Operating Systems Synchronization

Shyan-Ming Yuan

CS Department, NCTU

smyuan@gmail.com

Office hours: Wed. 8~10 (EC442)

Distributed Synchronization

- A distributed system consists of a collection of distinct processes that are spatially separated and run concurrently.
  - The concurrently executing processes may share system resources either cooperatively or competitively.
  - To guarantee correct interactions among cooperative or competitive sharing, certain rules of behavior must be obeyed.
- The rules for enforcing correct interaction are implemented in the form of synchronization mechanisms.
  - Clock synchronization & Event ordering
  - Mutual exclusion
  - Deadlock detection (and handling)
  - Leader Election

Time and Clock in a DS

- Time is an important practical issue in distributed systems.
  - In a DSM, it requires an absolute global time to support strict consistency model.
    - All writes instantaneously become visible to all processes.
  - In a DFS, it also requires an absolute global time to support Unix semantics.
    - A read always returns the result from the latest write.
- Time is also problematic in distributed systems.
  - Each computer has its own physical clock.
    - The crystals in different computers may oscillate in different frequencies.
      - The difference between the readings of any two clocks is called clock skew.
      - The difference of oscillation frequency is called clock drift.
  - These clocks may deviate and cannot be synchronized perfectly.

Coordinated Universal Time

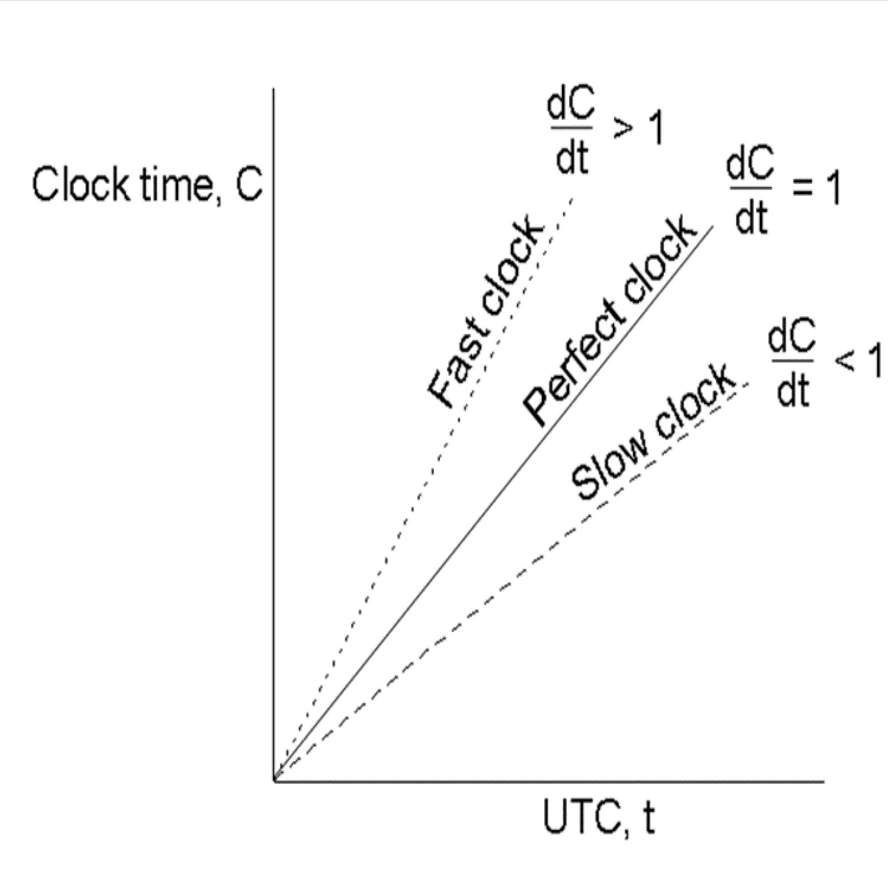
- The most accurate physical clocks use atomic oscillators.
  - The drift rate is about 10^{-13}.
  - In 1967, the standard second of International Atomic Time (IAT) has been defined as 9,192,631,770 periods of transition between the two hyperfine levels of the ground state of cesium 133 (Cs133).
    - The average of 50 Cs133 clocks in international labs world wide.
- Coordinated Universal Time (UTC) is an international standard for timekeeping that is based on the IAT.
  - A "leap second" is inserted or deleted occasionally to keep it in step with astronomical time.
  - It is broadcasted over shortwave radio station WWV.
    - The accuracy is 10 msec.
  - It is also broadcasted over Satellite for accuracy of 0.5 msec.

Physical Clock Synchronization

- There are two kinds of physical clock synchronization.
  - External synchronization: synchronized to a UTC source.
    - Assume that there is a bound $D > 0$ and a UTC source S, such that for all $i = 1, 2, \ldots, N$ and for all real times t in time interval T, $|S(t) - C_i(t)| < D$.
    - Then the clocks C_i are accurate to within the bound D.
  - Internal synchronization: synchronized within the DS.
    - Assume that there is a bound $D > 0$, such that for all $i, j = 1, 2, \ldots, N$ and for all real times t in time interval T, $|C_i(t) - C_j(t)| < D$.
    - Then the clocks C_i agree within the bound D.
- Externally synchronized clocks are also internally synchronized.
- However, internally synchronized clocks are not necessarily externally synchronized.
  - They may drift collectively from an external UTC source.

Timers and Clocks

- A computer timer usually goes off multiple times per sec.
  - It increases the count of ticks for each timer interrupt.
  - The value of clock on a host p is $C_p(t)$.
  - For a perfect clock w.r.t. UTC, $C_p(t) = t$ and $dC/dt = 1$.
- If the drift rates of all clocks w.r.t UTC are at most ρ .
  - To guarantee all clocks never differ by more than D, the clocks must re-synchronize every $D/(2\rho)$ seconds.

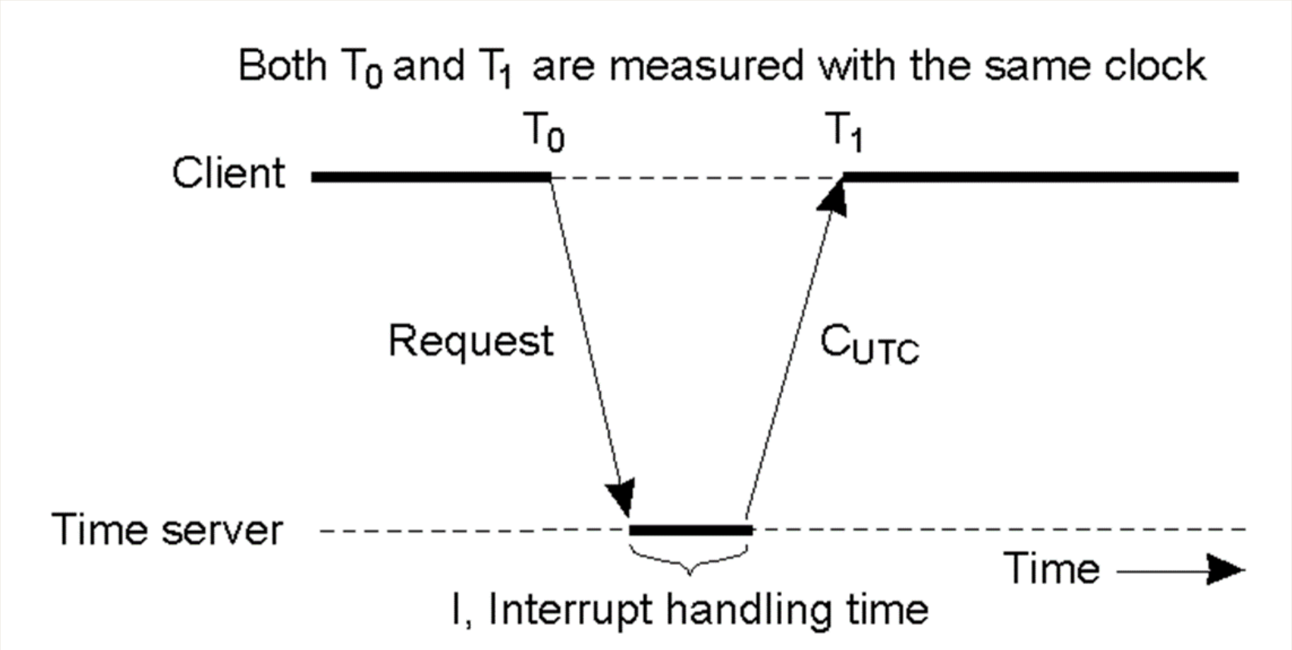


Clock Synchronization Algorithms

- Centralized Algorithms
  - The Cristian's Algorithm (1989)
  - The Berkeley Algorithm (1989)
- Decentralized Algorithms
  - Averaging Algorithms (e.g. NTP)

The Cristian's Algorithm (1)

- Assume one host (the time server S) has a UTC receiver.
  - All other hosts try to stay synchronized with the S.
- Every $D/(2\rho)$ seconds, each host sends a synch request to the time server S asking for the current time.
  - The S replies its current UTC time C_{UTC} to the requester.

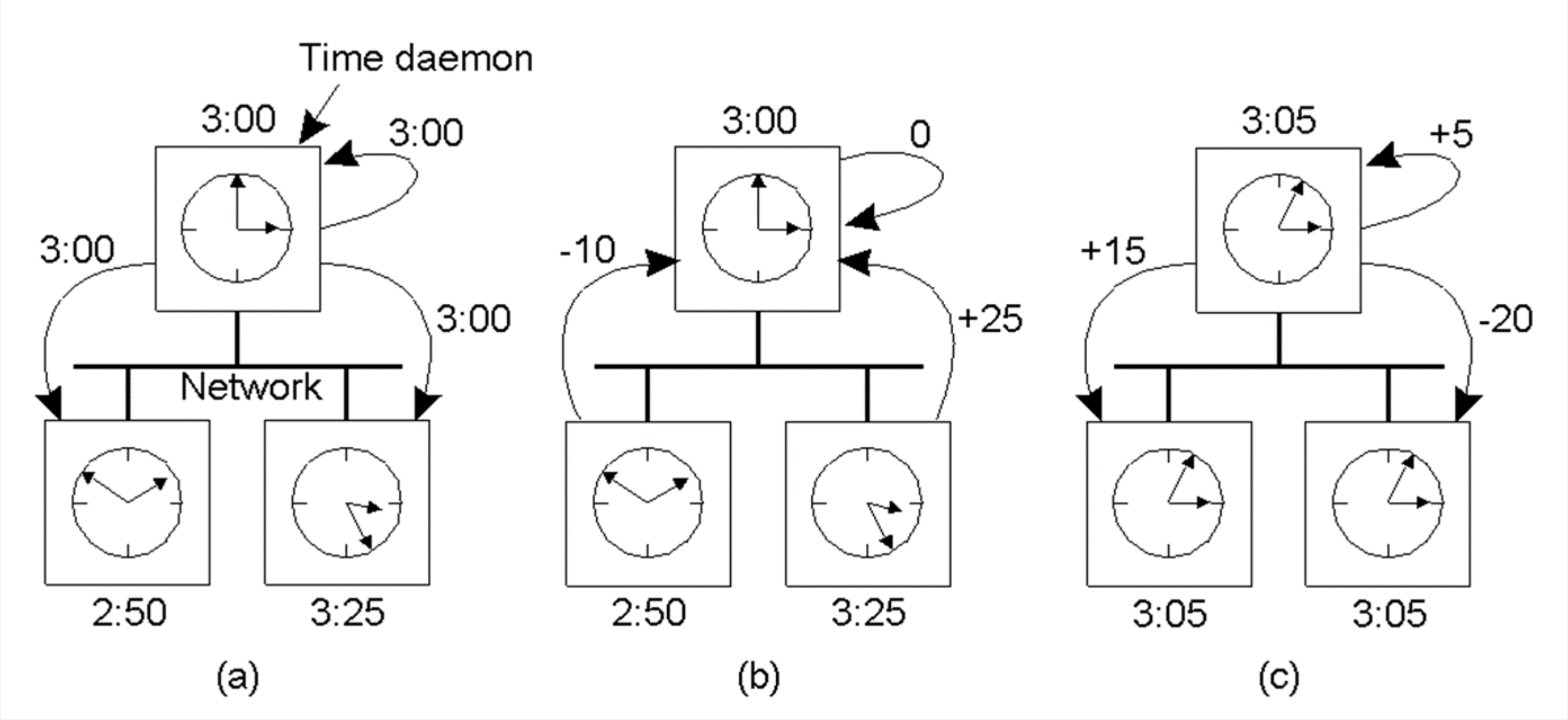


Cristian's Algorithm (2)

- In general, the client's clock may be adjusted to $C_{UTC}+(t_1-t_0-\mathbf{I})/2$
  - (t_1-t_0) is the round trip delay.
  - **I** is the server interrupt handling time.
    - It can be piggybacked in the C_{UTC} response.
- However, if the client's clock is faster than UTC then the $C_{UTC}+(t_1-t_0-I)/2$ may be less than t_1.
  - The client's clock will be turn back.
    - This is not acceptable for some applications.
  - The reasonable solution is to slow down client's clock by adding less value per tick.

The Berkeley Algorithm

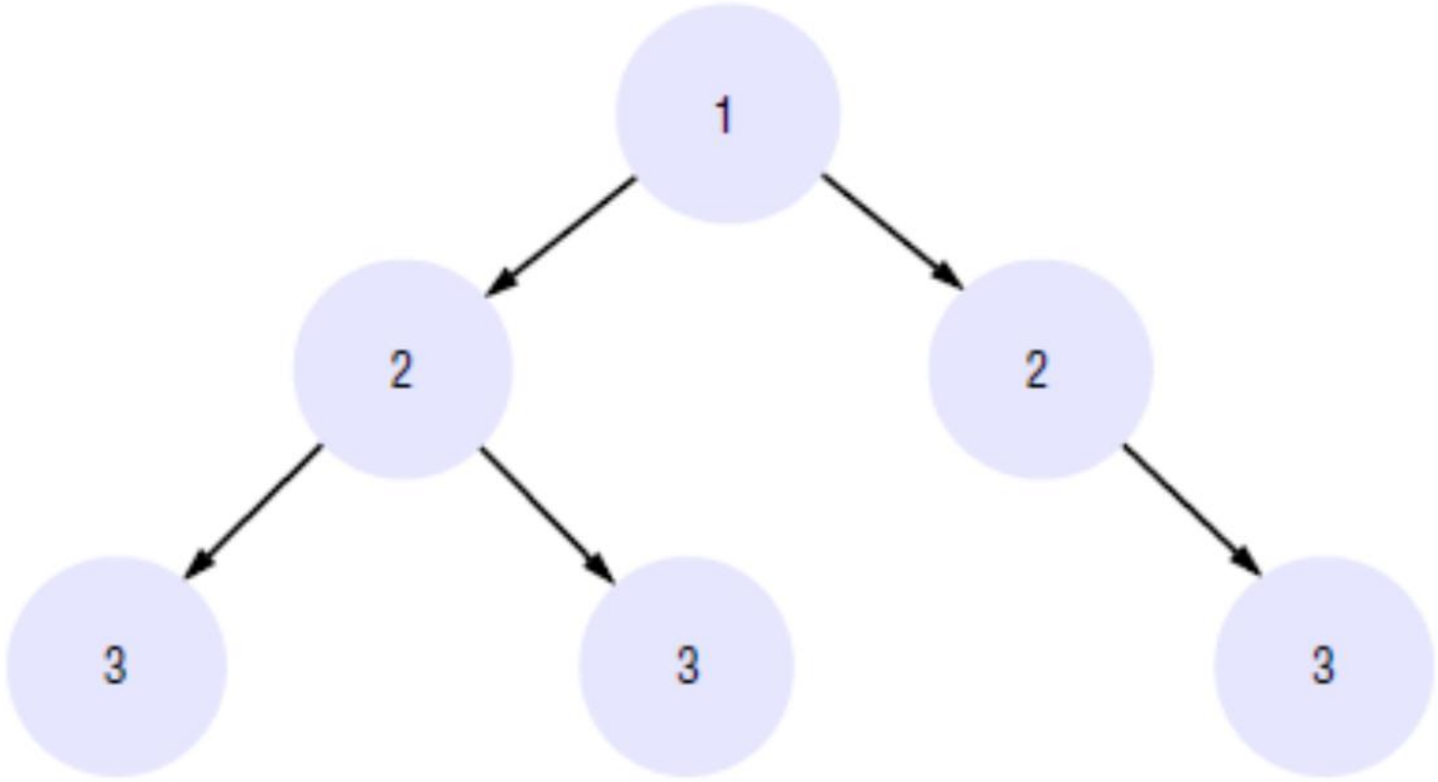
- A host is chosen to act as the time daemon which asks all the other hosts for their clock skews periodically.
  - The slave hosts reply their clock skews.
  - The time daemon computes the average clock skew.
- The time daemon tells every host how to adjust their clocks.



The Network Time Protocol

- Cristian's method and the Berkeley algorithm are intended for use within intranets.
- The Network Time Protocol (NTP) service is provided by a network of servers located across the Internet.
- The NTP servers are connected in a logical hierarchy.
  - The levels are called strata.
  - Primary servers are connected directly to a UTC time source.
    - They occupy stratum 1 and act as the root.
  - Secondary servers are synchronized with primary servers.
    - They occupy stratum 2.
  - Stratum 3 servers are synchronized with stratum 2 servers, and so on.
  - Errors are introduced at each level of synchronization.

An Example of an NTP implementation



Arrows denote synchronization control, numbers denote strata.

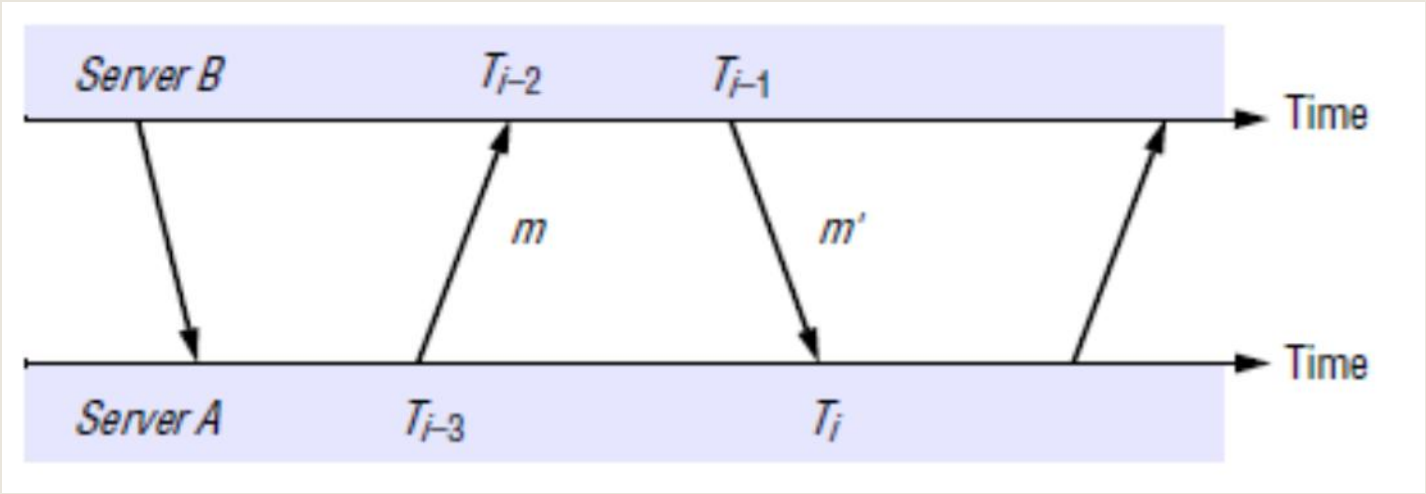
The NTP Synchronization (1)

- NTP servers are synchronized in one of three modes:
  - multicast, procedure-call and symmetric mode.
- Multicast mode is intended for use on a high-speed LAN.
  - One or more servers periodically multicasts their clock values to other hosts connected by the LAN.
  - Upon receiving a clock value, a host sets its clock by assuming a small delay.
  - This mode can achieve only relatively low accuracies, but it is considered sufficient for many purposes.
- Procedure-call mode is similar to the Cristian's algorithm.
  - In this mode, one server accepts requests from other computers.
  - Upon receiving a request, it replies its current clock reading.
  - It can achieved higher accuracy that multicast can not support.

The NTP Synchronization (2)

- Symmetric mode is intended for servers in the lower strata.
  - It can achieve the highest accuracy.
  - A pair of servers operating in symmetric mode exchange messages bearing timing information.
- In NTP, messages are delivered unreliably (e.g. UDP).
- In procedure-call and symmetric mode, processes exchange pairs of messages.
  - For each pair of messages sent between two servers the NTP calculates an offset o_{ij} which is an estimate of the actual offset between the two clocks and a delay d_{ij} which is the total transmission time for the two messages.

Clock Offset Estimation in NTP



If the clock skew of the clock at B relative to the clock at A is S, and the actual transmission times for m and m' are t(m) and t(m'), respectively.

Then $T_{i-2}=T_{i-3}+t(m)+S$, $T_i=T_{i-1}+t(m')-S$, $d_{AB}=t(m)+t(m')=T_{i-2}-T_{i-3}+T_i-T_{i-1}$

Let S_{AB} be $(T_{i-2}-T_{i-3}+T_{i-1}-T_i)/2$ then $S=S_{AB}+(t(m')-t(m))/2$

Since t(m) and t(m') ≥ 0, it can be shown that $S_{AB}-d_{AB}/2 \leqq S \leqq S_{AB}+d_{AB}/2$.
If S_{AB} is used as an estimated skew then d_{AB} is the accuracy of this estimation.

Event Ordering

- In a single computer, events can be ordered uniquely according to its local physical clock.
- However, it is very hard to perfectly synchronize clocks in a DS.
  - Therefore, it is extremely difficult to use physical clocks to determine the order of any arbitrary pair of events in a DS.
- Fortunately, there are simple and intuitively obvious facts that can be applied in a DS to order events occurred at different hosts.
  - FACT-1: If two events occurred at the same process p_i, then they occurred in the order in which p_i observes them.
  - FACT-2: Whenever a message is sent between processes, the sending message event occurred before the receiving message event.

Happened Before Relation

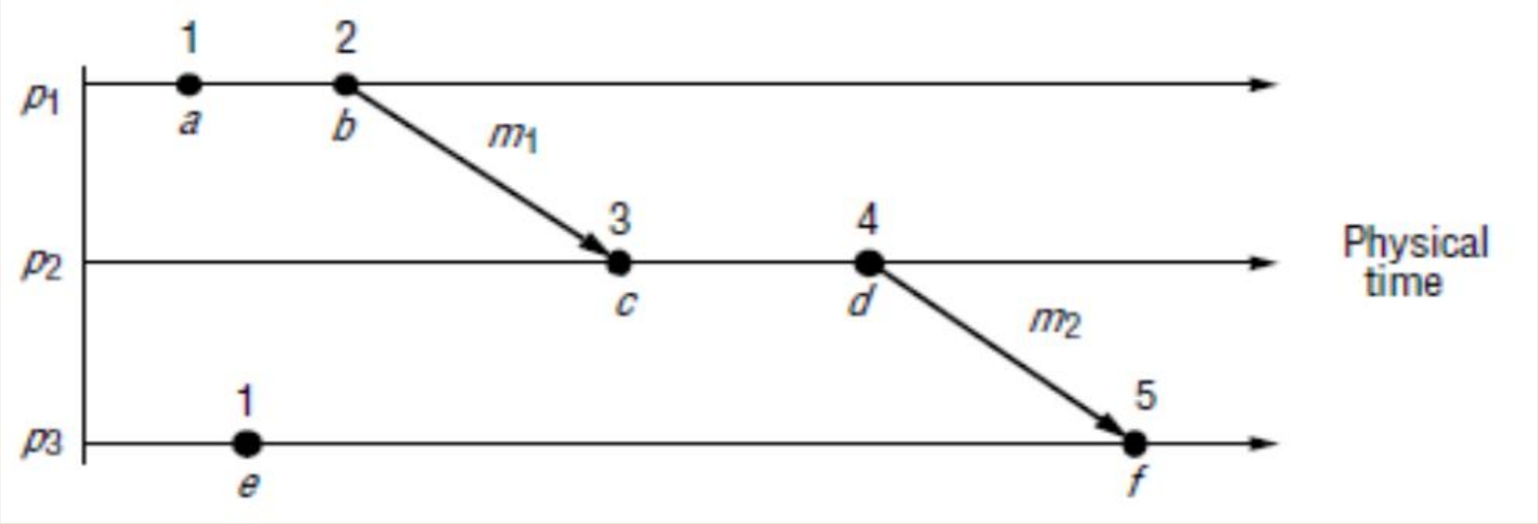
- Based on these two facts, Lamport had defined the happened before (HB) relation.
- The HB relation, denoted by $\rightarrow$, are defined as follows:
  - HB1: For any 2 events e and e' in the same process, if e' occurs after e, then $e \rightarrow e'$.
  - HB2: For any message m, send(m) $\rightarrow$ receive(m), where the send(m) is the event of sending m and the receive(m) is event of receiving m.
  - HB3: For any three events e, e', and e'',
    - if $e \rightarrow e'$ and $e' \rightarrow e''$, then $e \rightarrow e''$.
- Thus, for any 2 events e and e', if $e \rightarrow e'$, then there exists a series of events $e_1, e_2, \ldots, e_n$, where $e = e_1$ and $e' = e_n$, such that
  - for $i = 1, 2, \ldots, n-1$, either HB1 or HB2 holds between e_i and e_{i+1}.

Lamport's Logical Clock (1)

- Lamport invented a simple numerical mechanism, called logical clock, to capture the happened before relation.
  - A logical clock is a monotonically increasing counter.
- Each process p_i keeps its own logical clock, L_i, which is used for time stamping events in p_i.
  - The $L_i(e)$ is referred to the timestamp of event e at p_i and the $L(e)$ is the timestamp of event e.
- To capture the HB relation, processes update their logical clocks and exchange their logical clocks in messages as follows:
  - LC1: Before issuing an event, p_i updates $L_i \leftarrow L_i + 1$.
  - LC2a: On sending a message m, p_i uses L_i as timestamp for send(m).
  - LC2b: On receiving a message (m, t_m) by p_j, p_j sets $L_j \leftarrow \max(L_j, t_m)+1$ and uses L_i as timestamp for the receive(m).

Lamport's Logical Clock (2)

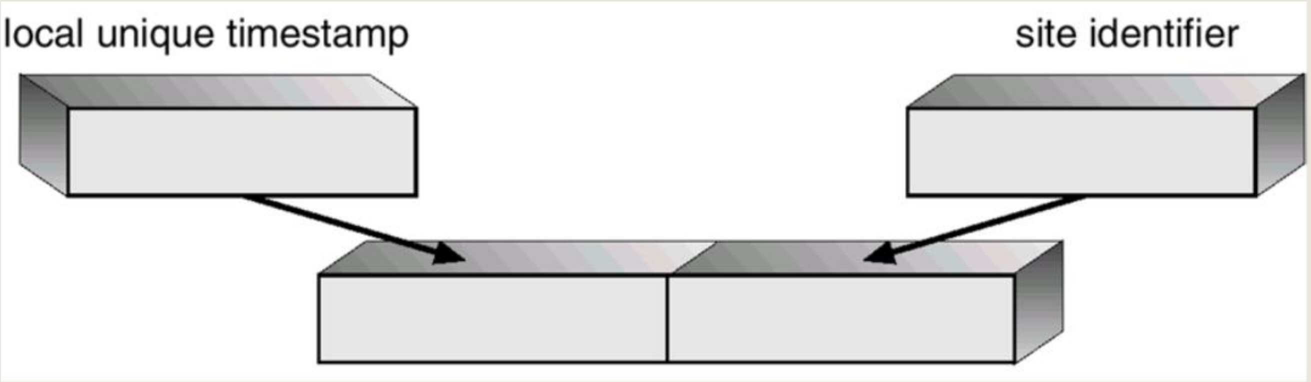
- It can be shown that any two events e and e',
  - if $e \rightarrow e'$ then $L(e) < L(e')$.
  - However, the converse is not true.
- For example, $L(b) > L(e)$ but neither $(b \rightarrow e)$ nor $(b \leftarrow e)$.
  - If neither $(b \rightarrow e)$ nor $(b \leftarrow e)$, then b and e are concurrent and denoted as $(b \parallel e)$.



Totally Ordered Logical Clocks

- HB is only a partial ordering relation: $L_i(e)$ may equal to $L_j(f)$.
- It is possible to create a total order on all events by taking into account the IDs of the processes at which events occur.
  - All events are ordered by their logical clocks with process IDs.
- Global logical clocks (GLC):
  - If e is an event of p_i with $L_i(e)$, then the GLC of e is $(L_i(e), i)$.
  - $(L_i(e), i) < (L_j(f), j)$ if and only if

    either $(L_i(e) < L_j(f))$ or $[(L_i(e) == L_j(f))$ and $(i < j)]$.



The Vector Clock (1)

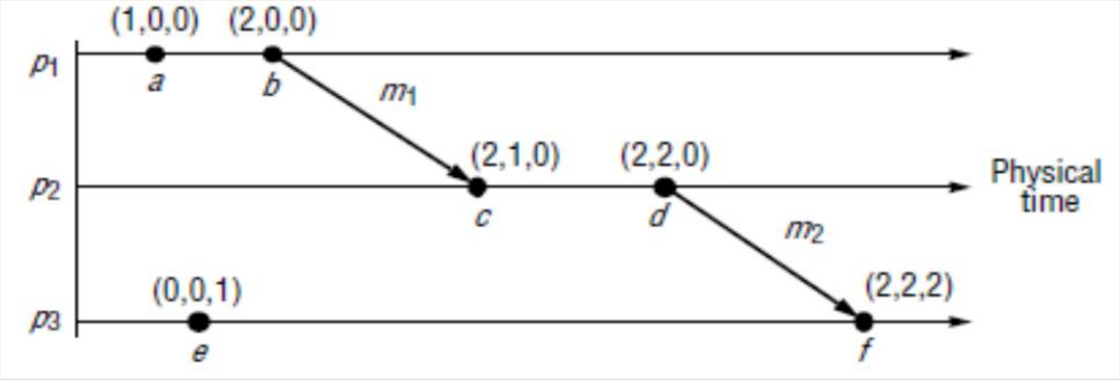
- The global logical clock proposed by Lamport can be used to support consistent ordering in many to many group communication.
- However in GLC, if $GLC(e) < GLC(e')$ does not imply $e \rightarrow e'$.
- Mattern and Fidge proposed vector clock (VC) scheme.
  - If $e \rightarrow e'$ then $VC(e) < VC(e')$.
  - If $VC(e) < VC(e')$ then $e \rightarrow e'$.
  - If $e \| e'$ then neither $VC(e) < VC(e')$ nor $VC(e) > VC(e')$.
  - If not $(VC(e) < VC(e'))$ and not$(VC(e) > VC(e'))$ then $e \| e'$.
- The VC scheme can be used to support Causal ordering in many to many group communication.

The Vector Clock (2)

- A vector clock for a system of N hosts is an array of N integers.
  - Each process p_i keeps its own vector clock, V_i, for timestamps.
- There are following rules for updating the vector clocks:
  - VC1: Initially, $V_i[j] = 0$, for all i, j = 1, 2, …, N .
  - VC2: before timestamping an event, p_i sets $V_i[i] \leftarrow V_i[i] + 1$.
  - VC3: on sending a message m, p_i uses V_i as the timestamp of m.
  - VC4: on receiving a message m (sent from p_j) with a timestamp T,
    - p_i sets $V_i[j] \leftarrow \max(V_i[j], T[j])$, for j = 1, 2, …, N.
- For each vector clock V_i,
  - $V_i[i]$ is the number of events that p_i has time stamped.
  - for all $j \neq i$, j = 1, 2, …, N
    - $V_i[j]$ is the number of events that have occurred at p_j and have affected p_i .

The Vector Clock (2)

- Vector Clocks have the following properties:
  - V == V', if and only if V[j] == V'[j] for all j=1,2…,N
  - V ≦ V', if and only if V[j]≦V'[j] for all j=1,2…,N
  - V < V', if and only if V≦V' and V≠V'
  - It can be shown that two events e and e',
    - if e → e' then V(e) < V(e').
    - if V(e) < V(e') then e →e'.
    - If e and e' are concurrent then V(e) and V(e') are not in any order.
    - Events e and b are concurrent because neither V(e) < V(b) nor V(b) < V(e) is true.



Distributed Mutual Exclusion (DME)

- If a collection of processes share a resource or collection of resources, then mutual exclusion is often required to prevent interference and ensure consistency of the shared resources.
- A "solution" for distributed mutual exclusion should be based solely on message passing.
- A particularly interesting example is where a collection of peer processes must coordinate their accesses to shared resources among themselves.
  - This occurs routinely on networks such as Ethernets and IEEE 802.11 wireless networks in 'ad hoc' mode, where network interfaces cooperate as peers so that only one node transmits at a time on the shared medium.

Algorithms for DME

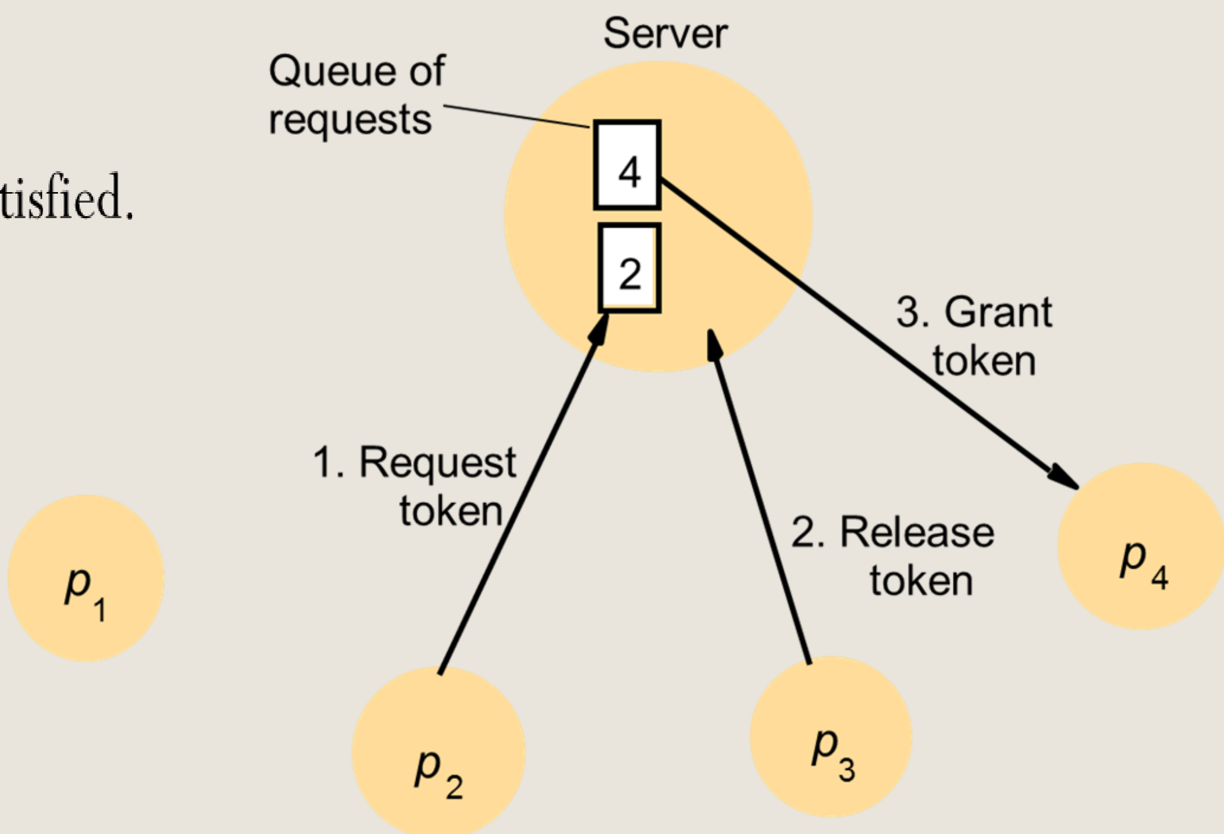
- Consider a system of N processes, named p_i, $i = 1, 2, \ldots, N$.
  - The processes access common resources in a critical section (CS).
  - The system is assumed to be asynchronous, processes are failure free, and the message delivery is reliable.
- Any message sent is eventually delivered undamaged and exactly once.
- The application-level protocol for executing a CS is as follows:
  - enter() // enter the CS – block if necessary
  - resourceAccesses() // access shared resources in the CS
  - exit() // leave the CS – other processes may now enter
- The essential requirements for mutual exclusion are as follows:
  - ME1: (safety) At most one process may execute in the CS at a time.
  - ME2: (liveness) Requests to enter and exit the CS eventually succeed.
    - ME2 implies freedom from both deadlock and starvation.
  - ME3: (HB ordering) Entry to the CS is granted in the HB order of requests.

A Central Server DME Algorithm

- Entering the CS takes 2 messages: a request and a grant.
- Exiting the CS takes 1 release message.

The ME1 and ME2 are satisfied.

The ME3 is not satisfied.

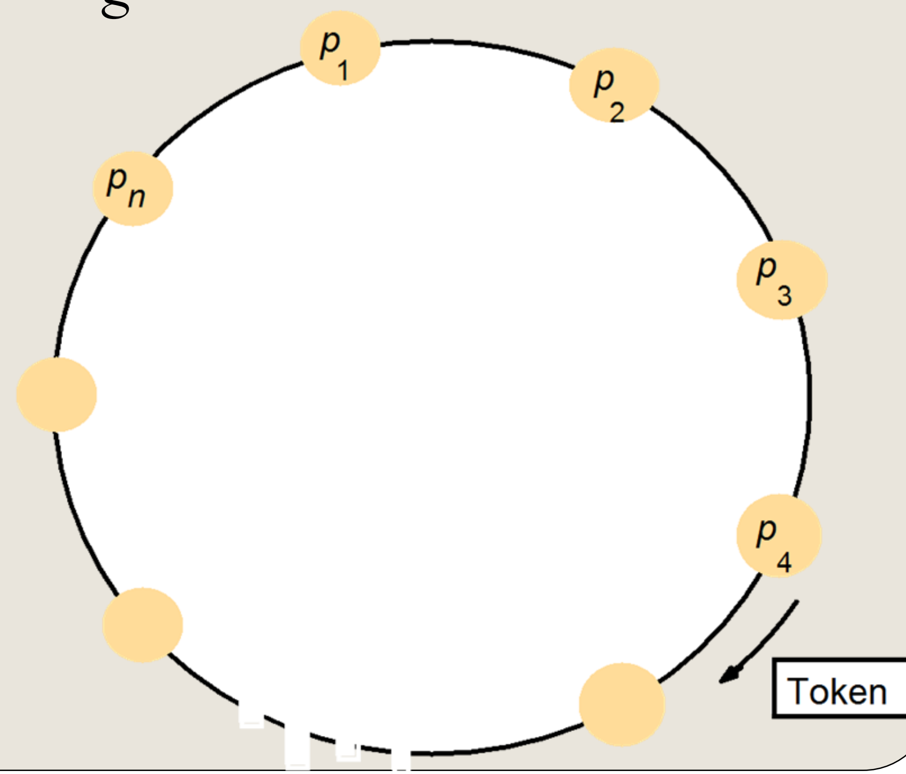


A Ring-Based DME Algorithm

- Entering the CS takes 0 ~ N messages.
  - 0: when it has just received the token.
  - N: when it has just passed out the token.
- Exiting the CS requires only one message.

The ME1 and ME2 are satisfied.

The ME3 is not satisfied.



The Ricart and Agrawala's DME Algorithm

- A process sends a request message to all other processes when it wants to enter the CS.
  - It can enter only if all the other processes have replied to this request message.
- The conditions under which a process replies to a request are designed to ensure that conditions ME1, ME2, and ME3 are met.
  - The processes $p_1, p_2, \ldots, p_N$ bear unique identifiers.
  - Each process p_i keeps a logical clock for time stamping.
  - There exists a reliable channel between any two processes.
  - Entry requesting messages are of the form $<T_i, i>$.
    - T_i is the sender's timestamp and i is the sender's identifier.

Pseudo Code for Ricart and Agrawala's DME Algorithm

On initialization at p_i

state := RELEASED; LC_i := any non-zero value;

To enter the critical section at p_i

state := WANTED;
$LC_i := LC_i + 1$;
send a *request* < T_i=LC_i, i> to all other processes;
Wait until (number of replies received == (N – 1));
state := IN-CS;

On receipt of a *request* <T_j, j> at p_i ($i \neq j$)

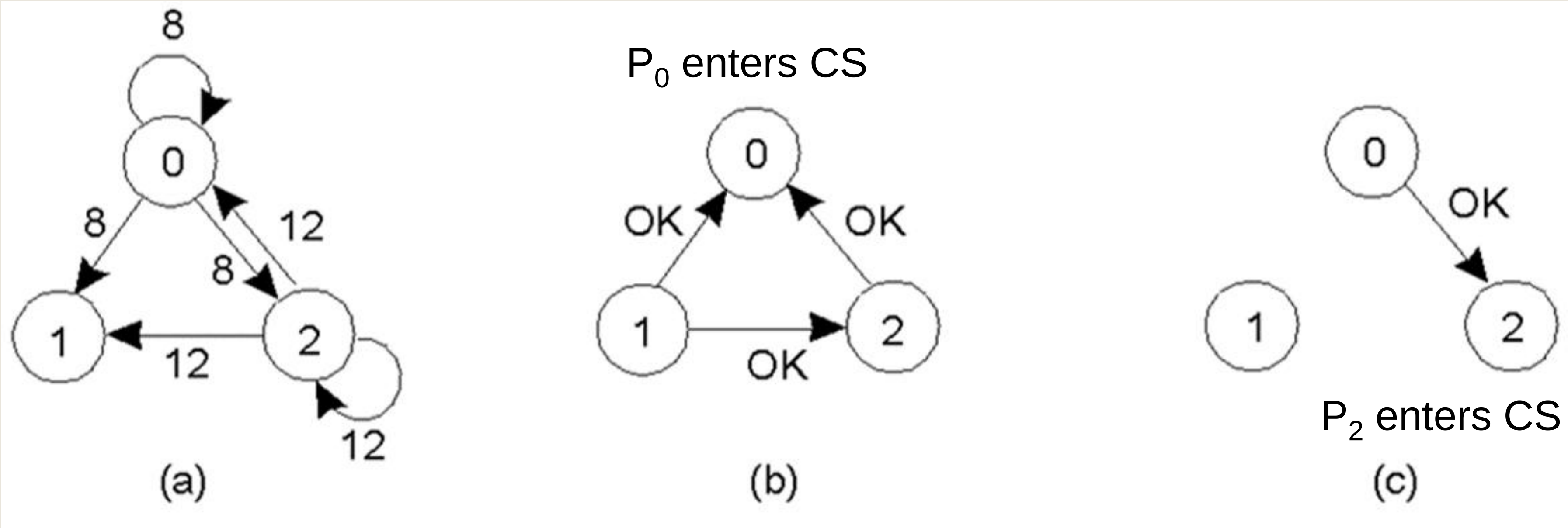
$LC_i := \max(T_j, LC_i) + 1$;
if (*state* == IN-CS or (*state* == WANTED and $(T_i, i) < (T_j, j)$))
// T_i is the timestamp of the p_i's requesting message
then
queue *request* <T_j, j> without replying;
else
reply immediately to p_j;

To exit the critical section at p_i

state := RELEASED;
reply all queued requests;

Properties of the Ricart's Algorithm

- Entering the CS takes 2(N-1) messages.
  - (N-1) messages for request and (N-1) messages for reply.
- Exiting the CS requires no message.
- All three requirements of ME1, ME2, and ME3 are met.



The Maekawa's DME Algorithm (1)

- Maekawa observed that in order for a process to enter a CS, it is not necessary for all of its peers to grant its access.
  - Processes can vote for each other to enter the CS.
  - A 'candidate' process must collect sufficient votes to enter the CS.
  - Each process must cast its vote to only one candidate at a time for ensuring that at most one process can enter the CS.
  - Majority voting is one simple mechanism of this scheme.
- However, the simple majority voting requires at least 3(1+N/2) message for each CS.
  - Entering CS takes (1+N/2) ~N requests and at least (1+N/2) grants.
  - Exiting CS takes at least (1+N/2) releases (for releasing grants).
- In addition, revoking-grant and re-grant messages may be needed to handle deadlock and starvation.

The Maekawa's DME Algorithm (2)

- Each process p_i is assigned with a voting set V_i, $i = 1, 2, \ldots, N$.
  - For all $V_i \subseteq \{p_1, p_2, \ldots, p_N\}$.
- The sets V_i are chosen so that, for all $i, j = 1, 2, \ldots, N$,
  - $p_i \in V_i$
  - $V_i \cap V_j \neq \emptyset$
  - $|V_i| = K$, each process has a voting set of the same size of K.
  - Each process p_j is contained in M voting sets.
- Maekawa showed that the optimal solution, which minimizes K and allows the processes to achieve mutual exclusion, has $K \sim \sqrt{N}$ and $M = K$.

Pseudo Code for Maekawa's DME Algorithm

On initialization
state := RELEASED;
voted := FALSE;

To enter the critical section at p_i
state := WANTED;
send *request* to all processes in V_i;
Wait until (# of replies received == *K*);
state := IN-CS;

On receipt of a *request* from p_j at p_i
if (*state* == IN-CS *or voted* == TRUE){
  queue *request* from p_j without replying;
} else {
  send *reply* to p_j; *voted* := TRUE;
}

To exit the critical section at p_i
state := RELEASED;
send *release* to all processes in V_i;

On receipt of a *release* from p_j at p_i
if (queue of *requests* is non-empty){
  remove head of queue (*request* from p_k);
  send *reply* to p_k; *voted* := TRUE;
} else {
  voted := FALSE;
}

Only ME1 is satisfied.
Both ME2 and ME3 are not satisfied.

Deadlocks in Meakawa's Algorithm

- Consider three processes, p_1, p_2, and p_3, where $V_1=\{p_1, p_2\}$, $V_2=\{p_2, p_3\}$, and $V_3=\{p_3, p_1\}$.
  - If they concurrently request to enter the CS, then it is possible for p_1 to reply to itself and hold off p_2, for p_2 to reply to itself and hold off p_3, and for p_3 to reply to itself and hold off p_1.
  - Each process has received only one replies and none can proceed.
- Later Sanders adjusted the algorithm to become deadlock-free.
  - It requires extra messages to handle deadlocks.
  - Maximum number of messages required per CS is $5\sqrt{N}$.
- To handle deadlocks, processes may change its vote for ensuring HB ordering of entering CS, so that requirement ME3 is also satisfied.

Deadlock Handling in Meakawa's Algorithm

- To avoid deadlock, a process p needs to yield a vote if the timestamp of p's request is larger than the timestamp of some other requests waiting for the same vote.
- When a higher priority request arrives and waits at a process p,
  - If p has sent a REPLY message to a lower priority request, a deadlock may occur.
- Deadlock handling requires three extra types of messages:
  - A FAILED message from p_i to p_j indicates that p_i can not grant p_j a vote because it has voted to a process with a higher priority request.
  - An INQUIRE message from p_i to p_j indicates that p_i would like to find out from p_j if it has all votes from processes in V_j.
  - A YIELD message from p_i to p_j indicates that p_i is returning the vote to p_j for yielding to a higher priority request at p_j .

Deadlock Free Maekawa's DME Algorithm (1)

On initialization

state := RELEASED; *failed* := ∅; *replied* := ∅;

voted := FALSE; *voted-to* := (∞,∞);

To enter the critical section at p_i

state := WANTED;

send request(L_i, i) to all processes in V_i;

wait until (*replied* == $|V_i|$);

state := IN-CS;

On receipt of a request(L_j, j) **from p_j at p_i**

```
if (state == IN-CS or voted == TRUE){
   queue request(L_j, j) in proper position;
   if ( (L_j, j) < voted-to )
       send INQUIRE to p_k, where voted-to is (L_k,k);
   else
       send FAILED to p_j;
} else {
   send reply to p_j; voted := TRUE; voted-to := (L_j, j);
}
```

Deadlock Free Maekawa's DME Algorithm (2)

On receipt of a *reply* from p_j at p_i

replied := *replied* ∪ {p_j};

failed := *failed* – {p_j};

On receipt of a *FAILED* from p_j at p_i

failed := *failed* ∪ {p_j};

On receipt of a *INQUIRE* from p_j at p_i

if ((failed ≠ ∅) or (state ≠ IN-CS)){

send *YIELD* to p_j; *replied* := *replied* - {p_j};

failed := failed ∪ {p_j};

}

On receipt of a *YIELD* from p_j at p_i

queue request(L_j, j) in proper position;

// where *voted-to* is (L_j, j)

remove head of queue (L_k, k);

send *reply* to p_k; *voted* := TRUE;

voted-to := (L_k, k);

To exit the critical section at p_i

state := RELEASED;

send *release* to all processes in V_i;

On receipt of a *release* from p_j at p_i

if (queue of requests is non-empty){

remove head of queue (L_k, k);

send *reply* to p_k;

voted := TRUE;

voted-to := (L_k, k);

} else {

voted := FALSE;

}

N=K(K-1)+1 is optimal
N=3*2+1=7

$V_1=\{1, 2, 3\}$
$V_2=\{2, 4, 6\}$
$V_3=\{3, 5, 6\}$
$V_4=\{4, 1, 5\}$
$V_5=\{5, 2, 7\}$
$V_6=\{6, 1, 7\}$
$V_7=\{7, 3, 4\}$

N=4*3+1=13

$V_1=\{1, 2, 3, 4\}$
$V_2=\{2, 5, 8, 11\}$
$V_3=\{3, 6, 8, 13\}$
$V_4=\{4, 6, 10, 11\}$
$V_5=\{5, 1, 6, 7\}$
$V_6=\{6, 2, 9, 12\}$
$V_7=\{7, 2, 10, 13\}$
$V_8=\{8, 1, 9, 10\}$
$V_9=\{9, 3, 7, 11\}$
$V_{10}=\{10, 3, 5, 12\}$
$V_{11}=\{11, 1, 12, 13\}$
$V_{12}=\{12, 4, 7, 8\}$
$V_{13}=\{13, 4, 5, 9\}$

N = 4*4 = 16, Arrange N processes in a 2D matrix.

Let V_i = {hosts in the row of p_i} ∪ {hosts in the column of p_i} .

$K=2\sqrt{N} - 1$

V_1={1, 2, 3, 4, 5, 9, 13}

V_2={1, 2, 3, 4, 6, 10, 14}

V_3={1, 2, 3, 4, 7, 11, 15}

V_4={1, 2, 3, 4, 8, 12, 16}

V_5={5, 6, 7, 8, 1, 9, 13}

V_6={5, 6, 7, 8, 2, 10, 14}

V_7={5, 6, 7, 8, 3, 11, 15}

V_8={5, 6, 7, 8, 4, 12, 16}

V_9= {9, 10, 11, 12, 1, 5, 13}

V_{10}={9, 10, 11, 12, 2, 6, 14}

V_{11}={9, 10, 11, 12, 3, 7, 15}

V_{12}={9, 10, 11, 12, 4, 8, 16}

V_{13}={13, 14, 15, 16, 1, 5, 9}

V_{14}={13, 14, 15, 16, 2, 6, 10}

V_{15}={13, 14, 15, 16, 3, 7, 11}

V_{16}={13, 14, 15, 16, 4, 8, 12}

Token Based DME Algorithms

- Inspired by the token ring and Ricart's DME algorithm, Suzuki and Kasami proposed to replace reply messages by a privilege (token) in such a way that each CS section only needs N messages.
  - N-1 requests and 1 token transfer.
  - Each node uses a vector RN of size N for recording the largest sequence number ever received from other nodes.
  - In addition, each node has a waiting queue Q to record all known pending requests.
  - The token is transferred with the RN and Q of the original holder.
- Later, Raymond proposed to arrange nodes as a minimum spanning (MSP) tree in such a way that all messages (request and token) are sent along the (undirected) edges of this tree.

The DME Algorithm of Suzuki and Kasami

On initialization

```
state := RELEASED;
HaveToken := TRUE or FALSE;
// TRUE for node 1, FALSE for others;
Q := Ø;
RN[j] := -1, for all j = 1,2, . . . , N;
LN[j] := -1, for all j = 1,2,. . . , N;
```

To enter the critical section at p_i

```
state := WANTED;
if (not HaveToken) {
  RN[i] := RN[i] + 1;
  for all (j≠i, j=1,2,…,N)
     send request (i, RN[i]) to p_j;
  wait until token (Q, LN) is received;
  HaveToken := TRUE;
  state := IN-CS;
}
```

On receipt of a *request* (j, n) from p_j at p_i

```
RN[j] := max(RN[j], n);
if (HaveToken and (state == RELEASED)
   and (RN[j]==LN[j]+1)){
     HaveToken := FALSE;
     send token(Q, LN) to p_j;
}
```

To exit the critical section at p_i

```
LN[i] := RN[i];
for all (j≠i, j=1,2,…,N)
  if ((j is not in Q) and (RN[j]==LN[j]+1))
     Q := append(Q, j);
if (Q ≠ Ø ){
   HaveToken := FALSE;
   Q := removehead(Q, k);
   send token(Q, LN) to p_k;
};
state := RELEASED;
```

Raymond's Tree-Based Algorithm (1)

- Initially, the root of the MSP tree holds the Token.
- The root has Holder pointing to itself and HaveToken := True.
  - All other nodes have Holder pointing to their parents in the tree and HaveToken := False.
- Each node has a FIFO queue Q for token requesting neighbors.

On initialization

```
state := RELEASED;
HaveToken := TRUE or FALSE;
// TRUE for root node, FALSE for others;
Q := Ø;
Holder := self or parent;
// self for root node, parent for others;
```

To enter the critical section at p_i

```
state := WANTED;
If (not HaveToken){
    If (Q == Ø )
        send a request token (i) to Holder;
    enqueue(Q, i);
}else
    state := IN-CS;
```

Raymond's Tree-Based Algorithm (2)

On receiving a request token (j) at p_i

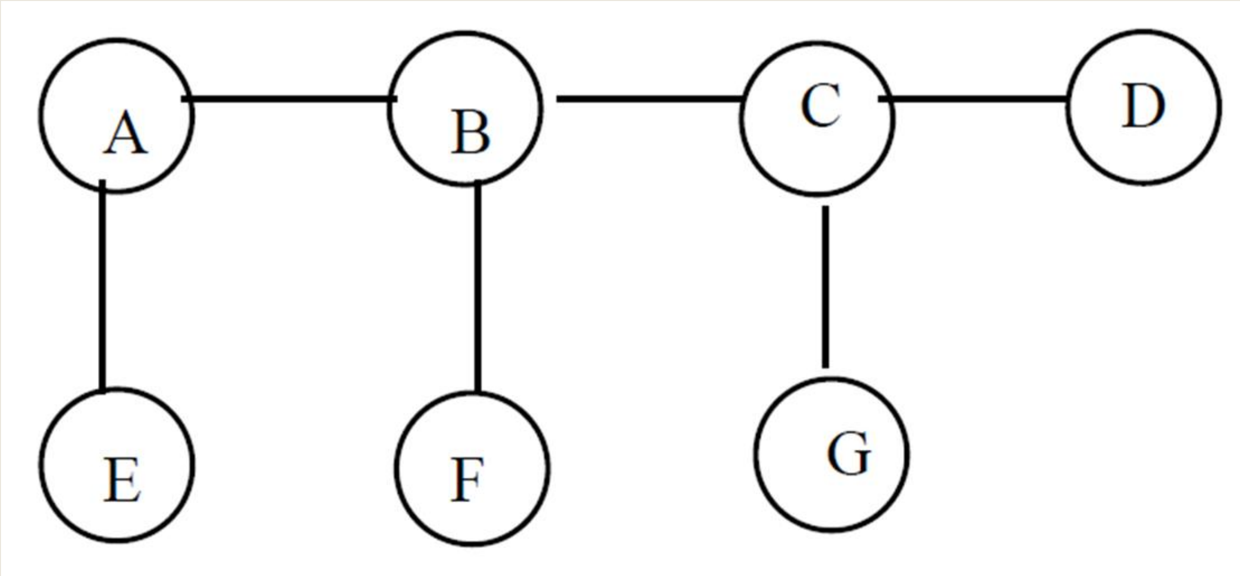
```
if (Q ≠ Ø or state == IN-CS) {
    enqueue(Q, j);
}else if (HaveToken) {
    send a token message to pj;
    HaveToken := False;
}else{
    enqueue(Q, j);
    send a request token(i) message to Holder;
};
```

To exit the critical section at p_i

```
If (Q ≠ Ø ){
    Holder := dequeue(Q);
    send a token message to Holder;
    HaveToken := False;
    if (Q ≠ Ø )
        send a request token(i) message to
    Holder;
}
```

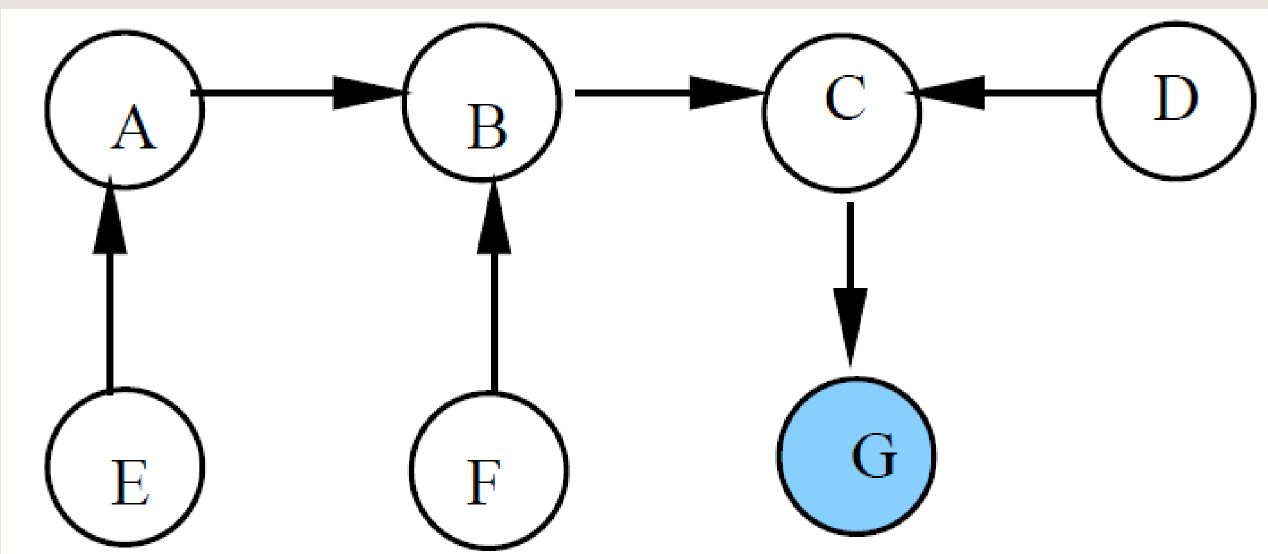
Upon receiving a token at p_i

```
Holder := dequeue(Q);
if (Holder==i){
    HaveToken := True;
    State := IN-CS;
}else{
    send a token message to Holder;
    if (Q ≠ Ø )
        send a request token(i) message to
    Holder;
};
```



The initial tree of 7 nodes.

All logically directed edges are pointing in a direction towards node G (the node has token.)



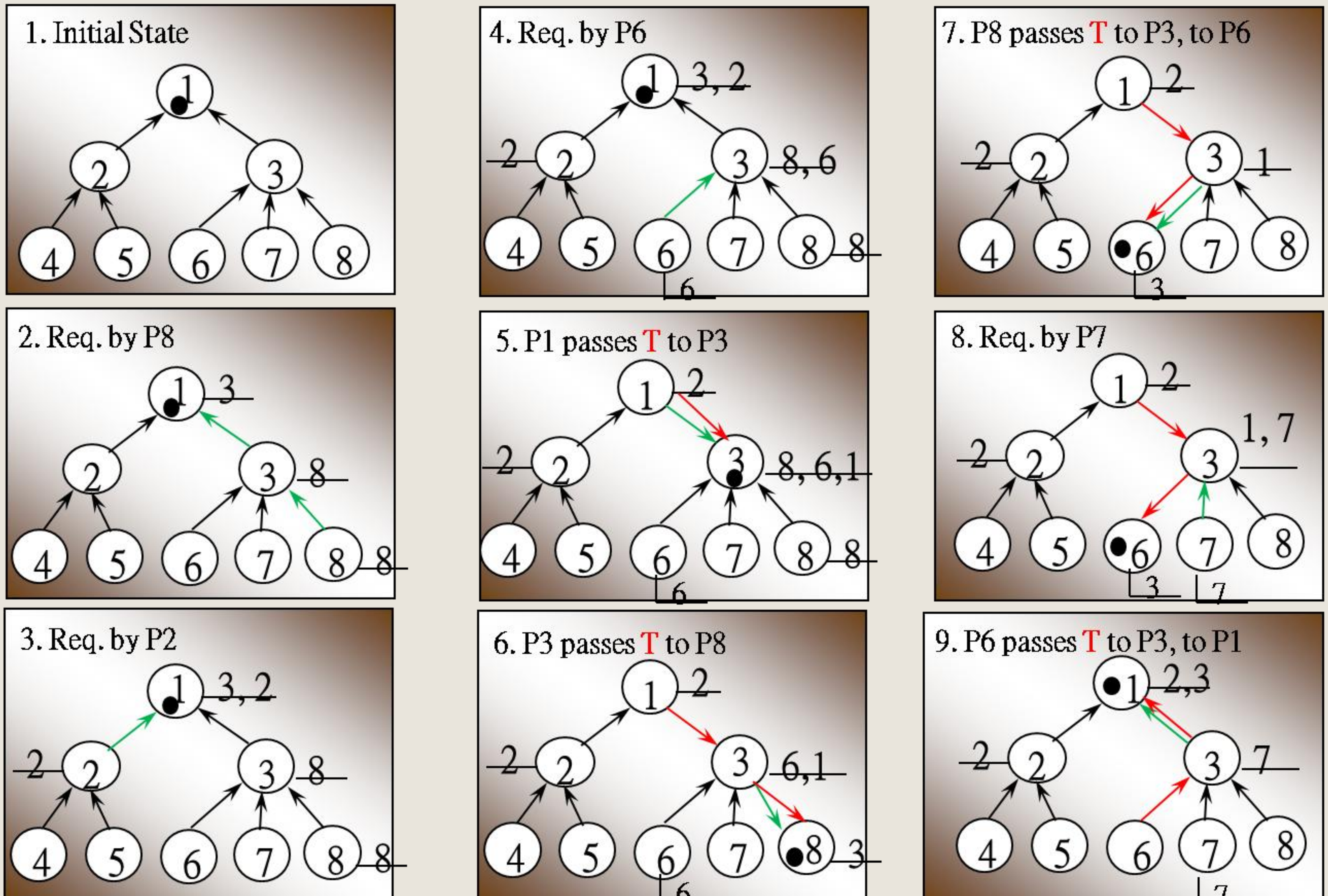
The values of the Holder for each node:

Holder(A) = B
Holder(B) = C
Holder(C) = G
Holder(D) = C
Holder(E) = A
Holder(F) = B
Holder(G) = G

Properties of Raymond's DME Algorithm

- In the worst-case, the algorithm requires (2 * longest path length of the tree) messages per CS execution.
  - This happens when the token is passed between nodes at either ends of the longest path of the minimal spanning tree.
  - The worst topology is where nodes are arranged in a straight line.
    - In this case, 2 * (N – 1) messages are needed per CS execution.
- In general, MSP trees with high fan-outs are preferred.
  - The longest path length of such trees is typically O(log N).
  - On average, O(log N) messages are needed per CS execution.
- Under heavy load, it exhibits an interesting property:
  - As the number of nodes requesting for the token increases, the number of messages exchanged per CS entry decreases.
  - In heavy load, it requires only 4 messages per CS execution.

An example of Raymond's DME Algorithm



Path Compressing in MSP Tree (1)

- Instead of using the static MSP tree, Naimia, Trehela, and Arnold dynamically adjust the MSP in such a way that the new root of the MSP is the next requester known by the original root.
  - All nodes in the path between both roots also changed the value of their holders to be the new root.
- It was shown that the dynamic MSP tree DME algorithm only needs about $1+\log_2 N$ messages per CS.

On initialization

state := RELEASED;
HaveToken := TRUE or FALSE;
// TRUE for root node, FALSE for others;
next := nil;
Holder := root or nil;
// nil for root, root for others

To enter the critical section at p_i

state := WANTED;
If (Holder ≠ nil){
    send a request token (i) to *Holder*;
    Holder := nil;
}else
    state := IN-CS;

Path Compression in MSP Tree (2)

On receiving a request token(j) at p_i

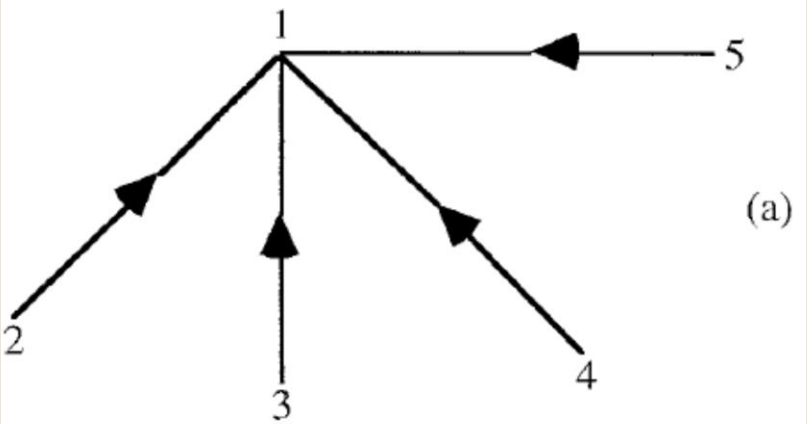
```
if (Holder == nil) {
    if (state == WANITED or state == IN-CS)
        next := j;
   } else {
        send a token message to pj;
        HaveToken := False;
   }
} else {
    send a request token(j) message to Holder;
};
Holder := j;
```

To exit the critical section at p_i

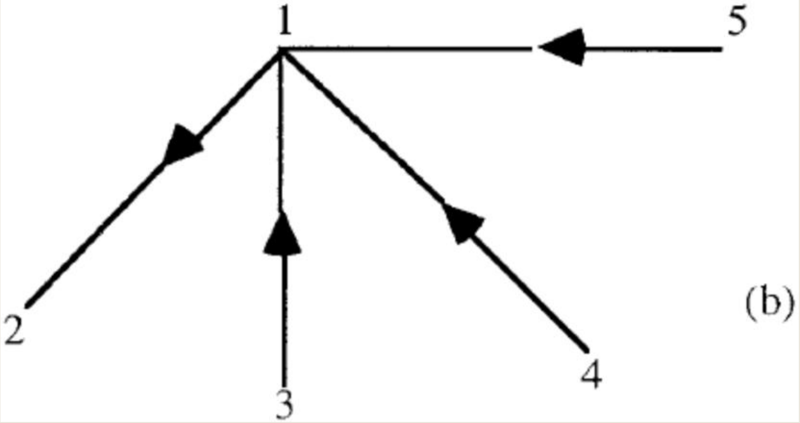
```
state := RELEASED;
If (next ≠ nil ){
    send a token message to next;
    HaveToken := False;
    next := nil;
    }
```

Upon receiving a token at p_i

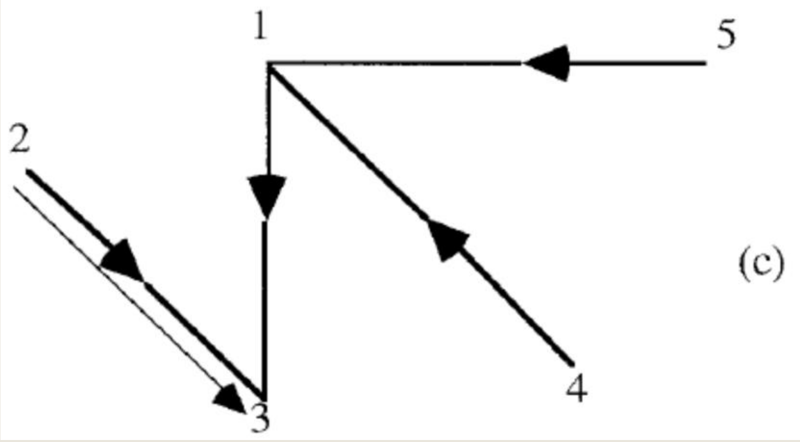
```
HaveToken := True;
if (state == WANTED)
    state := IN-CS;
```



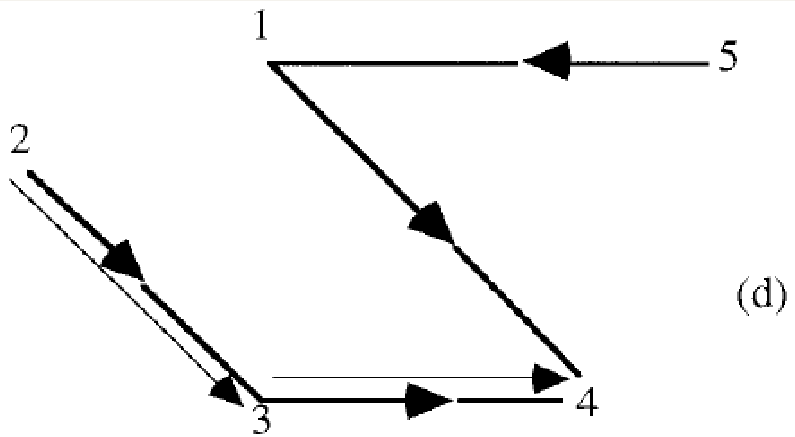
1. Initially, P_1 has the token and is the root.
2. All others have Holders pointing to P_1.



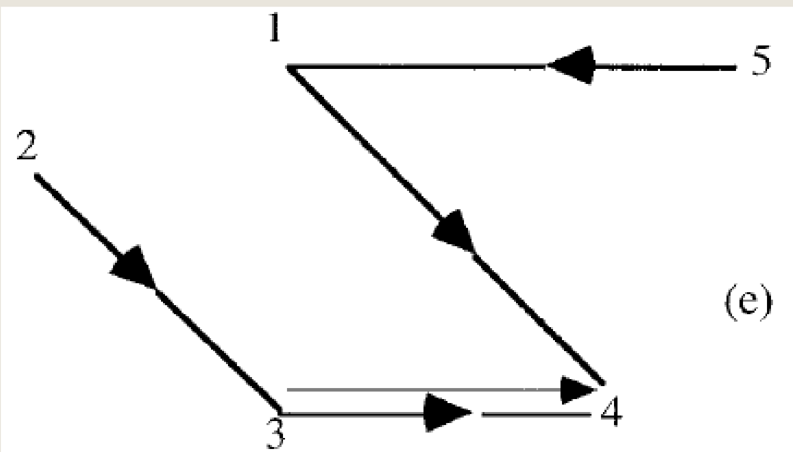
1. P_2 wants to enter CS, sends a request (2) to its Holder (P_1), and sets its Holder to nil.
2. P_1 sends the token to P_2,sets its Holder to P_2.
3. P_2 receives the token and enter CS.



1. P_3 wants to enter CS, sends a request (3) to its Holder (P_1), and sets its Holder to nil.
2. P_1 forward a request (3) to its Holder (P_2) and sets its Holder to P_3.
3. P_2 receives a request (3), sets its next to P_3, and sets its Holder to P_3 also.



P_4 wants to enter CS, sends a request (4) to its Holder $P_{1,}$ and sets its Holder to nil.
P_1 forwards a request (4) to its Holder (P_3) and sets its Holder to P_4.
P_3 receives a request (4), sets its next to P_4, and sets its Holder to P_4 also.



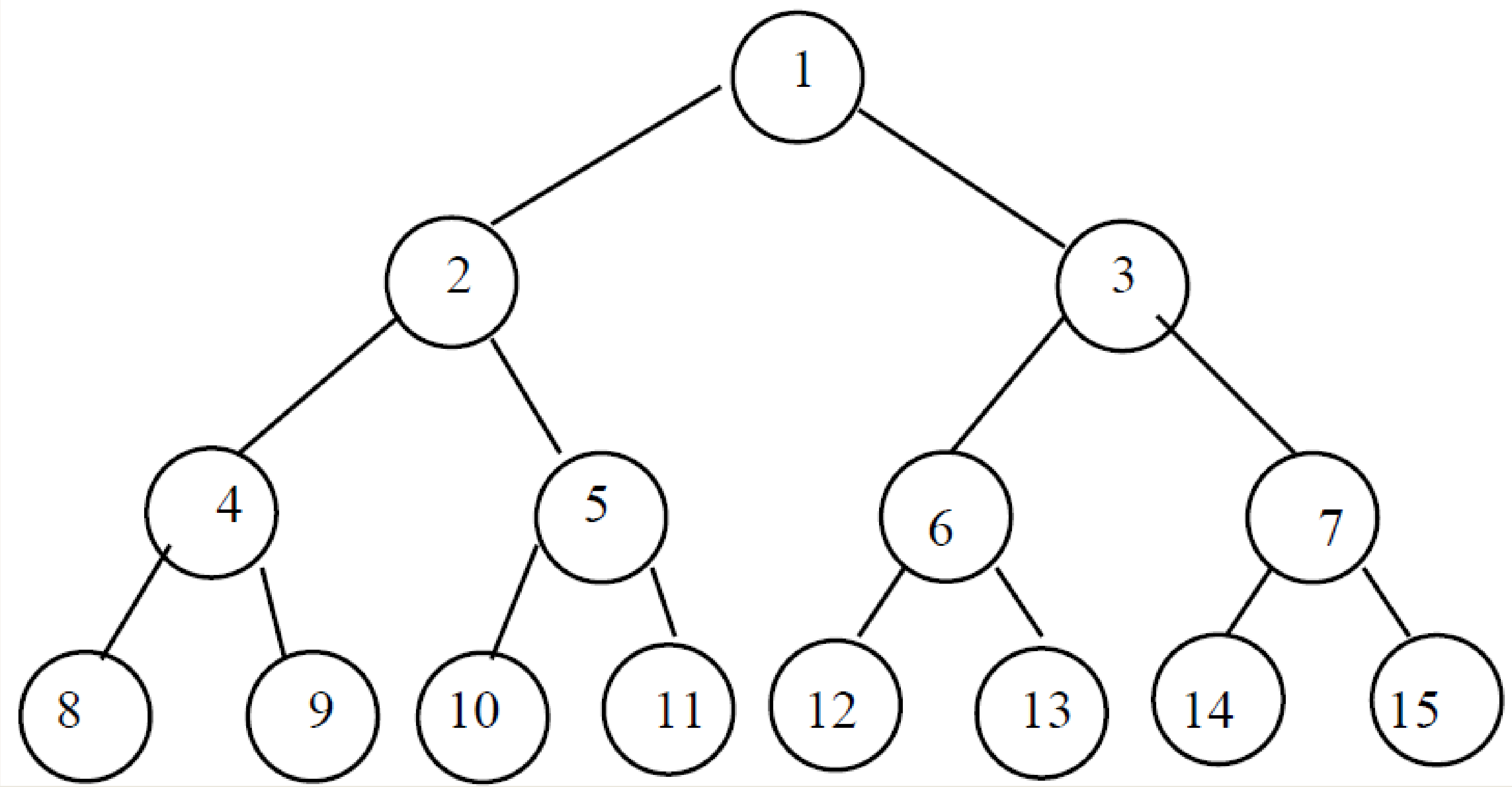
P_2 exits CS, sends the token to its next (P_3), and sets its next to nil.

A Fault Tolerant DME Algorithm

- Both token passing based (Suzuki, Raymond, Naimia, …) and quorum based (Ricart, Maekawa, Sanders, …) DME algorithms need to handle failures with special cares.
- Agrawal and Abbadi proposed a mechanism to dynamically create quorums for processes in a DS when failures occur.
  - A coterie is a set of sets with the property that any two members of a coterie have a nonempty intersection.
  - A DS is organized as a logical binary tree.
    - Each node in the tree is representing a process in the DS.
    - A valid quorum is a path from any leave node to the root.
    - If a node S fails, then any valid quorum Q which contains S can be adjusted to Q', where $Q' = Q - \{S\} + LQ_S + RQ_S$.
    - LQs and RQs are any valid quorum rooted by left and right children of S respectively.

An example of 15 nodes

- 8 possible valid quorums are
  - root-leaf paths: 1-2-4-8, 1-2-4-9, 1-2-5-10, 1-2-5-11,
    1-3-6-12, 1-3-6-13, 1-3-7-14 and 1-3-7-15.



Fault Tolerant Properties

- If any site fails, the quorums containing that site are adjusted by two possible paths starting from the site's two children and ending in leaf nodes.
- When node 3 fails, quorums are adjusted as following:
  - The possible paths starting from child 6 are 6-12 and 6-13, and from child 7 are 7-14 and 7-15.
  - Thus, the new valid 8 quorums are:

    {1,2,4,8},{1,2,4,9},{1,2,5,10}, {1,2,5,11},
    {1,6,12,7,14}, {1,6,12,7,15}, {1,6,13,7,14}, {1,6,13,7,15},
- It can tolerate up to $N-\lceil \log 2N \rceil$ node failures and still form a tree quorum.
- Nodes 1, 2, 4, 8 can form a quorum when all other nodes fail.

Leader Elections (LE) in DSs

- In a DS, it is a common practice to designate a process (a leader) as the coordinator of some forthcoming task.
  - In the central-server DME algorithm, the server is chosen from the processes p_i, ($i = 1, 2, \ldots, N$) that need to use the CS.
- Assume that a unique priority number is associated with each active process in the system, and assume that the priority number of process p_i is i.
  - The new coordinator is the process with the largest priority number.
- When a coordinator fails, the active process with the largest priority number must be elected as the new leader.
- Two algorithms, the bully algorithm and a ring algorithm, can be used to elect a new coordinator in case of failures.

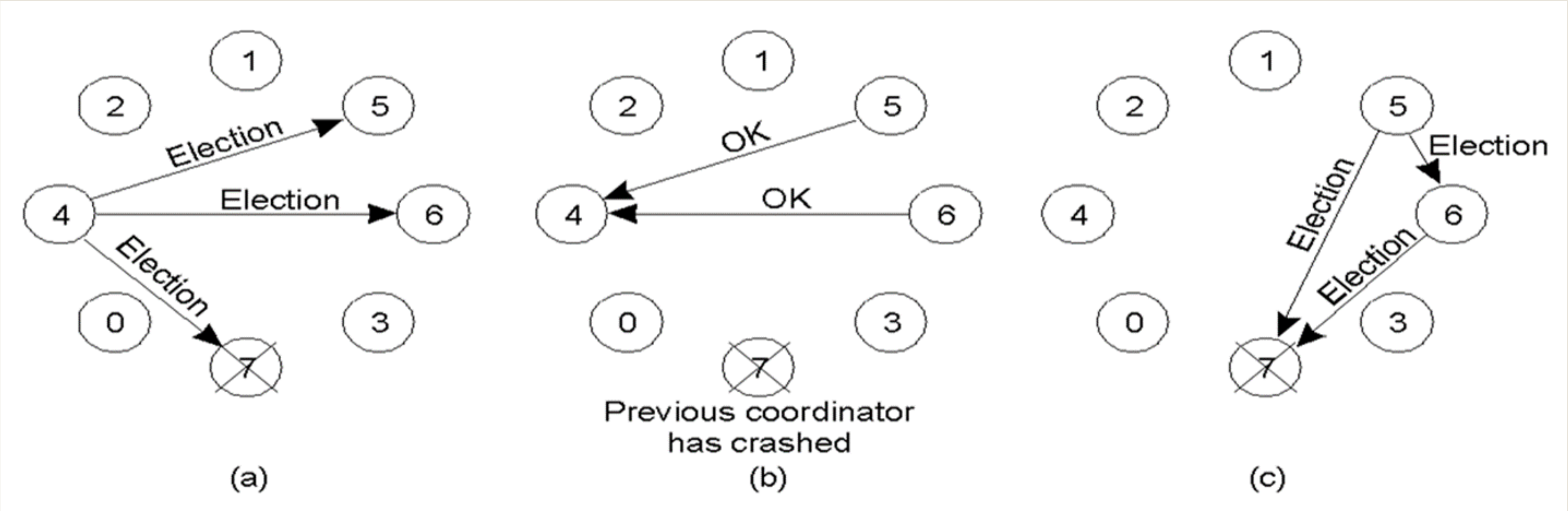
The Bully LE Algorithm (1)

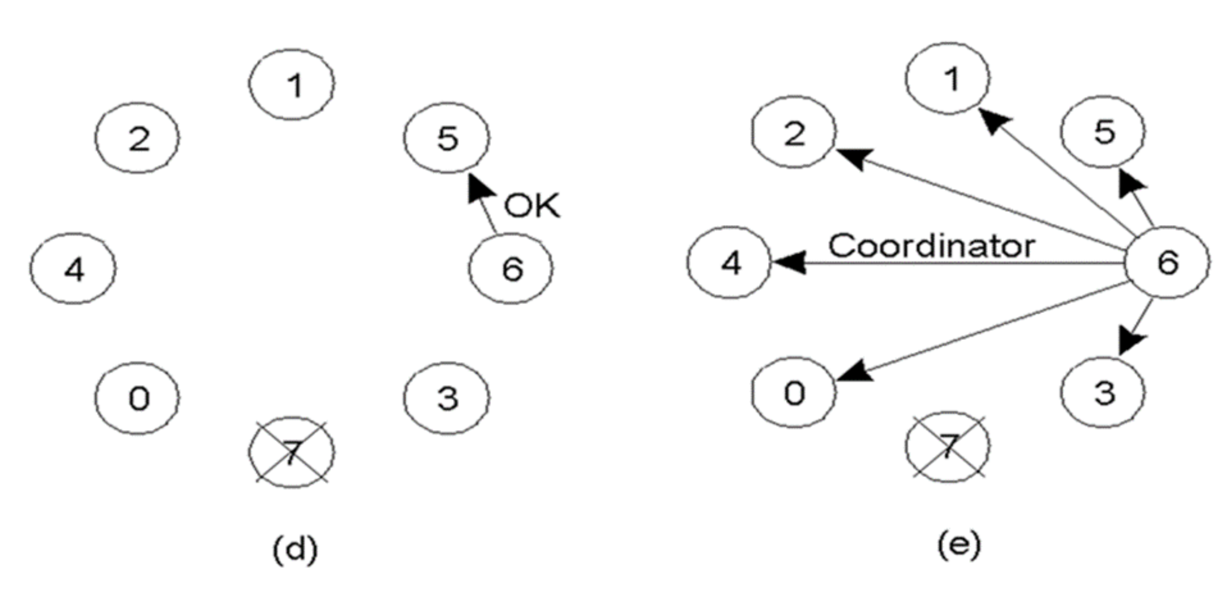
- It is assumed that every process can send a message to all other processes in the system.
- If process p_i sends a request that is not answered by the coordinator within a predefined time interval T,
  - the coordinator is assumed to have failed.
- The p_i then tries to elect itself as the new coordinator.
  - p_i sends an election(i) message to every process p_j with a higher priority number than p_i and waits for a response within T.
  - If no response within T,
    - it is assumed that all processes with higher priority have failed.
    - The p_i then elects itself as the new coordinator and sends coordinator(i) to all processes.

The Bully LE Algorithm (2)

  - If a response is received, p_i begins a time interval T' and waits for a coordinator(?) message.
    - If no coordinator(?) message is received within T',
      - it is assumed that the process with a higher number has failed.
      - p_i should restart the leader election algorithm again.
- If p_i is not the current coordinator, then p_i may receive one of the following two messages from process p_j.
  - If receiving a coordinator(j) message and (j > i),
    - p_i records that p_j is the new coordinator.
  - If receiving an election(j) message and (j < i):
    - p_i sends a response to p_j and begins its own election algorithm if p_i has not already initiated such an election.
- After recovery, a process immediately calls a new election if it has higher priority number than the current leader.

A Bully Algorithm Example





1. P4 calls an election.
2. P5 and P6 respond to stop P4.
3. Both P5 and P6 call an election.
4. P6 responds to stop P5.
5. P6 wins and tells everyone.

The Bully Algorithm Properties

- Assume there are N processes.
  - Worst Case:
    - The lowest priority process initiates election.
    - It requires $O(N^2)$ messages
  - Best Case:
    - The eventual leader initiates election.
    - It requires $O(N)$ messages

The Ring LE Algorithm (1)

- A DS can be organized as a ring logically or physically.
- Each process P_i can send messages to its right neighbor process in the ring, $P_{(i+1 \bmod N)}$.
- Each process maintains an active list (AC).
  - The AC will contain the priority numbers of all active processes in the system when the election terminates.
- If process P_i detects a coordinator failure,
  - P_i first creates a new AC and sets AC := $\emptyset$.
  - It then sends a message elect(i) to its right neighbor.
  - It also sets AC := AC $\cup \{i\}$.

The Ring LE Algorithm (2)

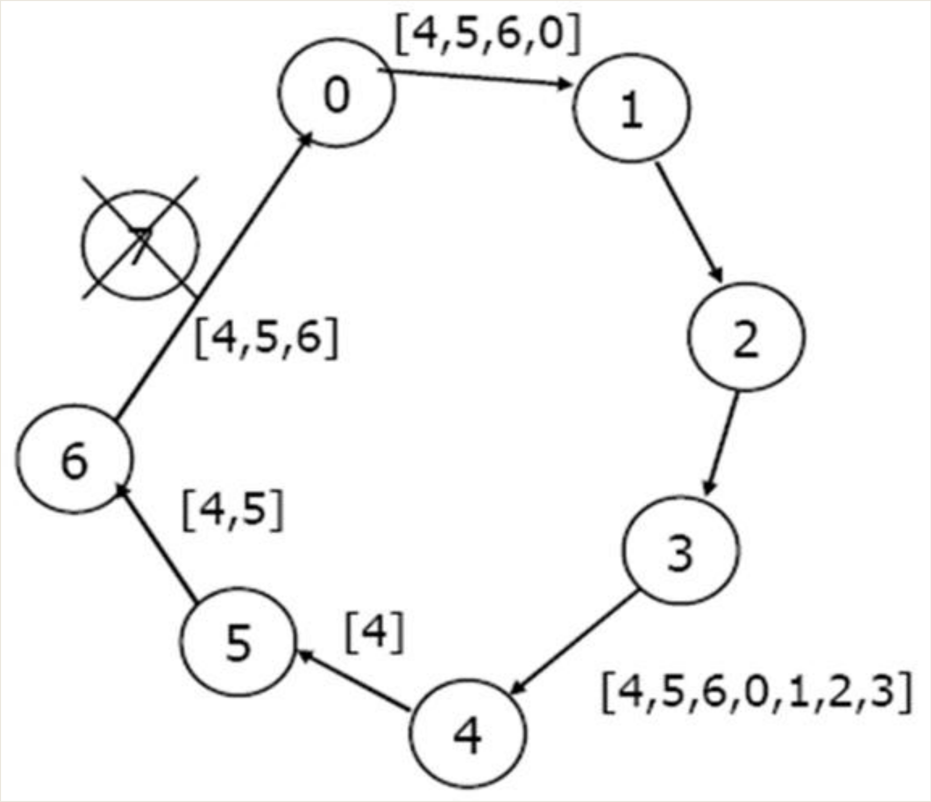
- If P_i receives a message elect(j), it must respond in one of 3 ways:
  - If this is the first elect message it has seen or sent,
    - P_i creates a new AC and sets AC := $\{i, j\}$.
    - It then sends the message elect(i) and the message elect(j) to its right neighbor.
  - If this is not the first elect message it has seen or sent and i $\neq$ j,
    - P_i sets AC := AC $\cup$ $\{j\}$ and forwards the message elect(j) to its right neighbor.
  - If i == j, then P_i receives its own elect message elect(i) back.
    - The AC of P_i now contains all the active processes in the system.
    - P_i can now determine the new coordinator process and send a message leader(k).
- If P_i receives a message leader(k), it must respond in one of 2 ways:
  - If it is new to P_i, then it sets P_k as the new leader and sends leader(k) to its right neighbor.
  - Otherwise, it stops the election process.

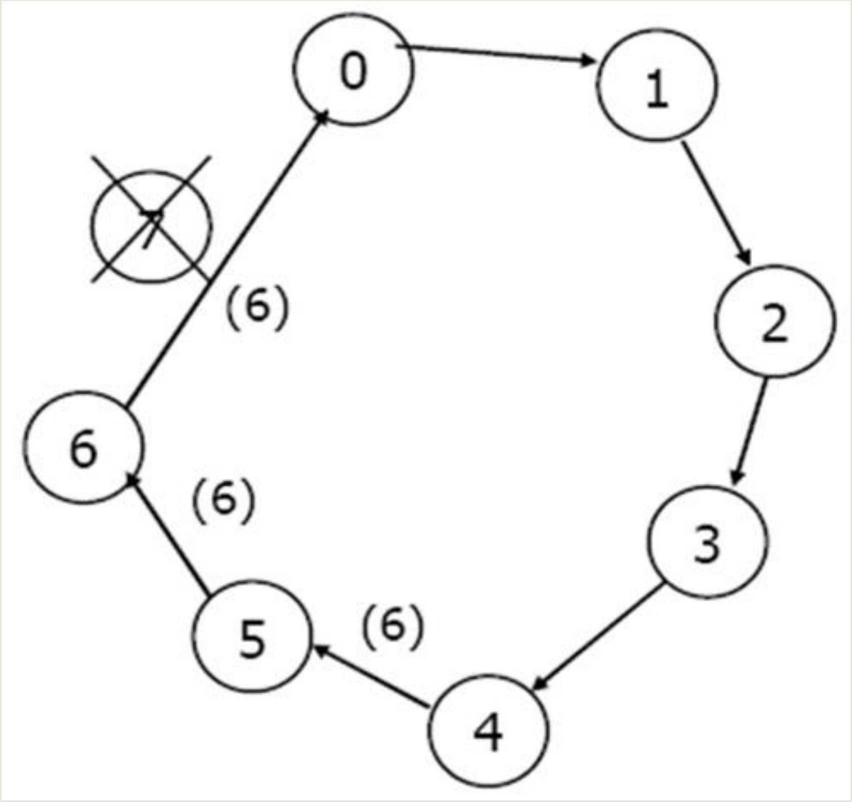
The Ring Algorithm Properties

- If only 1 process initiates election:
  - It requires 2N messages.
  - N for elect and N for leader.
- Two or more processes might simultaneously initiate elections.
  - It is ensured that the same leader will be elected.
  - The total messages may vary from (k+1)N to 2kN if there are k processes initiating election in the same time.

A Ring Algorithm Example

- P4 initiates the election.





Deadlocks in Distributed Systems

- Deadlocks in a DS are similar to deadlocks in a computer.
  - They are harder to avoid, prevent or even detect in a DS.
  - They are hard to recover because all relevant information is scattered over many machines.
- A deadlock can arise if and only if all of the following conditions hold simultaneously in a system:
  - Mutual Exclusion:
    - Each resource has only one copy and can only be accessed exclusively.
  - Hold some resources and Wait some others.
  - No Preemption:
    - Resources can not be forcibly revoked from a holding process.
  - Circular Wait:
    - There is a closed hold and wait cycle existed.

Strategies to Handle Deadlocks

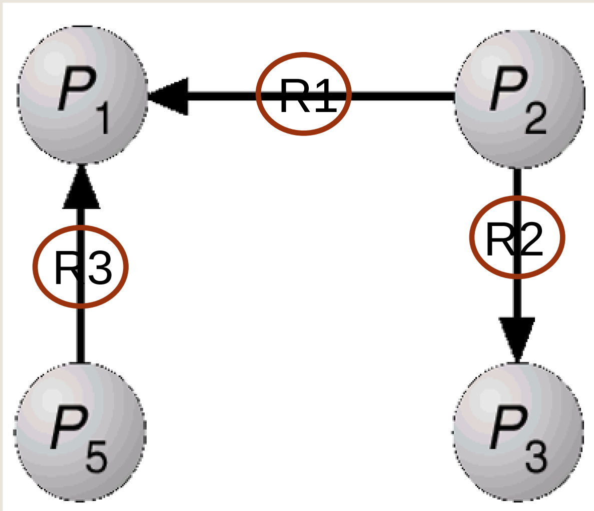
- Detection and Recovery
  - Let deadlocks occur, detect them, and try to recover.
- Prevention
  - Systematically make deadlocks impossible to happen.
    - Make one of 4 necessary conditions impossible.
    - Best possible mechanism is to define a linear ordering of resources and processes requesting resources according to the linear order.
- Avoidance
  - Avoid deadlocks by allocating resources carefully.
    - Do not start a process if its demands might lead to deadlock.
    - Do not grant an incremental resource request to a process if this allocation might lead to deadlock.

A Central Server Deadlock Detection

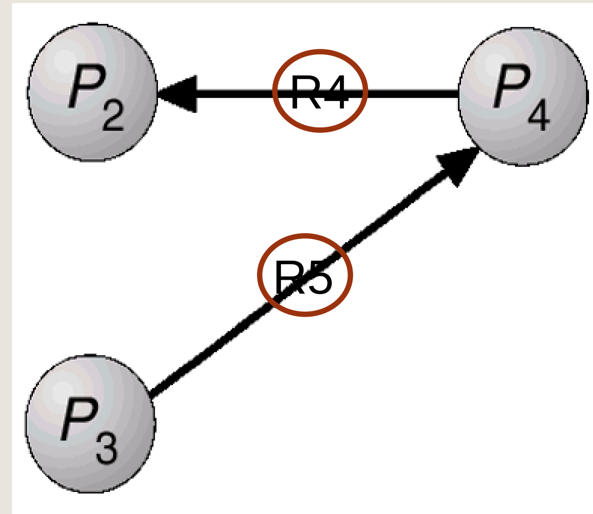
- Each host keeps a local wait-for graph (LWFG) .
  - The nodes of the graph are the processes that are currently holding or requesting any of resources managed by the host.
- A global wait-for graph (GWFG) is managed by a central server.
  - This graph is the union of all local wait-for graphs.
- There are three different options for constructing the GWFG:
  - Whenever a new edge is inserted or removed in one of the LWFG.
  - Periodically or when a number of changes have occurred in a LWFG.
  - Whenever the central server needs to invoke the deadlock detection algorithm.
- The GWFG may contain false cycles.

Two LWFGs and their corresponding GWFG

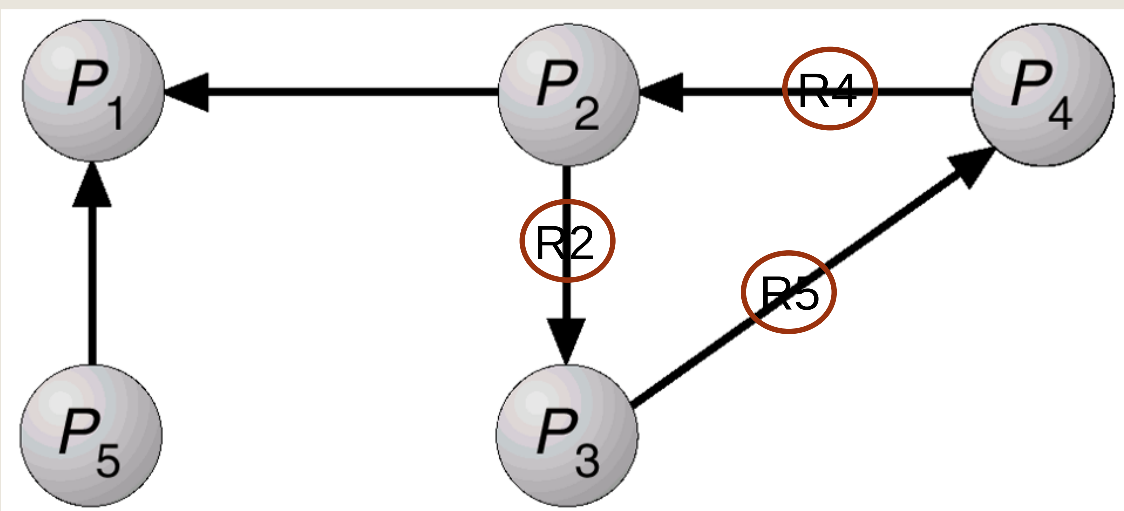
LWFG of host A



LWFG of host B



The combined GWFG



1. P2 in host A and P3 in host B.
2. P2 release(R4) then req(R2).
3. Host B reports its LWFG before receiving release(R4).

The GWFG contains a false cycle of P2→P3→P4→P2

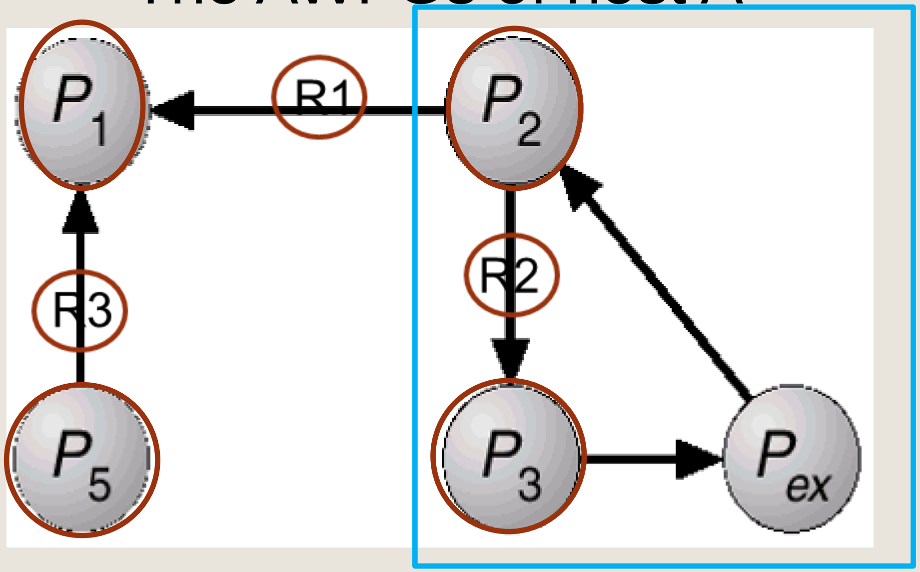
The False Cycle Free Deadlock Detection

- Append timestamps to resource requests among different hosts.
- When process P_i of host A, requests a resource (at host B) held by process P_j, a request message with timestamp TS is sent.
  - The edge $P_i \rightarrow P_j$ with the label TS is inserted in the LWGF of host A.
  - The edge is inserted in the LWFG of host B only if B has received the request message and cannot immediately grant the requested resource.
- To construct the GWFG:
  - The coordinator sends a start message to each site in the system.
  - On receiving the start message, a host sends its LWFG back.
  - When all LWFGs are received, the coordinator constructs the GWFG:
    - The GWFG contains a vertex for every process in the system.
    - The GWFG has an edge $P_i \rightarrow P_j$ if and only if
      - there is an edge $P_i \rightarrow P_j$ in one of the LWFGs, or
      - an edge $P_i \rightarrow P_j$ with a label TS appears in more than one LWFGs.
    - If the GWFG contains a cycle, it implies a deadlock.

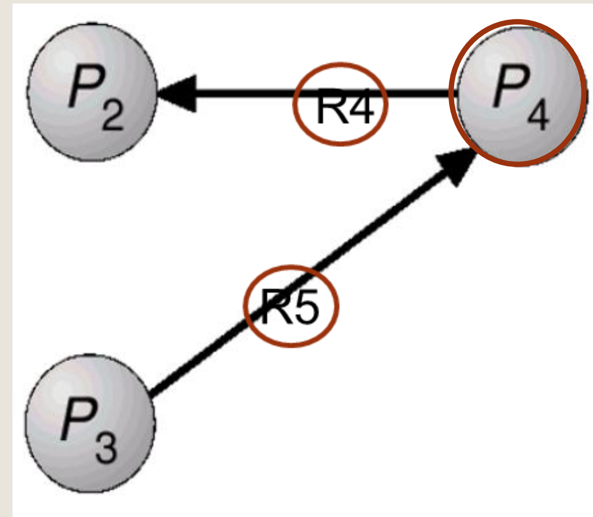
The Fully Distributed DD Algorithm

- All hosts share the same responsibility for detecting deadlock.
- Every host constructs an augmented wait-for graph (AWFG) that represents a part of the GWFG.
- One additional node P_{ex} is added to the LWFG for representing all external events to form the AWFG.
- If an AWFG contains a cycle that does not involve node P_{ex}, then the system is in a deadlock state.
- A cycle involving P_{ex} implies the possibility of a deadlock.
- To ascertain whether a deadlock does exist, an AWFG containing a cycle with P_{ex} must be propagated to other hosts for checking whether a real cycle existed.

The AWFGs of host A



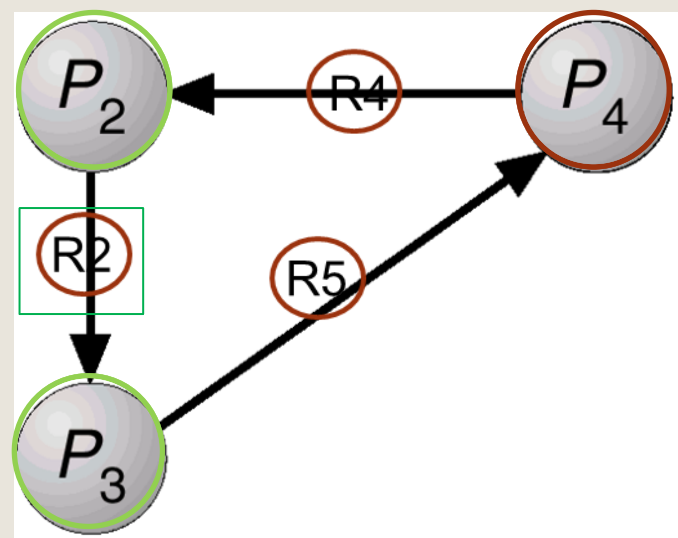
The AWFGs of host B



1. Since the AWFG of A has a cycle involving P_{ex}, host A sends the cycle to host B.

2. Host B updates its AWFG from the cycle and detects a real cycle $P_2 \rightarrow P_3 \rightarrow P_4 \rightarrow P_2$.

The modified AWFG of B



References

- Cristian, F., "Probabilistic Clock Synchronization," Distributed Computing, Vol. 3, pp. 146-158 (1989).
- Gusella, R., and Zatti, S., "The Accuracy of the Clock Synchronization Achieved by TEMPO in Berkeley UNIX 4.3BSD," IEEE Trans. on SE, 15(7), pp. 847-853 (1989).
- Mills, D. L., "Internet Time Synchronization: The Network Time Protocol," IEEE Trans. on Comm., 39(10), pp. 1482-1493 (1991).
- Lamport, L., "Time, Clocks, and the Ordering of Events in a Distributed System," CACM, 21(7), pp. 558-565 (1978).
- Mattern, F., "Virtual time and global states in distributed systems," in Proc. of the Workshop on Parallel and Distributed Algorithms, Amsterdam, North-Holland, pp. 215–226 (1989).
- Fidge, C., "Logical Time in Distributed Computing Systems," IEEE Computer, Vol. 24, pp. 28-33 (1991).
- Ricart, G. and Agrawala, A.K., "An optimal algorithm for mutual exclusion in computer networks," CACM, 24(1), pp. 9-17 (1981).

References

- Maekawa, M., "A $\sqrt{N}$ algorithm for mutual exclusion in decentralized systems," ACM Trans. on Computer Systems, 3(2), pp. 145-159 (1985).
- Sanders, B.A., "The information structure of distributed mutual exclusion algorithms," ACM Trans. on Computer Systems, 5(3), pp. 284-299 (1987).
- Suzuki, I. and Kasami, T., "A distributed mutual exclusion algorithm," ACM Trans. on Computer Systems, 3(4), pp. 344-349 (1985).
- Raymond, K., "A tree-based algorithm for distributed mutual exclusion," ACM Trans. on Computer Systems, 7(1), pp. 61-77 (1989).
- Naimia, M., Trehela, M., and Arnold, A., "A Log (N) Distributed Mutual Exclusion Algorithm Based on Path Reversal," Journal of Parallel and Distributed Computing, 34(1), pp. 1-13, (1996).
- Agrawal, D., and Abbadi, A.E., "An efficient and fault-tolerant solution for distributed mutual exclusion," ACM Trans. on Computer Systems, 9(1), pp. 1-20 (1991).
- Garcia-Molina, H. and Barbara, D., How to assign votes in a distributed system, JACM, 32(4), pp. 841-860 (1985).

References

- Garcia-Molina, H., "Elections in distributed computer systems," IEEE Trans. on Computers, C-31(1), pp. 48–59 (1982).