Distributed Operating Systems Issues
Chapters 16 and 18
Objectives

- To provide a high-level overview of distributed systems
- To discuss the general structure of distributed operating systems
- To describe various methods for achieving mutual exclusion in a distributed system
- To present schemes for handling deadlock prevention, deadlock avoidance, and deadlock detection in a distributed system
- To present distributed algorithms used in case of failure
Outline

- Motivation (16.1)
- Types of Distributed Operating Systems (16.2)
- Event Ordering (18.1)
- Mutual Exclusion (18.2)
- Deadlock Handling (18.5)
- Election Algorithms (18.6)
- Motivation …

- **Distributed system** is a collection of loosely coupled processors interconnected by a communications network.

- Processors variously called *nodes, computers, machines, hosts*
  - *Site* is the location of the processor.
Reasons for distributed systems

- Resource sharing
  - sharing and printing files at remote sites
  - processing information in a distributed database
  - using remote specialized hardware devices

- Computation speedup – load sharing

- Reliability – detect and recover from site failure, function transfer, reintegrate failed site

- Communication – message passing
- 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 the File Transfer Protocol (FTP) mechanism

- Distributed Operating Systems
  - Users not aware of multiplicity of machines
    - Access to remote resources similar to access to local resources
  - 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

- 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
- Event Ordering

- 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 $A \rightarrow C$
-- Relative Time for Three Concurrent Processes
-- 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 \( p_i \) 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
    - 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
- Distributed Mutual Exclusion (DME)

- **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
  - Centralized approach
  - Fully distributed 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
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.
The decision whether process $P_j$ replies immediately to a $\text{request}(P_i, TS)$ 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
Desirable Behavior of 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
  \[ 2 \times (n - 1) \]

This is the minimum number of required messages per critical-section entry when processes act independently and concurrently.
-- 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.

- 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.
-- 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 *logically* organized in a **ring structure**

- Algorithm similar to Chapter 6 algorithm 1 but token substituted for shared variable

- Unidirectional ring guarantees freedom from starvation

- Two types of failures
  - Lost token - election must be called
  - Failed processes - new logical ring established
- Deadlock Handling

The following 3 deadlock algorithms presented in Chapter 7 can be used with distributed systems, provided that appropriate modifications are made:

- Avoidance
  - Banker’s Algorithm

- Prevention

- Detection and recovery
-- Deadlock Avoidance

- **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
    - Every resource request must be channeled through the designated process.
    - Also implemented easily, but may require too much overhead
  
  - Not practical because the designated process may become a bottleneck due to excessive messages that it has to process
Deadlock Prevention

- Resource-ordering deadlock-prevention Scheme
- Time stamped Deadlock-Prevention Scheme
  - Wait-Die Scheme
  - Would-Wait Scheme
--- Resource-ordering deadlock-prevention Scheme

- 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.
-- Time stamped Deadlock-Prevention Scheme

- 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 $P_j$; 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
- Disadvantage - starvation
  - Solution :- priorities based on timestamps
    - Wait-die scheme (nonpreemptive)
    - Wound-wait scheme (preemptive)
--- Wait-Die Scheme

- 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)

- In short, if the requesting process is:
  - Old: waits
  - Young: 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
--- Would-Wait Scheme

- 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 $P_i$).

- In short, if the requesting process:
  - young: wait
  - old: never waits—wounds the young

- 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
--- Both (Wait-die and Wound-wait) schemes

- In Wait-die
  - Older waits for younger
  - Younger is not allowed to wait (Killed)

- In Wound-wait
  - Older never waits for younger
  - Younger is allowed to wait

- In both schemes unnecessary rollback can occur

- Both schemes can avoid starvation provided that, when a process is rolled back its timestamp doesn’t change.
-- Deadlock Detection

- Centralized Approach

- Fully Distributed Approach
--- 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
Local and Global Wait-For Graphs

- Site $S_1$:
  - $P_1 \rightarrow P_2$

- Site $S_2$:
  - $P_1 \rightarrow P_3$

- Coordinator:
  - $P_1 \rightarrow P_2 \rightarrow P_3$
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

With options 1 and 2, Unnecessary rollbacks may occur as a result of false cycles
… --- Centralized Approach …

- 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
Centralized Approach: Option 3-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
--- 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

Site $S_1$

Site $S_2$
---- Augmented Local Wait-For Graph in Site S2

site $S_2$
- Election Algorithms

- Determine where a new copy of the coordinator should be restarted

- 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 $i$

- Assume a one-to-one correspondence between processes and sites

- 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 algorithms, the bully algorithm and a ring algorithm, can be used to elect a new coordinator in case of failures
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 $T$. 
… -- Bully Algorithm …

- 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.
... -- 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 number to let it become the coordinator process, even if there is a currently active coordinator with a lower number
-- 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
...-- Ring Algorithm

- If $P_i$ receives a message $\text{elect}(j)$ from the process on the left, it must respond in one of three ways:

1. If this is the first $\text{elect}$ message it has seen or sent, $P_i$ creates a new active list with the numbers $i$ and $j$
   - It then sends the message $\text{elect}(i)$, followed by the message $\text{elect}(j)$
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 $\text{elect}(i)$
   - The active list for $P_i$ contains all the active processes in the system
   - $P_i$ can now determine the new coordinator process.
End of Chapter 16 and 18