

# Causal, Partial and Total Ordering of Messages in Distributed Systems

# Causal and Total Order

## Stronger orderings

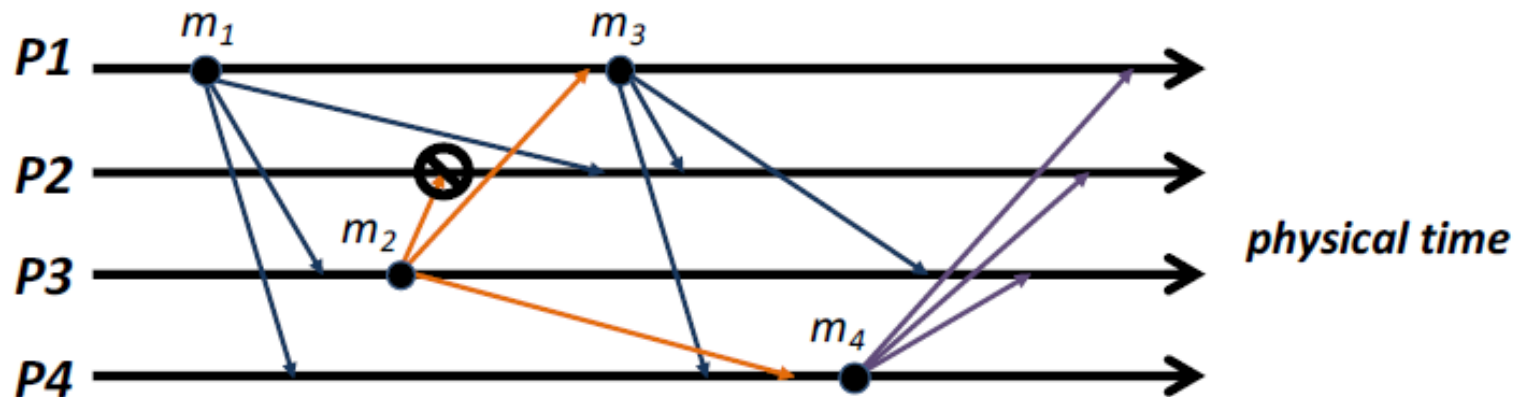
---

- Can also implement FIFO ordering by just using a reliable FIFO transport like TCP/IP
- But the general ‘receive versus deliver’ model also allows us to provide **stronger** orderings:
  - **Causal ordering**: if event  $\text{multicast}(g, m_1) \rightarrow \text{multicast}(g, m_2)$ , then all processes will see  $m_1$  before  $m_2$
  - **Total ordering**: if any processes delivers a message  $m_1$  before  $m_2$ , then all processes will deliver  $m_1$  before  $m_2$
- Causal ordering implies FIFO ordering, since any two multicasts by the same process are related by  $\rightarrow$
- Total ordering (as defined) does *not* imply FIFO (or causal) ordering, just says that all processes must agree
  - Often want **FIFO-total** ordering (combines the two)

# Causal Ordering

## Causal ordering

---

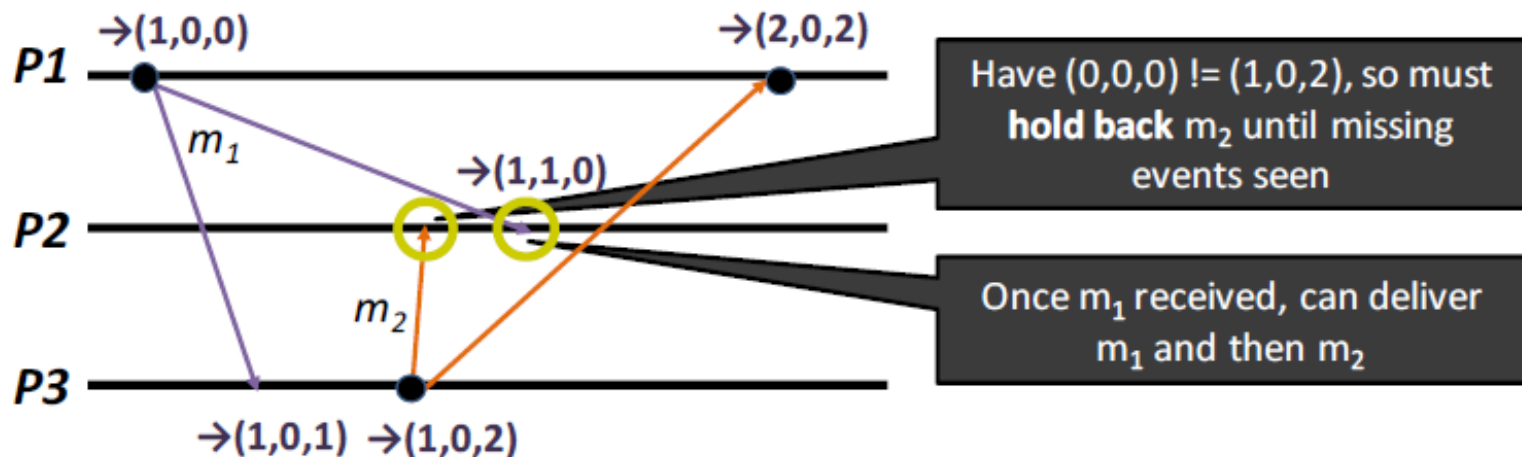


- Same example as previously, but now causal ordering means that
  - (a) everyone must see  $m_1$  before  $m_3$  (as with FIFO), **and**
  - (b) everyone must see  $m_1$  before  $m_2$  (due to happens-before)
- Is this ok?
  - No!  $m_1 \rightarrow m_2$ , but **P2** sees  $m_2$  before  $m_1$
  - To be correct, must hold back (delay) delivery of  $m_2$  at **P2**
  - But how do we know this?

# Causal Ordering

## Implementing causal ordering

- Turns out this is pretty easy!
  - Start with receive algorithm for FIFO multicast...
  - and replace sequence numbers with vector clocks



- Some care needed with dynamic groups

# Partial Order

partial order:

$\{A, B, C, D, E, F, G, H\}$

→ is an irreflexive partial order

- A set  $S$ , together with
- A binary relation, often written  $\leq$ , that lets you compare elements of  $S$ , and has the following properties:

X  
vacuously true ✓  
✓  
→

- Reflexivity: for all  $a \in S$ ,  $a \leq a$ .

- Antisymmetry: for all  $a, b \in S$ ,  
if  $a \leq b$ , and  $b \leq a$ ,  
 $a = b$

- Transitivity: for all  $a, b, c \in S$ ,  
if  $a \leq b$  and  $b \leq c$ ,  
then  $a \leq c$ .

# Total Order

## Total ordering of events

- A system of clocks that satisfy the Clock Condition can be used to totally order system events.
- To totally order the events in a system, the events are ordered according to their times of occurrence. In case two or more events occur at the same time, an arbitrary total ordering  $\prec$  of processes is used. To do this, the relation  $\Rightarrow$  is defined as follows:

If  $a$  is an event in process  $P_i$  and  $b$  is an event in process  $P_j$ , then  $a \Rightarrow b$  if and only if either:

- i.  $C_i\langle a \rangle < C_j\langle b \rangle$  or
- ii.  $C_i\langle a \rangle = C_j\langle b \rangle$  and  $P_i \prec P_j$

There is total ordering because for any two events in the system, it is clear which happened first.

- The total ordering of events is very useful for distributed system implementation.

# Group Communication

- Unicast vs. multicast vs. broadcast
- Network layer or hardware-assist multicast cannot easily provide:
  - ▶ Application-specific semantics on message delivery order
  - ▶ Adapt groups to dynamic membership
  - ▶ Multicast to arbitrary process set at each send
  - ▶ Provide multiple fault-tolerance semantics
- Closed group (source part of group) vs. open group
- # groups can be  $O(2^n)$

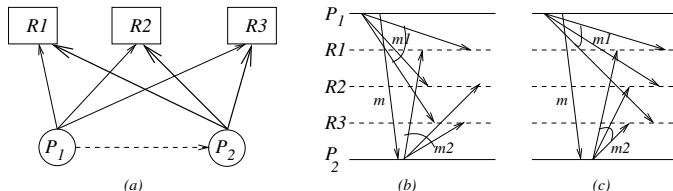


Figure 6.11: (a) Updates to 3 replicas. (b) Causal order (CO) and total order violated. (c) Causal order violated.

If  $m$  did not exist, (b,c) would not violate CO.

# Raynal-Schiper-Toueg (RST) Algorithm

(local variables)

**array of int**  $SENT[1 \dots n, 1 \dots n]$

**array of int**  $DELIV[1 \dots n]$  //  $DELIV[k] = \#$  messages sent by  $k$  that are delivered locally

(1) **send event**, where  $P_i$  wants to send message  $M$  to  $P_j$ :

(1a) **send**  $(M, SENT)$  to  $P_j$ ;

(1b)  $SENT[i, j] \leftarrow SENT[i, j] + 1$ .

(2) **message arrival**, when  $(M, ST)$  arrives at  $P_i$  from  $P_j$ :

(2a) **deliver**  $M$  to  $P_i$  when **for each** process  $x$ ,

(2b)  $DELIV[x] \geq ST[x, i]$ ;

(2c)  $\forall x, y, SENT[x, y] \leftarrow \max(SENT[x, y], ST[x, y])$ ;

(2d)  $DELIV[j] \leftarrow DELIV[j] + 1$ .

How does algorithm simplify if all msgs are broadcast?

## Assumptions/Correctness

- FIFO channels.
- Safety: Step (2a,b).
- Liveness: assuming no failures, finite propagation times

## Complexity

- $n^2$  ints/ process
- $n^2$  ints/ msg
- Time per send and rcv event:  $n^2$



# Optimal KS Algorithm for CO: Principles

$M_{i,a}$ :  $a^{th}$  multicast message sent by  $P_i$

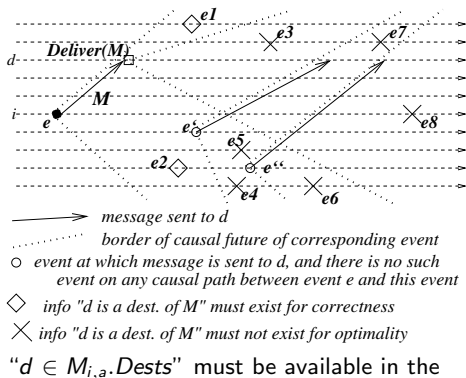
## Delivery Condition for correctness:

Msg  $M^*$  that carries information " $d \in M.Dests$ ", where message  $M$  was sent to  $d$  in the causal past of  $Send(M^*)$ , is not delivered to  $d$  if  $M$  has not yet been delivered to  $d$ .

## Necessary and Sufficient Conditions for Optimality:

- For how long should the information " $d \in M_{i,a}.Dests$ " be stored in the log at a process, and piggybacked on messages?
- *as long as and only as long as*
  - ▶ (*Propagation Constraint I:*) it is not known that the message  $M_{i,a}$  is delivered to  $d$ , and
  - ▶ (*Propagation Constraint II:*) it is not known that a message has been sent to  $d$  in the causal future of  $Send(M_{i,a})$ , and hence it is not guaranteed using a reasoning based on transitivity that the message  $M_{i,a}$  will be delivered to  $d$  in CO.
- $\Rightarrow$  if either (I) or (II) is false, " $d \in M.Dests$ " must *not* be stored or propagated, even to remember that (I) or (II) has been falsified.

# Optimal KS Algorithm for CO: Principles

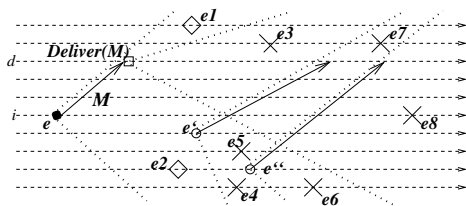


- In the causal future of  $Deliver_d(M_{i,a})$ , and  $Send(M_{k,c})$ , the information is redundant; elsewhere, it is necessary.
- Information about what messages have been delivered (or are guaranteed to be delivered without violating CO) is necessary for the Delivery Condition.
  - ▶ For optimality, this cannot be stored. Algorithm infers this using set-operation logic.

causal future of event  $e_{i,a}$ , but

- not in the causal future of  $Deliver_d(M_{i,a})$ , and
- not in the causal future of  $e_{k,c}$ , where  $d \in M_{k,c}.Dests$  and there is no other message sent causally between  $M_{i,a}$  and  $M_{k,c}$  to the same destination  $d$ .

# Optimal KS Algorithm for CO: Principles



→ message sent to  $d$   
 ..... border of causal future of corresponding event  
 ○ event at which message is sent to  $d$ , and there is no such event on any causal path between event  $e$  and this event

◇ info " $d$  is a dest. of  $M$ " must exist for correctness  
 ✕ info " $d$  is a dest. of  $M$ " must not exist for optimality

" $d \in M.Dests$ "

- must exist at  $e1$  and  $e2$  because (I) and (II) are true.
- must not exist at  $e3$  because (I) is false
- must not exist at  $e4, e5, e6$  because (II) is false
- must not exist at  $e7, e8$  because (I) and (II) are false

- Info about messages (i) not known to be delivered and (ii) not guaranteed to be delivered in CO, is *explicitly* tracked using (*source, ts, dest*).
- Must be deleted as soon as either (i) or (ii) becomes false.
- Info about messages already delivered and messages guaranteed to be delivered in CO is *implicitly* tracked without storing or propagating it:
  - ▶ derived from the explicit information.
  - ▶ used for determining when (i) or (ii) becomes false for the explicit information being stored/piggybacked.

# Optimal KS Algorithm for CO: Code (1)

## (local variables)

$clock_j \leftarrow 0;$  // local counter clock at node  $j$   
 $SR_j[1..n] \leftarrow \vec{0};$  //  $SR_j[i]$  is the timestamp of last msg. from  $i$  delivered to  $j$   
 $LOG_j = \{(i, clock_i, Dests)\} \leftarrow \{\forall i, (i, 0, \emptyset)\};$   
// Each entry denotes a message sent in the causal past, by  $i$  at  $clock_i$ .  $Dests$  is the set of remaining destinations for which it is not known that  $M_{i, clock_i}$  (i) has been delivered, or (ii) is guaranteed to be delivered in CO.

## SND: $j$ sends a message $M$ to $Dests$ :

- 1  $clock_j \leftarrow clock_j + 1;$
- 2 for all  $d \in M.Dests$  do:
 

$O_M \leftarrow LOG_j;$  //  $O_M$  denotes  $O_{M_j, clock_j}$   
 for all  $o \in O_M$ , modify  $o.Dests$  as follows:  
   if  $d \notin o.Dests$  then  $o.Dests \leftarrow (o.Dests \setminus M.Dests);$   
   if  $d \in o.Dests$  then  $o.Dests \leftarrow (o.Dests \setminus M.Dests) \cup \{d\};$   
   // Do not propagate information about indirect dependencies that are guaranteed to be transitively satisfied when dependencies of  $M$  are satisfied.

for all  $o_{s,t} \in O_M$  do  
   if  $o_{s,t}.Dests = \emptyset \wedge (\exists o'_{s,t'} \in O_M \mid t < t')$  then  $O_M \leftarrow O_M \setminus \{o_{s,t}\};$   
   // do not propagate older entries for which  $Dests$  field is  $\emptyset$

send  $(j, clock_j, M, Dests, O_M)$  to  $d;$
- 3 for all  $l \in LOG_j$  do  $l.Dests \leftarrow l.Dests \setminus Dests;$ 

// Do not store information about indirect dependencies that are guaranteed to be transitively satisfied when dependencies of  $M$  are satisfied.  
// purge  $l \in LOG_j$  if  $l.Dests = \emptyset$

Execute  $PURGE\_NULL\_ENTRIES(LOG_j);$

- 4  $LOG_j \leftarrow LOG_j \cup \{(j, clock_j, Dests)\}.$

# Optimal KS Algorithm for CO: Code (2)

**RCV:**  $j$  receives a message  $(k, t_k, M, Dests, O_M)$  from  $k$ :

- ① // Delivery Condition; ensure that messages sent causally before  $M$  are delivered.  
 for all  $o_m, t_m \in O_M$  do  
     if  $j \in o_m, t_m.Dests$  wait until  $t_m \leq SR_j[m]$ ;
- ② Deliver  $M$ ;  $SR_j[k] \leftarrow t_k$ ;
- ③  $O_M \leftarrow \{(k, t_k, Dests)\} \cup O_M$ ;  
 for all  $o_m, t_m \in O_M$  do  $o_m, t_m.Dests \leftarrow o_m, t_m.Dests \setminus \{j\}$ ;  
     // delete the now redundant dependency of message represented by  $o_m, t_m$  sent to  $j$
- ④ // Merge  $O_M$  and  $LOG_j$  by eliminating all redundant entries.  
     // Implicitly track "already delivered" & "guaranteed to be delivered in CO" messages.  
     for all  $o_m, t \in O_M$  and  $l_{s,t'} \in LOG_j$  such that  $s = m$  do  
         if  $t < t' \wedge l_{s,t} \notin LOG_j$  then mark  $o_m, t$ ;  
             //  $l_{s,t}$  had been deleted or never inserted, as  $l_{s,t}.Dests = \emptyset$  in the causal past  
         if  $t' < t \wedge o_m, t' \notin O_M$  then mark  $l_{s,t'}$ ;  
             //  $o_m, t' \notin O_M$  because  $l_{s,t'}$  had become  $\emptyset$  at another process in the causal past  
     Delete all marked elements in  $O_M$  and  $LOG_j$ ;  
     // delete entries about redundant information  
     for all  $l_{s,t'} \in LOG_j$  and  $o_m, t \in O_M$ , such that  $s = m \wedge t' = t$  do  
          $l_{s,t'}.Dests \leftarrow l_{s,t'}.Dests \cap o_m, t.Dests$ ;  
         // delete destinations for which Delivery  
         // Condition is satisfied or guaranteed to be satisfied as per  $o_m, t$   
         // information has been incorporated in  $l_{s,t'}$   
         Delete  $o_m, t$  from  $O_M$ ;  
         // merge nonredundant information of  $O_M$  into  $LOG_j$   
      $LOG_j \leftarrow LOG_j \cup O_M$ ;  
     // Purge older entries  $l$  for which  $l.Dests = \emptyset$
- ⑤  $PURGE.NULL.ENTRIES(LOG_j)$ .

**PURGE.NULL.ENTRIES( $Log_j$ ):**

// Purge older entries  $l$  for which  $l.Dests = \emptyset$  is implicitly inferred

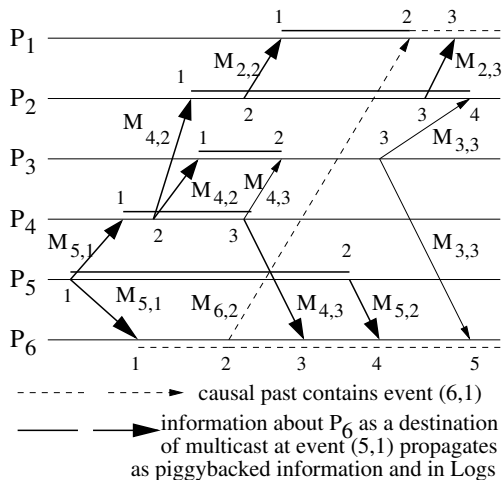
for all  $l_{s,t} \in Log_j$  do

if  $l_{s,t}.Dests = \emptyset \wedge (\exists l'_{s,t'} \in Log_j \mid t < t')$  then  $Log_j \leftarrow Log_j \setminus \{l_{s,t}\}$ .

# Optimal KS Algorithm for CO: Information Pruning

- Explicit tracking of  $(s, ts, dest)$  per multicast in  $Log$  and  $O_M$
- Implicit tracking of msgs that are (i) delivered, or (ii) guaranteed to be delivered in CO:
  - ▶ (Type 1:)  $\exists d \in M_{i,a}.Dests \mid d \notin l_{i,a}.Dests \vee d \notin o_{i,a}.Dests$ 
    - ★ When  $l_{i,a}.Dests = \emptyset$  or  $o_{i,a}.Dests = \emptyset$ ?
    - ★ Entries of the form  $l_{i,a_k}$  for  $k = 1, 2, \dots$  will accumulate
    - ★ Implemented in Step (2d)
  - ▶ (Type 2:) if  $a_1 < a_2$  and  $l_{i,a_2} \in LOG_j$ , then  $l_{i,a_1} \in LOG_j$ . (Likewise for messages)
    - ★ entries of the form  $l_{i,a_1}.Dests = \emptyset$  can be inferred by their absence, and should not be stored
    - ★ Implemented in Step (2d) and PURGE\_NULL\_ENTRIES

# Optimal KS Algorithm for CO: Example



| Message to dest.        | piggybacked $M_{5,1}.Dests$ |
|-------------------------|-----------------------------|
| $M_{5,1}$ to $P_4, P_6$ | $\{P_4, P_6\}$              |
| $M_{4,2}$ to $P_3, P_2$ | $\{P_6\}$                   |
| $M_{2,2}$ to $P_1$      | $\{P_6\}$                   |
| $M_{6,2}$ to $P_1$      | $\{P_4\}$                   |
| $M_{4,3}$ to $P_6$      | $\{P_6\}$                   |
| $M_{4,3}$ to $P_3$      | $\{\}$                      |
| $M_{5,2}$ to $P_6$      | $\{P_4, P_6\}$              |
| $M_{2,3}$ to $P_1$      | $\{P_6\}$                   |
| $M_{3,3}$ to $P_2, P_6$ | $\{\}$                      |

Figure 6.13: Tracking of information about  $M_{5,1}.Dests$

# Total Message Order

## Total order

For each pair of processes  $P_i$  and  $P_j$  and for each pair of messages  $M_x$  and  $M_y$  that are delivered to both the processes,  $P_i$  is delivered  $M_x$  before  $M_y$  if and only if  $P_j$  is delivered  $M_x$  before  $M_y$ .

Same order seen by all

Solves coherence problem

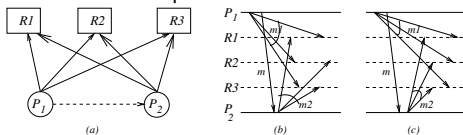


Fig 6.11: (a) Updates to 3 replicas. (b) Total order violated. (c) Total order not violated.

## Centralized algorithm

- (1) When  $P_i$  wants to multicast  $M$  to group  $G$ :  
 (1a) **send**  $M(i, G)$  to coordinator.
- (2) When  $M(i, G)$  arrives from  $P_i$  at coordinator:  
 (2a) **send**  $M(i, G)$  to members of  $G$ .
- (3) When  $M(i, G)$  arrives at  $P_j$  from coordinator:  
 (3a) **deliver**  $M(i, G)$  to application.

Time Complexity: 2 hops/ transmission

Message complexity:  $n$



# Total Message Order: 3-phase Algorithm Code

```

record Q_entry
    M: int;                                     // the application message
    tag: int;                                   // unique message identifier
    sender_id: int;                             // sender of the message
    timestamp: int;                             // tentative timestamp assigned to message
    deliverable: boolean;                       // whether message is ready for delivery

(local variables)
queue of Q_entry: temp-Q, delivery-Q
int: clock                                     // Used as a variant of Lamport's scalar clock
int: priority                                 // Used to track the highest proposed timestamp
(message types)
REVISE_TS(M, i, tag, ts)                       // Phase 1 message sent by  $P_i$ , with initial timestamp ts
PROPOSED_TS(j, i, tag, ts)                     // Phase 2 message sent by  $P_j$ , with revised timestamp, to  $P_i$ 
FINAL_TS(i, tag, ts)                           // Phase 3 message sent by  $P_i$ , with final timestamp

```

(1) When process  $P_i$  wants to multicast a message  $M$  with a tag  $tag$ :

```

(1a) clock = clock + 1;
(1b) send REVISE_TS(M, i, tag, clock) to all processes;
(1c) temp_ts = 0;
(1d) await PROPOSED_TS(j, i, tag, ts_j) from each process  $P_j$ ;
(1e)  $\forall j \in N$ , do temp_ts = max(temp_ts, ts_j);
(1f) send FINAL_TS(i, tag, temp_ts) to all processes;
(1g) clock = max(clock, temp_ts).
(2) When REVISE_TS(M, j, tag, clk) arrives from  $P_j$ :

```

```

(2a) priority = max(priority + 1, clk);
(2b) insert (M, tag, j, priority, undeliverable) in temp-Q;           // at end of queue
(2c) send PROPOSED_TS(i, j, tag, priority) to  $P_j$ .
(3) When FINAL_TS(j, tag, clk) arrives from  $P_j$ :

```

```

(3a) Identify entry Q_entry(tag) in temp-Q, corresponding to tag;
(3b) mark q_tag as deliverable;
(3c) Update Q_entry.timestamp to clk and re-sort temp-Q based on the timestamp field;
(3d) if head(temp-Q) = Q_entry(tag) then
(3e)     move Q_entry(tag) from temp-Q to delivery-Q;
(3f)     while head(temp-Q) is deliverable do
(3g)         move head(temp-Q) from temp-Q to delivery-Q.
(4) When  $P_i$  removes a message (M, tag, j, ts, deliverable) from head(delivery-Q):
(4a) clock = max(clock, ts) + 1.

```

# Total Order: Distributed Algorithm: Example and Complexity

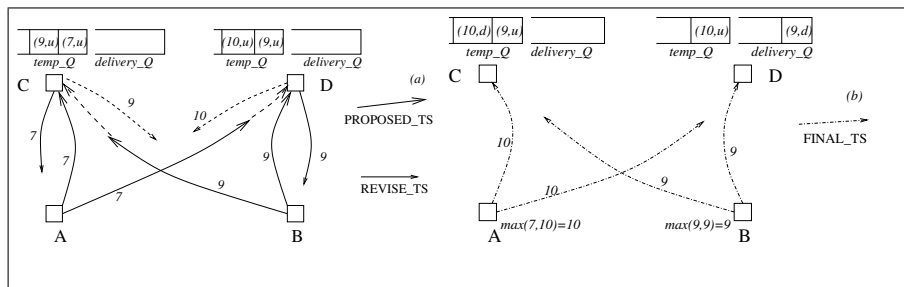
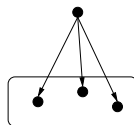


Figure 6.14: (a) A snapshot for PROPOSED\_TS and REVISE\_TS messages. The dashed lines show the further execution after the snapshot. (b) The FINAL\_TS messages.

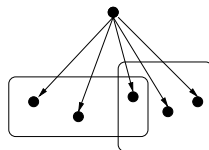
## Complexity:

- Three phases
- $3(n - 1)$  messages for  $n - 1$  dests
- Delay: 3 message hops
- Also implements causal order

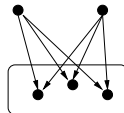
# A Nomenclature for Multicast



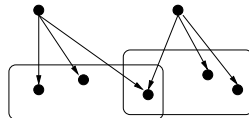
(a) *Single Source Single Group*



(c) *Single Source Multiple Groups*



(b) *Multiple Sources Single Group*



(d) *Multiple Sources Multiple Groups*

4 classes of source-dest relns for open groups:

- SSSG: Single source and single dest group
- MSSG: Multiple sources and single dest group
- SSMG: Single source and multiple, possibly overlapping, groups
- MSMG: Multiple sources and multiple, possibly overlapping, groups

Fig 6.15 : Four classes of source-dest relationships for open-group multicasts. For closed-group multicasts, the sender needs to be part of the recipient group.

SSSG, SSMG: easy to implement

MSSG: easy. E.g., Centralized algorithm

MSMG: Semi-centralized *propagation tree* approach

# Propagation Trees for Multicast: Definitions

- set of groups  $\mathcal{G} = \{G_1 \dots G_g\}$
- set of *meta-groups*  $\mathcal{MG} = \{MG_1, \dots, MG_h\}$  with the following properties.
  - ▶ Each process belongs to a single meta-group, and has the exact same group membership as every other process in that meta-group.
  - ▶ No other process outside that meta-group has that exact group membership.
- MSMG to groups  $\rightarrow$  MSSG to meta-groups
- A distinguished node in each meta-group acts as its manager.
- For each user group  $G_i$ , one of its meta-groups is chosen to be its *primary meta-group* (*PM*), denoted  $PM(G_i)$ .
- All meta-groups are organized in a *propagation forest/tree* satisfying:
  - ▶ For user group  $G_i$ ,  $PM(G_i)$  is at the lowest possible level (i.e., farthest from root) of the tree such that all meta-groups whose destinations contain any nodes of  $G_i$  belong to subtree rooted at  $PM(G_i)$ .
- Propagation tree is not unique!
  - ▶ Exercise: How to construct propagation tree?
  - ▶ Metagroup with members from more user groups as root  $\Rightarrow$  low tree height

# Propagation Trees for Multicast: Properties

- ① The primary meta-group  $PM(G)$  is the ancestor of all the other meta-groups of  $G$  in the propagation tree.
- ②  $PM(G)$  is uniquely defined.
- ③ For any meta-group  $MG$ , there is a unique path to it from the  $PM$  of any of the user groups of which the meta-group  $MG$  is a subset.
- ④ Any  $PM(G_1)$  and  $PM(G_2)$  lie on the same branch of a tree or are in disjoint trees. In the latter case, their groups membership sets are disjoint.

**Key idea:** Multicasts to  $G_i$  are sent first to the meta-group  $PM(G_i)$  as only the subtree rooted at  $PM(G_i)$  can contain the nodes in  $G_i$ . The message is then propagated down the subtree rooted at  $PM(G_i)$ .

- $MG_1$  *subsumes*  $MG_2$  if  $MG_1$  is a subset of each user group  $G$  of which  $MG_2$  is a subset.
- $MG_1$  *is joint with*  $MG_2$  if neither subsumes the other and there is some group  $G$  such that  $MG_1, MG_2 \subset G$ .

# Propagation Trees for Multicast: Example

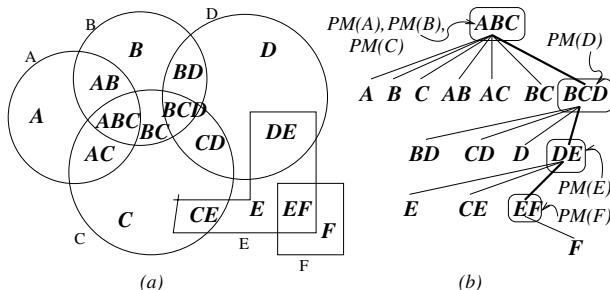


Fig 6.16: Example illustrating a propagation tree. Meta-groups in boldface. (a) Groups A, B, C, D, E and F, and their meta-groups. (b) A *propagation tree*, with the primary meta-groups labeled.

- $\langle ABC \rangle$ ,  $\langle AB \rangle$ ,  $\langle AC \rangle$ , and  $\langle A \rangle$  are meta-groups of user group  $\langle A \rangle$ .
- $\langle ABC \rangle$  is  $PM(A), PM(B), PM(C)$ .  $\langle B, C, D \rangle$  is  $PM(D)$ .  $\langle D, E \rangle$  is  $PM(E)$ .  $\langle E, F \rangle$  is  $PM(F)$ .
- $\langle ABC \rangle$  is joint with  $\langle CD \rangle$ . Neither subsumes the other and both are a subset of C.
- Meta-group  $\langle ABC \rangle$  is the primary meta-group  $PM(A), PM(B), PM(C)$ . Meta-group  $\langle BCD \rangle$  is the primary meta-group  $PM(D)$ . A multicast to D is sent to  $\langle BCD \rangle$ .

# Propagation Trees for Multicast: Logic

- Each process knows the propagation tree
- Each meta-group has a distinguished process (*manager*)
- $SV_i[k]$  at each  $P_i$ : # msgs multicast by  $P_i$  that will traverse  $PM(G_k)$ .  
Piggybacked on each multicast by  $P_i$ .
- $RV_{manager(PM(G_z))}[k]$ : # msgs sent by  $P_k$  received by  $PM(G_z)$
- At  $manager(PM(G_z))$ : process  $M$  from  $P_i$  if  $SV_i[z] = RV_{manager(PM(G_z))}[i]$ ;  
else buffer  $M$  until condition becomes true
- At manager of non-primary meta-group: msg order already determined, as it  
never receives msg directly from sender of multicast. Forward (2d-2g).

Correctness for Total Order: Consider  $MG_1, MG_2 \subset G_x, G_y$

- $\Rightarrow PM(G_x), PM(G_y)$  both subsume  $MG_1, MG_2$  and lie on the same branch of  
the propagation tree to either  $MG_1$  or  $MG_2$
- order seen by the "lower-in-the-tree" primary meta-group (+ FIFO) =  
order seen by processes in meta-groups subsumed by it

# Propagation Trees for Multicast (CO and TO): Code

```

(local variables)
array of integers:  $SV[1 \dots h]$ ;           //kept by each process.  $h$  is  $\#(\text{primary meta-groups})$ ,  $h \leq |\mathcal{G}|$ 
array of integers:  $RV[1 \dots n]$ ;         //kept by each primary meta-group manager.  $n$  is  $\#(\text{processes})$ 
set of integers:  $PM\_set$ ;                 //set of primary meta-groups through which message must traverse

```

(1) When process  $P_i$  wants to multicast message  $M$  to group  $G$ :

(1a) **send**  $M(i, G, SV_i)$  to manager of  $PM(G)$ , primary meta-group of  $G$ ;

(1b)  $PM\_set \leftarrow \{ \text{primary meta-groups through which } M \text{ must traverse} \}$ ;

(1c) **for all**  $PM_x \in PM\_set$  **do**

(1d)  $SV_i[x] \leftarrow SV_i[x] + 1$ .

(2) When  $P_i$ , the manager of a meta-group  $MG$  receives  $M(k, G, SV_k)$  from  $P_j$ :

// Note:  $P_i$  may not be a manager of any meta-group

(2a) **if**  $MG$  is a primary meta-group **then**

(2b) **buffer** the message **until**  $(SV_k[i] = RV_i[k])$ ;

(2c)  $RV_i[k] \leftarrow RV_i[k] + 1$ ;

(2d) **for each** child meta-group that is subsumed by  $MG$  **do**

(2e) **send**  $M(k, G, SV_k)$  to the manager of that child meta-group;

(2f) **if** there are no child meta-groups **then**

(2g) **send**  $M(k, G, SV_k)$  to each process in this meta-group.



# Propagation Trees for Multicast: Correctness for CO

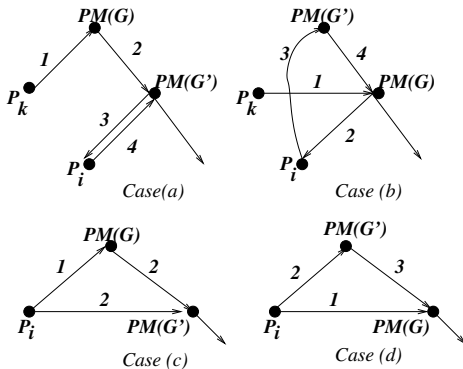
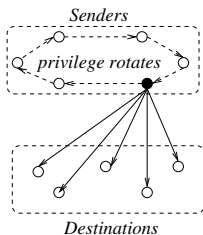


Fig 6.17: The four cases for the correctness of causal ordering. The sequence numbers indicate the order in which the msgs are sent.

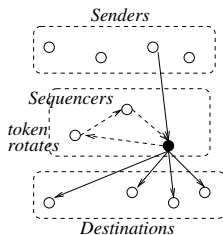
$M$  and  $M'$  multicast to  $G$  and  $G'$ , resp.  
Consider  $G \cap G'$

- Senders of  $M, M'$  are different.  
 $P_i$  in  $G$  receives  $M$ , then sends  $M'$ .  
 $\Rightarrow \forall MG_q \in G \cap G', PM(G), PM(G')$  are both ancestors of metagroup of  $P_i$ 
  - (a)  $PM(G')$  processes  $M$  before  $M'$
  - (b)  $PM(G)$  processes  $M$  before  $M'$
- FIFO  $\Rightarrow$  CO guaranteed for all in  $G \cap G'$
- $P_i$  sends  $M$  to  $G$ , then sends  $M'$  to  $G'$ .  
Test in lines (2a)-(2c)  $\Rightarrow$ 
  - $PM(G')$  will not process  $M'$  before  $M$
  - $PM(G)$  will not process  $M'$  before  $M$
- FIFO  $\Rightarrow$  CO guaranteed for all in  $G \cap G'$

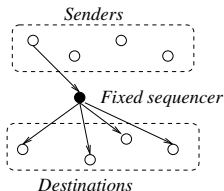
# Classification of Application-Level Multicast Algorithms



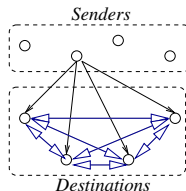
(a) Privilege-based



(b) Moving sequencer



(c) Fixed sequencer



(d) Destination agreement

- Communication-history based: RST, KS, Lamport, NewTop
- Privilege-based: Token-holder multicasts
  - ▶ processes deliver msgs in order of *seq\_no*.
  - ▶ Typically closed groups, and CO & TO.
  - ▶ E.g., Totem, On-demand.
- Moving sequencer: E.g., Chang-Maxemchuck, Pinwheel
  - ▶ Sequencers' token has *seq\_no* and list of msgs for which *seq\_no* has been assigned (these are sent msgs).
  - ▶ On receiving token, sequencer assigns *seq\_nos* to received but unsequenced msgs, and sends the newly sequenced msgs to dests.
  - ▶ Dests deliver in order of *seq\_no*
- Fixed Sequencer: simplifies moving sequencer approach. E.g., propagation tree, ISIS, Amoeba, Phoenix, Newtop-asymmetric
- Destination agreement:
  - ▶ Dests receive limited ordering info.
  - ▶ (i) Timestamp-based (Lamport's 3-phase)
  - ▶ (ii) Agreement-based, among dests.

# Semantics of Fault-Tolerant Multicast (1)

- Multicast is non-atomic!
- Well-defined behavior during failure  $\Rightarrow$  well-defined recovery actions
- if one correct process delivers  $M$ , what can be said about the other correct processes and faulty processes being delivered  $M$ ?
- if one faulty process delivers  $M$ , what can be said about the other correct processes and faulty processes being delivered  $M$ ?
- For causal or total order multicast, if one correct or faulty process delivers  $M$ , what can be said about other correct processes and faulty processes being delivered  $M$ ?
- (*Uniform*) specifications: specify behavior of faulty processes (benign failure model)

## Uniform Reliable Multicast of $M$ .

**Validity.** If a correct process multicasts  $M$ , then all correct processes will eventually deliver  $M$ .

(*Uniform*) **Agreement.** If a correct (*or faulty*) process delivers  $M$ , then all correct processes will eventually deliver  $M$ .

(*Uniform*) **Integrity.** Every correct (*or faulty*) process delivers  $M$  at most once, and only if  $M$  was previously multicast by  $sender(M)$ .

# Semantics of Fault-Tolerant Multicast (2)

(Uniform) FIFO order. If a process broadcasts  $M$  before it broadcasts  $M'$ , then no correct (or faulty) process delivers  $M'$  unless it previously delivered  $M$ .

(Uniform) Causal Order. If a process broadcasts  $M$  causally before it broadcasts  $M'$ , then no correct (or faulty) process delivers  $M'$  unless it previously delivered  $M$ .

(Uniform) Total Order. If correct (or faulty) processes  $a$  and  $b$  both deliver  $M$  and  $M'$ , then  $a$  delivers  $M$  before  $M'$  if and only if  $b$  delivers  $M$  before  $M'$ .

Specs based on global clock or local clock (needs clock synchronization)

(Uniform) Real-time  $\Delta$ -Timeliness. For some known constant  $\Delta$ , if  $M$  is multicast at real-time  $t$ , then no correct (or faulty) process delivers  $M$  after real-time  $t + \Delta$ .

(Uniform) Local  $\Delta$ -Timeliness. For some known constant  $\Delta$ , if  $M$  is multicast at local time  $t_m$ , then no correct (or faulty) process delivers  $M$  after its local time  $t_m + \Delta$ .

# Reverse Path Forwarding (RPF) for Constrained Flooding

Network layer multicast exploits topology, e.g., bridged LANs use spanning trees for learning destinations and distributing information, IP layer RPF approximates DVR/LSR-like algorithms at lower cost

- Broadcast gets curtailed to approximate a spanning tree
- Approx. to rooted spanning tree is identified without being computed/stored
- # msgs closer to  $|N|$  than to  $|L|$

(1) When  $P_i$  wants to multicast  $M$  to group  $Dests$ :

(1a) **send**  $M(i, Dests)$  on all outgoing links.

(2) When a node  $i$  receives  $M(x, Dests)$  from node  $j$ :

(2a) **if**  $Next\_hop(x) = j$  **then** // this will necessarily be a new message

(2b) **forward**  $M(x, Dests)$  on all other incident links besides  $(i, j)$ ;

(2c) **else** ignore the message.

# Steiner Trees

## Steiner tree

Given a weighted graph  $(N, L)$  and a subset  $N' \subseteq N$ , identify a subset  $L' \subseteq L$  such that  $(N', L')$  is a subgraph of  $(N, L)$  that connects all the nodes of  $N'$ .

A *minimal Steiner tree* is a minimal weight subgraph  $(N', L')$ .

NP-complete  $\Rightarrow$  need heuristics

Cost of routing scheme  $R$ :

- Network cost:  $\sum$  cost of Steiner tree edges
- Destination cost:  $\frac{1}{N'} \sum_{i \in N'} \text{cost}(i)$ , where  $\text{cost}(i)$  is cost of path  $(s, i)$

# Kou-Markowsky-Berman Heuristic for Steiner Tree

Input: weighted graph  $G = (N, L)$ , and  $N' \subseteq N$ , where  $N'$  is the set of Steiner points

- 1 Construct the complete undirected distance graph  $G' = (N', L')$  as follows.  
 $L' = \{(v_i, v_j) \mid v_i, v_j \text{ in } N'\}$ , and  $wt(v_i, v_j)$  is the length of the shortest path from  $v_i$  to  $v_j$  in  $(N, L)$ .
- 2 Let  $T'$  be the minimal spanning tree of  $G'$ . If there are multiple minimum spanning trees, select one randomly.
- 3 Construct a subgraph  $G_s$  of  $G$  by replacing each edge of the MST  $T'$  of  $G'$ , by its corresponding shortest path in  $G$ . If there are multiple shortest paths, select one randomly.
- 4 Find the minimum spanning tree  $T_s$  of  $G_s$ . If there are multiple minimum spanning trees, select one randomly.
- 5 Using  $T_s$ , delete edges as necessary so that all the leaves are the Steiner points  $N'$ . The resulting tree,  $T_{Steiner}$ , is the heuristic's solution.

- Approximation ratio = 2 (even without steps (4) and (5) added by KMB)
- Time complexity: Step (1):  $O(|N'| \cdot |N|^2)$ , Step (2):  $O(|N'|^2)$ , Step (3):  $O(|N|)$ , Step (4):  $O(|N|^2)$ , Step (5):  $O(|N|)$ . Step (1) dominates, hence  $O(|N'| \cdot |N|^2)$ .

# Constrained (Delay-bounded) Steiner Trees

- $\mathcal{C}(l)$  and  $\mathcal{D}(l)$ : cost, integer delay for edge  $l \in L$

## Definition

For a given delay tolerance  $\Delta$ , a given source  $s$  and a destination set  $Dest$ , where  $\{s\} \cup Dest = N' \subseteq N$ , identify a spanning tree  $T$  covering all the nodes in  $N'$ , subject to the constraints below.

- $\sum_{l \in T} \mathcal{C}(l)$  is minimized, subject to
- $\forall v \in N', \sum_{l \in path(s,v)} \mathcal{D}(l) < \Delta$ , where  $path(s, v)$  denotes the path from  $s$  to  $v$  in  $T$ .
- *constrained cheapest path* between  $x$  and  $y$  is the cheapest path between  $x$  and  $y$  having delay  $< \Delta$ .
- its cost and delay denoted  $\mathcal{C}(x, y)$ ,  $\mathcal{D}(x, y)$ , resp.



# Constrained (Delay-Bounded) Steiner Trees: Algorithm

$C(l), \mathcal{D}(l);$  // cost, delay of edge  $l$   
 $T;$  // constrained spanning tree to be constructed  
 $P(x, y);$  // path from  $x$  to  $y$   
 $\mathcal{P}_C(x, y), \mathcal{P}_D(x, y);$  // cost, delay of constrained cheapest path from  $x$  to  $y$   
 $\mathcal{C}_d(x, y);$  // cost of the cheapest path with delay exactly  $d$

Input: weighted graph  $G = (N, L)$ , and  $N' \subseteq N$ , where  $N'$  is the set of Steiner points and source  $s$ ;  $\Delta$  is the constraint on delay.

- 1 Compute the closure graph  $G'$  on  $(N', L)$ , to be the complete graph on  $N'$ . The closure graph is computed using the all-pairs constrained cheapest paths using a dynamic programming approach analogous to Floyd's algorithm. For any pair of nodes  $x, y \in N'$ :
  - $\mathcal{P}_C(x, y) = \min_{d < \Delta} \mathcal{C}_d(x, y)$  This selects the cheapest constrained path, satisfying the condition of  $\Delta$ , among the various paths possible between  $x$  and  $y$ . The various  $\mathcal{C}_d(x, y)$  can be calculated using DP as follows.
  - $\mathcal{C}_d(x, y) = \min_{z \in N} \{ \mathcal{C}_{d - \mathcal{D}(z, y)}(x, z) + \mathcal{C}(z, y) \}$  For a candidate path from  $x$  to  $y$  passing through  $z$ , the path with weight exactly  $d$  must have a delay of  $d - \mathcal{D}(z, y)$  for  $x$  to  $z$  when the edge  $(z, y)$  has delay  $\mathcal{D}(z, y)$ .

In this manner, the complete closure graph  $G'$  is computed.  $\mathcal{P}_D(x, y)$  is the constrained cheapest path that corresponds to  $\mathcal{P}_C(x, y)$ .

- 2 Construct a constrained spanning tree of  $G'$  using a greedy approach that sequentially adds edges to the subtree of the constrained spanning tree  $T$  (thus far) until all the Steiner points are included. The initial value of  $T$  is the singleton  $s$ . Consider that node  $u$  is in the tree and we are considering whether to add edge  $(u, v)$ . The following two edge selection criteria (heuristics) can be used to decide whether to include edge  $(u, v)$  in the tree.

- Heuristic  $CST_{CD}$ :  $f_{CD}(u, v) = \begin{cases} \frac{C(u, v)}{\Delta - (\mathcal{P}_D(s, u) + \mathcal{D}(u, v))}, & \text{if } \mathcal{P}_D(s, u) + \mathcal{D}(u, v) < \Delta \\ \infty, & \text{otherwise} \end{cases}$

The numerator is the "incremental cost" of adding  $(u, v)$  and the denominator is the "residual delay" that could be afforded. The goal is to minimize the incremental cost, while also maximizing the residual delay by choosing an edge that has low delay.

- Heuristic  $CST_C$ :  $f_C = \begin{cases} C(u, v), & \text{if } \mathcal{P}_D(s, u) + \mathcal{D}(u, v) < \Delta \\ \infty, & \text{otherwise} \end{cases}$

Picks the lowest cost edge between the already included tree edges and their nearest neighbour, provided total delay  $< \Delta$ .

The chosen node  $v$  is included in  $T$ . This step 2 is repeated until  $T$  includes all  $|N'|$  nodes in  $G'$ .

- 3 Expand the edges of the constrained spanning tree  $T$  on  $G'$  into the constrained cheapest paths they represent in the original graph  $G$ . Delete/break any loops introduced by this expansion.

# Constrained (Delay-Bounded) Steiner Trees: Example

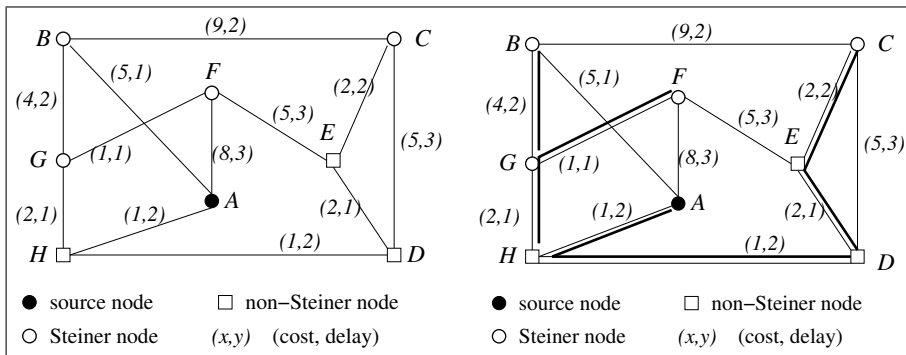


Figure 6.19: (a) Network graph. (b,c) MST and Steiner tree (optimal) are the same and shown in thick lines.

# Constrained (Delay-Bounded) Steiner Trees: Heuristics, Time Complexity

Heuristic  $CST_{CD}$ : Tries to choose low-cost edges, while also trying to maximize the remaining allowable delay.

Heuristic  $CST_C$ : Minimizes the cost while ensuring that the delay bound is met.

- step (1) which finds the constrained cheapest shortest paths over all the nodes costs  $O(n^3\Delta)$ .
- Step (2) which constructs the constrained MST on the closure graph having  $k$  nodes costs  $O(k^3)$ .
- Step (3) which expands the constrained spanning tree, involves expanding the  $k$  edges to up to  $n - 1$  edges each and then eliminating loops. This costs  $O(kn)$ .
- Dominating step is step (1).

# Core-based Trees

Multicast tree constructed dynamically, grows on demand.  
Each group has a *core* node(s)

- 1 A node wishing to join the tree as a receiver sends a unicast `join` message to the core node.
- 2 The `join` marks the edges as it travels; it either reaches the core node, or some node already part of the tree. The path followed by the `join` till the core/multicast tree is grafted to the multicast tree.
- 3 A node on the tree multicasts a message by using a flooding on the core tree.
- 4 A node not on the tree sends a message towards the core node; as soon as the message reaches any node on the tree, it is flooded on the tree.

# Causal Ordering of Messages

BSS Protocol for Broadcast

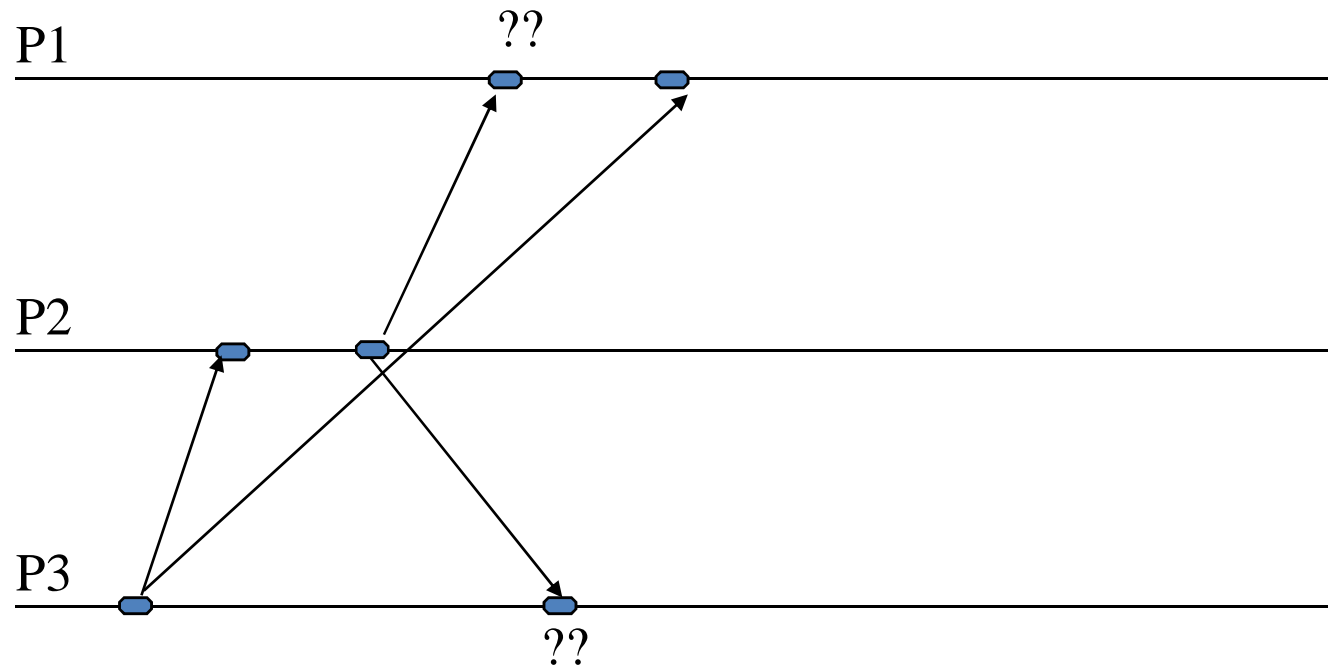
# BSS Algorithm

- BSS: Birman-Schiper-Stephenson Protocol
- Broadcast based: a message sent is received by all other processes.
- Deliver a message to a process only if the message preceding it immediately, has been delivered to the process.
- Otherwise, buffer the message.
- Accomplished by using a vector accompanying the message.

# BSS Algorithm ...

1. Process  $P_i$  increments the vector time  $VT_{pi}[i]$ , time stamps, and broadcasts the message  $m$ .
2.  $P_j \neq P_i$  receives  $m$ .  $m$  is delivered when:
  - a.  $VT_{pj}[i] == VT_m[i] - 1$
  - b.  $VT_{pj}[k] \geq VT_m[k]$  for all  $k$  in  $\{1, 2, \dots, n\} - \{i\}$ ,  $n$  is the total number of processes. Delayed messages are queued in a sorted manner.
  - c. Concurrent messages are ordered by time of receipt.
3. When  $m$  is delivered at  $P_j$ ,  $VT_{pj}$  is updated according to Rule 2 of vector clocks.

# BSS Algorithm ...





# Causal Ordering of Messages

**SES Protocol for Unicast and  
Multicast**

# Schipper-Eggli-Sandoz Protocol

- The goal of this protocol is to ensure that messages are given to the receiving processes in order of sending.
- Unlike the Birman-Schipper-Stephenson protocol, it does not require using broadcast messages.
- Each message has an associated vector that contains information for the recipient to determine if another message preceded it. Clocks are updated only when messages are sent.

# Notations used in SES Protocol

- $n$  processes
- $P_i$  process
- $C_i$  vector clock associated with process  $P_i$ ;  $j$ th element is  $C_i[j]$  and contains  $P_i$ 's latest value for the current time in process  $P_k$
- $t^m$  vector timestamp for message  $m$  (stamped after local clock is incremented)
- $t^i$  current time at process  $P_i$
- $V_i$  vector of  $P_i$ 's previously sent messages;  $V_i[j] = t^m$ , where  $P_j$  is the destination process and  $t^m$  the vector timestamp of the message;  $V_i[j][k]$  is the  $k$ th component of  $V_i[j]$ .
- $V^m$  vector accompanying message  $m$

# SES Protocol

**$P_i$  sends a message to  $P_j$**

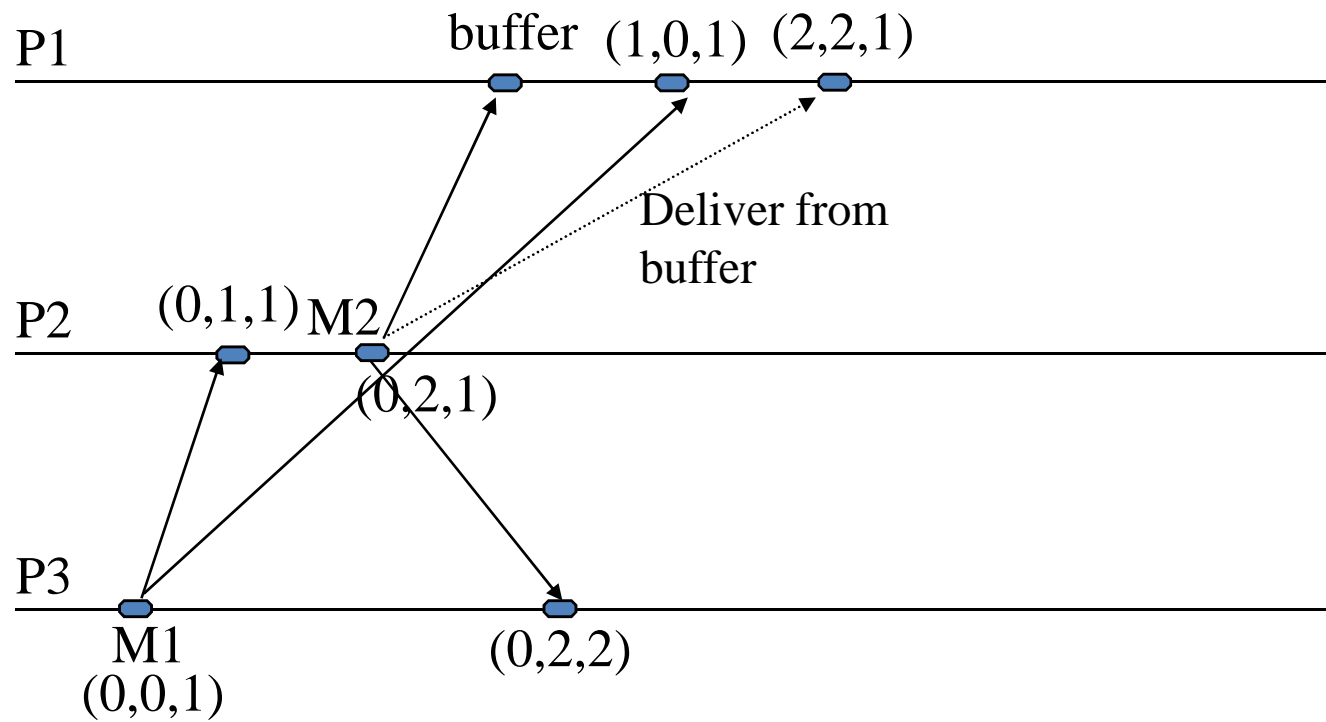
- $P_i$  sends message  $m$ , timestamped  $t^m$ , and  $V_i$ , to process  $P_j$
- $P_i$  sets  $V_i[j] = t^m$

# SES Protocol

**$P_j$  receives a message from  $P_i$**

- When  $P_j, j \neq i$ , receives  $m$ , it delays the message's delivery if both:
  - $V^m[j]$  is set; and
  - $V^m[j] < t^j$
- When the message is delivered to  $P_j$ , update all set elements of  $V_j$  with the corresponding elements of  $V^m$ , except for  $V_j[j]$ , as follows:
  - If  $V_j[k]$  and  $V^m[k]$  are uninitialized, do nothing.
  - If  $V_j[k]$  is uninitialized and  $V^m[k]$  is initialized, set  $V_j[k] = V^m[k]$ .
  - If both  $V_j[k]$  and  $V^m[k]$  are initialized, set  $V_j[k][k'] = \max(V_j[k][k'], V^m[k][k'])$  for all  $k' = 1, \dots, n$
- Update  $P_j$ 's vector clock.
- Check buffered messages to see if any can be delivered.

# SES Protocol



# Chapter 2: A Model of Distributed Computations

Ajay Kshemkalyani and Mukesh Singhal

Distributed Computing: Principles, Algorithms, and Systems

Cambridge University Press

# A Distributed Program

- A distributed program is composed of a set of  $n$  asynchronous processes,  $p_1, p_2, \dots, p_i, \dots, p_n$ .
- The processes do not share a global memory and communicate solely by passing messages.
- The processes do not share a global clock that is instantaneously accessible to these processes.
- Process execution and message transfer are asynchronous.
- Without loss of generality, we assume that each process is running on a different processor.
- Let  $C_{ij}$  denote the channel from process  $p_i$  to process  $p_j$  and let  $m_{ij}$  denote a message sent by  $p_i$  to  $p_j$ .
- The message transmission delay is finite and unpredictable.



# A Model of Distributed Executions

- The execution of a process consists of a sequential execution of its actions.
- The actions are atomic and the actions of a process are modeled as three types of events, namely, internal events, message send events, and message receive events.
- Let  $e_i^x$  denote the  $x$ th event at process  $p_i$ .
- For a message  $m$ , let  $send(m)$  and  $rec(m)$  denote its send and receive events, respectively.
- The occurrence of events changes the states of respective processes and channels.
- An internal event changes the state of the process at which it occurs.
- A send event changes the state of the process that sends the message and the state of the channel on which the message is sent.
- A receive event changes the state of the process that receives the message and the state of the channel on which the message is received.

# A Model of Distributed Executions

- The events at a process are linearly ordered by their order of occurrence.
- The execution of process  $p_i$  produces a sequence of events  $e_i^1, e_i^2, \dots, e_i^x, e_i^{x+1}, \dots$  and is denoted by  $\mathcal{H}_i$  where

$$\mathcal{H}_i = (h_i, \rightarrow_i)$$

$h_i$  is the set of events produced by  $p_i$  and  
binary relation  $\rightarrow_i$  defines a linear order on these events.

- Relation  $\rightarrow_i$  expresses causal dependencies among the events of  $p_i$ .

# A Model of Distributed Executions

- The send and the receive events signify the flow of information between processes and establish causal dependency from the sender process to the receiver process.
- A relation  $\rightarrow_{msg}$  that captures the causal dependency due to message exchange, is defined as follows. For every message  $m$  that is exchanged between two processes, we have

$$send(m) \rightarrow_{msg} rec(m).$$

- Relation  $\rightarrow_{msg}$  defines causal dependencies between the pairs of corresponding send and receive events.

# A Model of Distributed Executions

- The evolution of a distributed execution is depicted by a space-time diagram.
- A horizontal line represents the progress of the process; a dot indicates an event; a slant arrow indicates a message transfer.
- Since we assume that an event execution is atomic (hence, indivisible and instantaneous), it is justified to denote it as a dot on a process line.
- In the Figure 2.1, for process  $p_1$ , the second event is a message send event, the third event is an internal event, and the fourth event is a message receive event.

# A Model of Distributed Executions

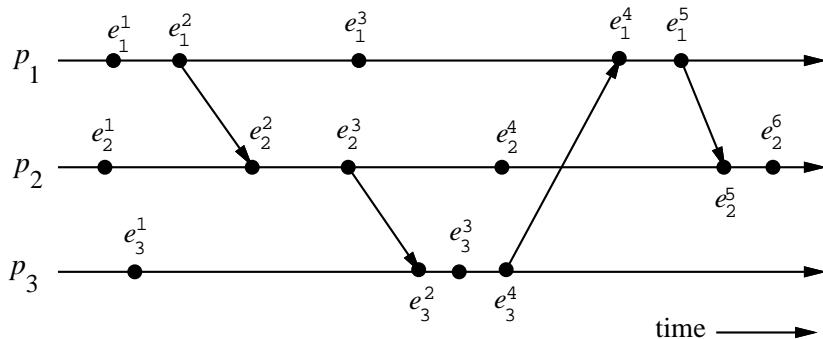


Figure 2.1: The space-time diagram of a distributed execution.

# A Model of Distributed Executions

## Causal Precedence Relation

- The execution of a distributed application results in a set of distributed events produced by the processes.
- Let  $H = \cup_i h_i$  denote the set of events executed in a distributed computation.
- Define a binary relation  $\rightarrow$  on the set  $H$  as follows that expresses causal dependencies between events in the distributed execution.

$$\forall e_i^x, \forall e_j^y \in H, \quad e_i^x \rightarrow e_j^y \quad \Leftrightarrow \quad \left\{ \begin{array}{l} e_i^x \rightarrow_i e_j^y \quad i.e., (i = j) \wedge (x < y) \\ or \\ e_i^x \rightarrow_{msg} e_j^y \\ or \\ \exists e_k^z \in H : e_i^x \rightarrow e_k^z \wedge e_k^z \rightarrow e_j^y \end{array} \right.$$

- The causal precedence relation induces an irreflexive partial order on the events of a distributed computation that is denoted as  $\mathcal{H} = (H, \rightarrow)$ .

# A Model of Distributed Executions

## ... Causal Precedence Relation

- Note that the relation  $\rightarrow$  is nothing but Lamport's "happens before" relation.
- For any two events  $e_i$  and  $e_j$ , if  $e_i \rightarrow e_j$ , then event  $e_j$  is directly or transitively dependent on event  $e_i$ . (Graphically, it means that there exists a path consisting of message arrows and process-line segments (along increasing time) in the space-time diagram that starts at  $e_i$  and ends at  $e_j$ .)
- For example, in Figure 2.1,  $e_1^1 \rightarrow e_3^3$  and  $e_3^3 \rightarrow e_2^6$ .
- The relation  $\rightarrow$  denotes flow of information in a distributed computation and  $e_i \rightarrow e_j$  dictates that all the information available at  $e_i$  is potentially accessible at  $e_j$ .
- For example, in Figure 2.1, event  $e_2^6$  has the knowledge of all other events shown in the figure.

# A Model of Distributed Executions

## ... Causal Precedence Relation

- For any two events  $e_i$  and  $e_j$ ,  $e_i \not\rightarrow e_j$  denotes the fact that event  $e_j$  does not directly or transitively dependent on event  $e_i$ . That is, event  $e_i$  does not causally affect event  $e_j$ .
- In this case, event  $e_j$  is not aware of the execution of  $e_i$  or any event executed after  $e_i$  on the same process.
- For example, in Figure 2.1,  $e_1^3 \not\rightarrow e_3^3$  and  $e_2^4 \not\rightarrow e_3^1$ .

Note the following two rules:

- For any two events  $e_i$  and  $e_j$ ,  $e_i \not\rightarrow e_j \not\Rightarrow e_j \not\rightarrow e_i$ .
- For any two events  $e_i$  and  $e_j$ ,  $e_i \rightarrow e_j \Rightarrow e_j \not\rightarrow e_i$ .



# A Model of Distributed Executions

## Concurrent events

- For any two events  $e_i$  and  $e_j$ , if  $e_i \not\rightarrow e_j$  and  $e_j \not\rightarrow e_i$ , then events  $e_i$  and  $e_j$  are said to be concurrent (denoted as  $e_i \parallel e_j$ ).
- In the execution of Figure 2.1,  $e_1^3 \parallel e_3^3$  and  $e_2^4 \parallel e_3^1$ .
- The relation  $\parallel$  is not transitive; that is,  $(e_i \parallel e_j) \wedge (e_j \parallel e_k) \not\Rightarrow e_i \parallel e_k$ .
- For example, in Figure 2.1,  $e_3^3 \parallel e_2^4$  and  $e_2^4 \parallel e_1^5$ , however,  $e_3^3 \not\parallel e_1^5$ .
- For any two events  $e_i$  and  $e_j$  in a distributed execution,  $e_i \rightarrow e_j$  or  $e_j \rightarrow e_i$ , or  $e_i \parallel e_j$ .

# A Model of Distributed Executions

## Logical vs. Physical Concurrency

- In a distributed computation, two events are logically concurrent if and only if they do not causally affect each other.
- Physical concurrency, on the other hand, has a connotation that the events occur at the same instant in physical time.
- Two or more events may be logically concurrent even though they do not occur at the same instant in physical time.
- However, if processor speed and message delays would have been different, the execution of these events could have very well coincided in physical time.
- Whether a set of logically concurrent events coincide in the physical time or not, does not change the outcome of the computation.
- Therefore, even though a set of logically concurrent events may not have occurred at the same instant in physical time, we can assume that these events occurred at the same instant in physical time.

# Models of Communication Networks

- There are several models of the service provided by communication networks, namely, FIFO, Non-FIFO, and causal ordering.
- In the FIFO model, each channel acts as a first-in first-out message queue and thus, message ordering is preserved by a channel.
- In the non-FIFO model, a channel acts like a set in which the sender process adds messages and the receiver process removes messages from it in a random order.

# Models of Communication Networks

- The “causal ordering” model is based on Lamport’s “happens before” relation.
- A system that supports the causal ordering model satisfies the following property:

CO: For any two messages  $m_{ij}$  and  $m_{kj}$ , if  $send(m_{ij}) \longrightarrow send(m_{kj})$ , then  $rec(m_{ij}) \longrightarrow rec(m_{kj})$ .

- This property ensures that causally related messages destined to the same destination are delivered in an order that is consistent with their causality relation.
- Causally ordered delivery of messages implies FIFO message delivery. (Note that  $CO \subset FIFO \subset Non-FIFO$ .)
- Causal ordering model considerably simplifies the design of distributed algorithms because it provides a built-in synchronization.

# Global State of a Distributed System

“A collection of the local states of its components, namely, the processes and the communication channels.”

- The state of a process is defined by the contents of processor registers, stacks, local memory, etc. and depends on the local context of the distributed application.
- The state of channel is given by the set of messages in transit in the channel.
- The occurrence of events changes the states of respective processes and channels.
- An internal event changes the state of the process at which it occurs.
- A send event changes the state of the process that sends the message and the state of the channel on which the message is sent.
- A receive event changes the state of the process that or receives the message and the state of the channel on which the message is received.

# ... Global State of a Distributed System

## Notations

- $LS_i^x$  denotes the state of process  $p_i$  after the occurrence of event  $e_i^x$  and before the event  $e_i^{x+1}$ .
- $LS_i^0$  denotes the initial state of process  $p_i$ .
- $LS_i^x$  is a result of the execution of all the events executed by process  $p_i$  till  $e_i^x$ .
- Let  $send(m) \leq LS_i^x$  denote the fact that  $\exists y: 1 \leq y \leq x :: e_i^y = send(m)$ .
- Let  $rec(m) \not\leq LS_i^x$  denote the fact that  $\forall y: 1 \leq y \leq x :: e_i^y \neq rec(m)$ .

# ... Global State of a Distributed System

## A Channel State

- The state of a channel depends upon the states of the processes it connects.
- Let  $SC_{ij}^{x,y}$  denote the state of a channel  $C_{ij}$ .

The state of a channel is defined as follows:

$$SC_{ij}^{x,y} = \{m_{ij} \mid send(m_{ij}) \leq e_i^x \wedge rec(m_{ij}) \not\leq e_j^y\}$$

Thus, channel state  $SC_{ij}^{x,y}$  denotes all messages that  $p_i$  sent upto event  $e_i^x$  and which process  $p_j$  had not received until event  $e_j^y$ .

# ... Global State of a Distributed System

## Global State

- The global state of a distributed system is a collection of the local states of the processes and the channels.
- Notationally, global state  $GS$  is defined as,

$$GS = \{ \bigcup_i LS_i^{x_i}, \bigcup_{j,k} SC_{jk}^{y_j, z_k} \}$$

- For a global state to be meaningful, the states of all the components of the distributed system must be recorded at the same instant.
- This will be possible if the local clocks at processes were perfectly synchronized or if there were a global system clock that can be instantaneously read by the processes. (However, both are impossible.)



# ... Global State of a Distributed System

## A Consistent Global State

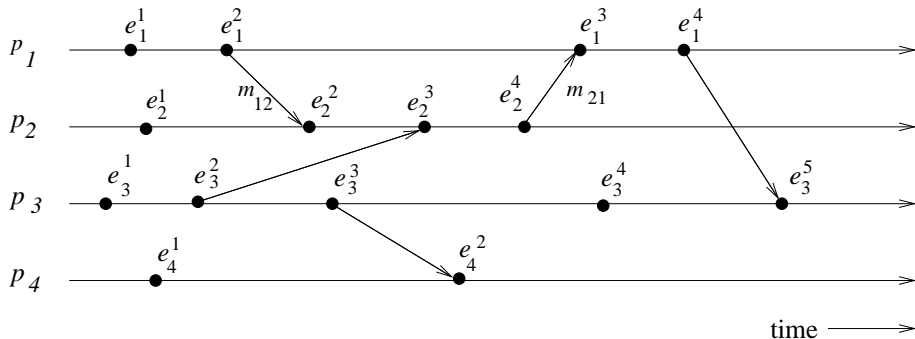
- Even if the state of all the components is not recorded at the same instant, such a state will be meaningful provided every message that is recorded as received is also recorded as sent.
- Basic idea is that a state should not violate causality – an effect should not be present without its cause. A message cannot be received if it was not sent.
- Such states are called *consistent global states* and are meaningful global states.
- Inconsistent global states are not meaningful in the sense that a distributed system can never be in an inconsistent state.
- A global state  $GS = \{\bigcup_i LS_i^{x_i}, \bigcup_{j,k} SC_{jk}^{y_j, z_k}\}$  is a *consistent global state* iff
 
$$\forall m_{ij} : send(m_{ij}) \not\leq LS_i^{x_i} \Leftrightarrow m_{ij} \notin SC_{ij}^{x_i, y_j} \wedge rec(m_{ij}) \not\leq LS_j^{y_j}$$
- That is, channel state  $SC_{ij}^{y_i, z_k}$  and process state  $LS_j^{z_k}$  must not include any message that process  $p_i$  sent after executing event  $e_i^{x_i}$ .

# ... Global State of a Distributed System

## An Example

Consider the distributed execution of Figure 2.2.

Figure 2.2: The space-time diagram of a distributed execution.



## ... Global State of a Distributed System

In Figure 2.2:

- A global state  $GS_1 = \{LS_1^1, LS_2^3, LS_3^3, LS_4^2\}$  is inconsistent because the state of  $p_2$  has recorded the receipt of message  $m_{12}$ , however, the state of  $p_1$  has not recorded its send.
- A global state  $GS_2$  consisting of local states  $\{LS_1^2, LS_2^4, LS_3^4, LS_4^2\}$  is consistent; all the channels are empty except  $C_{21}$  that contains message  $m_{21}$ .

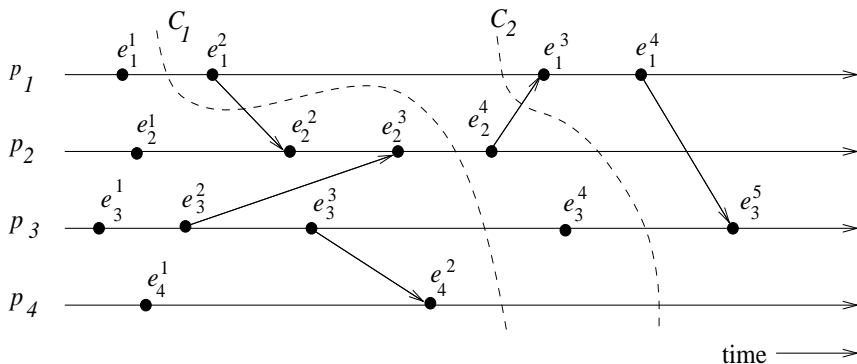
# Cuts of a Distributed Computation

“In the space-time diagram of a distributed computation, a *cut* is a zigzag line joining one arbitrary point on each process line.”

- A cut slices the space-time diagram, and thus the set of events in the distributed computation, into a PAST and a FUTURE.
- The PAST contains all the events to the left of the cut and the FUTURE contains all the events to the right of the cut.
- For a cut  $C$ , let  $PAST(C)$  and  $FUTURE(C)$  denote the set of events in the PAST and FUTURE of  $C$ , respectively.
- Every cut corresponds to a global state and every global state can be graphically represented as a cut in the computation's space-time diagram.
- Cuts in a space-time diagram provide a powerful graphical aid in representing and reasoning about global states of a computation.

# ... Cuts of a Distributed Computation

Figure 2.3: Illustration of cuts in a distributed execution.



## ... Cuts of a Distributed Computation

- In a consistent cut, every message received in the PAST of the cut was sent in the PAST of that cut. (In Figure 2.3, cut  $C_2$  is a consistent cut.)
- All messages that cross the cut from the PAST to the FUTURE are in transit in the corresponding consistent global state.
- A cut is *inconsistent* if a message crosses the cut from the FUTURE to the PAST. (In Figure 2.3, cut  $C_1$  is an inconsistent cut.)

# Past and Future Cones of an Event

## Past Cone of an Event

- An event  $e_j$  could have been affected only by all events  $e_i$  such that  $e_i \rightarrow e_j$ .
- In this situation, all the information available at  $e_i$  could be made accessible at  $e_j$ .
- All such events  $e_i$  belong to the past of  $e_j$ .

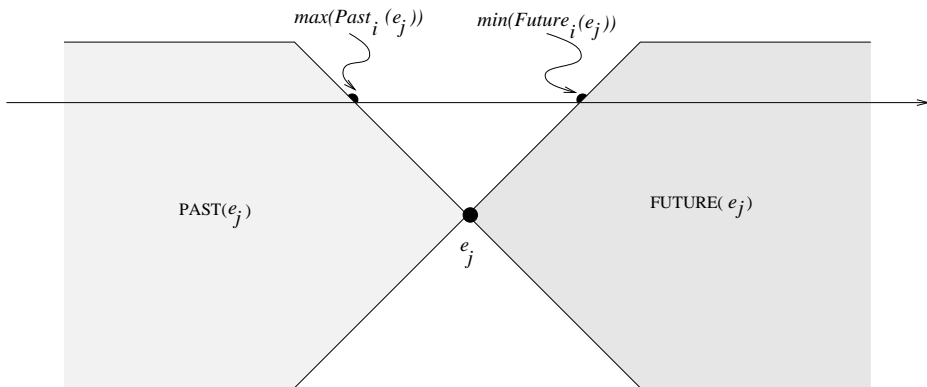
Let  $Past(e_j)$  denote all events in the past of  $e_j$  in a computation  $(H, \rightarrow)$ . Then,

$$Past(e_j) = \{e_i \mid \forall e_i \in H, e_i \rightarrow e_j\}.$$

- Figure 2.4 (next slide) shows the past of an event  $e_j$ .

# ... Past and Future Cones of an Event

Figure 2.4: Illustration of past and future cones.





## ... Past and Future Cones of an Event

- Let  $Past_i(e_j)$  be the set of all those events of  $Past(e_j)$  that are on process  $p_i$ .
- $Past_i(e_j)$  is a totally ordered set, ordered by the relation  $\rightarrow_i$ , whose maximal element is denoted by  $max(Past_i(e_j))$ .
- $max(Past_i(e_j))$  is the latest event at process  $p_i$  that affected event  $e_j$  (Figure 2.4).

## ... Past and Future Cones of an Event

- Let  $Max\_Past(e_j) = \bigcup_{(i)} \{max(Past_i(e_j))\}$ .
- $Max\_Past(e_j)$  consists of the latest event at every process that affected event  $e_j$  and is referred to as the *surface of the past cone* of  $e_j$ .
- $Past(e_j)$  represents all events on the past light cone that affect  $e_j$ .

### Future Cone of an Event

- The future of an event  $e_j$ , denoted by  $Future(e_j)$ , contains all events  $e_i$  that are causally affected by  $e_j$  (see Figure 2.4).
- In a computation  $(H, \rightarrow)$ ,  $Future(e_j)$  is defined as:

$$Future(e_j) = \{e_i | \forall e_i \in H, e_j \rightarrow e_i\}.$$

## ... Past and Future Cones of an Event

- Define  $Future_i(e_j)$  as the set of those events of  $Future(e_j)$  that are on process  $p_i$ .
- define  $\min(Future_i(e_j))$  as the first event on process  $p_i$  that is affected by  $e_j$ .
- Define  $Min\_Future(e_j)$  as  $\bigcup_{(\forall i)} \{\min(Future_i(e_j))\}$ , which consists of the first event at every process that is causally affected by event  $e_j$ .
- $Min\_Future(e_j)$  is referred to as the *surface of the future cone* of  $e_j$ .
- All events at a process  $p_i$  that occurred after  $\max(Past_i(e_j))$  but before  $\min(Future_i(e_j))$  are concurrent with  $e_j$ .
- Therefore, all and only those events of computation  $H$  that belong to the set " $H - Past(e_j) - Future(e_j)$ " are concurrent with event  $e_j$ .

# Models of Process Communications

- There are two basic models of process communications – synchronous and asynchronous.
- The *synchronous* communication model is a blocking type where on a message send, the sender process blocks until the message has been received by the receiver process.
- The sender process resumes execution only after it learns that the receiver process has accepted the message.
- Thus, the sender and the receiver processes must synchronize to exchange a message. On the other hand,
- *asynchronous* communication model is a non-blocking type where the sender and the receiver do not synchronize to exchange a message.
- After having sent a message, the sender process does not wait for the message to be delivered to the receiver process.
- The message is buffered by the system and is delivered to the receiver process when it is ready to accept the message.

## ... Models of Process Communications

- Neither of the communication models is superior to the other.
- Asynchronous communication provides higher parallelism because the sender process can execute while the message is in transit to the receiver.
- However, A buffer overflow may occur if a process sends a large number of messages in a burst to another process.
- Thus, an implementation of asynchronous communication requires more complex buffer management.
- In addition, due to higher degree of parallelism and non-determinism, it is much more difficult to design, verify, and implement distributed algorithms for asynchronous communications.
- Synchronous communication is simpler to handle and implement.
- However, due to frequent blocking, it is likely to have poor performance and is likely to be more prone to deadlocks.

# Chapter 4: Global State and Snapshot Recording Algorithms

Ajay Kshemkalyani and Mukesh Singhal

Distributed Computing: Principles, Algorithms, and Systems

Cambridge University Press

# Introduction

- Recording the global state of a distributed system on-the-fly is an important paradigm.
- The lack of globally shared memory, global clock and unpredictable message delays in a distributed system make this problem non-trivial.
- This chapter first defines consistent global states and discusses issues to be addressed to compute consistent distributed snapshots.
- Then several algorithms to determine on-the-fly such snapshots are presented for several types of networks.

# System model

- The system consists of a collection of  $n$  processes  $p_1, p_2, \dots, p_n$  that are connected by channels.
- There are no globally shared memory and physical global clock and processes communicate by passing messages through communication channels.
- $C_{ij}$  denotes the channel from process  $p_i$  to process  $p_j$  and its state is denoted by  $SC_{ij}$ .
- The actions performed by a process are modeled as three types of events: Internal events, the message send event and the message receive event.
- For a message  $m_{ij}$  that is sent by process  $p_i$  to process  $p_j$ , let  $send(m_{ij})$  and  $rec(m_{ij})$  denote its send and receive events.



# System model

- At any instant, the state of process  $p_i$ , denoted by  $LS_i$ , is a result of the sequence of all the events executed by  $p_i$  till that instant.
- For an event  $e$  and a process state  $LS_i$ ,  $e \in LS_i$  iff  $e$  belongs to the sequence of events that have taken process  $p_i$  to state  $LS_i$ .
- For an event  $e$  and a process state  $LS_i$ ,  $e \notin LS_i$  iff  $e$  does not belong to the sequence of events that have taken process  $p_i$  to state  $LS_i$ .
- For a channel  $C_{ij}$ , the following set of messages can be defined based on the local states of the processes  $p_i$  and  $p_j$

**Transit:**  $transit(LS_i, LS_j) = \{m_{ij} \mid send(m_{ij}) \in LS_i \wedge rec(m_{ij}) \notin LS_j\}$

# Models of communication

Recall, there are three models of communication: FIFO, non-FIFO, and Co.

- In FIFO model, each channel acts as a first-in first-out message queue and thus, message ordering is preserved by a channel.
- In non-FIFO model, a channel acts like a set in which the sender process adds messages and the receiver process removes messages from it in a random order.
- A system that supports causal delivery of messages satisfies the following property: “For any two messages  $m_{ij}$  and  $m_{kj}$ , if  $send(m_{ij}) \longrightarrow send(m_{kj})$ , then  $rec(m_{ij}) \longrightarrow rec(m_{kj})$ ”.

# Consistent global state

- The global state of a distributed system is a collection of the local states of the processes and the channels.
- Notationally, global state  $GS$  is defined as,

$$GS = \{\bigcup_i LS_i, \bigcup_{i,j} SC_{ij}\}$$

- A global state  $GS$  is a *consistent global state* iff it satisfies the following two conditions :

C1:  $\text{send}(m_{ij}) \in LS_i \Rightarrow m_{ij} \in SC_{ij} \oplus \text{rec}(m_{ij}) \in LS_j$ . ( $\oplus$  is Ex-OR operator.)

C2:  $\text{send}(m_{ij}) \notin LS_i \Rightarrow m_{ij} \notin SC_{ij} \wedge \text{rec}(m_{ij}) \notin LS_j$ .

## Interpretation in terms of cuts

- A cut in a space-time diagram is a line joining an arbitrary point on each process line that slices the space-time diagram into a PAST and a FUTURE.
- A consistent global state corresponds to a cut in which every message received in the PAST of the cut was sent in the PAST of that cut.
- Such a cut is known as a *consistent cut*.
- For example, consider the space-time diagram for the computation illustrated in Figure 4.1.
- Cut C1 is inconsistent because message m1 is flowing from the FUTURE to the PAST.
- Cut C2 is consistent and message m4 must be captured in the state of channel  $C_{21}$ .

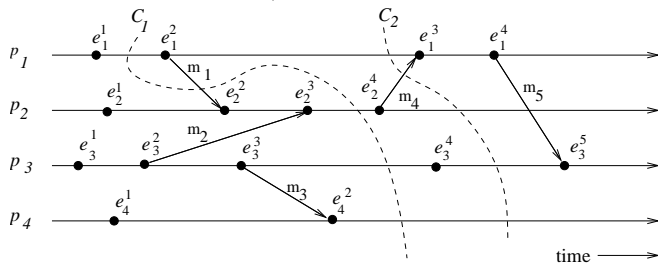


Figure 4.1: An Interpretation in Terms of a Cut.

# Issues in recording a global state

The following two issues need to be addressed:

- 11: How to distinguish between the messages to be recorded in the snapshot from those not to be recorded.

- Any message that is sent by a process before recording its snapshot, must be recorded in the global snapshot (from **C1**).
- Any message that is sent by a process after recording its snapshot, must not be recorded in the global snapshot (from **C2**).

- 12: How to determine the instant when a process takes its snapshot.

- A process  $p_j$  must record its snapshot before processing a message  $m_{ij}$  that was sent by process  $p_i$  after recording its snapshot.

# Snapshot algorithms for FIFO channels

## Chandy-Lamport algorithm

- The Chandy-Lamport algorithm uses a control message, called a *marker* whose role in a FIFO system is to separate messages in the channels.
- After a site has recorded its snapshot, it sends a *marker*, along all of its outgoing channels before sending out any more messages.
- A marker separates the messages in the channel into those to be included in the snapshot from those not to be recorded in the snapshot.
- A process must record its snapshot no later than when it receives a marker on any of its incoming channels.

# Chandy-Lamport algorithm

- The algorithm can be initiated by any process by executing the “Marker Sending Rule” by which it records its local state and sends a marker on each outgoing channel.
- A process executes the “Marker Receiving Rule” on receiving a marker. If the process has not yet recorded its local state, it records the state of the channel on which the marker is received as empty and executes the “Marker Sending Rule” to record its local state.
- The algorithm terminates after each process has received a marker on all of its incoming channels.
- All the local snapshots get disseminated to all other processes and all the processes can determine the global state.



# Chandy-Lamport algorithm

## Marker Sending Rule for process $i$

- 1 Process  $i$  records its state.
- 2 For each outgoing channel  $C$  on which a marker has not been sent,  $i$  sends a marker along  $C$  before  $i$  sends further messages along  $C$ .

## Marker Receiving Rule for process $j$

On receiving a marker along channel  $C$ :

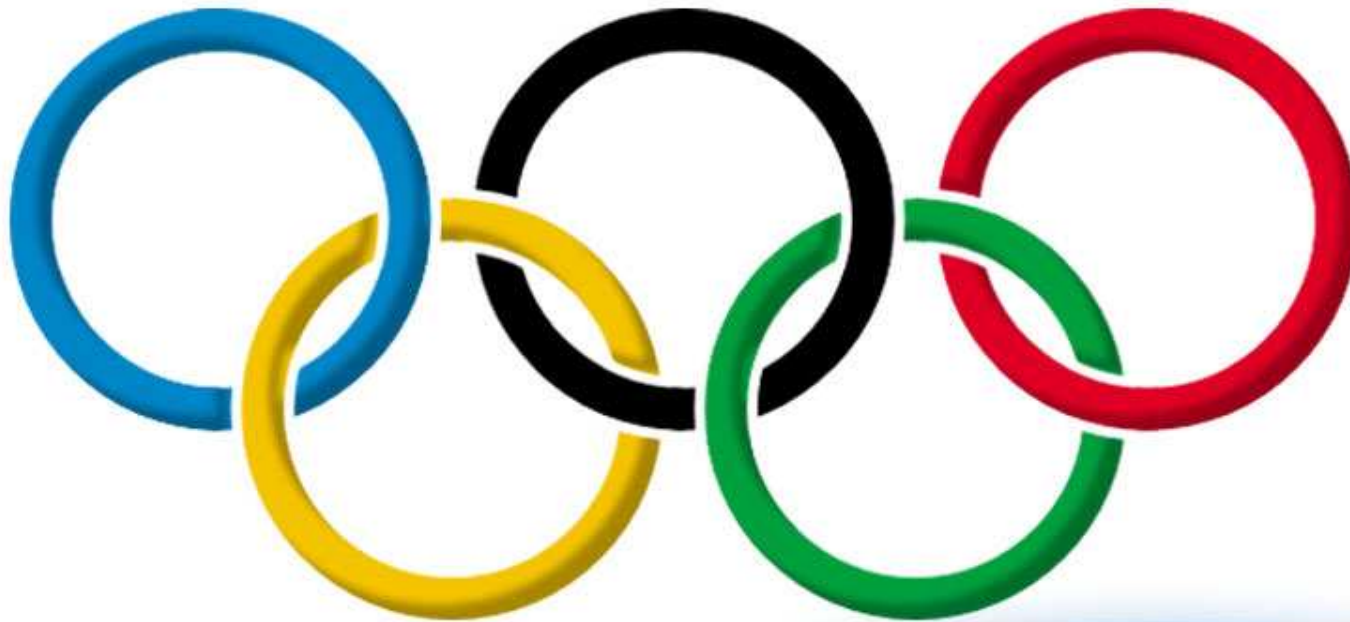
**if**  $j$  has not recorded its state **then**

Record the state of  $C$  as the empty set

Follow the “Marker Sending Rule”

**else**

Record the state of  $C$  as the set of messages received along  $C$  after  $j$ 's state was recorded and before  $j$  received the marker along  $C$

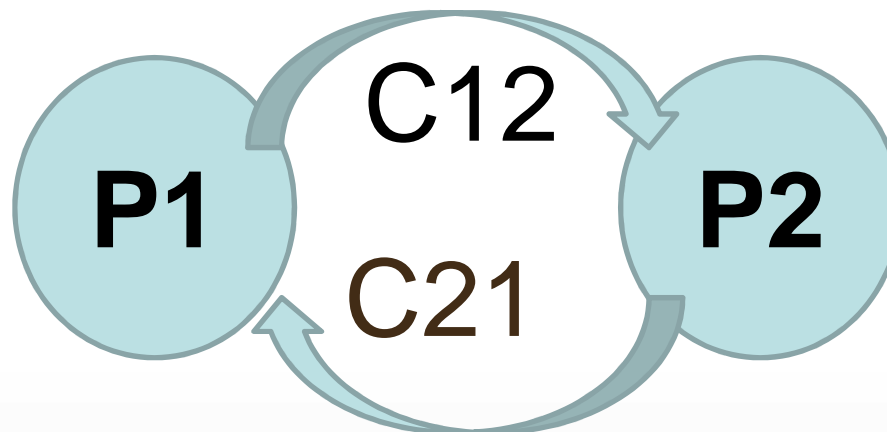


# **Chandy-Lamport Snapshot Algorithm - Global State**

**Y. V. Lokeswari , AP/ CSE**

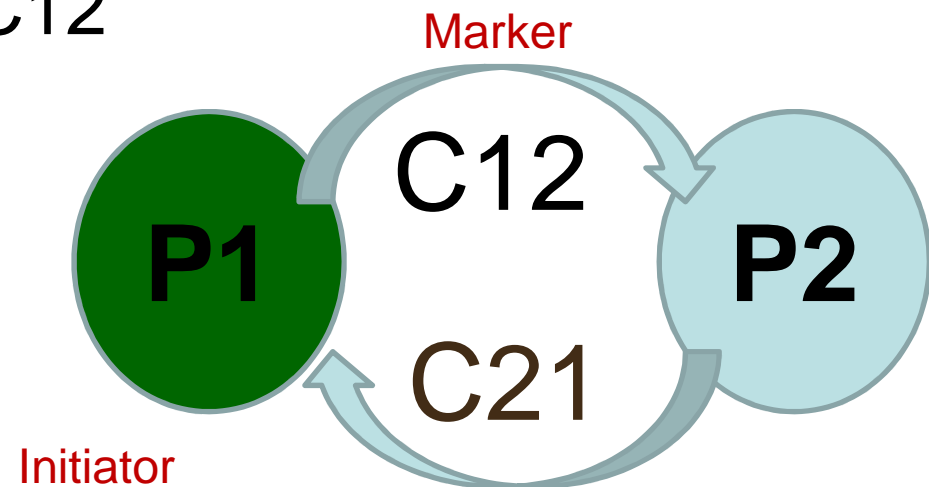
# Chandy-Lamport Algorithm - Example

- Consider 2 processes P1 and P2
- Local state of P1 is recorded as LS1
- Local State of P2 is recorded as LS2
- Channel C12 sends message from P1 to P2. Local state is SC12
- Channel C21 sends message from P2 to P1. Local state is SC21



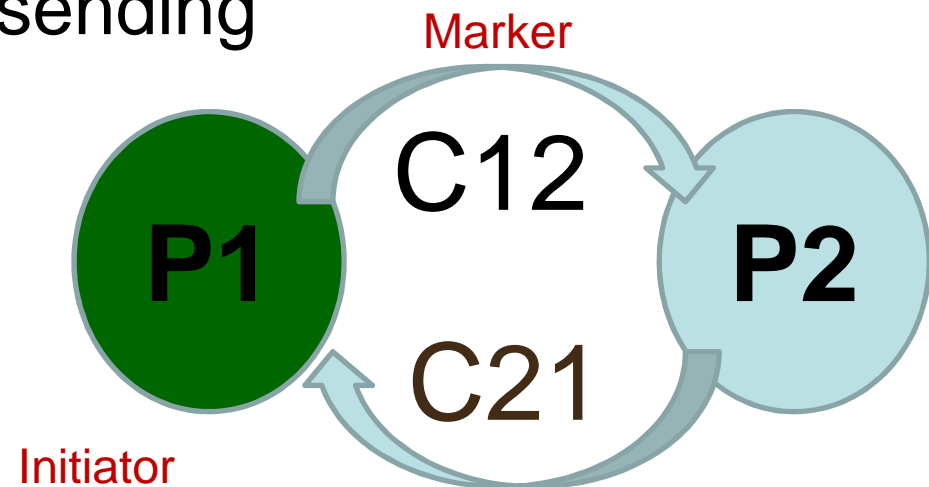
# Chandy-Lamport Algorithm - Example

- P1 initiates Marker Sending
  - P1 records its own local state LS1 and
  - Sends marker to all outgoing channels
  - Here it is Channel C12



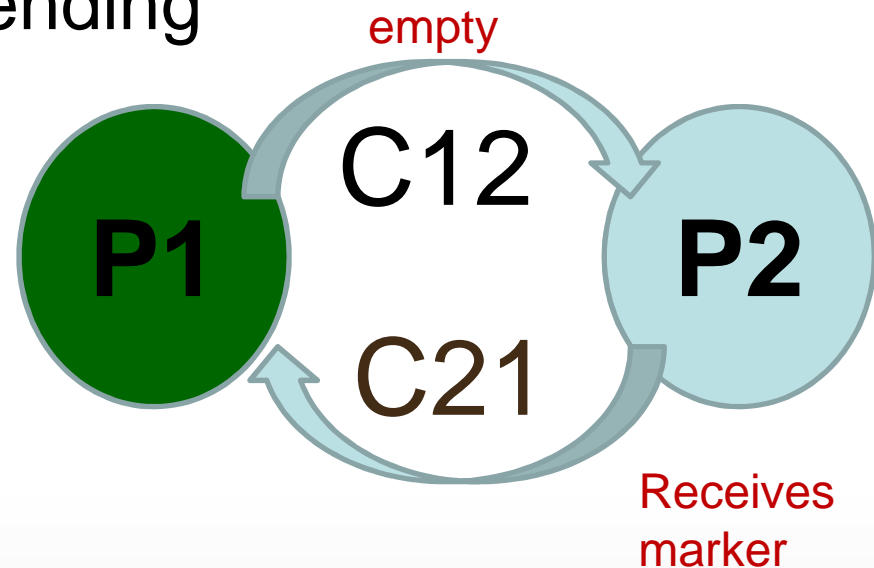
# Chandy-Lamport Algorithm - Example

- P2 receives Marker
  - P2's local state is not recorded
  - P2 records channel C12 as empty
  - P2 initiates marker sending



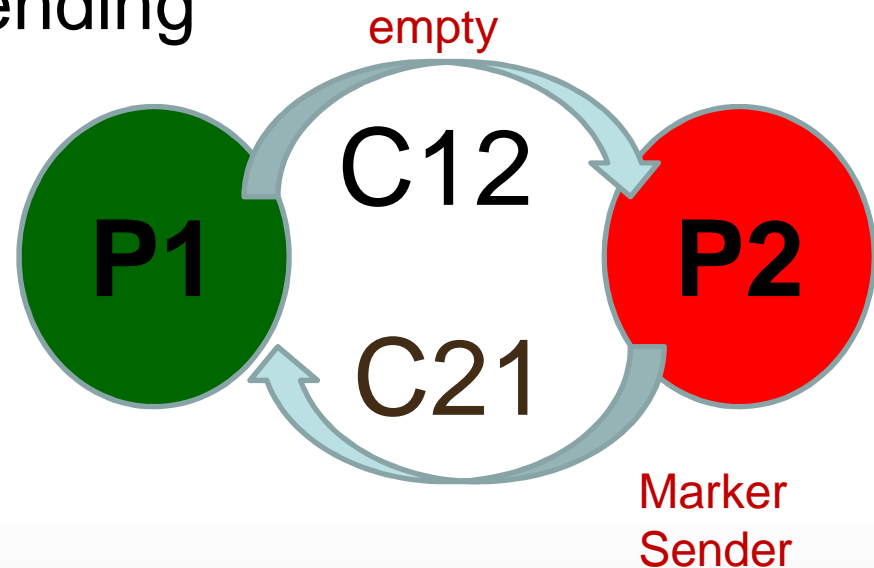
# Chandy-Lamport Algorithm - Example

- P2 receives Marker
  - P2's local state is not recorded
  - P2 records channel C12 as empty
  - P2 initiates marker sending



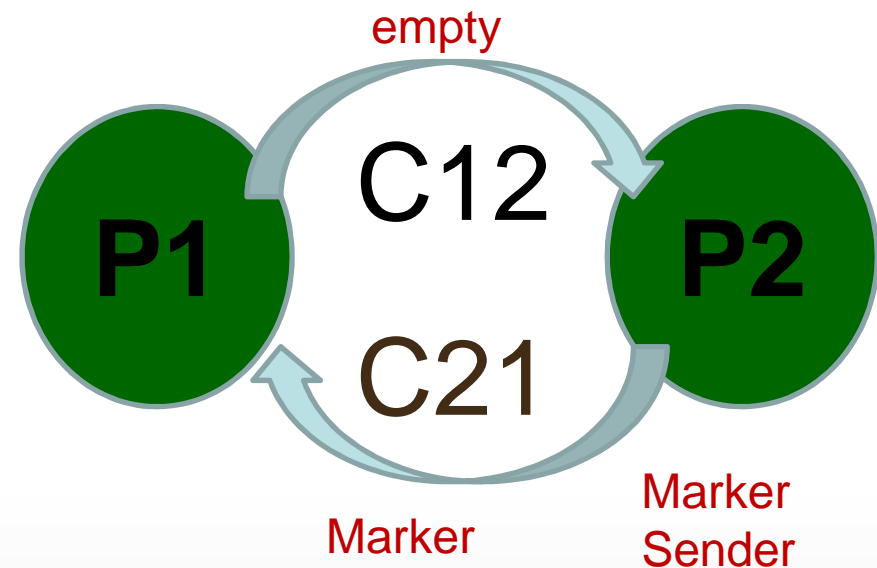
# Chandy-Lamport Algorithm - Example

- P2 receives Marker
  - P2's local state is not recorded
  - P2 records channel C12 as empty
  - P2 initiates marker sending



# Chandy-Lamport Algorithm - Example

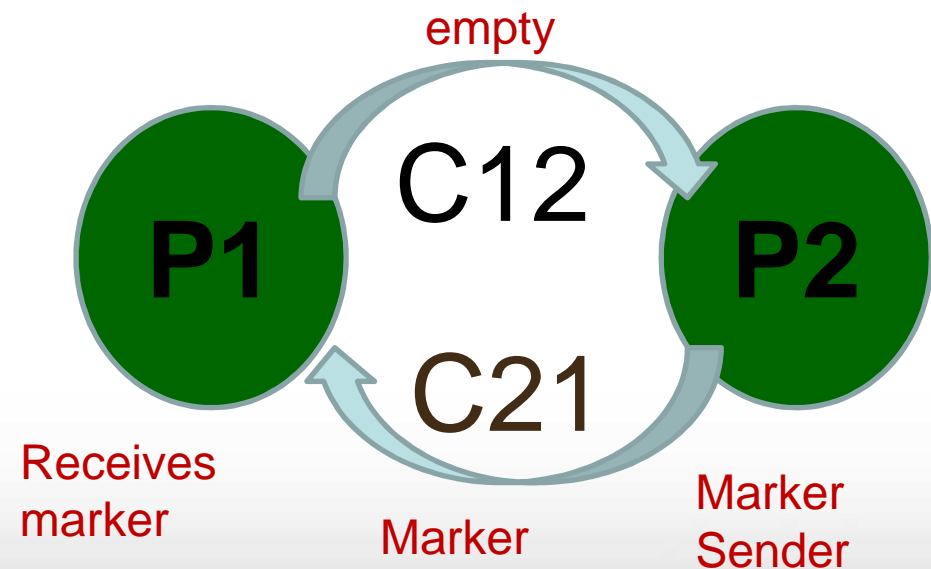
- P2 follows Marker sending rule
  - P2's local state LS2 is now recorded
  - P2 sends marker to all of its outgoing channels
  - Here channel C21





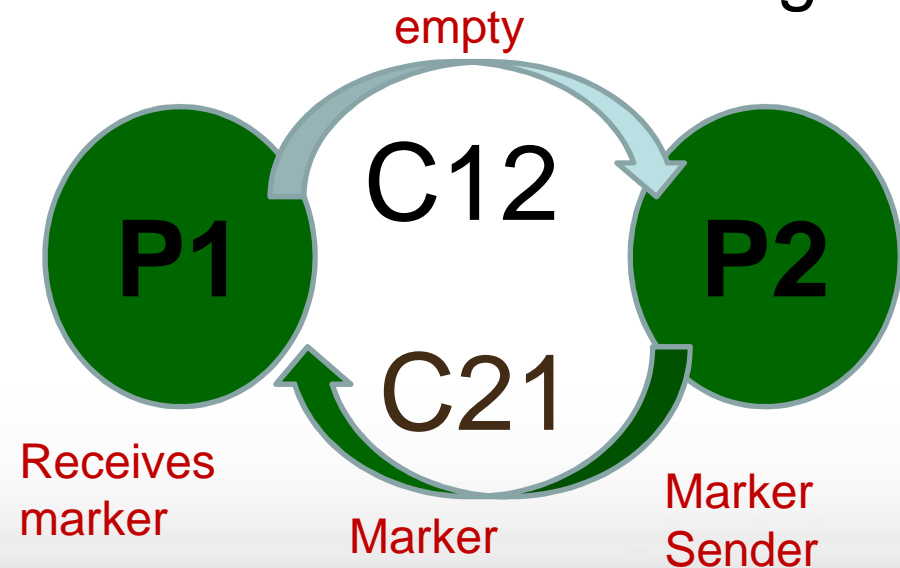
# Chandy-Lamport Algorithm - Example

- P1 receives marker
  - P1's local state LS1 is already recorded
  - Records all the messages in channel C21 right from the point P1 recorded its local state



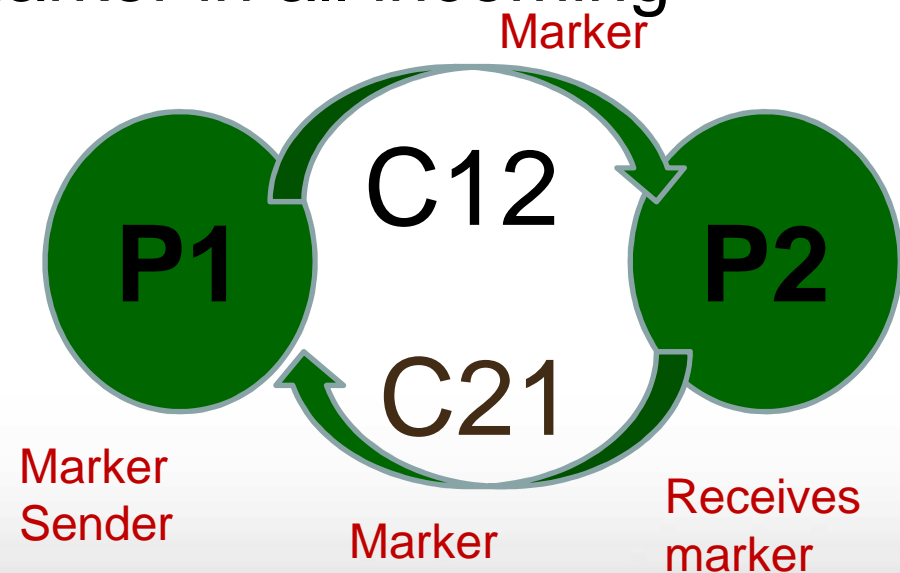
# Chandy-Lamport Algorithm - Example

- P1 receives marker
  - P1's local state LS1 is already recorded
  - Records all the messages in channel C21 right from the point P1 recorded its local state till P1 receives second marker in all incoming channels

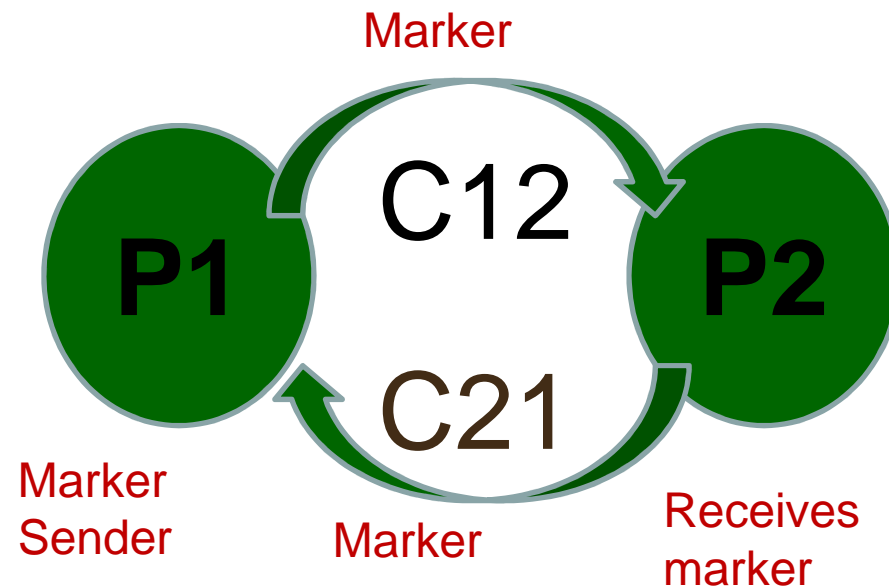


# Chandy-Lamport Algorithm - Example

- P1 follows marker sending and P2 receives it
  - P2's local state LS2 is already recorded
  - Records all the messages in channel C12 right from the point P2 recorded its local state till P2 receives second marker in all incoming channels

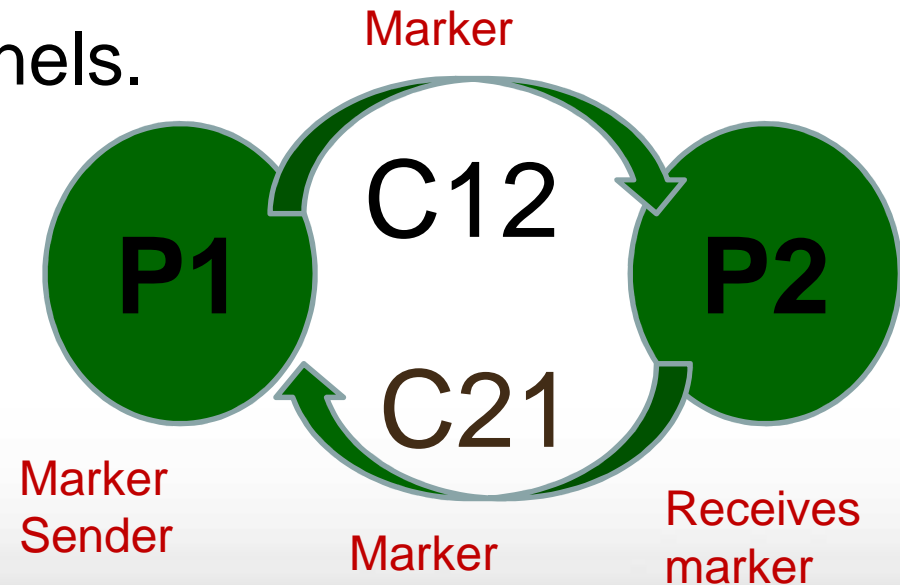


# Chandy-Lamport Algorithm - Example

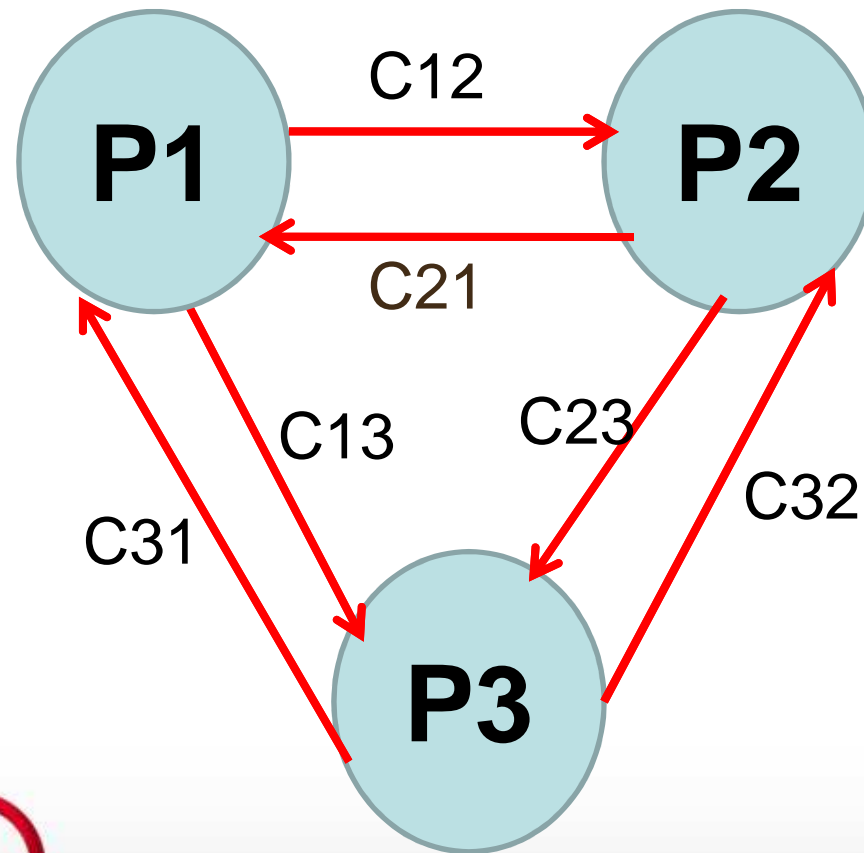


# Chandy-Lamport Algorithm - Example

- **When does algorithm terminates:**
  - The local state is disseminated to all other nodes in the form of second marker message. When all of the processes in the Distributed systems have received second marker in all of its incoming channels.

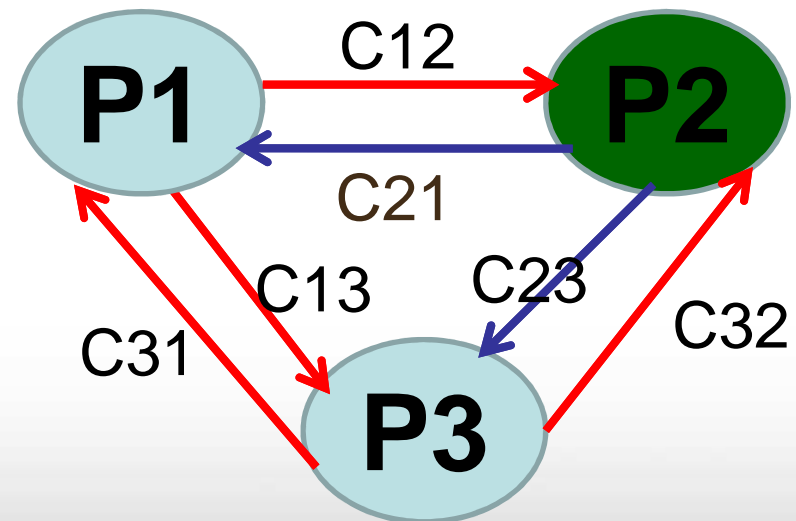


# Chandy-Lamport Algorithm - Example



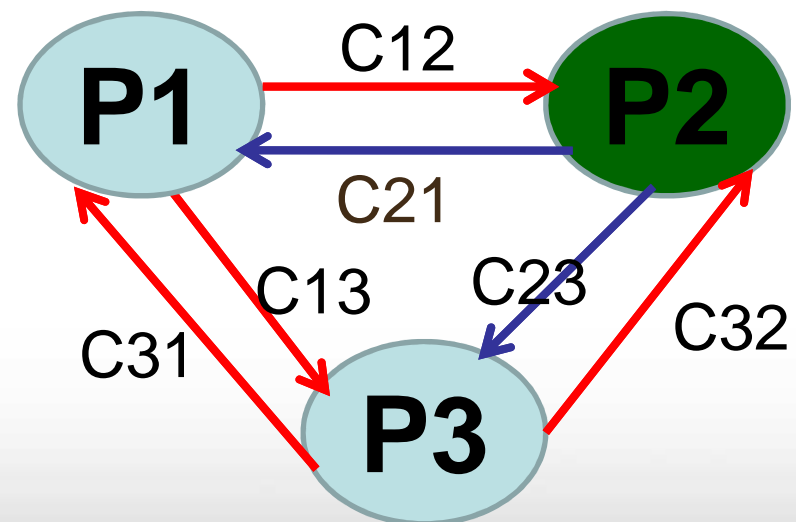
# Chandy-Lamport Algorithm - Example

- P2 is initiator.
  - P2 records its local state LS2 and P2 sends marker as follows  
P2 -----C21----- > P1  
P2 -----C23 ----- > P3
  - P1 and P3 receives marker and their states are not recorded.



# Chandy-Lamport Algorithm - Example

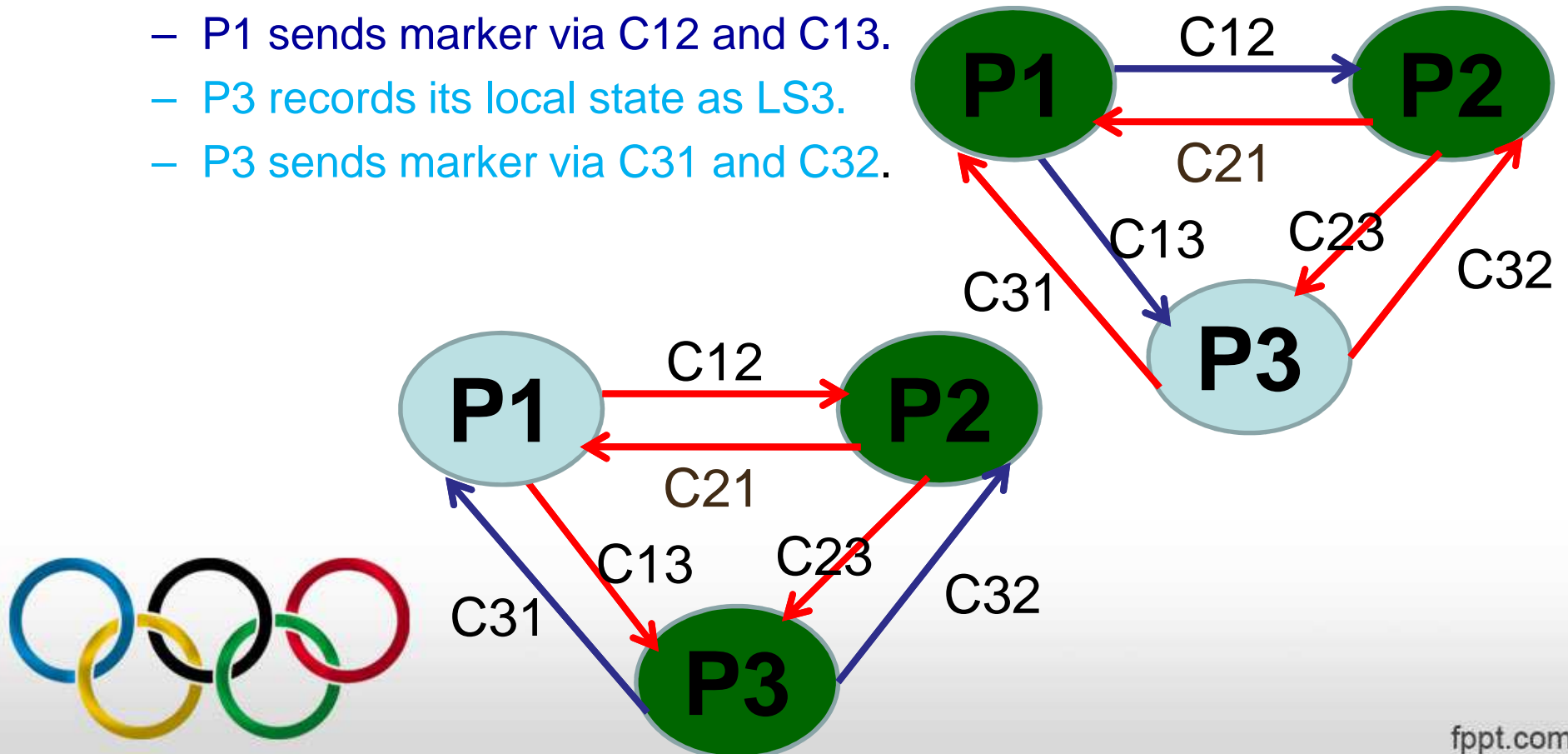
- P1 and P3 receives.
  - P1 records C21 as empty.
  - P3 records C23 as empty
  - P1 and P3 follows Marker sending rule.





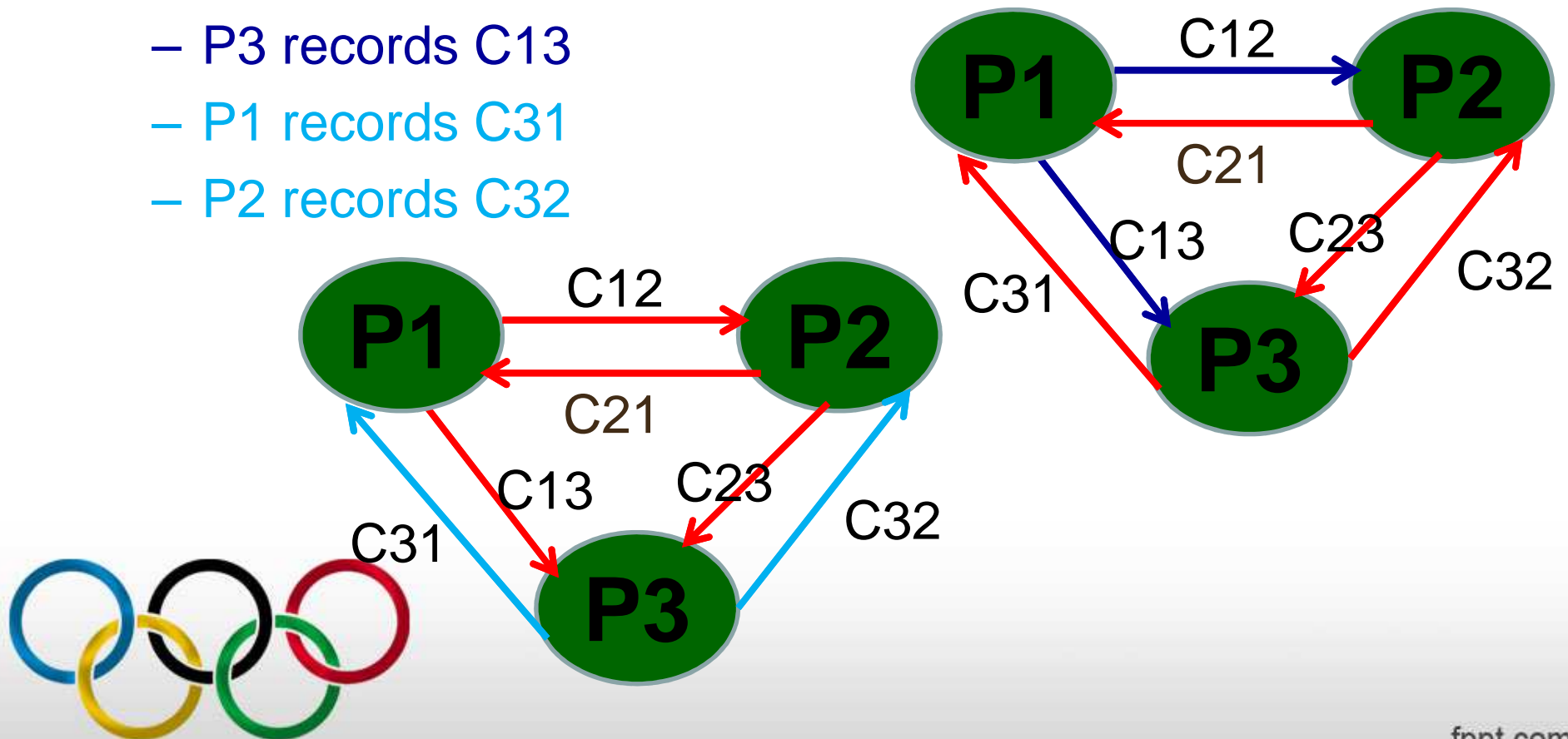
# Chandy-Lamport Algorithm - Example

- P1 and P3 follows Marker sending rule
  - P1 records its local state as LS1.
  - P1 sends marker via C12 and C13.
  - P3 records its local state as LS3.
  - P3 sends marker via C31 and C32.



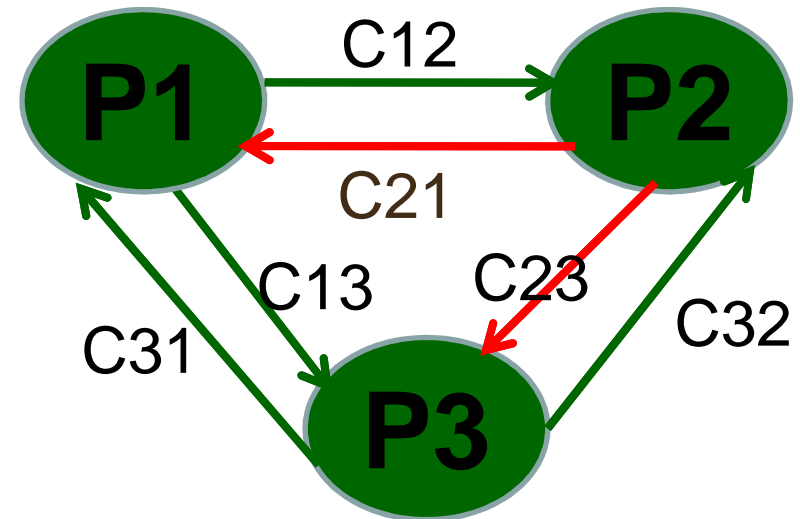
# Chandy-Lamport Algorithm - Example

- P1, P2 & P3 follows Marker Receiving rule
  - P2 records C12
  - P3 records C13
  - P1 records C31
  - P2 records C32



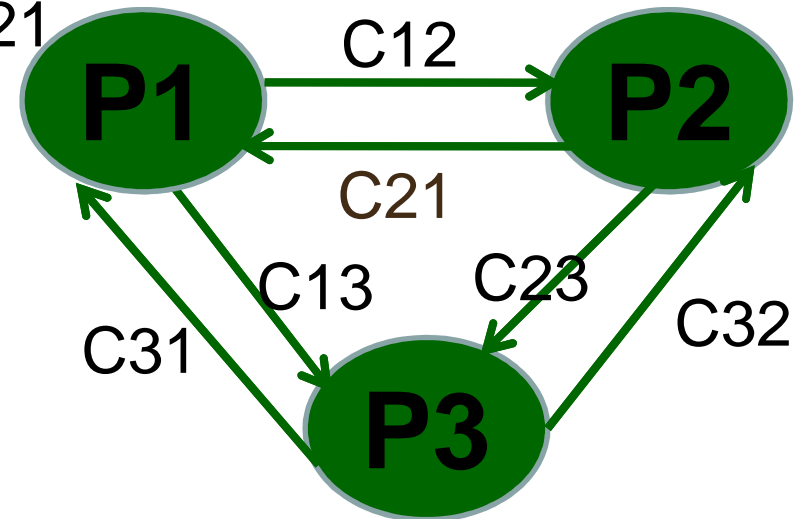
# Chandy-Lamport Algorithm - Example

- P1, P2, P3 receives marker and P1, P2, P3 follow marker sending rule as its states are already recorded, they just record the state of channel right from the point P1, P2 and P3 recorded their states.



# Chandy-Lamport Algorithm - Example

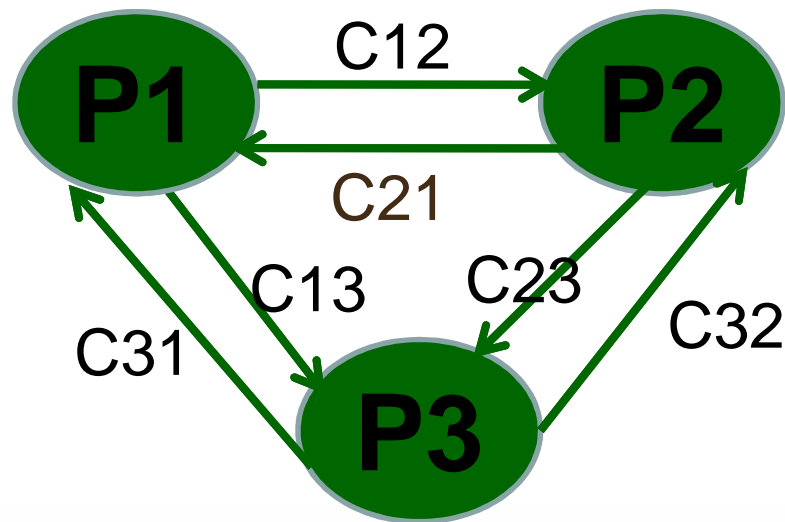
- P3 sends marker to P1 via C31 and P2 via C32 (Already recorded)
- P2 sends marker to P1 via C21 and P3 via C23 (C21 and C23 are recorded)
- P1 sends marker to P2 via C12 and P3 via C13 (Already recorded)



# Chandy-Lamport Algorithm - Example

- All the processes received marker in its all incoming channels.

Chandy-Lamport Snapshot algorithm terminates



# Chandy-Lamport Algorithm - Example

- Every process will exchange its recorded local state and channel state to all the other processes.
- All the local states are combined to form a Global State.
- To find whether a global state is consistent or not, introduce cuts and check for Consistency or Inconsistency




# Synchronizing Physical Clock Logical Clock & Vector Clock

Y. V. Lokeswari

ASP/CSE



# Overview

- Physical Clocks
  - Synchronizing Physical Clock (Algorithms)
  - Problems with Physical Clock
  - Lamport's Logical Clock
  - Problems with Logical Clock
  - Vector Clock
  - Drawbacks of Vector Clocks
  - Applications of Vector Clocks
- 



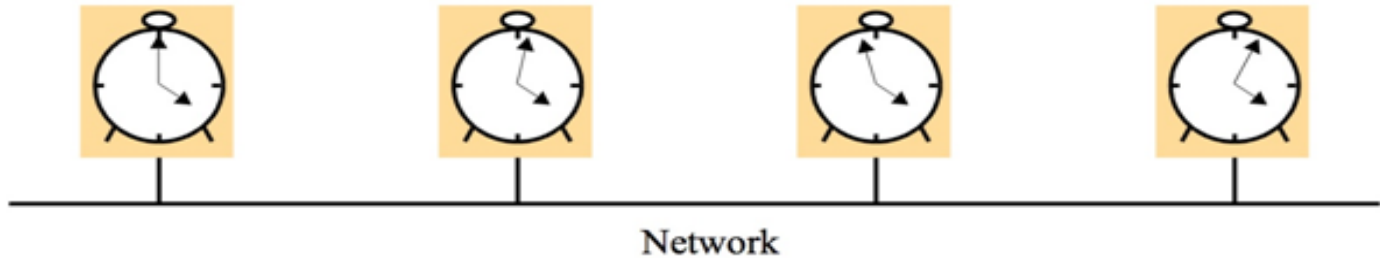
# Clock Synchronization

- Temporal ordering of events produced by concurrent processes
- Synchronization between senders and receivers of messages
- Serialization of concurrent access for shared objects
- **Physical Clock:** It is the internal clock present in a computer.  
(Time of a day)
- **Logical Clock:** keeps track of event ordering among related  
(causal) events.

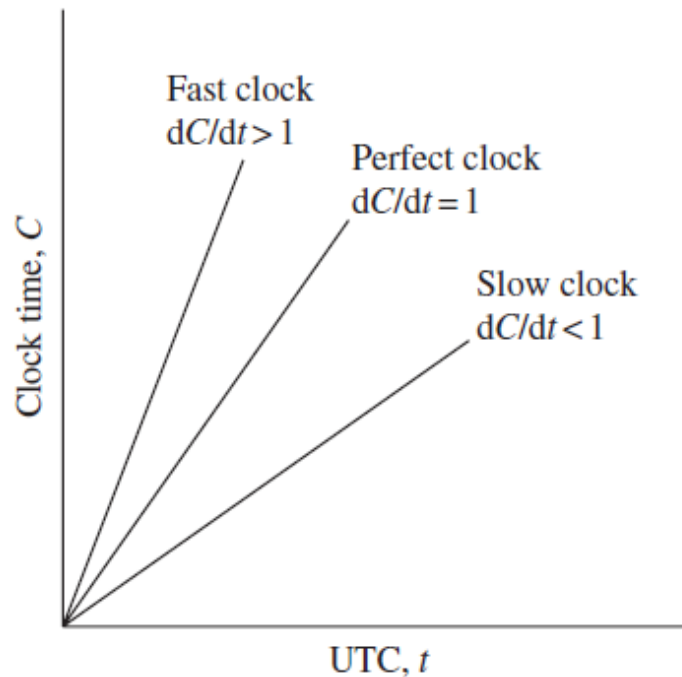


# Clock Synchronization

- Getting two systems to agree on time
  - Two clocks hardly ever agree
  - **Quartz oscillators oscillate at slightly different frequencies**
- **Clock Drift** : Clocks tick at different rates. Create ever-widening gap in perceived time
  - Clock drift in ordinary clocks based on quartz crystal is  $10^{-6}$  seconds.
  - This creates a difference of 1 sec for every 11.6 days (1,000,000 sec)
  - Clock drift of high precision clock is  $10^{-7}$  to  $10^{-8}$
- **Clock Skew** : Difference between two clocks at one point in time.



# Clock Synchronization

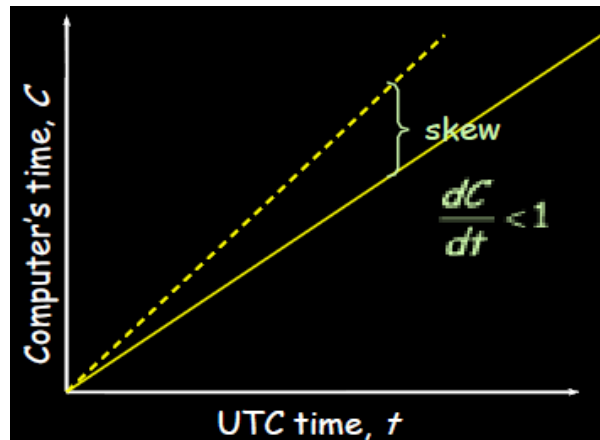
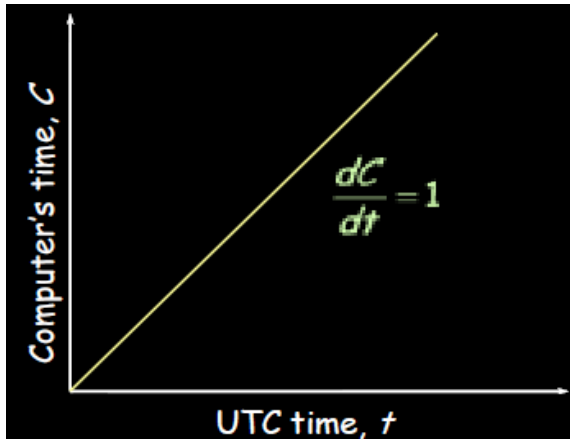


**Figure 3.8** The behavior of fast, slow, and perfect clocks with respect to UTC.

# Clock Synchronization

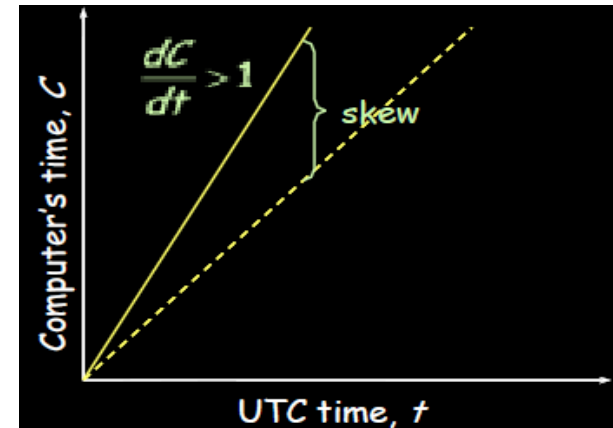
- Dealing with Clock Drift: Go for gradual clock correction

UTC : Universal Coordinated Time



If slow:

Make clock run faster until  
it Synchronizes

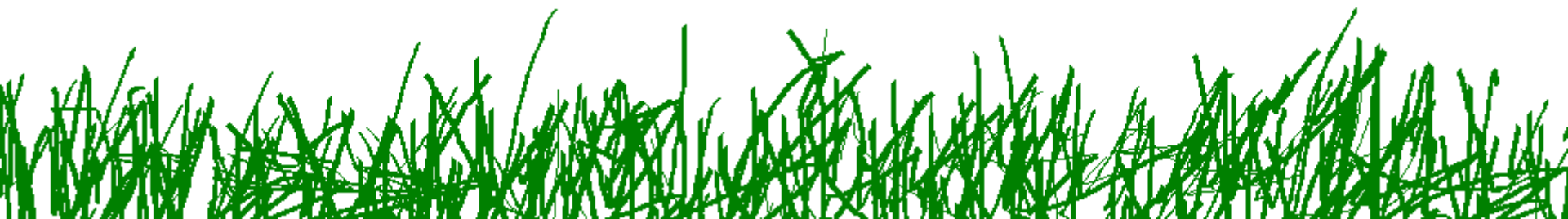


If fast:

Make clock run slower  
until it Synchronizes

# Clock Synchronization

- **External Synchronization** : Clock synchronizes to correct time from external timing elements like Radio / Satellite time.
- **Internal Synchronization** : Clock synchronizes to correct time by getting timings from other computers.
- 3 - Algorithms for Synchronizing Physical Clock
  1. Cristian's Algorithm
  2. Berkeley Algorithm
  3. Network Time Protocol (NTP)

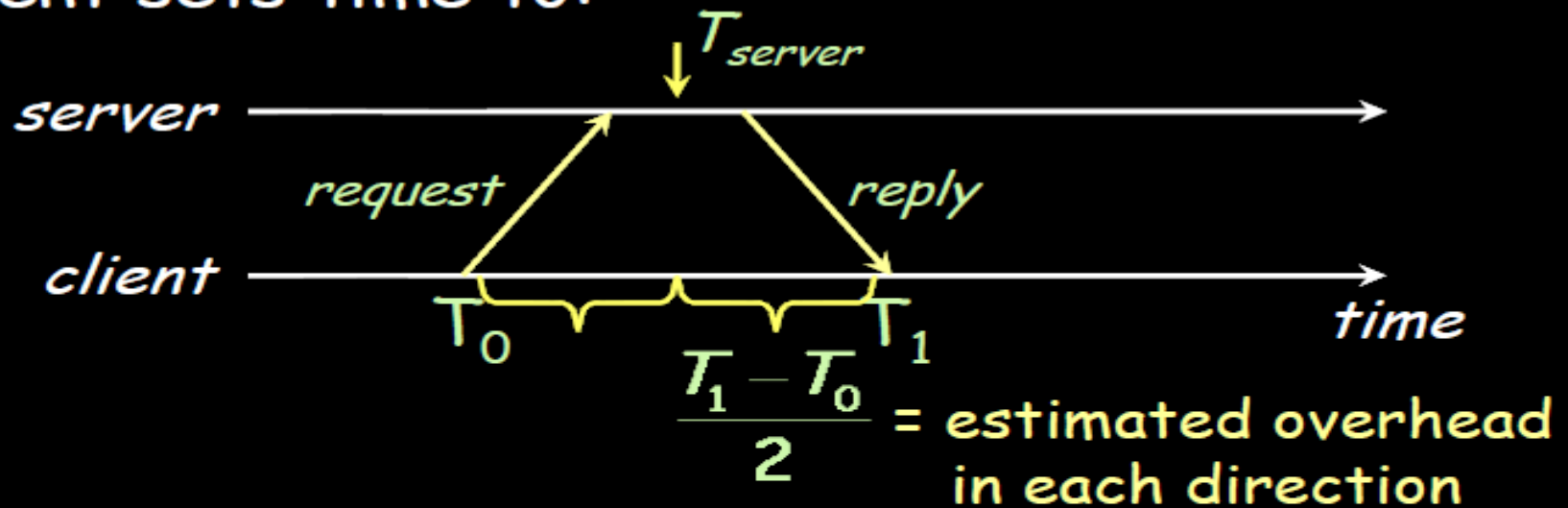


# Clock Synchronization

## 1. Cristian's Algorithm

$(T_1 - T_0)/2$  is round-trip time

Client sets time to:



$$T_{new} = T_{server} + \frac{T_1 - T_0}{2}$$

# Clock Synchronization

## 1. Cristian's Algorithm – Example

- Send request at 5:08:15.100 (T<sub>0</sub>)
- Receive response at 5:08:15.900 (T<sub>1</sub>)
- Response contains 5:09:25.300 (T<sub>server</sub>)
- Elapsed time is T<sub>1</sub> - T<sub>0</sub>

$$5:08:15.900 - 5:08:15.100 = 800 \text{ msec}$$

- Best guess: timestamp was generated

$$400 \text{ msec ago } (800/2)$$

- Set time to T<sub>server</sub> + elapsed time

$$5:09:25.300 + 400 = 5:09:25.700$$


# Clock Synchronization

## 2. Berkeley Algorithm

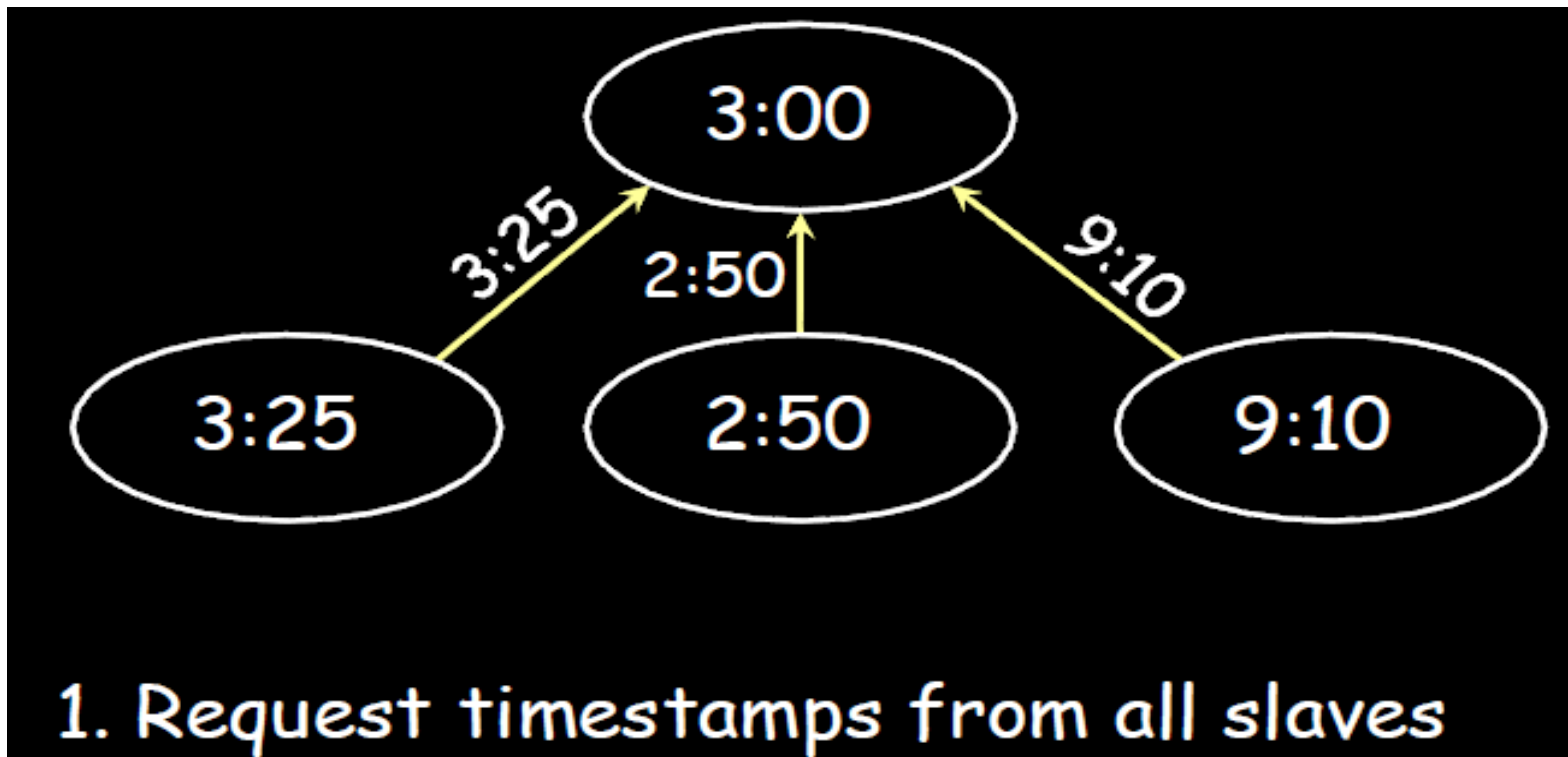
- Machines run **time daemon** Process that implements protocol
- One machine is elected (or designated) as the server (**master**) Others are **slaves**
- Master polls each machine periodically and ask each machine for time
- Can use Cristian's algorithm to compensate for network latency
- When results are received, master computes average of times - Including master's time
- **Hope**: Average cancels out individual clock's tendencies to run fast or slow (clock drift)
- Send offset by which each clock needs adjustment to each slave.
- Algorithm has provisions for ignoring readings from clocks whose skew is too great
- Compute a **fault-tolerant average**
- If master fails - any slave can take over





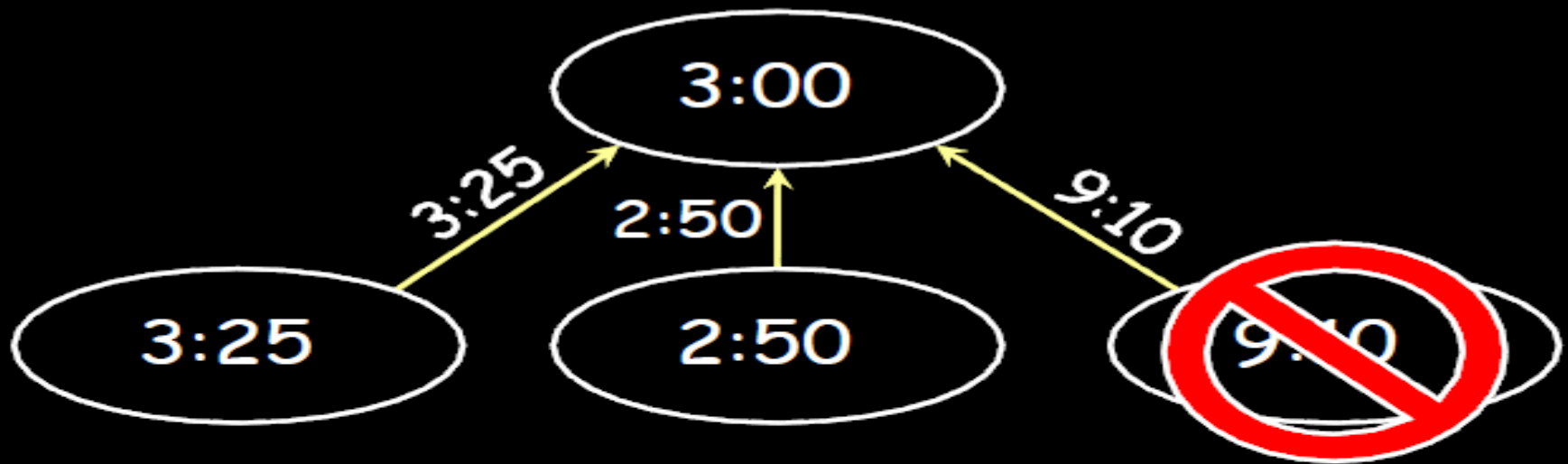
# Clock Synchronization

## 2. Berkeley Algorithm Example



# Clock Synchronization

## 2. Berkeley Algorithm Example

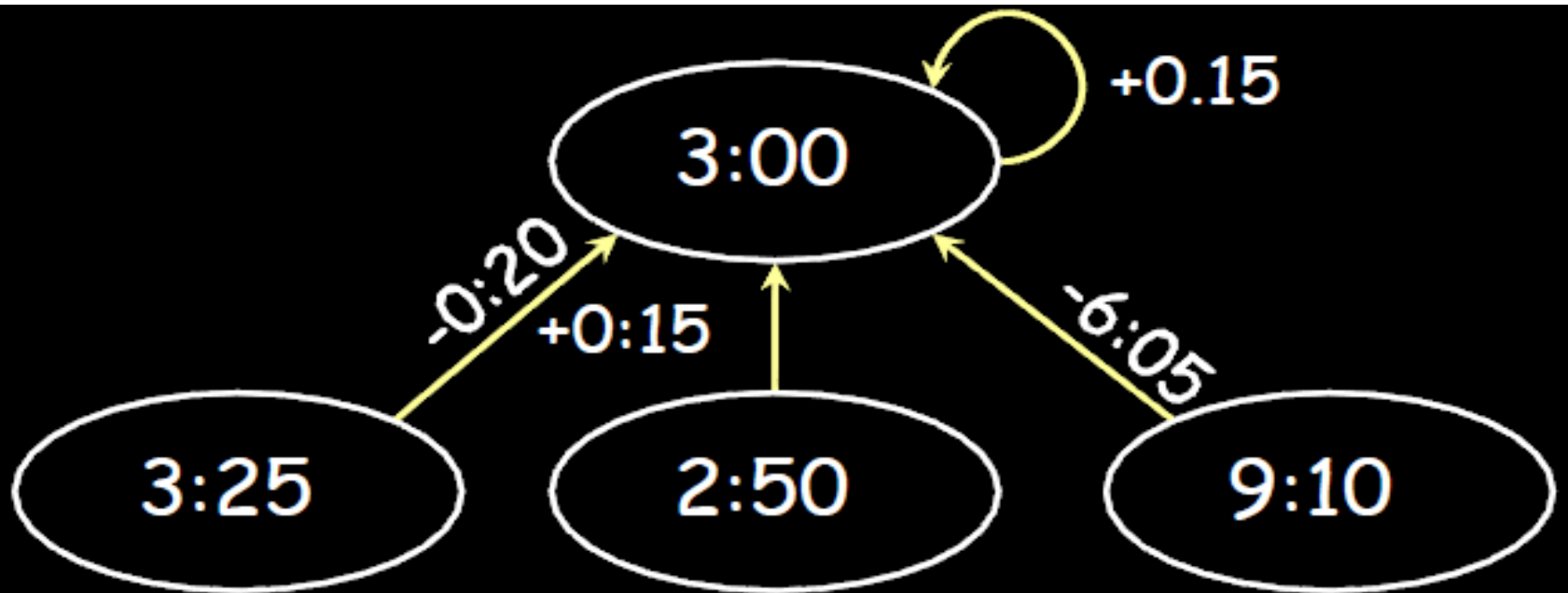


2. Compute fault-tolerant average:

$$\frac{3:25 + 2:50 + 3:00}{3} = 3:05$$

# Clock Synchronization

## 2. Berkeley Algorithm Example



3. Send offset to each client

# Clock Synchronization

## 3. Network Time Protocol

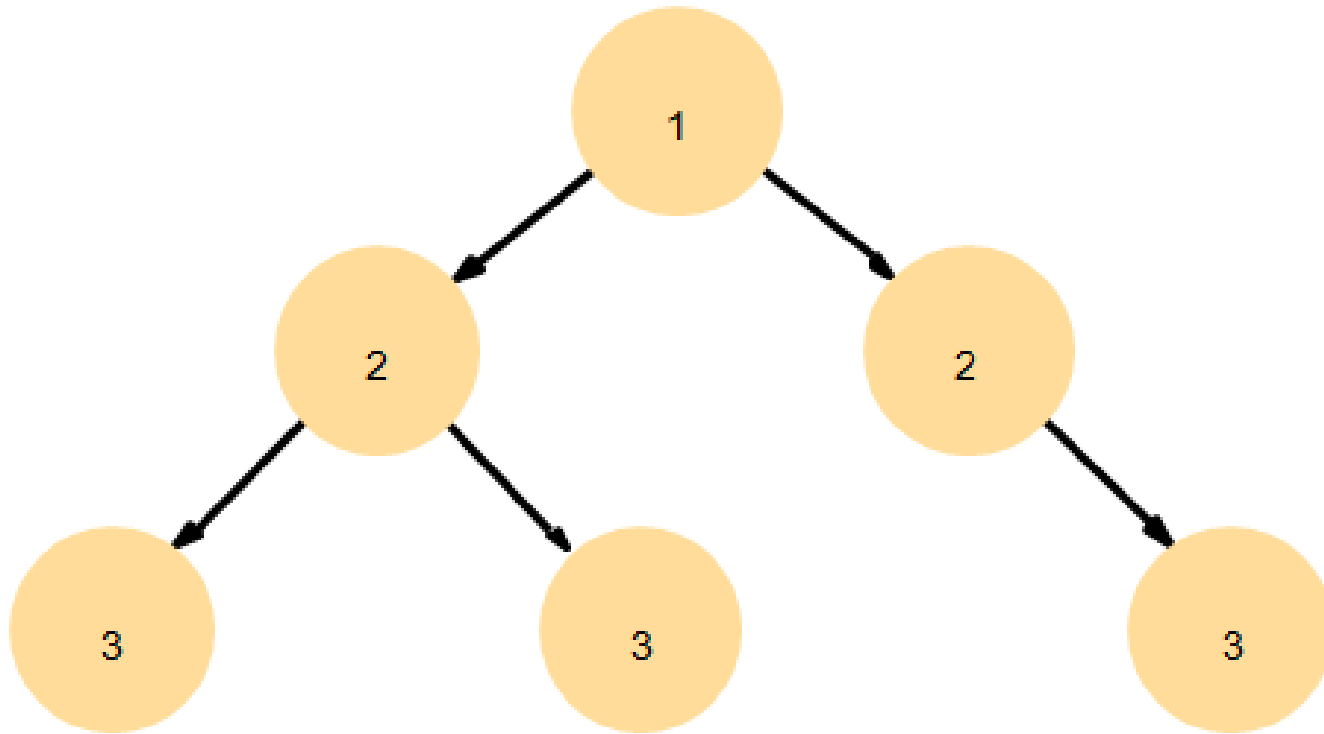
- Enable clients across Internet to be accurately synchronized to UTC despite message delays
- Primary servers are connected directly to a time source such as a radio clock receiving UTC; secondary servers are synchronized with primary servers.
- The servers are connected in a logical hierarchy called a **synchronization subnet** whose levels are called **strata**.
- Primary servers occupy stratum 1: they are at the root.
- Stratum 2 servers are secondary servers that are synchronized directly with the primary servers;
- Stratum 3 servers are synchronized with stratum 2 servers, and so on.



# Clock Synchronization

## 3. Network Time Protocol

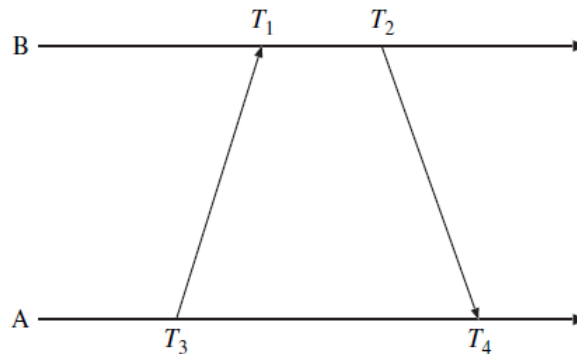
Arrows denote synchronization control, numbers denote strata.



# Clock Synchronization

## Network Time Protocol

Figure 3.9 Offset and delay estimation [15].



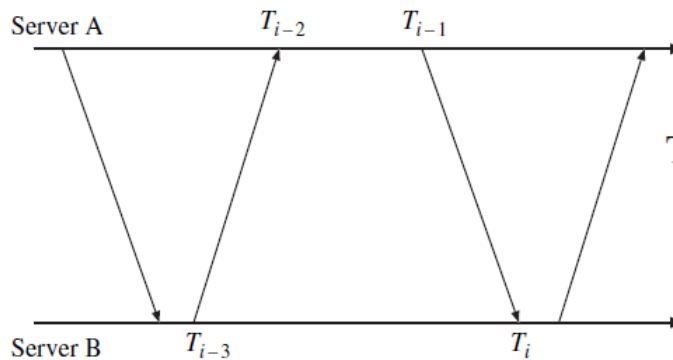
Let  $a = T_1 - T_3$  and  $b = T_2 - T_4$ .

$$\theta = \frac{a+b}{2}, \quad \delta = |a-b|$$

the offset  $O_i$  can be estimated as

$$O_i = (T_{i-2} - T_{i-3} + T_{i-1} - T_i)/2.$$

Figure 3.10 Timing diagram for the two servers [15].



The round-trip delay is estimated as

$$D_i = (T_i - T_{i-3}) - (T_{i-1} - T_{i-2}).$$

# Clock Synchronization

## 3. Network Time Protocol (Synchronization Modes)

- **Multicast mode**
  - Every node sends its time to all the other nodes in the LAN
  - For high speed LANs
  - Lower accuracy but efficient
- **Procedure call mode**
  - Similar to Cristian's algorithm
- **Symmetric mode**
  - Intended for master servers
  - Pair of servers exchange messages and retain data to improve synchronization over time

All messages delivered unreliably with UDP

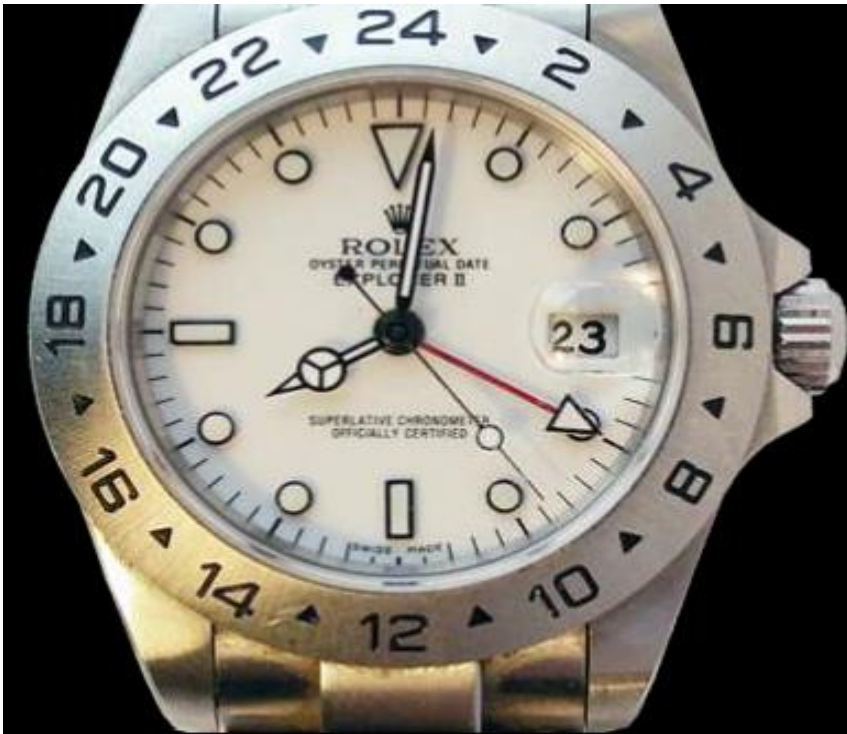


# Problems with Physical Clocks

- **Clock Drift** : Clocks tick at different rates. Create ever-widening gap in perceived time
- **Clock Skew** : Difference between two clocks at one point in time.
- For quartz crystal clocks, typical drift rate is about one second every  $10^6$  seconds = 11.6 days
- Best atomic clocks have drift rate of one second in  $10^{13}$  seconds = 300,000 years.
- **Quartz clock run at rate of 1.5 microseconds slower for every 35 days.**



# Problems with Physical Clocks



8:01:24

Skew = +84 seconds  
+84 seconds/35 days  
Drift = +2.4 sec/day

Oct 23, 2006  
8:00:00



8:01:48

Skew = +108 seconds  
+108 seconds/35 days  
Drift = +3.1 sec/day

# Logical Clocks



# Need for Logical Clock

- For many purposes, it is sufficient to know the order in which events occurred.
- Lamport (1978) — introduce logical (*virtual*) time, synchronize *logical clocks*.
- An event may be an instruction execution, may be a function execution, etc.
- Events include message send / receive

## Within a single process, or between two processes on the same computer

- The order in which two **events** occur **can** be determined using the **physical clock**

## Between two different computers in a distributed system

- The order in which two events occur **cannot** be determined **using local physical clocks**, since those clocks cannot be synchronized perfectly



# Happened Before Relation

- Lamport defined the happened before relation (denoted as “ $\rightarrow$ ”), which describes a **causal ordering of events**:
  1. if **a** and **b** are events in the same process, and **a** occurred before **b**, then **a**  $\rightarrow$  **b**
  2. if **a** is the event of sending a message **m** in one process, and **b** is the event of receiving that message **m** in another process, then **a**  $\rightarrow$  **b**
  3. if **a**  $\rightarrow$  **b**, and **b**  $\rightarrow$  **c**, then **a**  $\rightarrow$  **c** (i.e., the relation “ $\rightarrow$ ” is transitive)



# Causality

- Past events influence future events
- This influence among causally related events (those that can be ordered by “ $\rightarrow$ ”) is referred to **causally affects**.
- If  $a \rightarrow b$ , event **a** causally affects event **b**

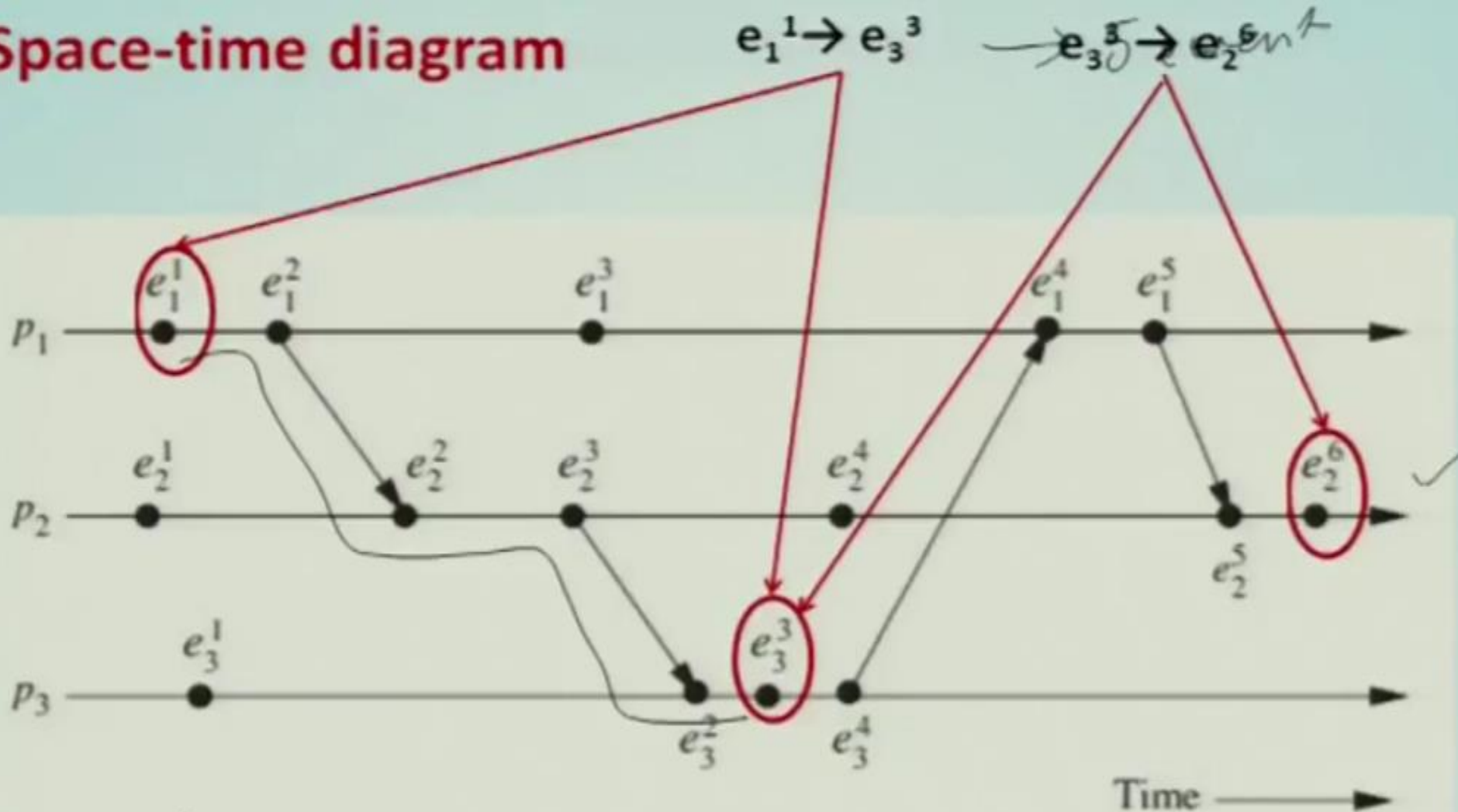
## Concurrent events

- Two distinct events **a** and **b** are said to be concurrent (denoted “ $a \parallel b$ ”), if neither  $a \rightarrow b$  nor  $b \rightarrow a$
- In other words, concurrent events do not causally affect each other
- For any two events  $a$  and  $b$  in a system, either:  $a \rightarrow b$  or  $b \rightarrow a$  or  $a \parallel b$



# Causal Events

Space-time diagram

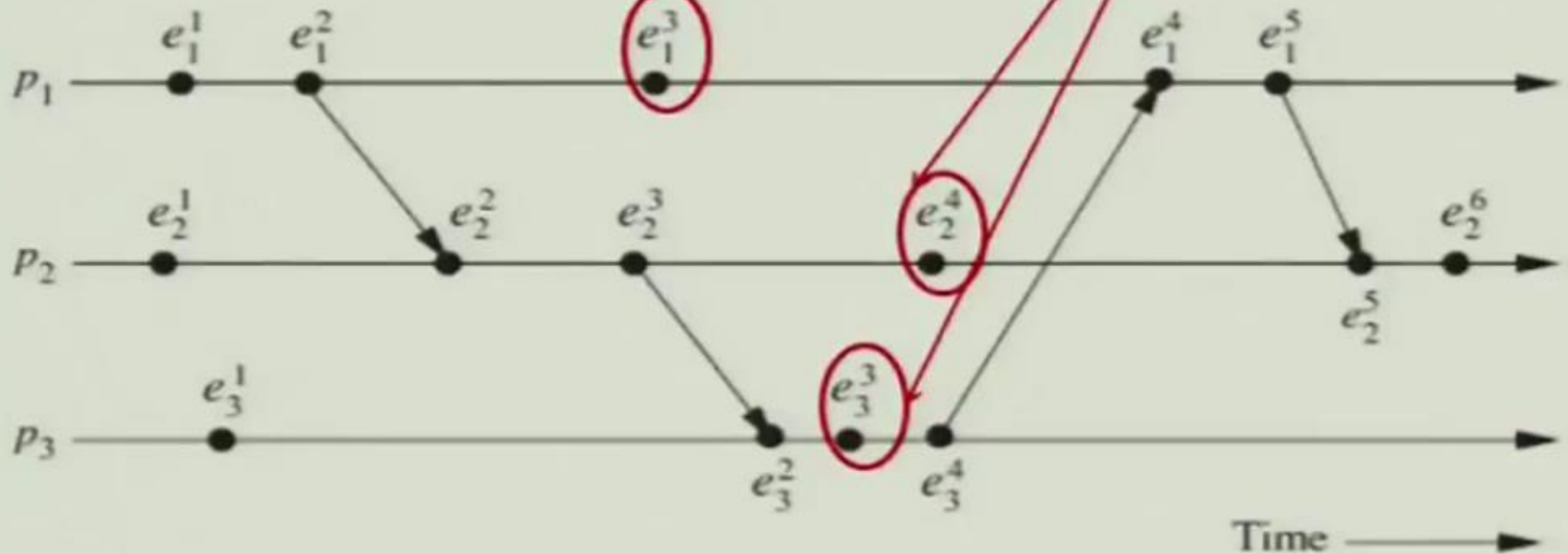




# Concurrent Events

## Space-time diagram

events in the set  $\{e_1^3, e_2^4, e_3^3\}$  are logically concurrent.



# Lamport's Logical Clock

- To implement “ $\rightarrow$ ” in a distributed system, Lamport (1978) introduced the concept of logical clocks, which captures “ $\rightarrow$ ” numerically
- Each process  $P_i$  has a logical clock  $C_i$
- Clock  $C_i$  can assign a value  $C_i(a)$  to any event **a** in process  $P_i$
- The value  $C_i(a)$  is called the timestamp of event **a** in process  $P_i$
- The value  $C(a)$  is called the timestamp of event **a** in whatever process it occurred.
- The timestamps have no relation to physical time, which leads to the term logical clock.
- The logical clocks assign monotonically increasing timestamps, and can be implemented by simple counters



# Lamport's Logical Clock

- **Clock condition:** if  $a \rightarrow b$ , then  $C(a) < C(b)$
- If event  $a$  happens before event  $b$ , then the clock value (timestamp) of  $a$  should be less than the clock value of  $b$
- Note that we **can not say:** if  $C(a) < C(b)$ , then  $a \rightarrow b$
- **Correctness conditions (must be satisfied by the logical clocks to meet the clock condition above):**
- [C1] For any two events  $a$  and  $b$  in the same process  $P_i$ ,  
if  $a$  happens before  $b$ , then  $C_i(a) < C_i(b)$
- [C2] If event  $a$  is the event of sending a message  $m$  in process  $P_i$ , and event  $b$  is the event of receiving that same message  $m$  in a different process  $P_k$ , then  $C_i(a) < C_k(b)$

# Lamport's Logical Clock

- **[IR1]** Clock  $C_i$  must be incremented between any two successive events in process  $P_i$

$$C_i := C_i + d, \quad (d > 0) \text{ (usually } d=1)$$

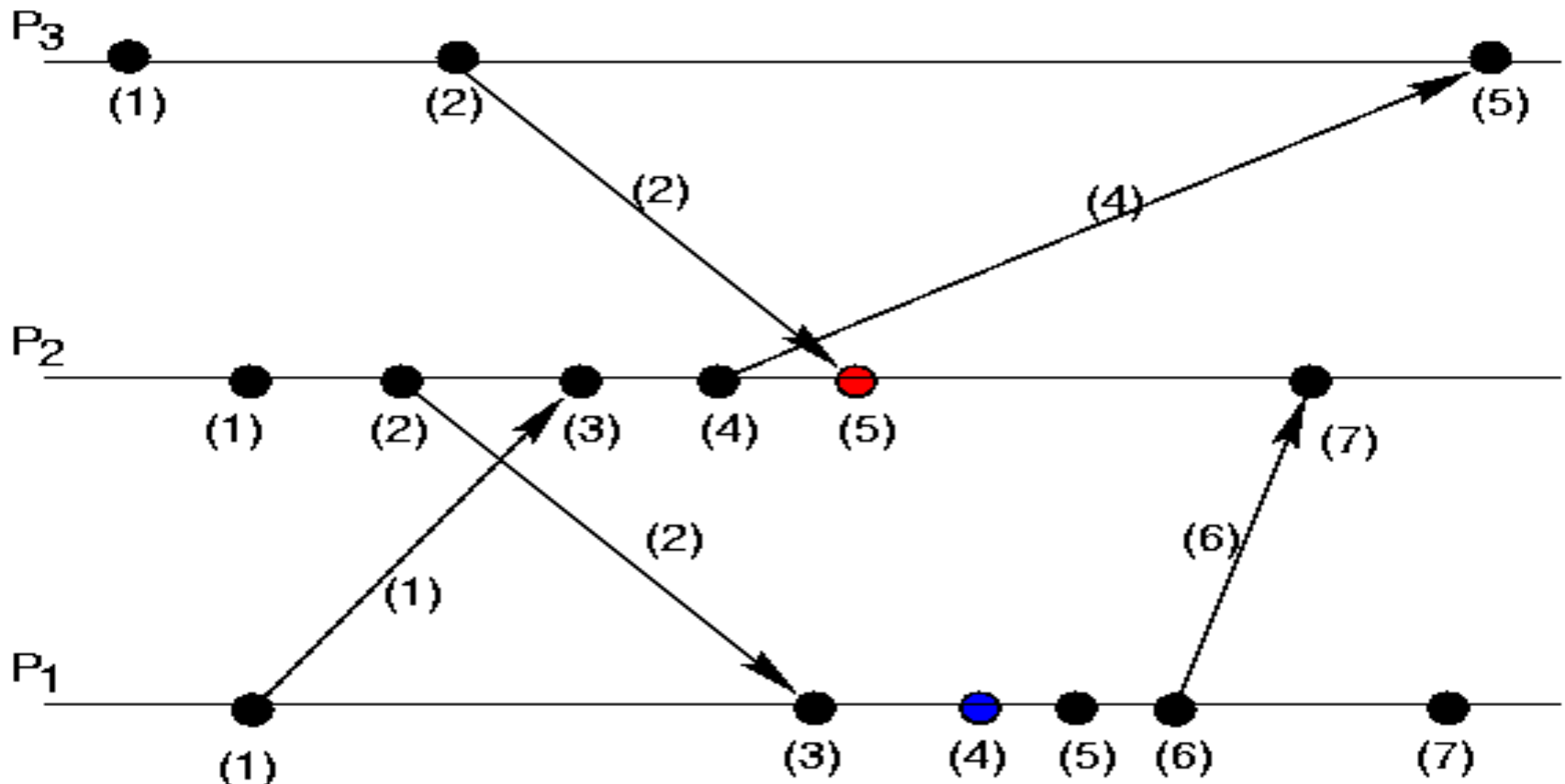
- **[IR2]** If event  $a$  is the event of sending a message  $m$  in process  $P_i$ , then message  $m$  is assigned a timestamp  $C_{msg} = C_i(a)$ . When that same message  $m$  is received by a different process  $P_k$ ,  $C_k$  is set to a value greater than or equal to its present value, and greater than  $C_{msg}$ .

$$C_k := \max(C_k, C_{msg} + d), \quad (d > 0) \text{ (usually } d=1)$$


# Lamport's Logical Clock

IR 1:  $C_i := C_i + d$  , ( $d > 0$ ), (usually  $d=1$ )

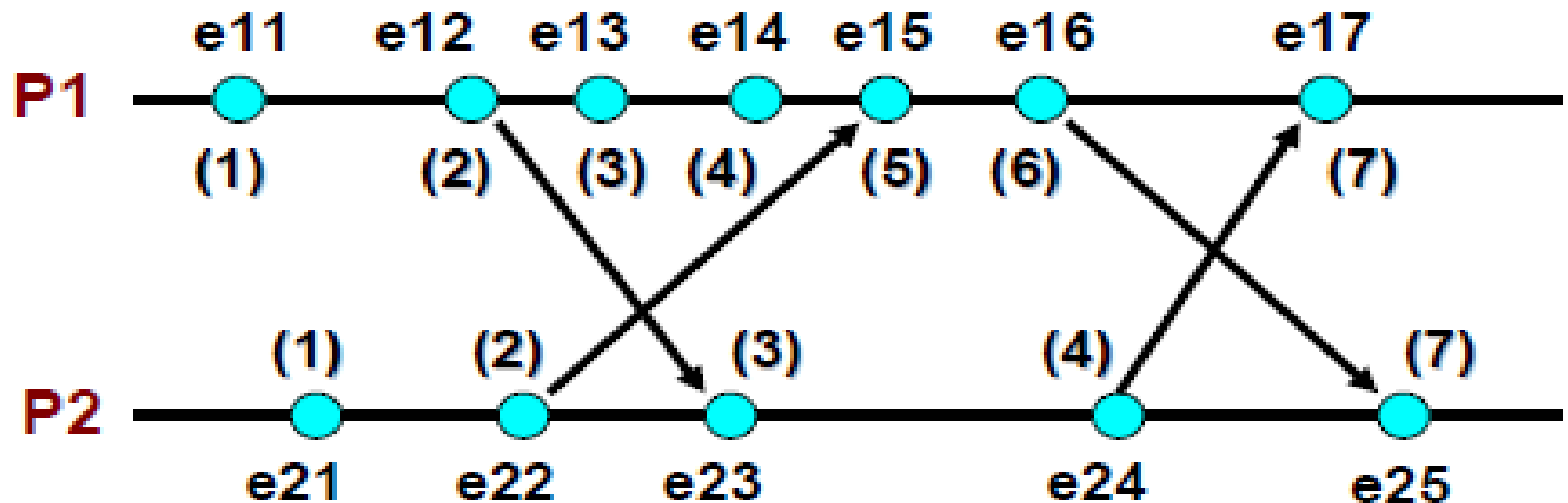
IR2:  $C_k := \max(C_k, C_{msg} + d)$  , ( $d > 0$ ) (Usually  $d=1$ )



# Lamport's Logical Clock

IR 1:  $C_i := C_i + d$ , ( $d > 0$ ), (usually  $d=1$ )

IR2:  $C_k := \max(C_k, C_{msg} + d)$ , ( $d > 0$ ) (Usually  $d=1$ )



# Lamport's Logical Clock

- A total order of events (“ $\Rightarrow$ ”) can be obtained as follows:
- If **a** is any event in process **P<sub>i</sub>**, and **b** is any event in process **P<sub>k</sub>**, then **a**  $\Rightarrow$  **b** if and only if either:
- $C_i(a) < C_k(b)$  or
- $C_i(a) = C_k(b)$  and  $P_i \ll P_k$  (if scalar timestamps of **a** and **b** are equal, ordering of events need to done through some mechanism).

where “ $\ll$ ” denotes a relation that totally orders the processes. Process identifier is used to break ties.

**Lower the process identifier in ranking higher the priority.**

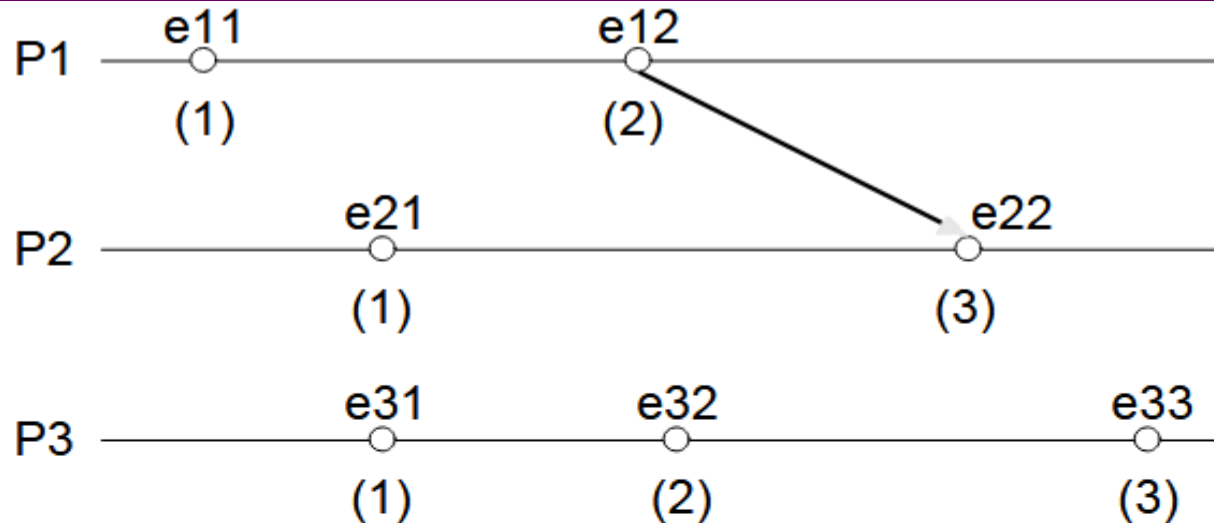


# Limitations of Lamport's Logical Clock

- With Lamport's logical clocks,
- if  $a \rightarrow b$ , then  $C(a) < C(b)$
- The following is **not necessarily true if events a and b occur in** different processes:
- if  $C(a) < C(b)$ , then  $a \rightarrow b$  **is NOT TRUE**, if **a** and **b** events occur in two different processes.
- Scalar time is not strongly consistent.



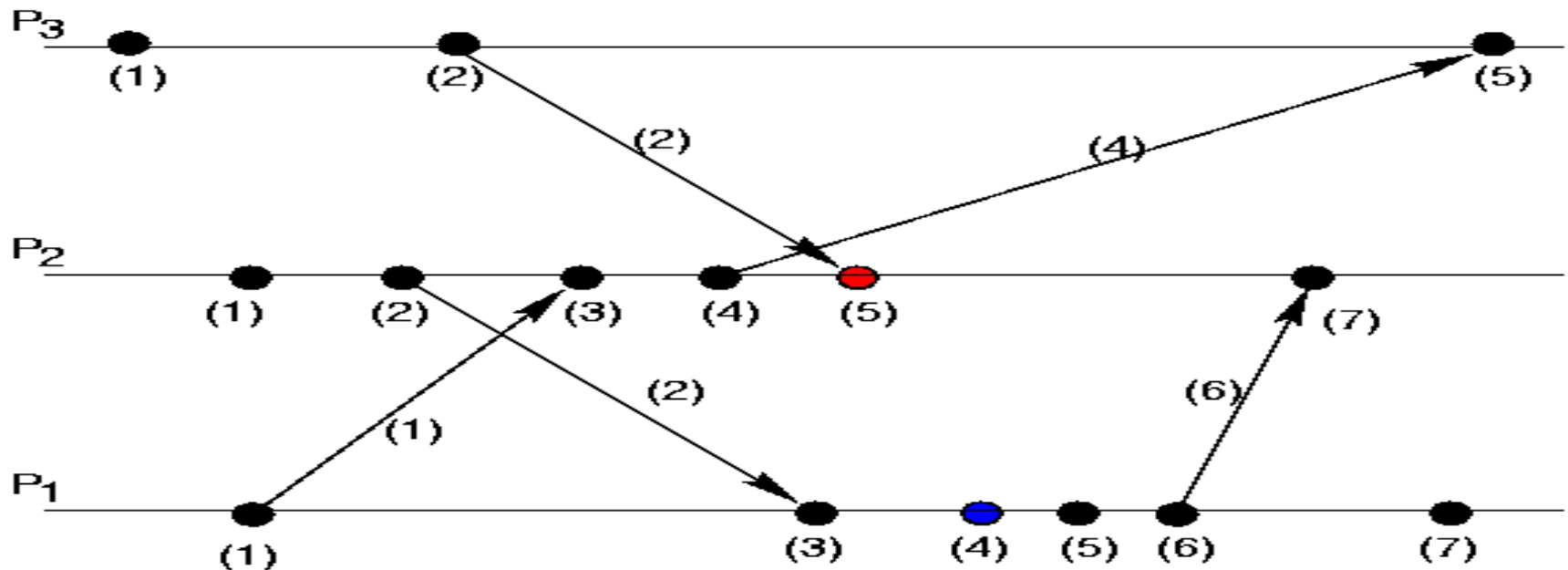
# Limitations of Lamport's Logical Clock



- With Lamport's logical clocks, if  $a \rightarrow b$ , then  $C(a) < C(b)$ 
  - ◆ The following is **not** necessarily true if events  $a$  and  $b$  occur in different processes: if  $C(a) < C(b)$ , then  $a \rightarrow b$
  - ◆  $C(e11) < C(e22)$ , and  $e11 \rightarrow e22$  is true
  - ◆  $C(e11) < C(e32)$ , but  $e11 \rightarrow e32$  is false
- Cannot determine whether two events are causally related from timestamps

# Limitations of Lamport's Logical Clock

There is **drastic increase in clock values** due to events that occur in different processes and that causally affects the other events. This is **not captured** in logical clocks.





# Vector Clocks



# Vector Clock

- Maintain a vector of values for every event that happens in all processes.
- Update happens for group of values in every event.



# Vector Clock

- [IR1] Clock  $C_i$  must be incremented between any two successive events in process  $P_i$  :

$$C_i[i] := C_i[i] + d, \quad (d > 0, \text{ usually } d=1)$$

- [IR2] If event  $a$  is the event of sending a message  $m$  in process  $P_i$ , then message  $m$  is assigned a vector timestamp  $tm = C_i(a)$ . When that same message  $m$  is received by a different process  $P_k$ ,  $C_k$  is updated as follows:

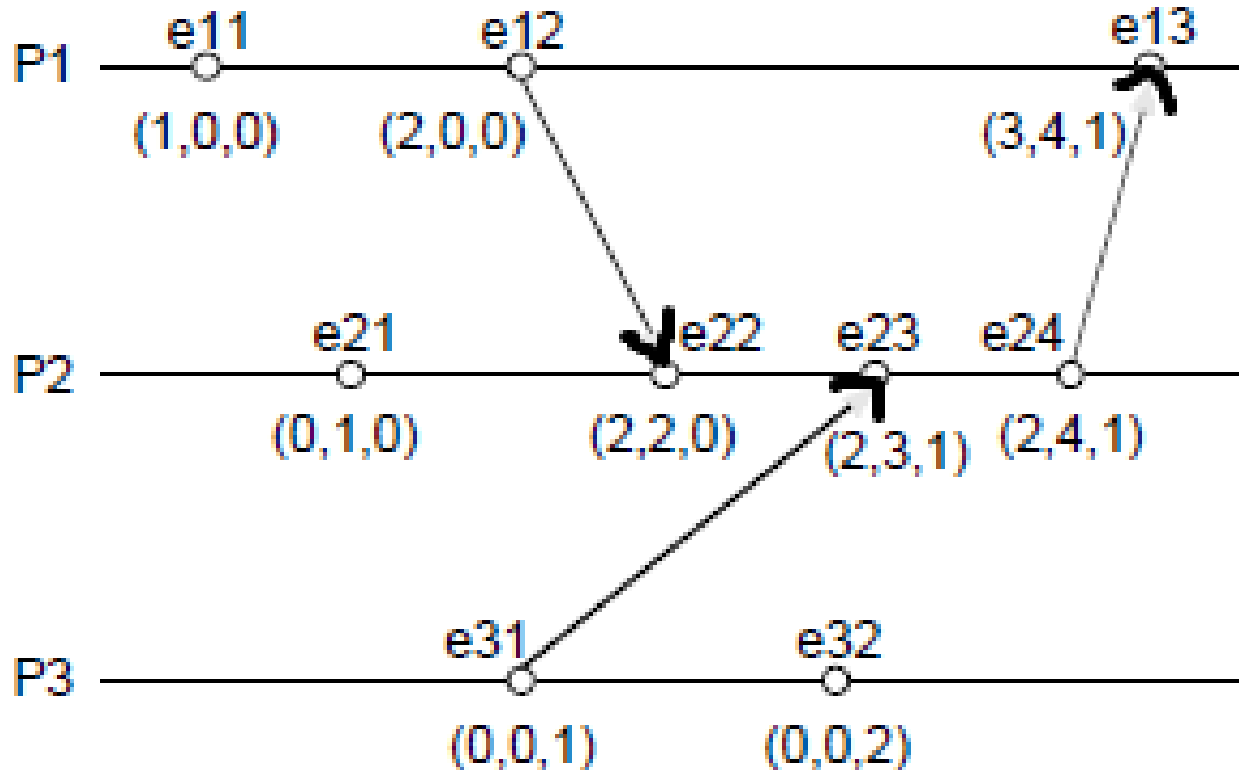
$$\forall p, C_k[p] := \max(C_k[p], tm[p] + d), \quad (\text{usually } d=0 \text{ unless needed to model network delay})$$

- It can be shown that  $\forall i, \forall k : C_i[i] \geq C_k[i]$
- Rules for comparing timestamps can also be established so that
- if  $ta < tb$ , then  $a \rightarrow b$  / Solves the problem with Lamport's clocks

# Vector Clock

IR1:  $C_i[i] := C_i[i] + d$ . ( $d=1$ )

IR2:  $\forall p, C_k[p] := \max(C_k[p], tm[p] + d)$ . ( $d=0$ )



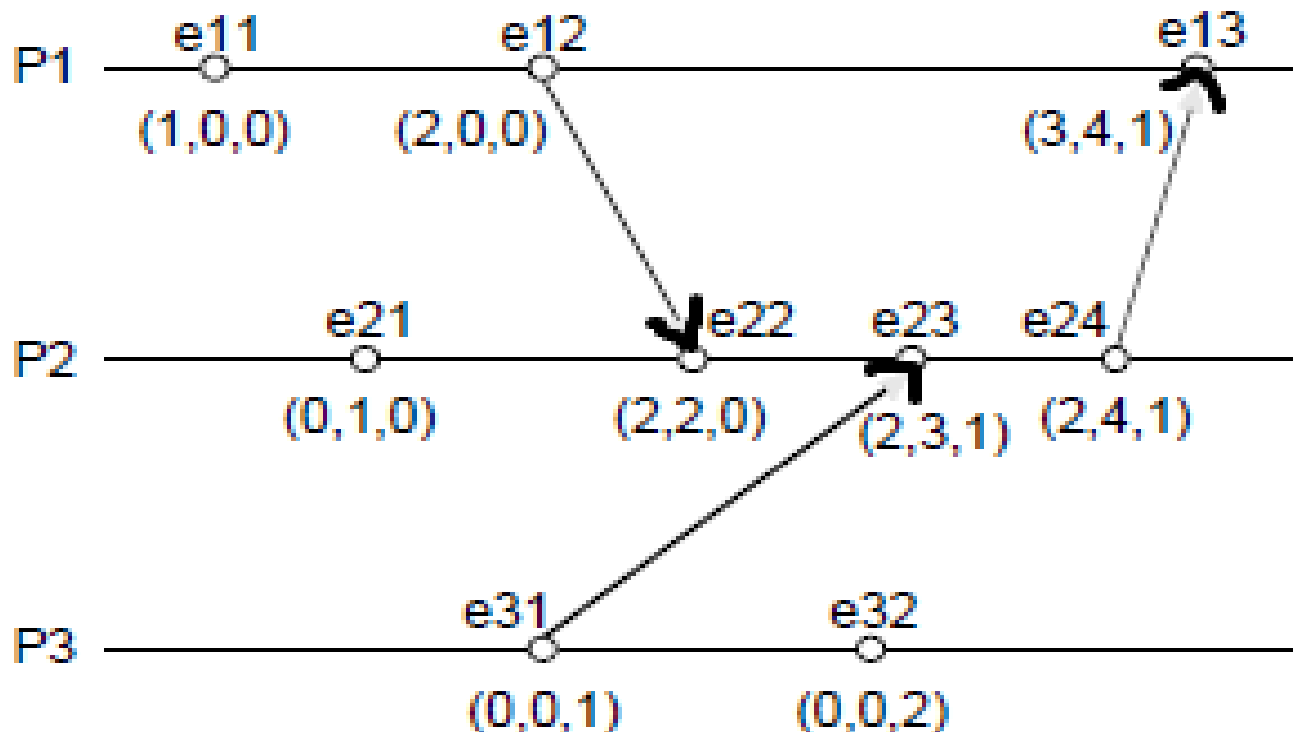
"enn" is  
event;  
"(n,n,n)" is  
clock value

# Vector Clock – Solving Logical Clock Problem

if  $t(e_{31}) < t(e_{23})$  then  $e_{31} \rightarrow e_{23}$  is true.

if  $t(e_{12}) < t(e_{24})$  then  $e_{12} \rightarrow e_{24}$  is also true.

Concurrent events  $e_{32} \parallel e_{24}$ ;  $e_{32} \not\rightarrow e_{24}$  and  $e_{24} \not\rightarrow e_{32}$



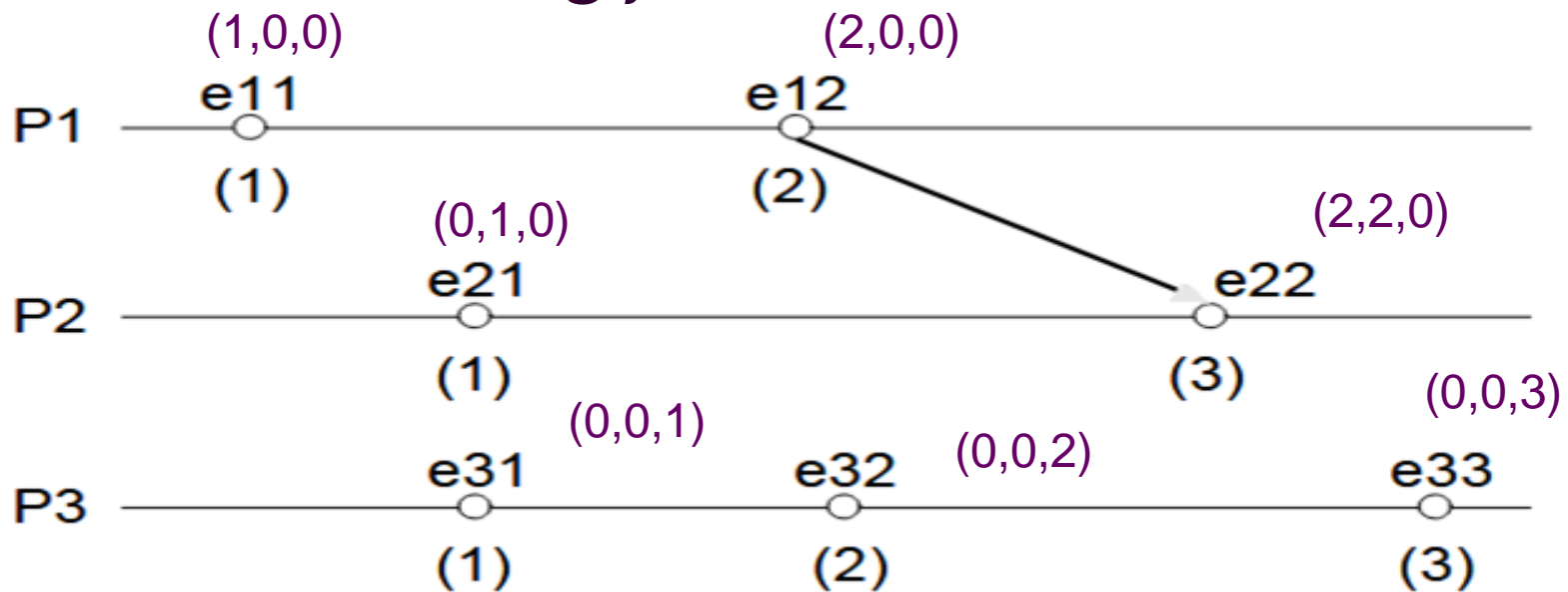
"enn" is  
event;  
"(n,n,n)" is  
clock value

# Vector Clock – Solving Logical Clock Problem

if  $C(e_{11}) < C(e_{22})$  then  $e_{11} \rightarrow e_{22}$  is true.

if  $C(e_{11}) < C(e_{32})$  then  $e_{11} \rightarrow e_{32}$  is also true.

Vector time is strongly consistent.



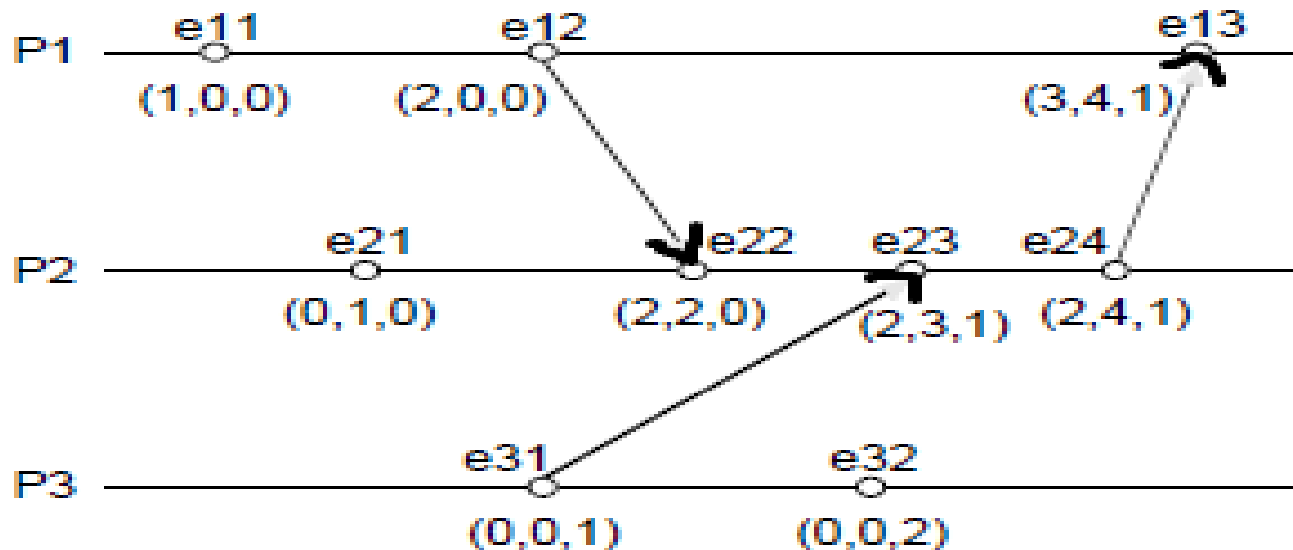
# Vector Clock – Concurrent Events

Concurrent events  $e_{32} \parallel e_{24}$ ;  $e_{32} \not\rightarrow e_{24}$  and  $e_{24} \not\rightarrow e_{32}$

$C(e_{32}) = (0,0,2)$  and  $C(e_{24}) = (2,4,1)$

$C(e_{32})$  is neither less nor greater than  $C(e_{24})$

$e_{11} \parallel e_{21}$ ,  $e_{11} \parallel e_{31}$ ,  $e_{11} \parallel e_{32}$ ,  $e_{21} \parallel e_{32}$ ,  $e_{12} \parallel e_{32}$ ,  
 $e_{22} \parallel e_{32}$ ,  $e_{32} \parallel e_{13}$ .



"enn" is  
 event;  
 "(n,n,n)" is  
 clock value

# Vector Clock

## Example of Vector Clock

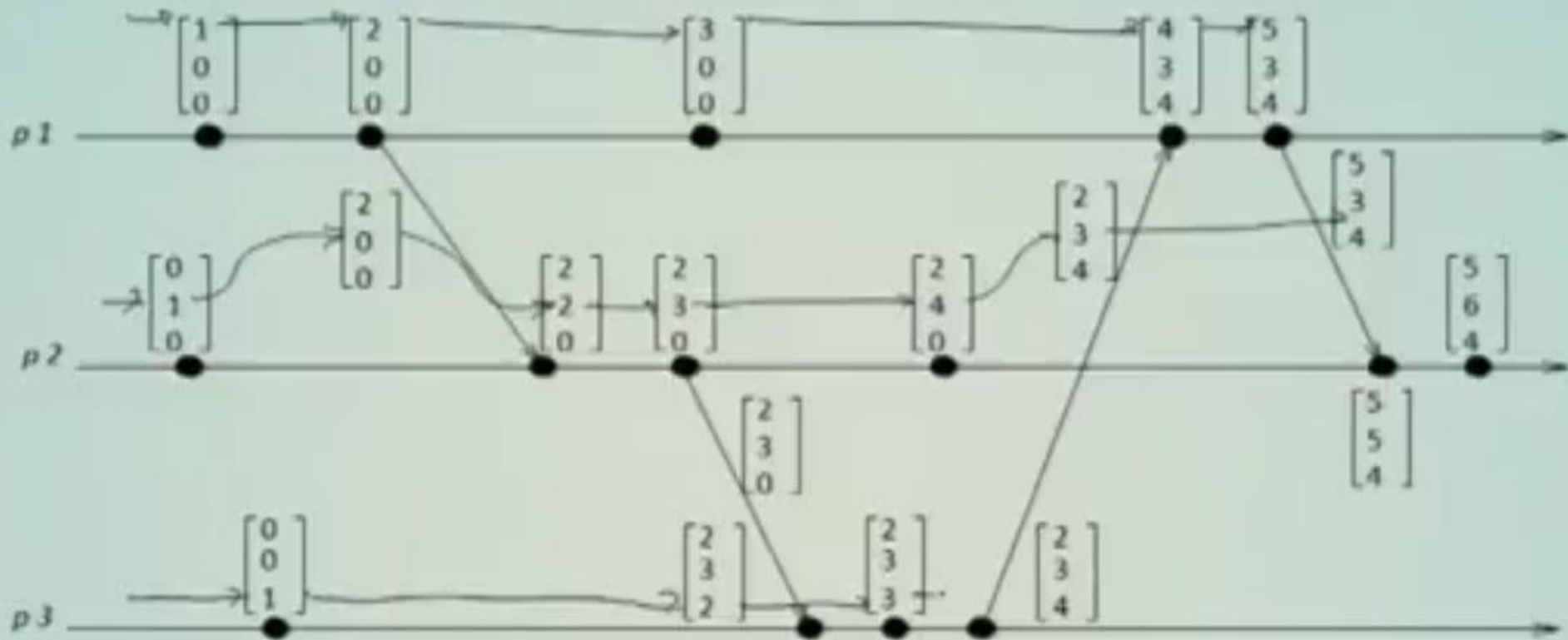


Figure 4.3: Evolution of vector time.



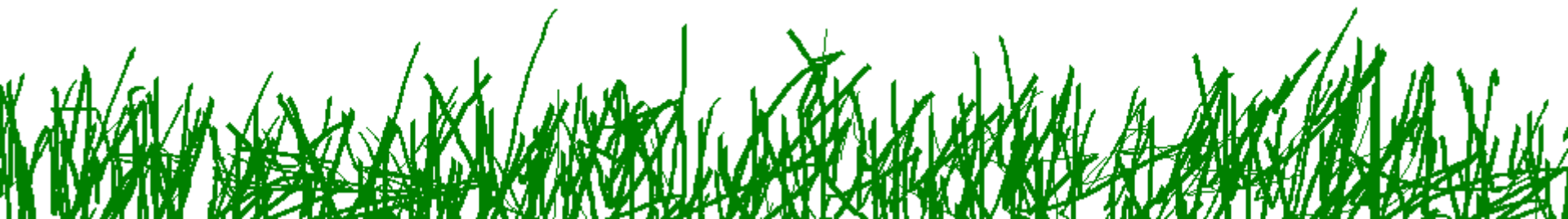
# Drawbacks of Vector Clock

- Need to **maintain memory** units for tracking all events in all processes as a vector.
- **The total numbers of processes may not be known in advance.** Number of processes may be created or terminated at any time.
- Unnecessary wastage of maximum allocation of memory units.



# Applications of Vector Clock

- **Distributed debugging.**
- Implementations of **causal ordering** communication and **causal distributed shared memory**.
- Establishment of **global breakpoints**.
- Determining the **consistency of checkpoints** in optimistic recovery.

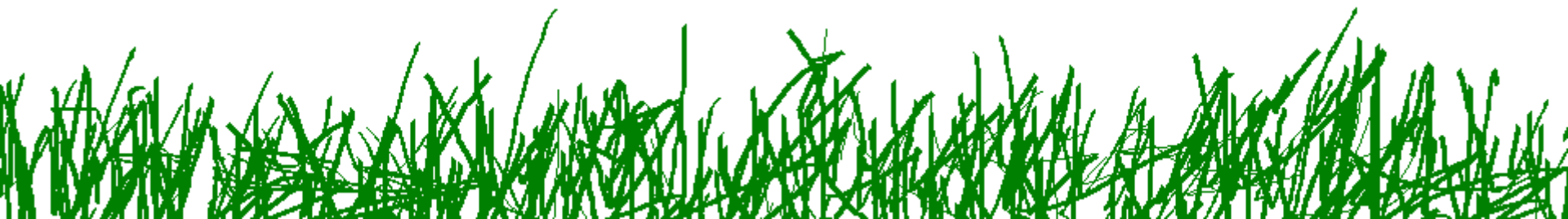


# Summary

- Physical Clocks
- Synchronizing Physical Clock (Algorithms)
- Problems with Physical Clock
- Lamport's Logical Clock
- Problems with Logical Clock
- Vector Clock
- Drawbacks of Vector Clocks
- Applications of Vector Clocks

# References

1. Kshemkalyani, Ajay D., and Mukesh Singhal. Distributed computing: principles, algorithms, and systems. Cambridge University Press, 2011
2. Mukesh Singhal & N.G. Shivaratri, Advanced Concepts in Operating Systems
3. George Coulouris, Jean Dollimore and Tim Kindberg, “Distributed Systems Concepts and Design”, Fifth Edition, Pearson Education, 2012



Thank You

