COP 4600 – Summer 2010

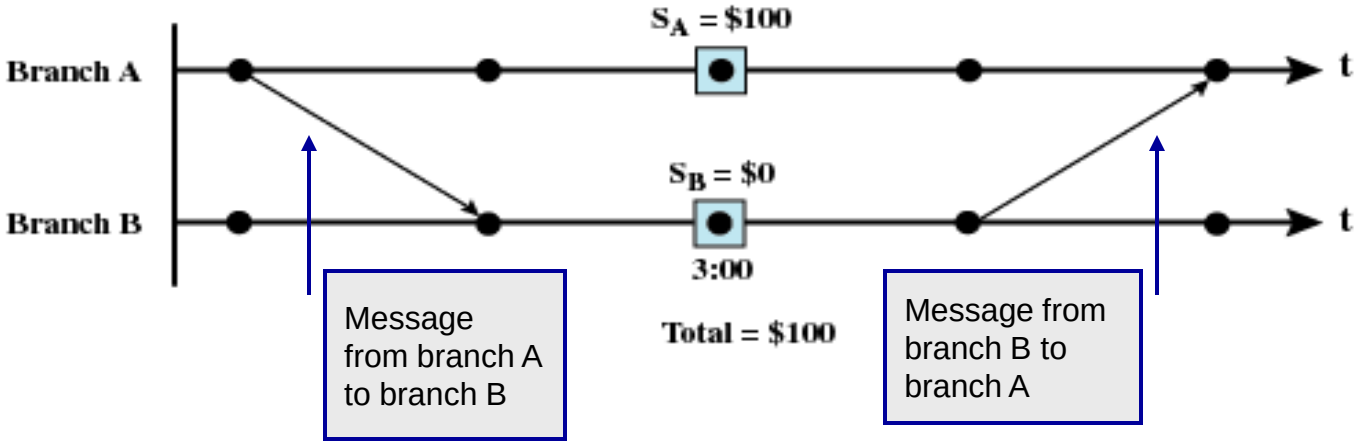
Introduction To Operating Systems

Distributed Process Management – Part 2

Instructor : Dr. Mark Llewellyn
markl@cs.ucf.edu
HEC 236, 407-823-2790
http://www.cs.ucf.edu/courses/cop4600/sum2010

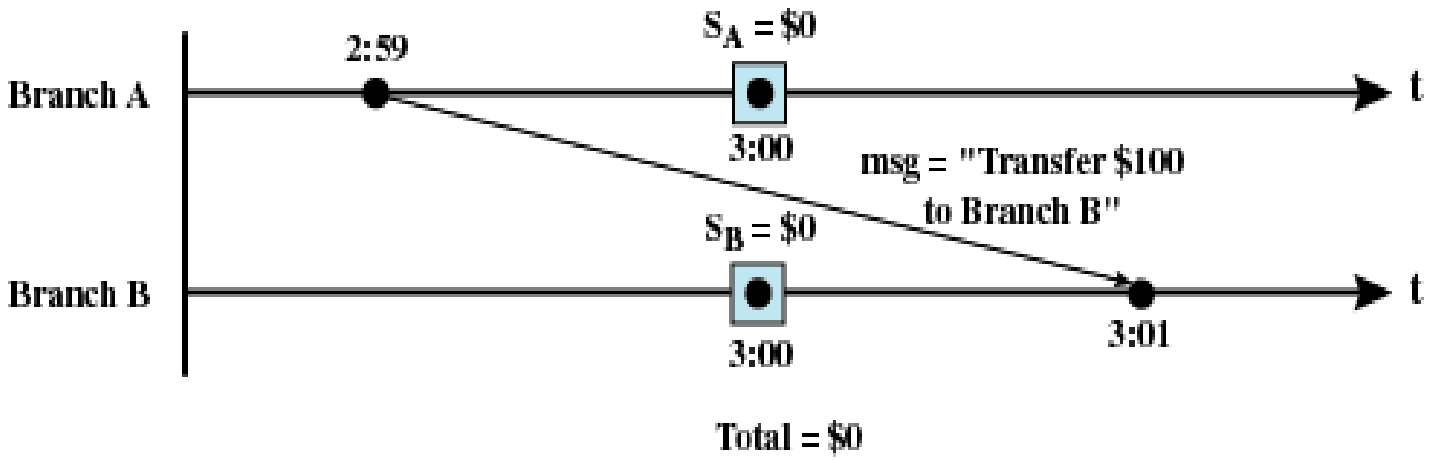
School of Electrical Engineering and Computer Science
University of Central Florida

Example Scenarios For Bank Example Process/Event Graph



One possible scenario – in this case reported account balance is correct

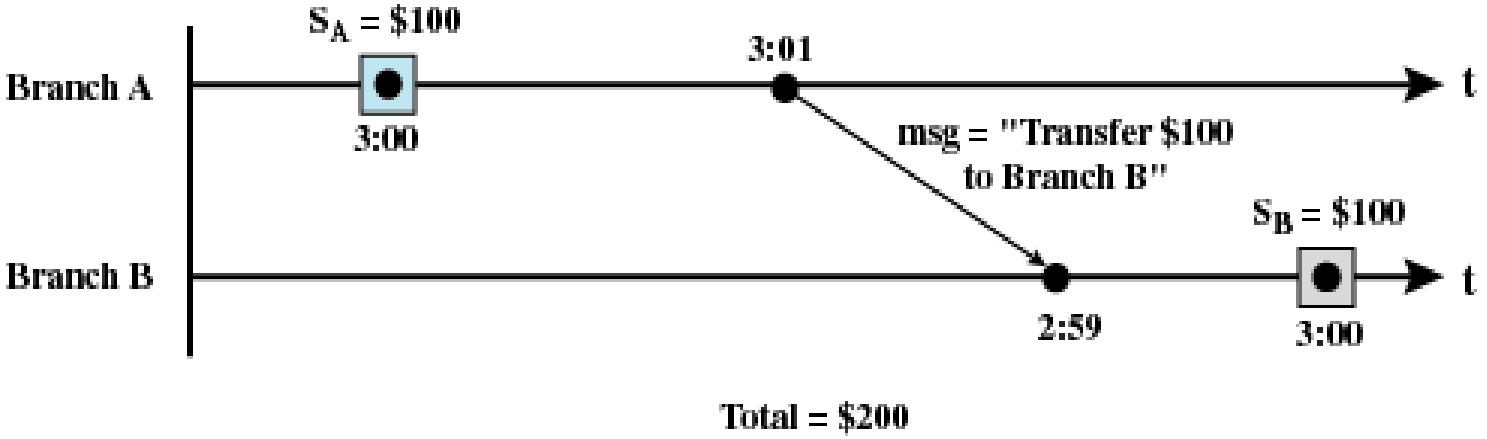
Example Scenarios For Bank Example



If at the time of balance determination, the balance from branch A is in transit to branch B. In this case balance determined at 3:00 pm is incorrect.

All messages in transit must be examined at time of observation. The correct total consists of balance at both branches and amount in any message in transit.

Example Scenarios For Bank Example



If the clocks at the two branches are not perfectly synchronized a problem can arise. Suppose that a transfer message is initiated at branch A at local time 3:01 pm. This message arrives at branch B at 2:59 local time. The balance calculated at 3:00 pm will now show the incorrect amount of $200. The amount is incorrectly counted twice.

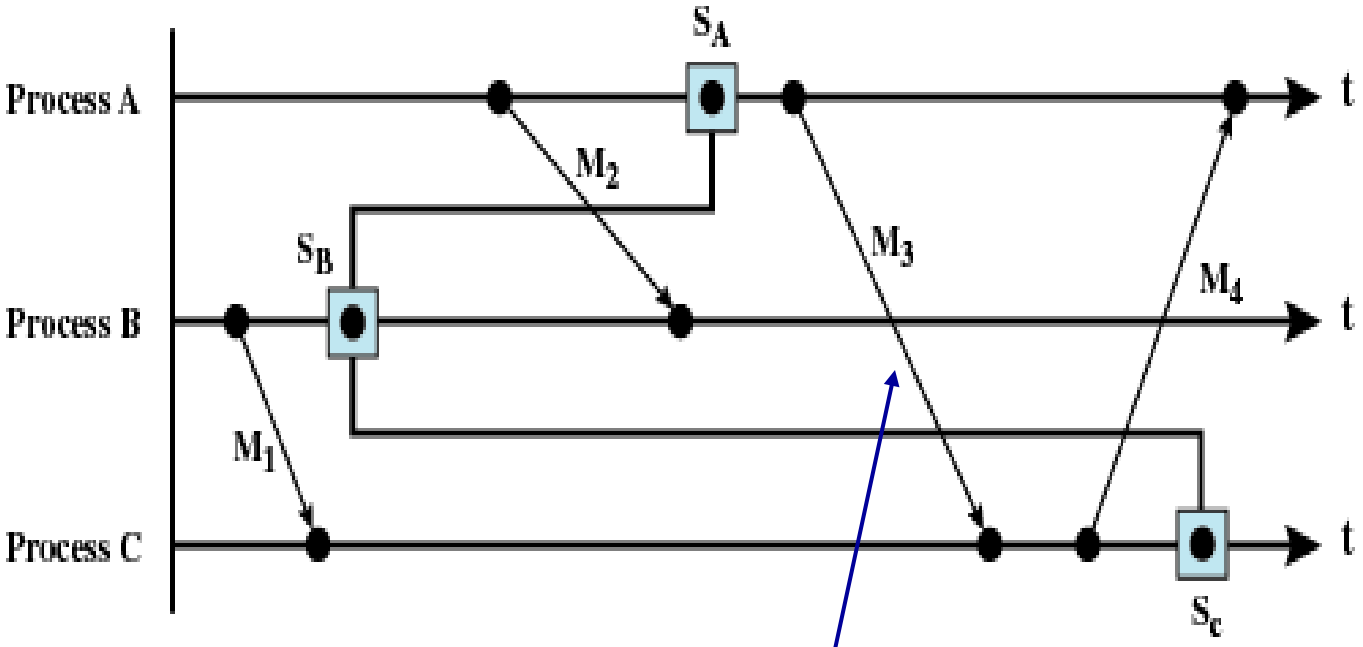
Some Terms

- Channel
  - Exists between two processes if they exchange messages.
- State
  - Sequence of messages that have been sent and received along channels incident with the process.
- Snapshot
  - Records the state of a process.
- Global state
  - The combined state of all processes.
- Distributed Snapshot
  - A collection of snapshots, one for each process.

Global States and Distributed Snapshots

- The problem with a distributed system is that a true global state cannot be determined because of the time lapse associated with message transfer.
- We can attempt to define a global state by collecting snapshots from all processes.
- For example, in the figure on the next page, at the time of taking the snapshot, there is a message in transit on the <A,B> channel (message 2), one in transit on the <A,C> channel (message 3), and one in transit on the <C,A> channel (message 4). Messages 2 and 4 are properly represented, however, message 3 is not.
  - The distributed snapshot indicates that message 3 has been received but not yet sent!

An Inconsistent Global State



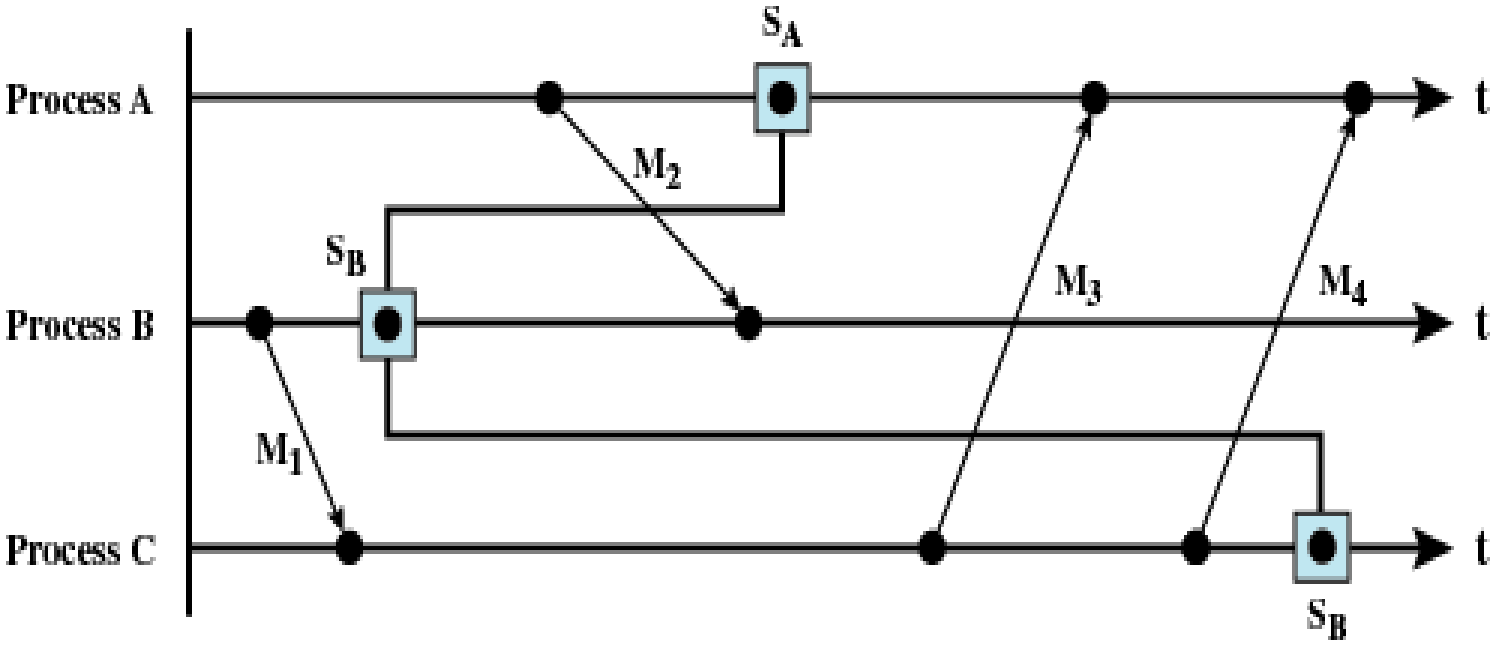
Global snapshot indicates message 3 has not yet been sent, but in fact has been received. Snapshot of Process A shows no record of sending message 3, snapshot of Process C indicates receiving message 3!

Global State
Snapshot process A: Message 2 sent
Snapshot process B: Message 1 sent
Snapshot process C: Message 1 received
Message 3 received
Message 4 sent

Consistent Global States

- We need the distributed snapshot to record a consistent global state.
- A global state is consistent if for every process state that records the receipt of a message, the sending of that message is recorded in the process state of the process that sent the message.
- In the previous slide, the global state was inconsistent because process C has recorded the receipt of message 3, but no process has a record of having sent message 3.
- In contrast, the process/event graph on the next page illustrates a consistent global state.

A Consistent Global State



Global State
Snapshot process A: Message 2 sent
Snapshot process B: Message 1 sent
Snapshot process C: Message 1 received
Message 3 sent
Message 4 sent

Distributed Snapshot Algorithm

- Several different algorithms which record a consistent global state have been developed. We'll examine a fairly popular one as follows:
- The algorithm assumes that messages are delivered in the order that they are sent and no messages are lost. (A reliable transport protocol such as TCP satisfies these requirements.)
- The algorithm uses a special control message, called a marker.
- Some process initiates the algorithm by recording its state and sending a marker on all outgoing channels before any more messages are sent.
- Each process *p* then proceeds as follows. Upon the first receipt of the marker (say from process *q*), receiving process *p* performs the following:
  1. Process *p* records its local state.
  2. Process *p* records the state of the incoming channel from *q* to *p* as empty.
  3. Process *p* propagates the marker to all of its neighbors along all outgoing channels.

Distributed Snapshot Algorithm (cont.)

- The previous three steps must be performed atomically; i.e., no messages can be sent or received by process *p* until all three steps are performed.
- At any time after recording its state, when process *p* receives a marker from another incoming channel (say from process *r*), it performs the following action:

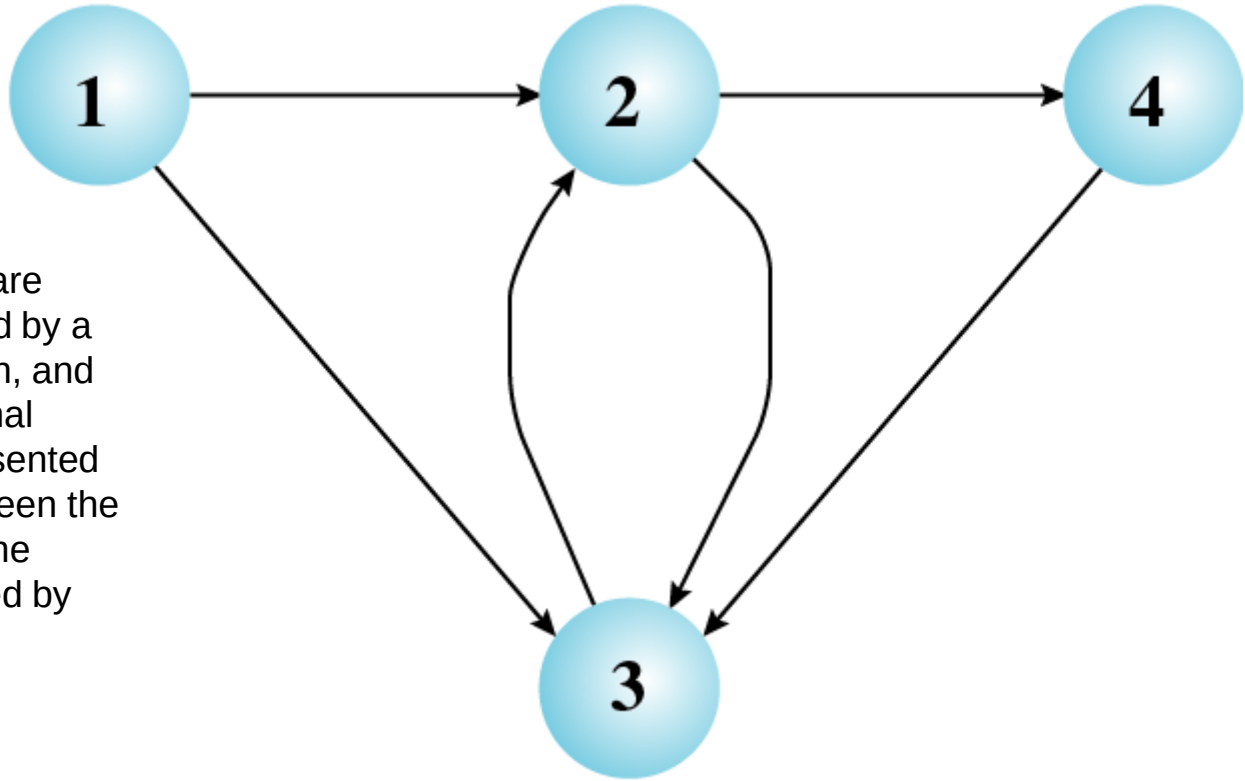
  1. Process *p* records the state of the channel from *r* to *p* as the sequence of messages process *p* has received from process *r* from the time process *p* recorded its local state S_p to the time it received the marker from process *r*.

- The algorithm terminates at a process once the marker has been received along every incoming channel.

Distributed Snapshot Algorithm (cont.)

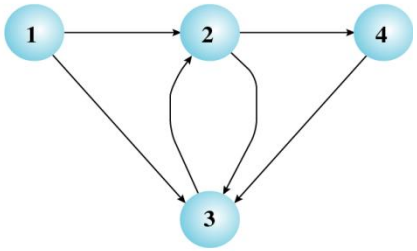
- The following points can be made about this algorithm:
  - Since any process may start the algorithm by sending out a marker, if several nodes independently decided to record their state and send out the marker, the algorithm will still work properly.
  - The algorithm will terminate in a finite amount of time, if every message is delivered in finite time.
  - Since this is a distributed algorithm, each process is responsible for recording its own state and the state of all incoming channels.
  - Once all of the states have been recorded (the algorithm has terminated at all processes), the consistent global state obtained by the algorithm can be assembled by having every process send the state data that it has recorded along every outgoing channel and having every process forward the state data that it receives along every outgoing channel. Alternatively, the initiating process could poll all processes to acquire the global state.

Distributed Snapshot Algorithm - Example



Four processes are each represented by a node in the graph, and each unidirectional channel is represented by an edge between the two nodes with the direction indicated by the arrowhead.

Distributed Snapshot Algorithm - Example



Process 1 initiates after sending 6 messages

Process 3 sent 8 messages on its outgoing channel prior to recording its state. Process 3 received 3 messages from process 1 before recording its state, leaving messages 4, 5, and 6 to be associated with the channel. Process 3 received 3 messages from process 2 recording its state, leaving message 4 to be associated with the channel. Process 3 received 3 messages from process 4.

Process 1

Outgoing channels
2 sent 1, 2, 3, 4, 5, 6
3 sent 1, 2, 3, 4, 5, 6
Incoming channels

Process 3

Outgoing channels
2 sent 1, 2, 3, 4, 5, 6, 7, 8
Incoming channels
1 received 1, 2, 3 stored 4, 5, 6
2 received 1, 2, 3 stored 4
4 received 1, 2, 3

Process 2

Outgoing channels
3 sent 1, 2, 3, 4
4 sent 1, 2, 3, 4
Incoming channels
1 received 1, 2, 3, 4 stored 5, 6
3 received 1, 2, 3, 4, 5, 6, 7, 8

Process 4

Outgoing channels
3 sent 1, 2, 3
Incoming channels
2 received 1, 2 stored 3, 4

Process 4 initiates after sending 3 messages

Process 2 sent four messages on each of the two outgoing channels prior to recording its states. Process 2 received 4 messages from process 1 before recording its state, leaving messages 5 and 6 to be associated with the channel.

Distributed Snapshot Algorithm - Example

Process 1 Outgoing channels 2 sent 1, 2, 3, 4, 5, 6 3 sent 1, 2, 3, 4, 5, 6 Incoming channels	**Process 3** Outgoing channels 2 sent 1, 2, 3, 4, 5, 6, 7, 8 Incoming channels 1 received 1, 2, 3 stored 4, 5, 6 2 received 1, 2, 3 stored 4 4 received 1, 2, 3
Process 2 Outgoing channels 3 sent 1, 2, 3, 4 4 sent 1, 2, 3, 4 Incoming channels 1 received 1, 2, 3, 4 stored 5, 6 3 received 1, 2, 3, 4, 5, 6, 7, 8	**Process 4** Outgoing channels 3 sent 1, 2, 3 Incoming channels 2 received 1, 2 stored 3, 4

Snapshot consistency check: Have all messages recorded as sent been either received or recorded as in transit?

Process 1: sent 6 messages to process 2 – process 2 has recorded 4 messages with 2 messages in the channel.

Process 1: sent 6 messages to process 3 – process 3 has recorded 3 messages with 3 messages in the channel.

Process 2: sent 4 messages to process 3 – process 3 has recorded 3 messages with 1 message in the channel.

Process 2: sent 4 messages to process 4 – process 4 has recorded 2 messages with 2 messages in the channel.

Process 3: sent 8 messages to process 2 – process 2 has recorded 8 messages

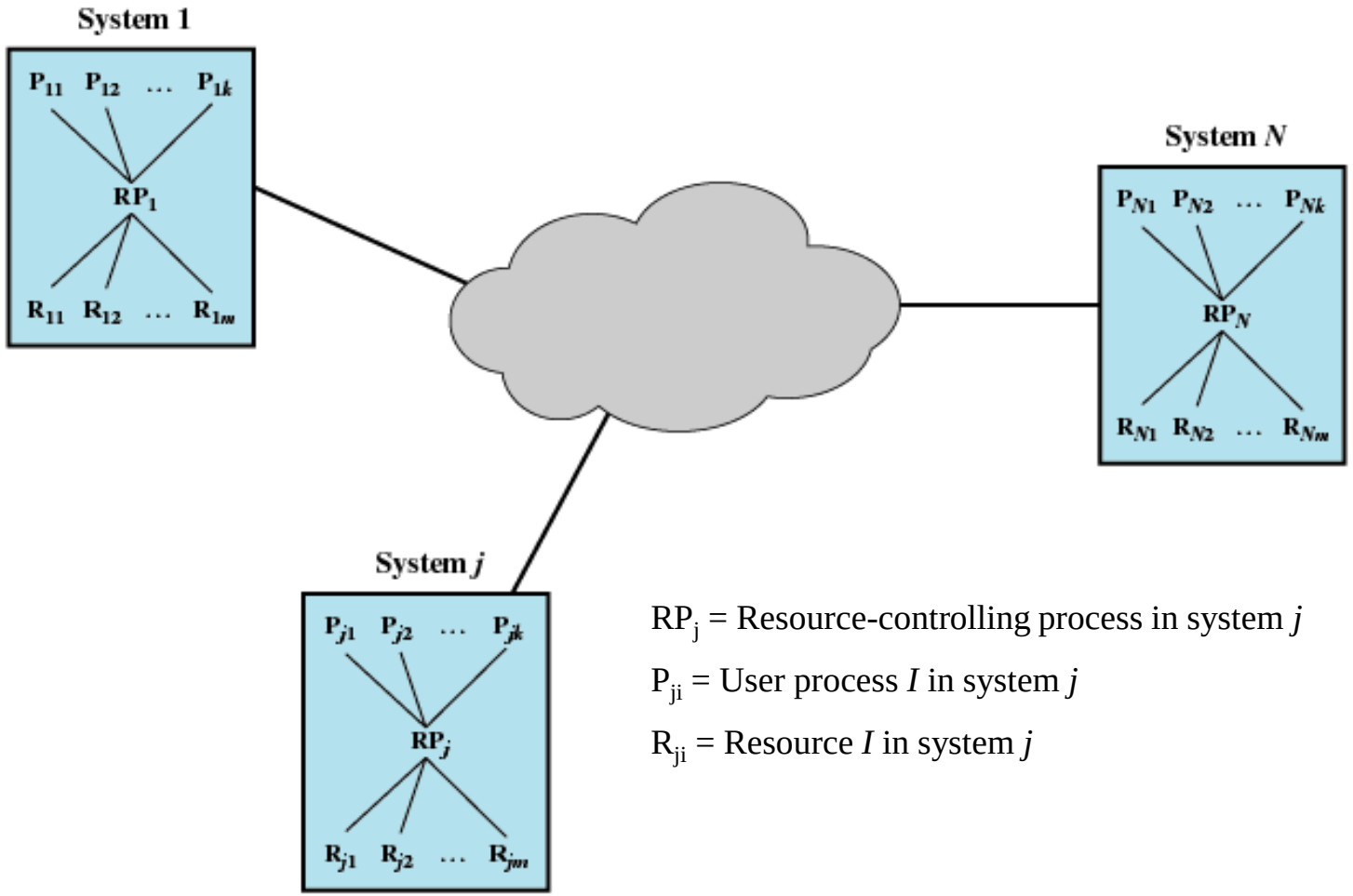
Process 4: sent 3 messages to process 3 – process 3 has recorded 3 messages

All messages sent by all processes have either been received or are in the channel: snapshot is consistent.

Distributed Mutual Exclusion Concepts

- Whenever two or more processes compete for the use of system resources, there is a need for a mechanism to enforce mutual exclusion.
- Any facility that is to provide support for mutual exclusion should meet the following criteria:
  - Mutual exclusion must be enforced: only one process at a time is allowed in its critical section.
  - A process that halts in its noncritical section must do so without interfering with other processes.
  - It must not be possible for a process requiring access to a critical section to be delayed indefinitely: no deadlock or starvation.
  - When no process is in a critical section, any process that requests entry to its critical section must be permitted to enter without delay.
  - No assumptions are made about relative process speeds or number of processors.
  - A process remains inside its critical section for a finite time only.

Distributed Mutual Exclusion Concepts



RP_j = Resource-controlling process in system j

P_{ji} = User process I in system j

R_{ji} = Resource I in system j

Centralized Algorithm for Mutual Exclusion

- One node is designated as the control node.
- This node control access to all shared objects.
- Two key properties of the centralized algorithm are:
  - Only the control node makes resource-allocation decision.
  - All necessary information is concentrated in the control node, including the identity and location of all resources and the allocation status of each resource.
- The centralized approach is straightforward, and it is easy to see how mutual exclusion is enforced: The control node will not grant a request for a resource until that resource is released by the process currently holding it.
- The centralized approach has severe drawbacks: (1) If the control node fails, mutual exclusion breaks down. (2) every allocation/deallocation requires an exchange of messages resulting in a bottleneck at the control node.

Distributed Algorithm

- All nodes have equal amount of information, on average.
- Each node has only a partial picture of the total system and must make decisions based on this information.
- All nodes bear equal responsibility for the final decision.
- All nodes expend equal effort, on average, in effecting a final decision.
- Failure of a node, in general, does not result in a total system collapse.
- There exists no system-wide common clock with which to regulate the time of events.

Ordering of Events

- Events must be ordered to ensure mutual exclusion and avoid deadlock.
- Clocks are not synchronized.
- Communication delays exist.
- Need to consistently say that one event occurs before another event.
- Messages are sent when want to enter critical section and when leaving critical section.
- Time-stamping
  - Orders events on a distributed system
  - System clock is not used

Time-Stamping Algorithm

- Each system on the network maintains a counter which functions as a clock (C_i).
- Each site has a numerical identifier.
- When a message is received, the receiving system sets is counter to one more than the maximum of its current value and the incoming time-stamp (counter).
- If two messages have the same time-stamp, they are ordered by the number of their sites.
- Messages have the form: (m, T_i, i) where m = the message content, T_i = the timestamp for the message set equal to C_i, and i = numerical identifier for the site.
- When a message is received, the receiving site j sets its clock to one more than the maximum of its current value and the incoming timestamp: $C_j = 1 + \max[C_j, T_i]$.
- At each site, the ordering of event is determine by the following rules: For a message x from site i and a message y from site j, x is said to precede y if one of the following conditions holds: (1) if $T_i < T_j$ or (2) $T_i = T_j$ and $i < j$.
- For this method to work, each message is sent from one process to all other processes.
  - Ensures all sites have same ordering of messages.
  - For mutual exclusion and deadlock all processes must be aware of the situation.

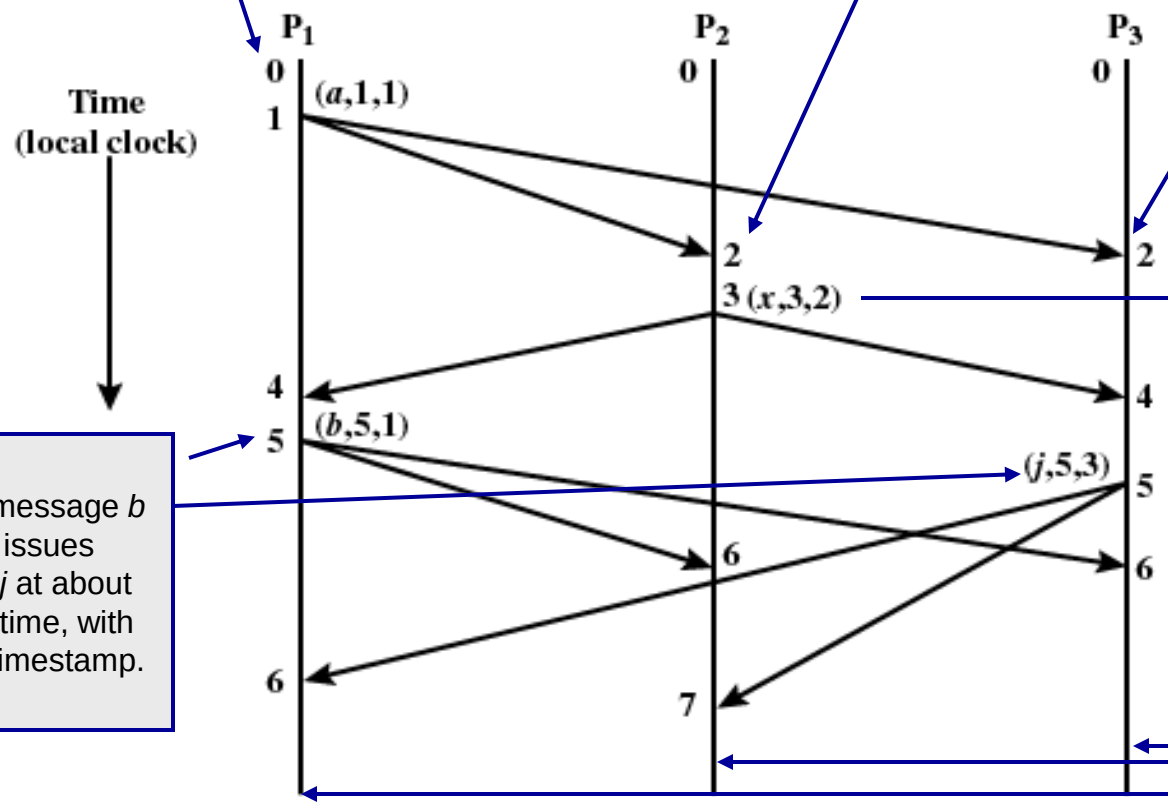
Time-Stamping Algorithm - Example

1

At time 0, P_1 transmits message *a* and increments its clock by 1

2

Message *a* received by P_2 and P_3 . Since both local clocks have time 0, they are reset to 2 = 1 + max[0,1].



3

P_2 issues message *x*, first incrementing its clock to 3. Upon receipt of this message both P_1 and P_3 both increment their clocks to 4.

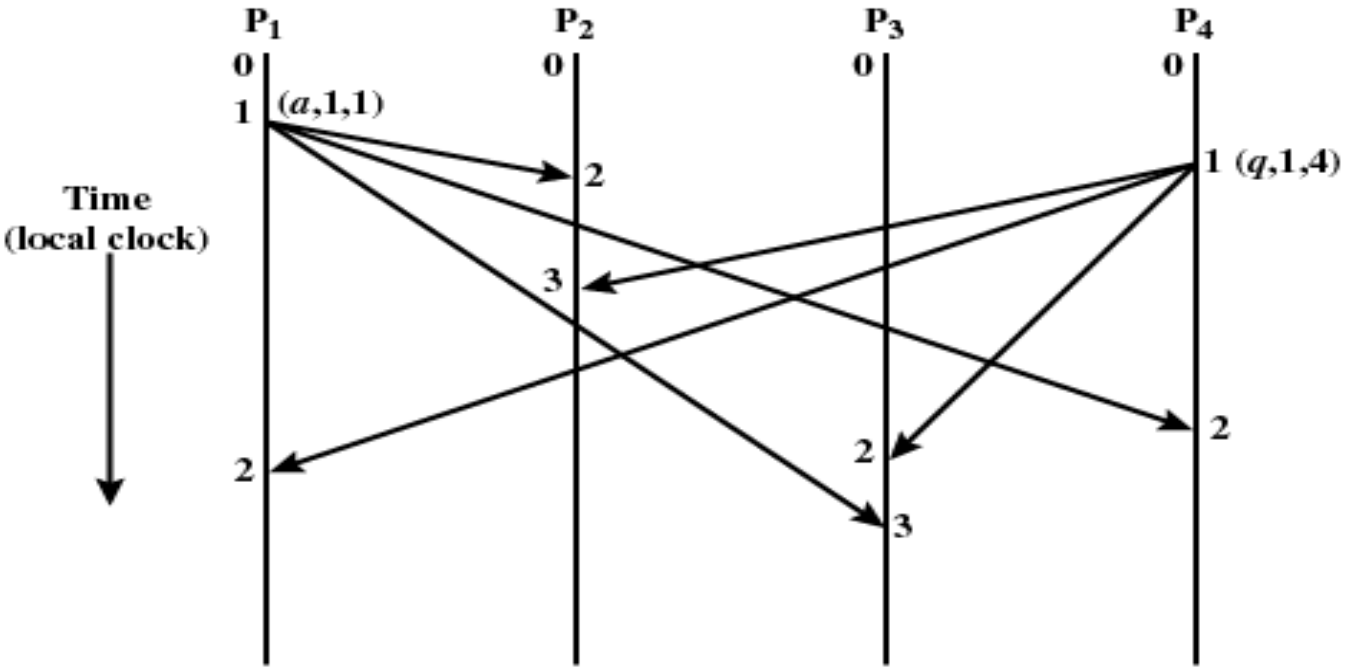
4

P_1 issues message *b* and P_3 issues message *j* at about the same time, with the same timestamp.

5

After all events have occurred, the ordering of messages is the same at all sites, namely {a, x, b, j}

Time-Stamping Algorithm – Another Example



Verify that the order of events (messages received) at each site is {a, q}.

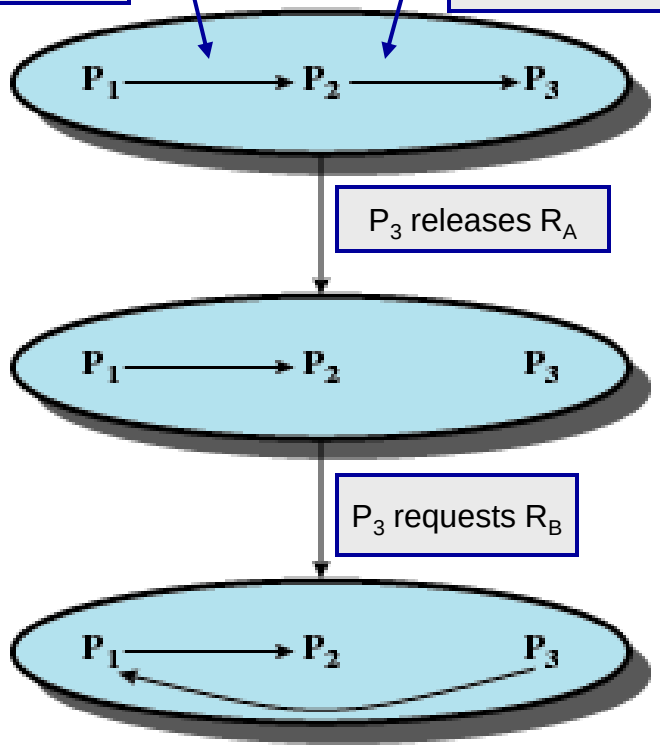
Deadlock in Resource Allocation

- Deadlock in resource allocation exists only if all of the following conditions are met:
  - Mutual exclusion
  - Hold and wait
  - No preemption
  - Circular wait
- The aim of an algorithm that deals with deadlock is either to prevent the formation of a circular wait or to detect its actual or potential occurrence.

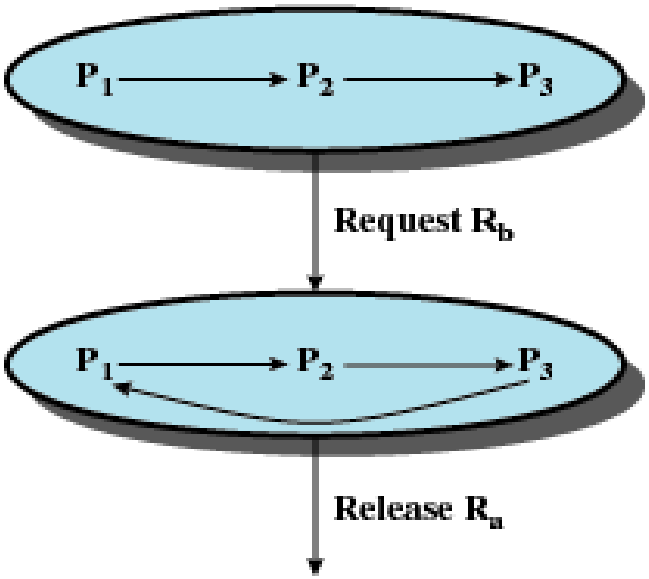
Phantom Deadlock

P_1 is halted waiting for a resource held by P_2. P_1 holds R_B

P1 is halted waiting for a resource held by P2. P_3 holds R_A



(a) If P_3's release of R_A arrives before its request for R_B then all is OK



(b) If P_3's release of R_A arrives after its request for R_B then deadlock may be falsely detected

Deadlock Prevention

- Circular-wait condition can be prevented by defining a linear ordering of resource types.
- Hold-and-wait condition can be prevented by requiring that a process request all of its required resource at one time, and blocking the process until all requests can be granted simultaneously.

Deadlock Avoidance

- Distributed deadlock avoidance is impractical
  - Every node must keep track of the global state of the system.
  - The process of checking for a safe global state must be mutually exclusive.
  - Checking for safe states involves considerable processing overhead for a distributed system with a large number of processes and resources.

Distributed Deadlock Detection

- Each site only knows about its own resources.
  - Deadlock may involve distributed resources
- Centralized control – one site is responsible for deadlock detection.
- Hierarchical control – lowest node above the nodes involved in deadlock.
- Distributed control – all processes cooperate in the deadlock detection function.

Summary of Distributed Deadlock Detection Strategies

Centralized Algorithms		Hierarchical Algorithms		Distributed Algorithms	
Strengths	Weaknesses	Strengths	Weaknesses	Strengths	Weaknesses
Algorithms are conceptually simple and easy to implement. Central site has complete information and can optimally resolve deadlocks.	Considerable communications overhead; every node must send state information to the central node. Vulnerable to the failure of the central node.	Not vulnerable to single point failure. Deadlock resolution activity is limited if most potential deadlocks are relatively localized.	May be difficult to configure the system so that most potential deadlocks are localized; otherwise there may actually be more overhead than in a distributed approach.	Not vulnerable to single point failure. No node is swamped with deadlock detection activity.	Deadlock resolution is cumbersome because several sites may detect the same deadlock and may not be aware of other nodes involved in the deadlock. Algorithms are difficult to design because of timing considerations.