# 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 <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 $C_i$.



# 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

  <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 – 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 contains a <ready *T*> 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 <commit *T*> record in its log, the *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.
  - Blocking problem – *T* is blocked pending the recovery of site $S_i$.




## 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 grater 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

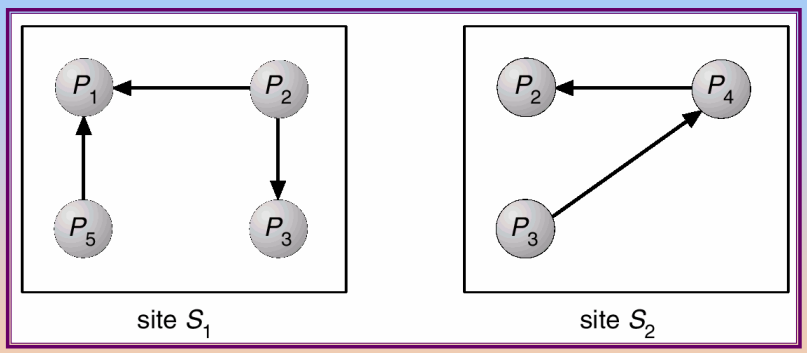
- Based on a preemptive technique; counterpart to the wait-die system.
- If $P_i$ requests a resource currently held by $P_j$, $P_i$ is allowed to wait only if it has a larger timestamp than does $P_j$ ($P_i$ is younger than $P_j$). Otherwise $P_j$ is rolled back ($P_j$ is wounded by $P_i$).
- Example: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



## Deadlock Detection – Centralized Approach

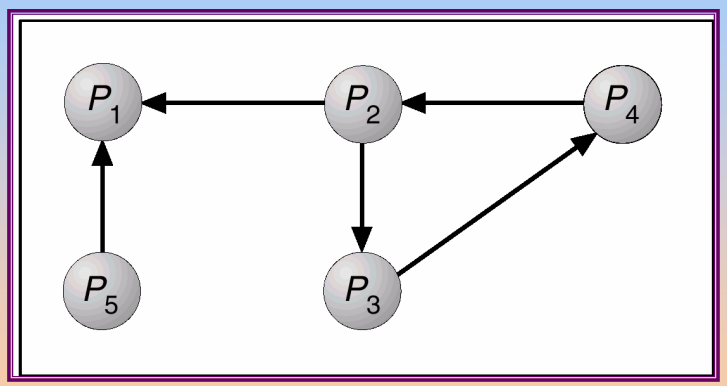
- Each site keeps a *local* wait-for graph. The nodes of the graph correspond to all the processes that are currently either holding or requesting any of the resources local to that site.
- A global wait-for graph is maintained in a *single* coordination process; this graph is the union of all local 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*.

## Two Local Wait-For Graphs






## Global Wait-For Graph



## Detection Algorithm Based on Option 3

- Append unique identifiers (timestamps) to requests form 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.

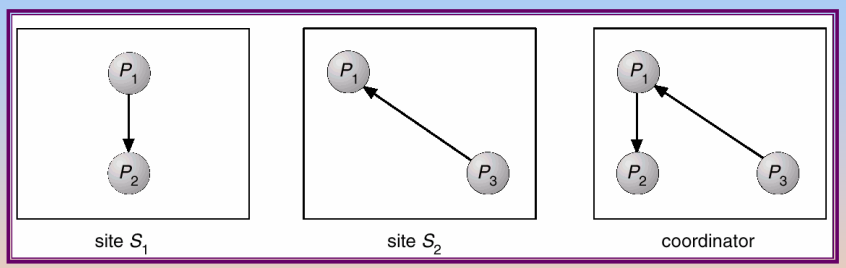



## The Algorithm

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

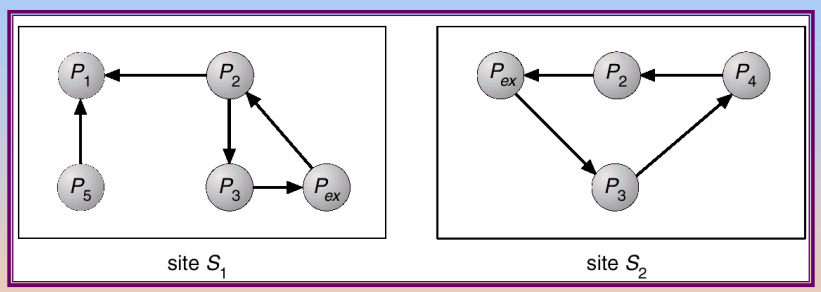





# Fully Distributed Approach

- 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 S2