

# Chapter 18: Distributed Coordination





# Chapter 18 Distributed Coordination

- Event Ordering
- Mutual Exclusion
- Atomicity
- Concurrency Control
- Deadlock Handling
- Election Algorithms
- Reaching Agreement





# Chapter Objectives

- To describe various methods for achieving mutual exclusion in a distributed system
- To explain how atomic transactions can be implemented in a distributed system
- To show how some of the concurrency-control schemes discussed in Chapter 6 can be modified for use in a distributed environment
- To present schemes for handling deadlock prevention, deadlock avoidance, and deadlock detection in a distributed system





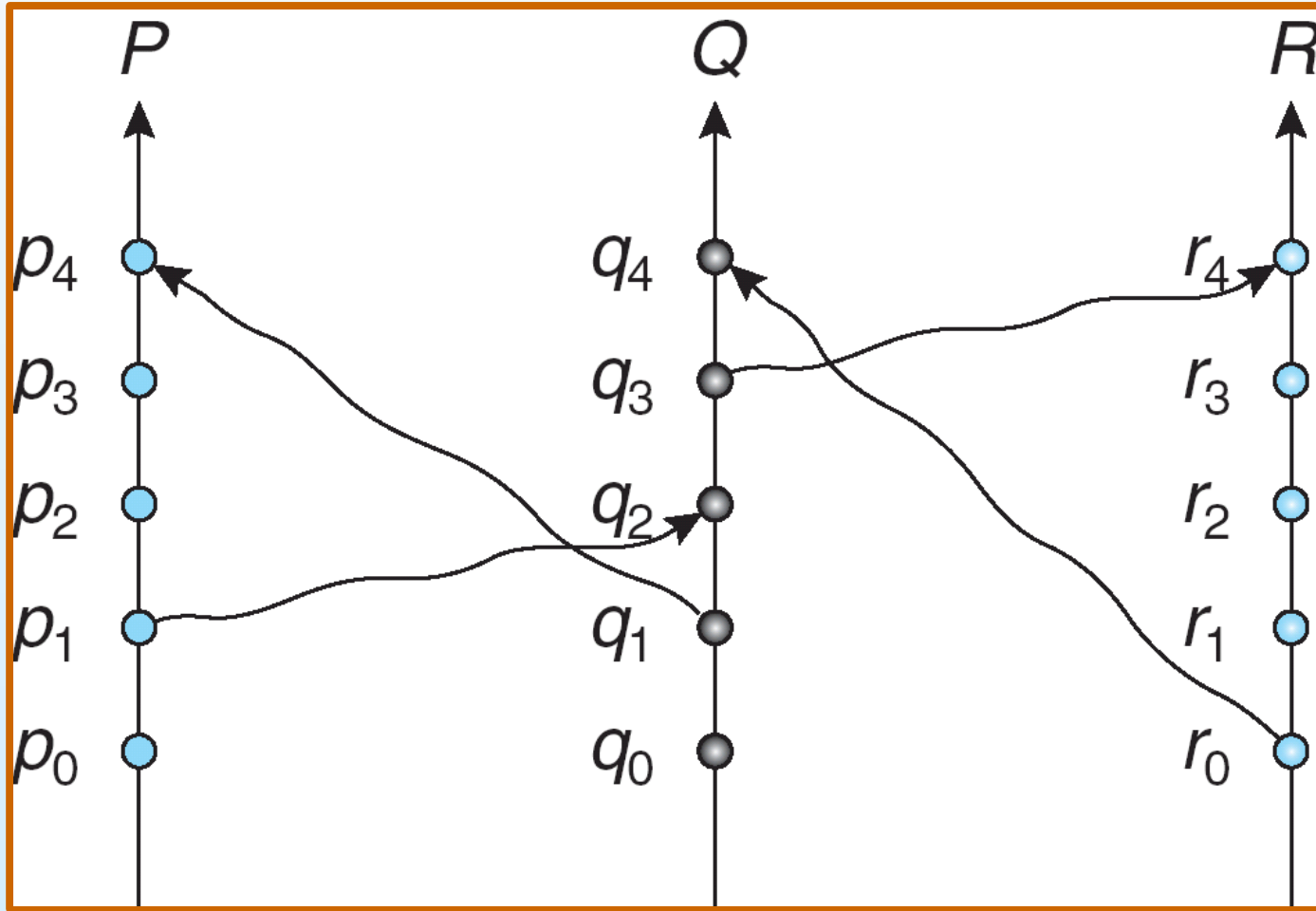
# Event Ordering

- *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 $\rightarrow$

- 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
    - ▶ 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





# DME: 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







# DME: 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





# DME: 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





# 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





# 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 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





# 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  $S_i$  be  $C_i$





# Phase 1: Obtaining a Decision

- $C_i$  adds  $\langle \text{prepare } T \rangle$  record to the log
- $C_i$  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  $C_i$  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  $C_i$







# 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*





# 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





# Failure Handling in 2PC – Site Failure

- The log contains a  $\langle \text{commit } T \rangle$  record
  - In this case, the site executes **redo**( $T$ )
- The log contains an  $\langle \text{abort } T \rangle$  record
  - In this case, the site executes **undo**( $T$ )
- The log contains a  $\langle \text{ready } T \rangle$  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**( $T$ )





# Failure Handling in 2PC – Coordinator $C_i$ Failure

- If an active site contains a  $\langle \text{commit } T \rangle$  record in its log, the  $T$  must be committed
- If an active site contains an  $\langle \text{abort } T \rangle$  record in its log, then  $T$  must be aborted
- If some active site does *not* contain the record  $\langle \text{ready } T \rangle$  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  $\langle \text{ready } T \rangle$  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  $S_i$





# 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





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





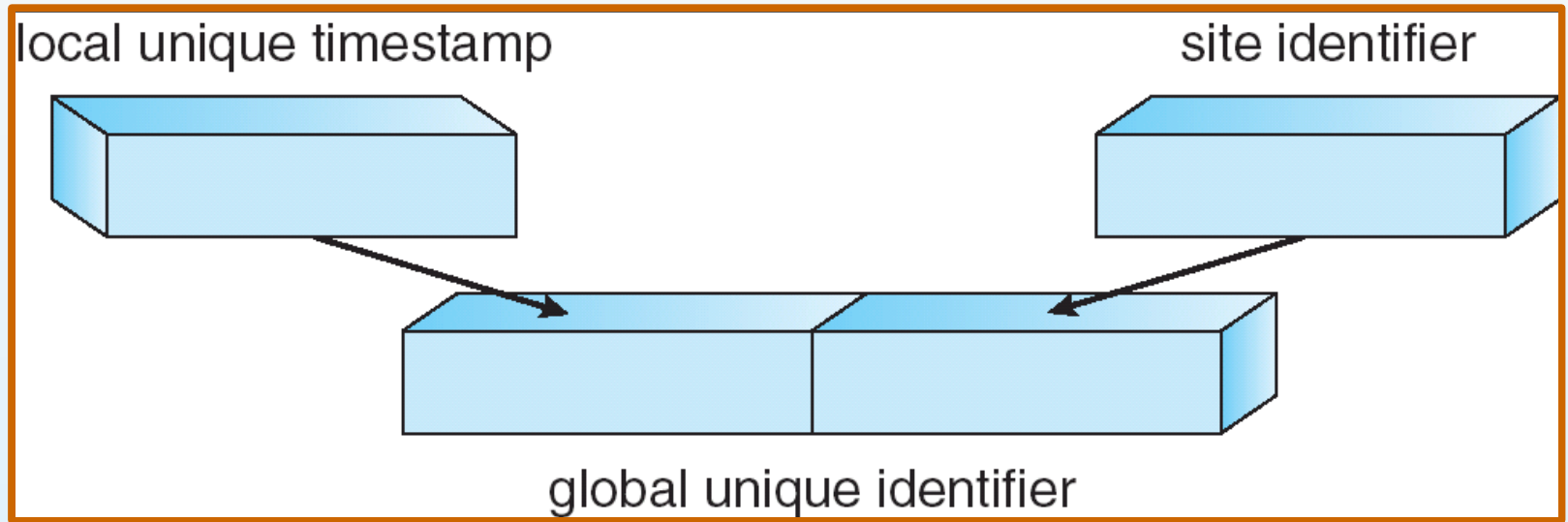
# Timestamping

- Generate unique timestamps in distributed scheme:
  - Each site generates a unique local timestamp
  - 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
  
- 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





# Generation of Unique Timestamps





# Deadlock Prevention

- 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





# Timestamped 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





# 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  $t$ , 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

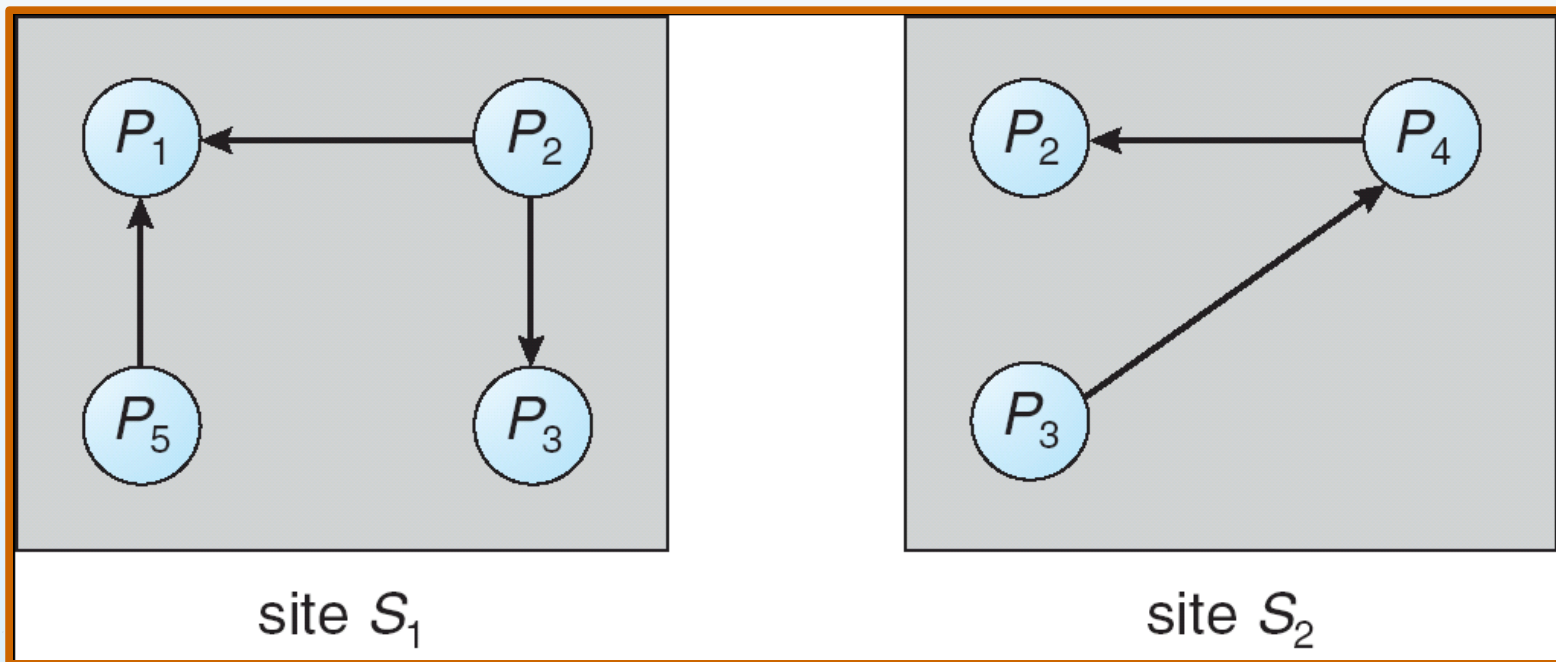
- Based on a preemptive technique; counterpart to the wait-die system
- If  $P_i$  requests a resource currently held by  $P_j$ ,  $P_i$  is allowed to wait only if it has a larger timestamp than does  $P_j$  ( $P_i$  is younger than  $P_j$ ). Otherwise  $P_j$  is rolled back ( $P_j$  is wounded by  $P_i$ )
- 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





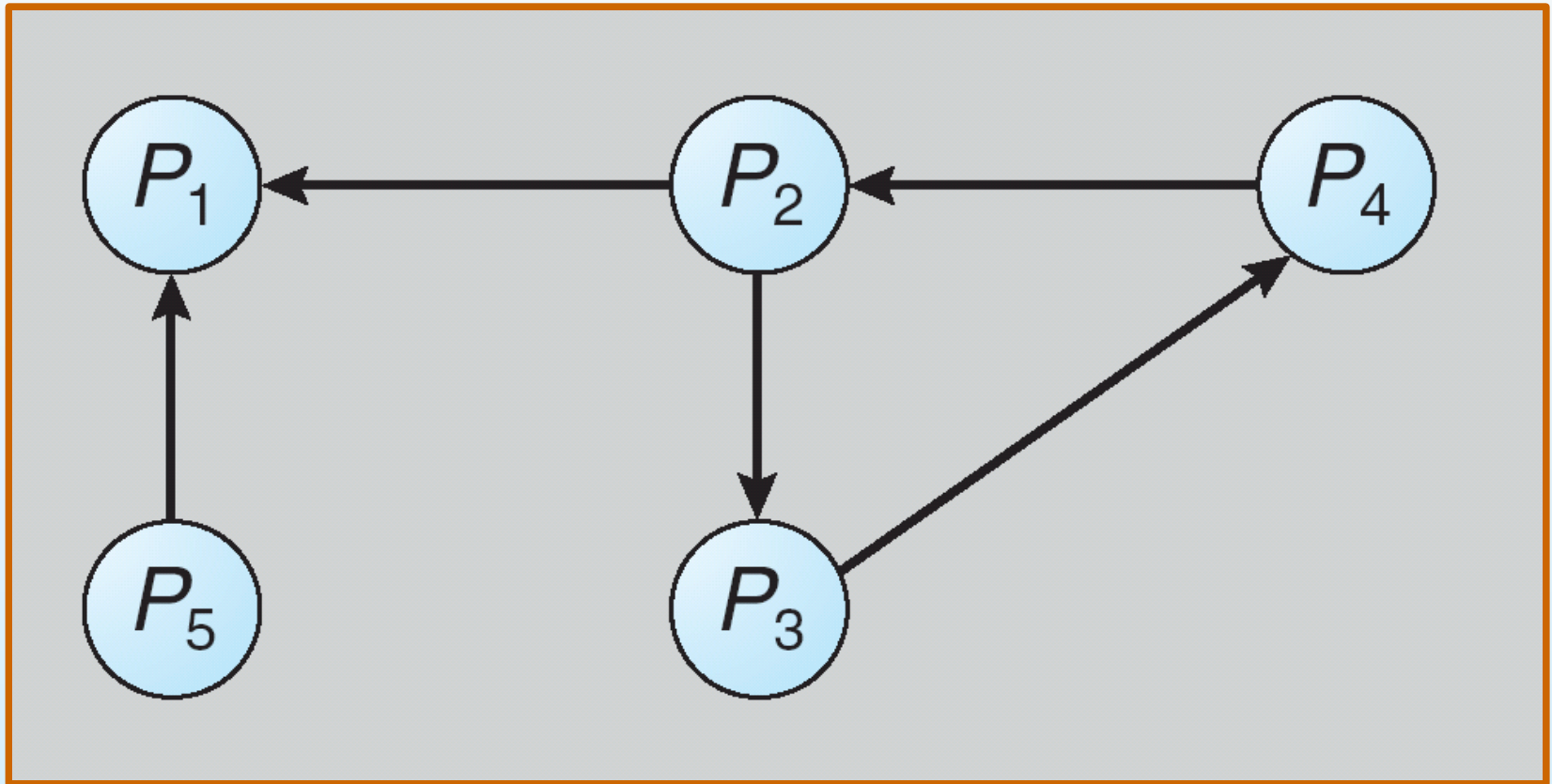


# Two Local Wait-For Graphs





# Global Wait-For Graph





# 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

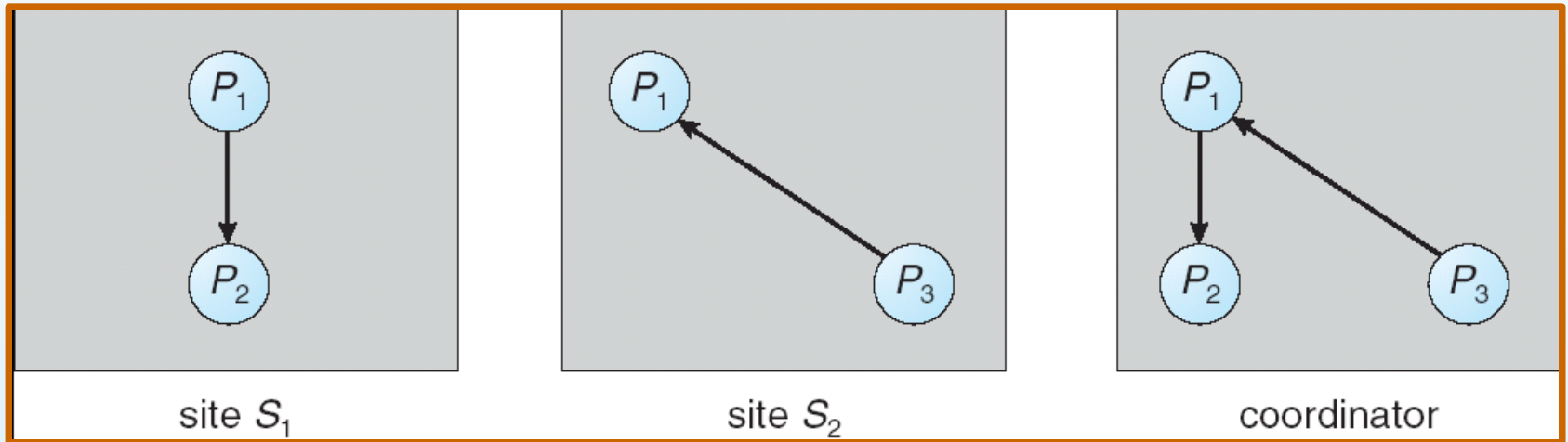
1. The controller sends an initiating message to each site in the system
2. On receiving this message, a site sends its local wait-for graph to the coordinator
3. When the controller has received a reply from each site, it constructs a graph as follows:
  - (a) The constructed graph contains a vertex for every process in the system
  - (b) The graph has an edge  $P_i \rightarrow P_j$  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





# Local and Global Wait-For Graphs





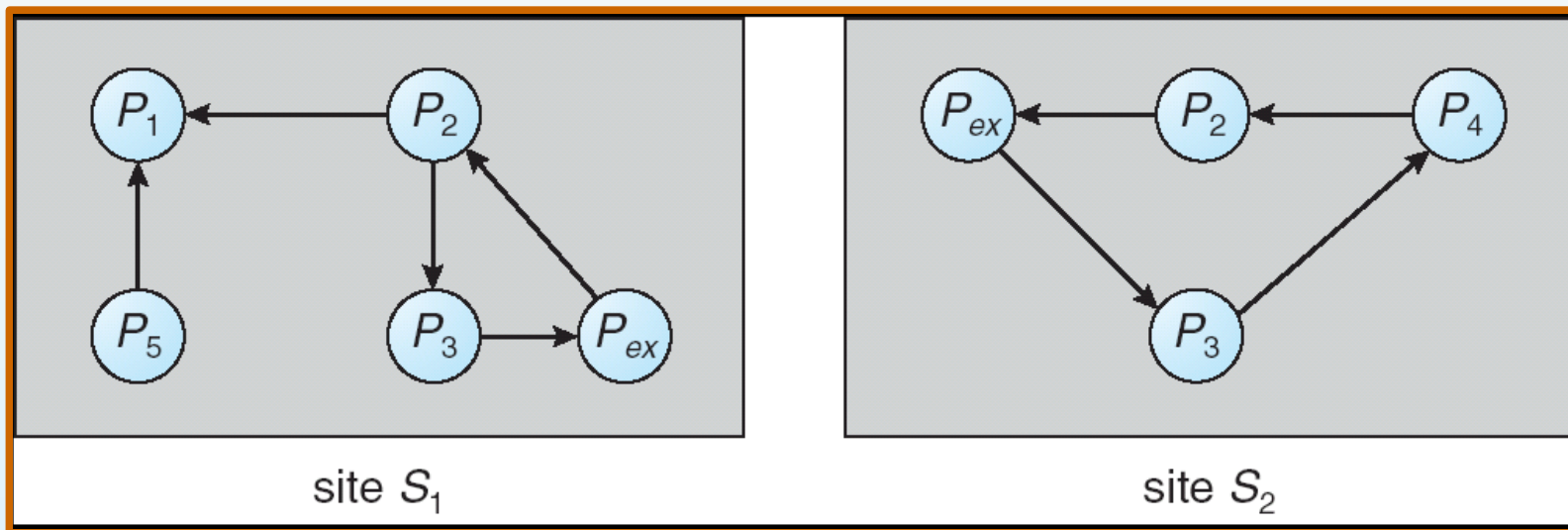
# Fully Distributed Approach

- 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





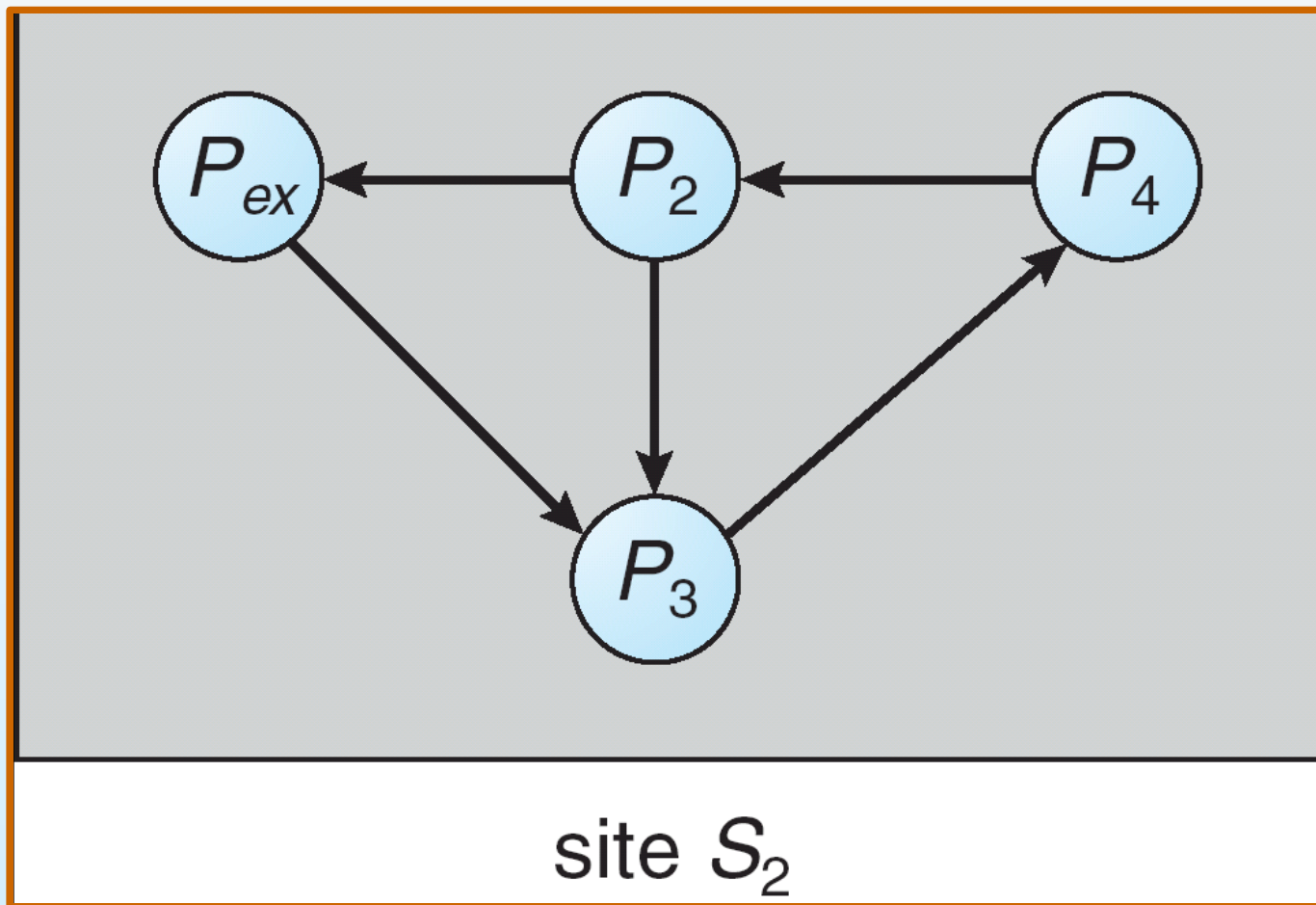
# Augmented Local Wait-For Graphs







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





# 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





# 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  $T$





# 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





# 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  $P_i$  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 (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.





# Reaching Agreement

- There are applications where a set of processes wish to agree on a common “value”
- Such agreement may not take place due to:
  - Faulty communication medium
  - Faulty processes
    - ▶ Processes may send garbled or incorrect messages to other processes
    - ▶ A subset of the processes may collaborate with each other in an attempt to defeat the scheme







# Faulty Communications

- Process  $P_i$  at site  $A$ , has sent a message to process  $P_j$  at site  $B$ ; to proceed,  $P_i$  needs to know if  $P_j$  has received the message
- Detect failures using a time-out scheme
  - When  $P_i$  sends out a message, it also specifies a time interval during which it is willing to wait for an acknowledgment message from  $P_j$
  - When  $P_j$  receives the message, it immediately sends an acknowledgment to  $P_i$
  - If  $P_i$  receives the acknowledgment message within the specified time interval, it concludes that  $P_j$  has received its message
    - ▶ If a time-out occurs,  $P_i$  needs to retransmit its message and wait for an acknowledgment
  - Continue until  $P_i$  either receives an acknowledgment, or is notified by the system that  $B$  is down





# Faulty Communications (Cont.)

- Suppose that  $P_j$  also needs to know that  $P_i$  has received its acknowledgment message, in order to decide on how to proceed
  - 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 respective states





# Faulty Processes (Byzantine Generals Problem)

- Communication medium is reliable, but processes can fail in unpredictable ways
- Consider a system of  $n$  processes, of which no more than  $m$  are faulty
  - Suppose that each process  $P_i$  has some private value of  $V_i$
- Devise an algorithm that allows each nonfaulty  $P_i$  to construct a vector  $X_i = (A_{i,1}, A_{i,2}, \dots, A_{i,n})$  such that:
  - If  $P_j$  is a nonfaulty process, then  $A_{ij} = V_j$ .
  - If  $P_i$  and  $P_j$  are both nonfaulty processes, then  $X_i = X_j$ .
- Solutions share the following properties
  - 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





# Faulty Processes (Cont.)

- An algorithm for the case where  $m = 1$  and  $n = 4$  requires two 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
- 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  $X_i = (A_{i,1}, A_{i,2}, A_{i,3}, A_{i,4})$  as follows:
  - $A_{i,j} = V_i$
  - 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



# End of Chapter 18

