the coordinator should be restarted
  - can be used to elect a new coordinator in case of failures
- Assume that a unique priority number is associated with each active process in the system,
  - assume the priority number of process P_i is i
- Assume a one-to-one correspondence between processes and sites
- The coordinator is always the process with the largest priority number. If a coordinator fails, the algorithm must elect that active process with the largest priority number
- Election algorithms,
  - the bully algorithm
  - the ring algorithm

The 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 them to answer within T

The Bully Algorithm (Cont.)

- 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, P_i 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

The 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 algorithm, provided that *Pi* 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 number to let it become the coordinator process, even if there is a currently active coordinator with a lower number

The 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 Pi detects a coordinator failure, I 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

---

The 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 *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 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
3. If $i = j$, then P_i receives the message *elect(i)*
   - The active list for P_i contains all the active processes in the system
     - P_i can now determine the new coordinator process.

Selected Topics of Chapter 16 & 18

Operating System Concepts, 7th Ed. A. Siblerschatz, P. Galvin, and G. Gagne. Addison Wesley, 2005