Distributed Systems

4. Synchronization

Causality (1)

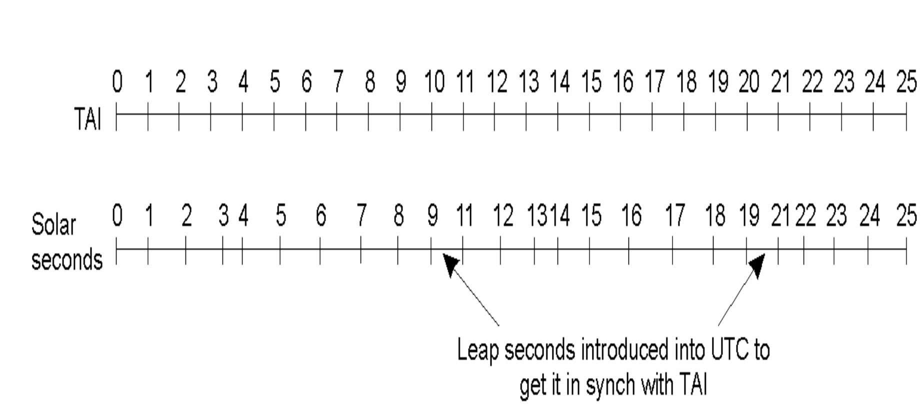
- *Distributed systems lack of a global state, their nature is asynchronous*
- Non-instantaneous communication
  - Different observers may observe the same event at different times and different events at the same time
  - Reason: propagation delay, contention for network resources, retransmission (due to lost messages) etc.
- Relativistic effects
  - Synchronizing by time is unreliable
  - Reason: clocks tend to drift apart
- Interruptions
  - Even if two computers receive a message at the same time, their reaction may need different time
  - Reason: Complex computer systems with unpredictable execution times due to CPU contention, interrupts, page faults, cache misses, garbage collection etc.

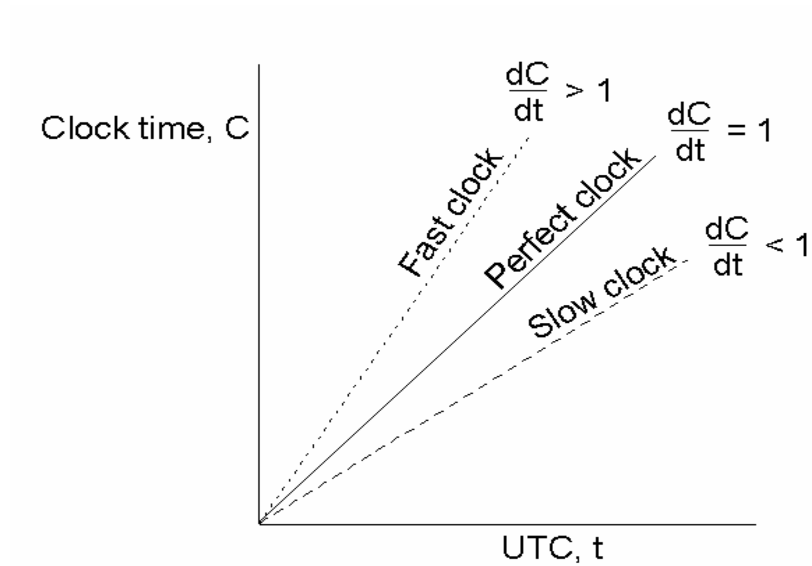
Causality (2)

- Distributed systems are *causal*
  - The cause precedes the effect
  - Traveling backward in time is excluded
- Physical and logical clocks
  - Synchronization of the physical (wall) clocks is possible, but hard and often not necessary
  - A logical clock cares only for the proper *order*
- Basic notions for logical clocks
  - Suppose the distributed system is composed of the set of processors $P = \{p_1, ..., p_m\}$.
  - The set of all events of a distributed system is E, the set of all events of processor p is E_p.

Universal Coordinated Time (UTC)

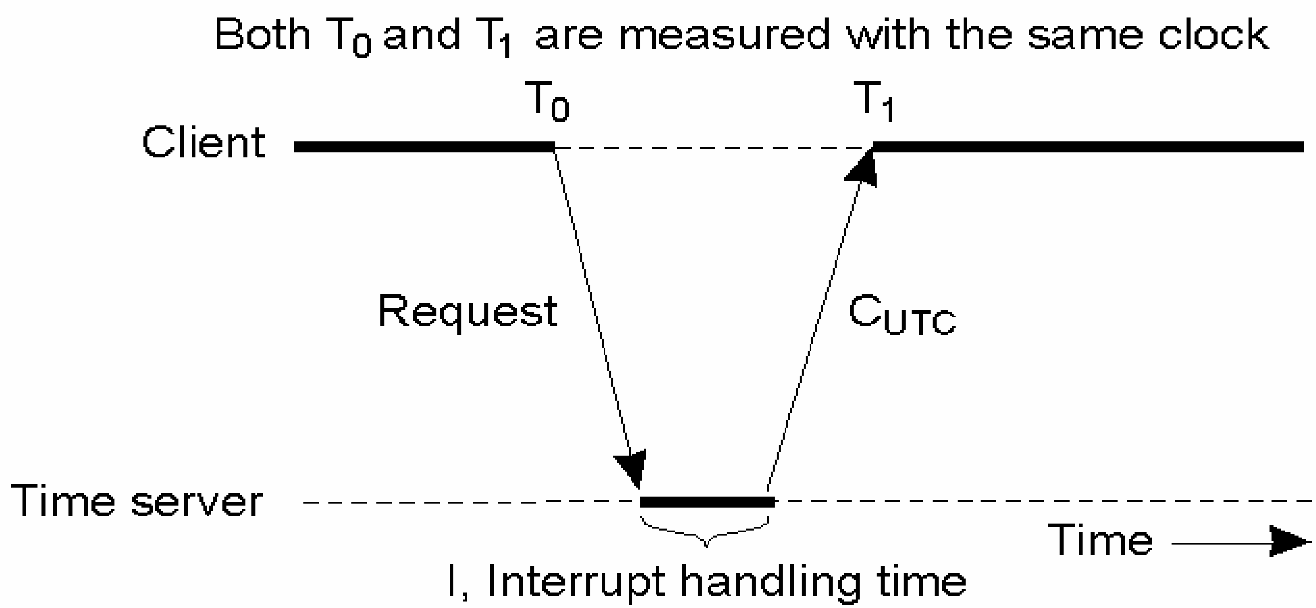
- Abbreviation from French: UTC – not UCT
- International Atomic Time (TAI): cesium clock
- UTC is based on a combination of TAI and mean solar second
- Radio and satellite servers provide periodically UTC
- Electricity 50 (Eu.) resp. 60 Hz (USA) is based on UTC
  - Frequency raised to 51/61 at leap seconds
- Clock time (C) and UTC may differ, as clocks tick at different rates (*drift*)
  - $1-p \le dC/dt \le 1+p$
  - p: max. drift rate



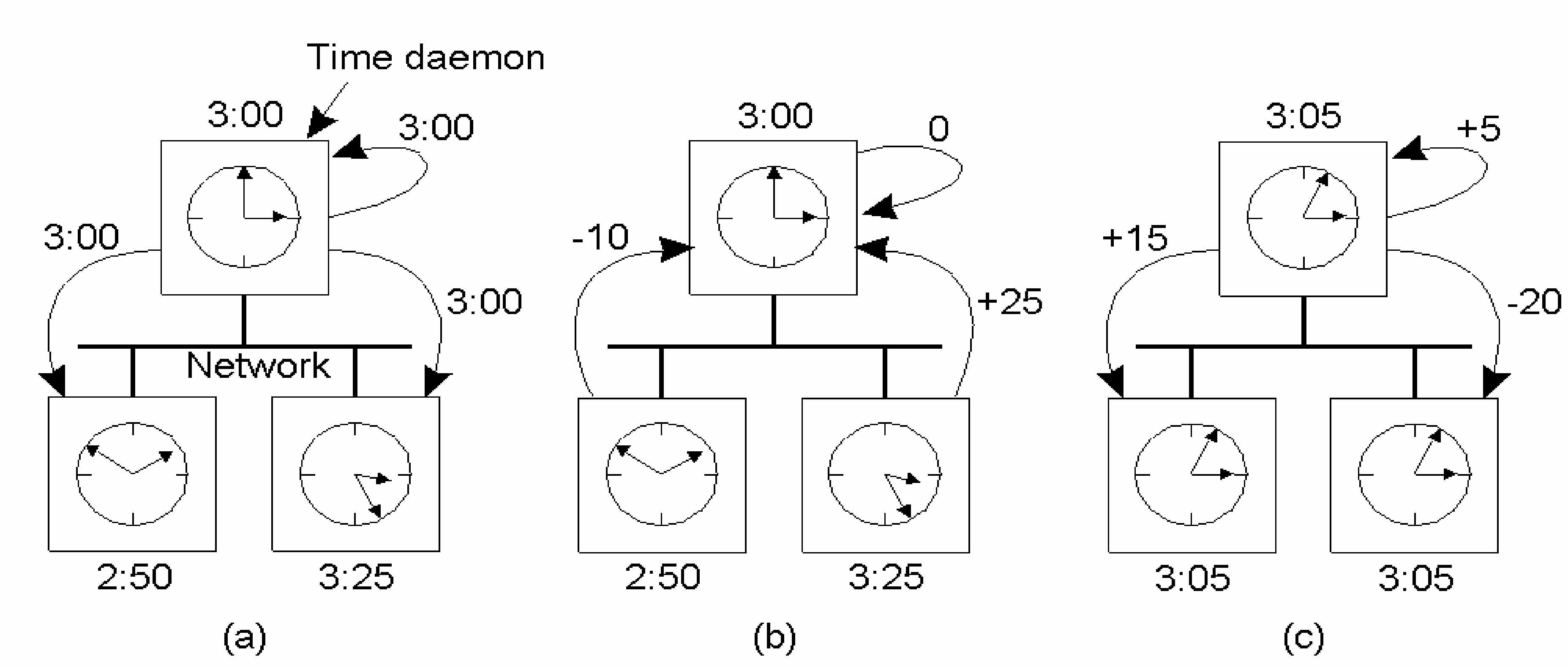


Cristian's Algorithm for Physical Clock Synch.

- Getting the current time from a *passive* time server
  - If $C_{UTC} > T_{client} \Rightarrow T_{client} := C_{UTC}$ (or speed up, if the difference is big)
  - If $C_{UTC} < T_{client} \Rightarrow$ the client's clock slows down (time must go forward)
  - The answer of the server costs time
    - The difference of two time values are still quite accurate
    - The average of a series of queries should be taken (disregard extreme values)



Berkeley Algorithm for Physical Clock Synch.



a) The *active* time daemon (TS) asks all the other machines for their clock
b) The machines answer
c) The TS computes average and tells everyone how to adjust their clock

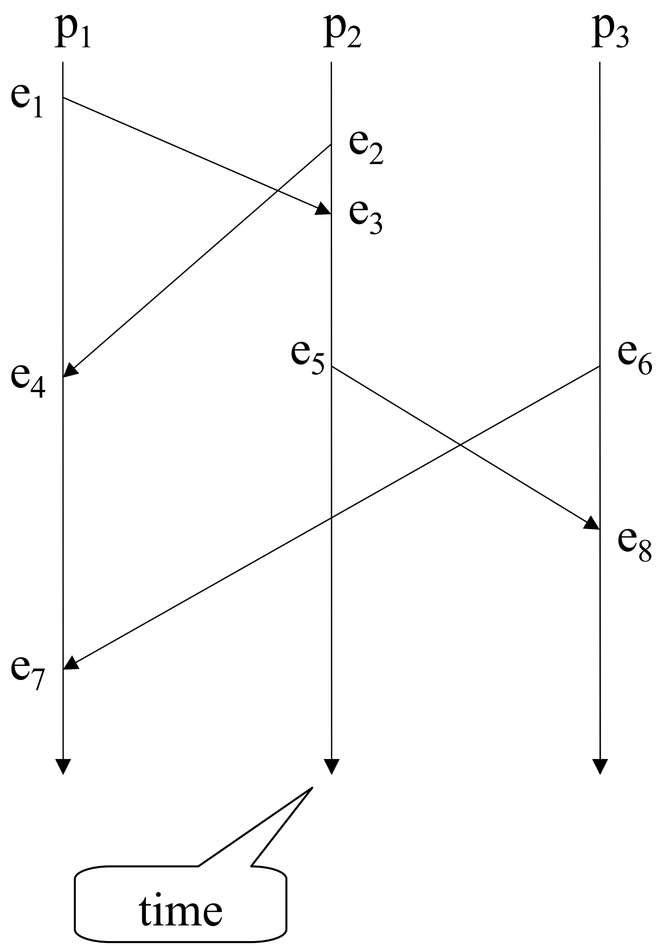
Happens-before (1)

- If event e_1 occurred before e_2, we write:
  - $e_1 < e_2$ (or $e_1 \rightarrow e_2$)
  - We say: e_1 *happened before* e_2
- If this is based on information X, we write:
  - $e_1 <_X e_2$
- Events of the same processor are *totally* ordered
  - If $e_1 \in E_p$ and $e_2 \in E_p$: either $e_1 <_p e_2$ or $e_2 <_p e_1$
- Sending of a message happens always before receiving it
  - If e_s is the sending of message *m* and e_r the receipt of *m*, then $e_s <_m e_r$

Happens-before (2)

- Happened-before relation: The transitive closure of the processor and message passing orderings:
  - If $e_1 <_p e_2$ then $e_1 <_H e_2$
  - If $e_1 <_m e_2$ then $e_1 <_H e_2$
  - If $e_1 <_H e_2$ and $e_2 <_H e_3$ then $e_1 <_H e_3$
- Causation
  - If e_1 happened before e_2 than e_1 *might have caused* e_2
- The happened-before relation is a *partial order* :
  - It is possible to have two events e_1 and e_2 that neither $e_1 <_H e_2$ nor $e_2 <_H e_1$
  - Such events are called *concurrent* (or disjoint)

Example Happens-before



$e_1 <_{p1} e_4 <_{p1} e_7$ ($\forall$ events on same proc.)

$e_2 <_{p2} e_3 <_{p2} e_5$ ($\forall$ events on same proc.)

...

$e_1 <_m e_3$ (send and receipt of same mess.)

$e_5 <_m e_8$ (send and receipt of same mess.)

...

$e_1 <_H e_3 <_H e_5 <_H e_8$, etc. $\Rightarrow$ $e_1 <_H e_8$ (transitivity)

$\{e_1, e_6\}$, $\{e_1 , e_2\}$ $\{e_2, e_6\}$, are concurrent

- Happens-before DAG (H-DAG)
  - The vertices are the events in *E*
  - The directed edge (e_1, e_2) is in the edge set, E_H iff (if and only if) $e_1 <_p e_2$ or $e_1 <_m e_2$.

Lamport Time Stamps (1)

- A *logical* global clock assigns a total order over all events
- The happened-before relation defines a partial order
  - Theoretically we can just apply topological sort on $<_H$
- Leslie Lamport's algorithm
  - creates total order "on the fly"
  - entirely distributed
  - fault tolerant
  - efficient
  - orders concurrent events arbitrarily

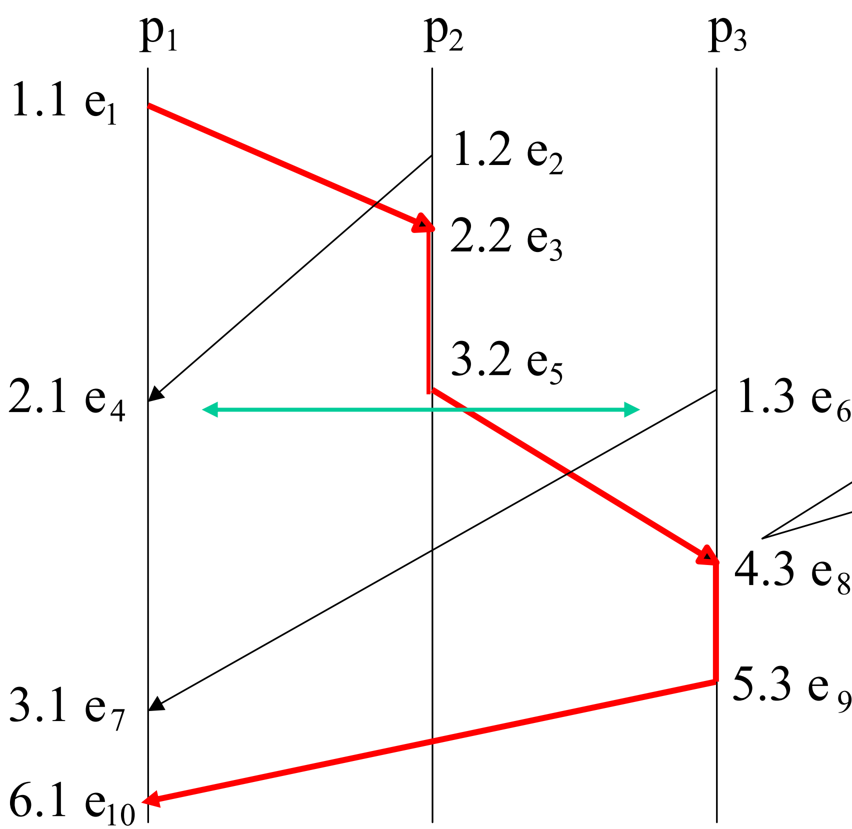
  - Each event *e* has a timestamp *e.TS*
  - Each processor maintains a local timestamp *my_TS*
  - Processor address (or number) is used for the lowest order bits of the timestamp (to avoid identical stamps in different processors)
  - Each event and each message get a timestamp assigned as:

Lamport Time Stamps (2)

```
my_TS = 0;                      // inital assignment
On event e,
        if e is the receipt of message m,
                my_TS = max(m.TS, my_TS)
        //local stamp “jumps” forward, if m is in the “future”
        my_TS++
        e.TS = my_TS
        if e is the sending of message m,
                m.TS = my_TS
```

- A logical global clock assigns a total order over all events

Example Lamport Time Stamps



Total order on all events:
1.1, 1.2, 1.3, ↔ 2.1, 2.2,
3.1, 3.2, 4.3, 5.3, 6.1...

e_8 must be labeled 4.3, (not 2.3!), because it follows event 3.2

Events along *happens before* build a sequence, e.g.
1.1, 2.2, 3.2, 4.3, 5.3, 6.1...

Time Stamp Implementation (1)

```
import java.io.*;
public class TimeStamp implements Serializable {
    // A time stamp object cannot be changed
    private final int time;
    private final int host;

    TimeStamp (int timeStamp, int hostNum) {
                time = timeStamp; host = hostNum; }

    public int getTime() { return time; }
    public int getHost() { return host; }
    public String toString () { return time + "." + host; }

} // TimeStamp
```

Time Stamp Implementation (2)

```
import java.util.*;
public class Lamport {
    static private int time = new Random().nextInt(100);
    static private int host = MyHost.NameToNum(MyHost.Name());
    static private TimeStamp lastReceived = new TimeStamp(time, host);

    public static synchronized void Adapt (TimeStamp received) {
        lastReceived = received;
        if (time < lastReceived.getTime())
            time = lastReceived.getTime();
    } // Adapt

    public static synchronized TimeStamp Next () {
        return new TimeStamp(++time, host);
    } // Next
} // Lamport
```

Time Stamp Producer

```
class Producer extends Thread  {
   private MessagePassing buf = null;
   private String target; private boolean stopped = false;
   Producer (MessagePassing sendBuf, String targetName)  {
       buf = sendBuf; target = targetName;}
   public void stopp () {stopped = true;}
   public void run ()  {     // produces and sends time stamps in a loop
       while (!stopped)  {
          TimeStamp nextTS = Lamport.Next();          // Sets a time stamp
          System.out.println("sends= " +  nextTS.toString());
          try { buf.send(nextTS); }
          catch (java.rmi.RemoteException e)
               {System.out.println("send-error" + e); }
          try { Thread.sleep(1000);}
          catch (java.lang.InterruptedException e) { }
}   }    } // while, run, Producer
```

Time Stamp Consumer

```
class Consumer extends Thread  {
   private MessagePassing buf = null;
   private boolean stopped = false;
   Consumer (MessagePassing recBuf)  {buf = recBuf;}
   public void stopp () {stopped = true;}
   public void run ()  {     // receives time stamps in a loop
       while (!stopped)  {
          TimeStamp receivedTS = null;
          try  { receivedTS = (TimeStamp) buf.receive();
               Lamport.Adapt(receivedTS);
               System.out.println("rcvd= "+receivedTS.toString());
          }  catch (java.rmi.RemoteException e)
               {System.out.println("receive-error" + e); }
          try { Thread.sleep(1000);}
          catch (java.lang.InterruptedException e) { }
}  }    } // while, run, Consumer
```

Time Stamp Execution

Connection established to
rmi://lpc1/MessagePassing
Connection established to
rmi://lpc2/MessagePassing
Connection established to
rmi://lpc3/MessagePassing
sends = 47.1
sends = 48.1
sends = 49.1
sends = 50.1
received = 74.2
sends = 75.1
received = 75.2
sends = 76.1
received = 76.2
sends = 77.1
received = 78.3
received = 77.2
received = 80.3
received = 82.3
received = 85.3
sends = 86.1
received = 79.2
sends = 87.1
received = 87.3
received = 88.3
sends = 89.1

Connection established to
rmi://lpc2/MessagePassing
Connection established to
rmi://lpc3/MessagePassing
Connection established to
rmi://lpc1/MessagePassing
sends = 73.2
sends = 74.2
received = 47.1
sends = 75.2
received = 49.1
sends = 76.2
received = 75.1
sends = 77.2
received = 76.1
sends = 78.2
sends = 79.2
received = 79.3
sends = 80.2
received = 81.3
sends = 82.2
received = 83.3
sends = 84.2
received = 84.3
sends = 85.2
sends = 86.2
received = 77.1

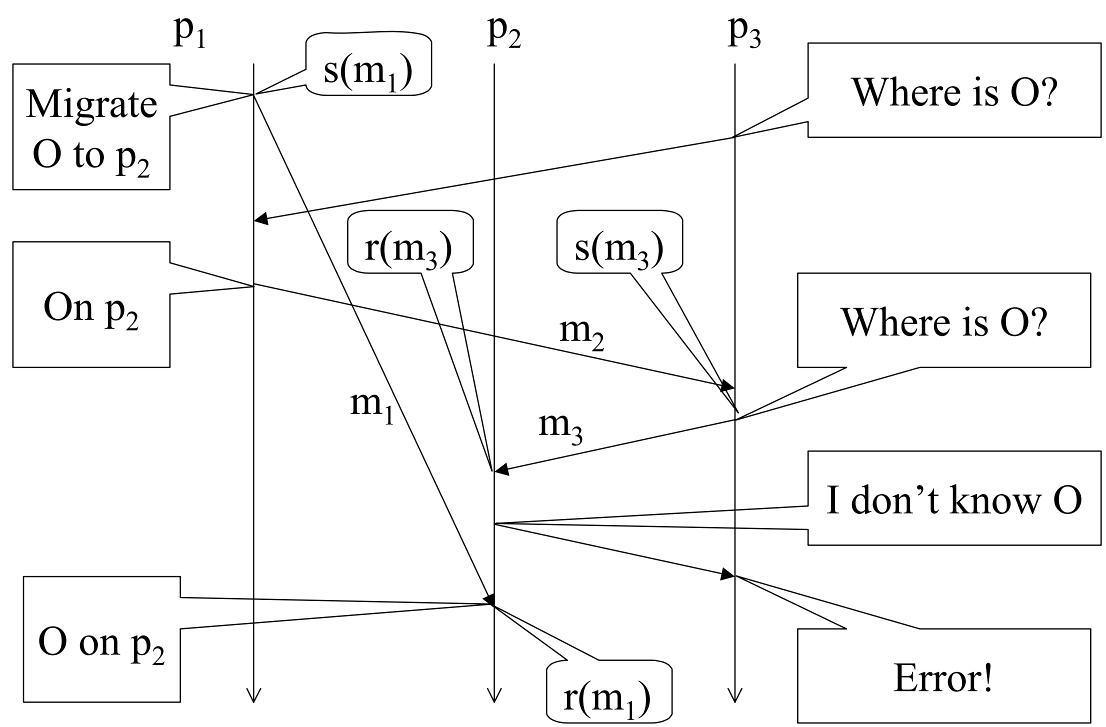
Process on lpc3 starts
Connection established to
rmi://lpc3/MessagePassing
Connection established to
rmi://lpc1/MessagePassing
Connection established to
rmi://lpc2/MessagePassing
sends = 78.3
sends = 79.3
received = 48.1
sends = 80.3
received = 73.2
sends = 81.3
received = 78.2
sends = 82.3
sends = 83.3
received = 80.2
sends = 84.3
sends = 85.3
received = 82.2
sends = 86.3
sends = 87.3
received = 84.2
received = 85.2
sends = 88.3
received = 87.2
sends = 89.3

Vector Time Stamps (1)

- Lamport algorithm guarantees:
  - If e_1 happens before e_2 then it has a smaller time stamp:
  - $e_1 <_H e_2 \Rightarrow$ e1.TS < e2.TS
- It does *not* guarantee:
  - If e_1 has a smaller time stamp than e_2 then it happened before:
  - e1.TS < e2.TS $\not\Rightarrow e_1 <_H e_2$
  - Because: concurrent events are ordered arbitrarily
- We often need the causality relation
  - E.g. to test *causality violation*:
  - the effect arrives earlier than the (potential) cause

Vector Time Stamps (2)

- Example (object *O* can migrate between processors)
- The request of p_3 causally follows the transfer of *O* from p_1 to p_2, but is processed before the transfer at p_2
- $s(m_1) <_H s(m_3)$ but $r(m_3) <_{p2} r(m_1)$



Vector Time Stamps (3)

- Let *s(m)* be the sending and *r(m)* the receipt of message *m*
- m_1 causally precedes m_2 ($m_1 <_c m_2$) if $s(m_1) <_H s(m_2)$
- *Causality violation*: $m_1 <_c m_2$, but $r(m_2) <_p r(m_1)$:
  - Message m_1 is sent before m_2, but m_2 received on p before m_1
- To *detect* causality violation, a timestamp *VT* is needed with comparison function $<_V$ such that $e_1 <_H e_2$ iff $e_1.VT <_V e_2.VT$
  - $<_V$ must be a partial order (since $<_H$ is)
  - e.VT must contain information about the other processors
  - We need a vector of integers of size *N* (no. of processors)
  - If e.VT[i] = k then *e* causally follows the first *k* events of processor_i (per definition, an event follows itself)
  - $e_1.VT \leq_V e_2.VT$ iff e_2 follows every event that e_1 follows
    - $e_1.VT \leq_{VT} e_2.VT$ iff $e_1.VT[i] \leq e_2.VT[i] \quad \forall\, i = 1, \ldots, N$
    - $e_1.VT <_V e_2.VT$ iff $e_1.VT \leq_{VT} e_2.VT$ and $e_1.VT \# e_2.VT$

Vector Time Stamps (4)

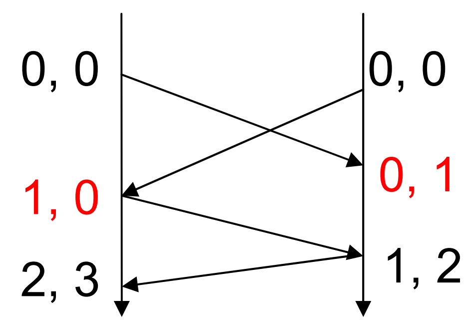
```
my_VT = [0, ..., 0];            // inital assignment
  On event e,
      if e is the receipt of message m,
            for i = 1 to N
                  my_VT[i] = max(m.VT[i], my_VT[i])
            // VT[i] "jumps" forward, if m is in the "future"
      my_VT[self]++
      e.VT = my_VT
      if e is the sending of message m,
            m.VT = my_VT
```

Vector Time Stamps (5)

- Proof
  - $e_1 <_H e_2 \Rightarrow e_1.VT <_{VT} e_2.VT$, because the algorithm ensures that for every event $e_1 <_p e_2$ or $e_1 <_m e_2$:

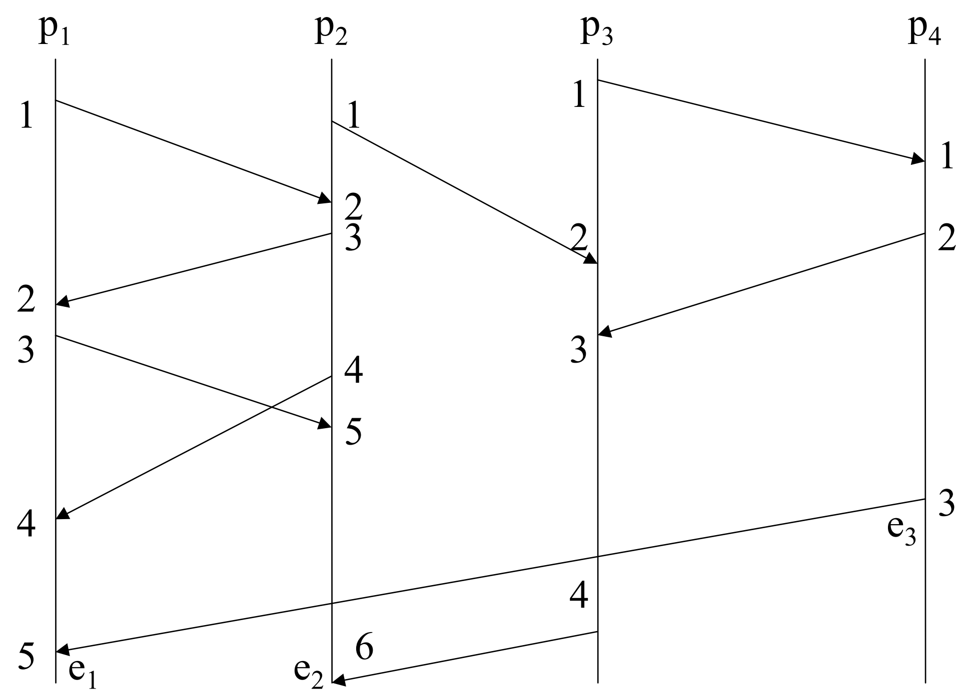
    $e_1.VT <_{VT} e_2.VT$ (very similar to Lamport's algorithm)
  - Suppose: $e_1 \neg<_H e_2$. Is to show: $e_1.VT \neg<_{VT} e_2.VT$
    - Suppose e_2 is the k^{th} event on processor p and that $e_1.VT[p] = j$, $j > k$
    - Then, there must be a path in the H-DAG from the j^{th} event on processor p to event e_1
    - So, if $e_1 \neg<_H e_2$ then $e_1.VT \neg<_{VT} e_2.VT$
    - If e_1 and e_2 are concurrent then

      $e_1.VT \neg<_{VT} e_2.VT$ and $e_2.VT \neg<_{VT} e_1.VT$



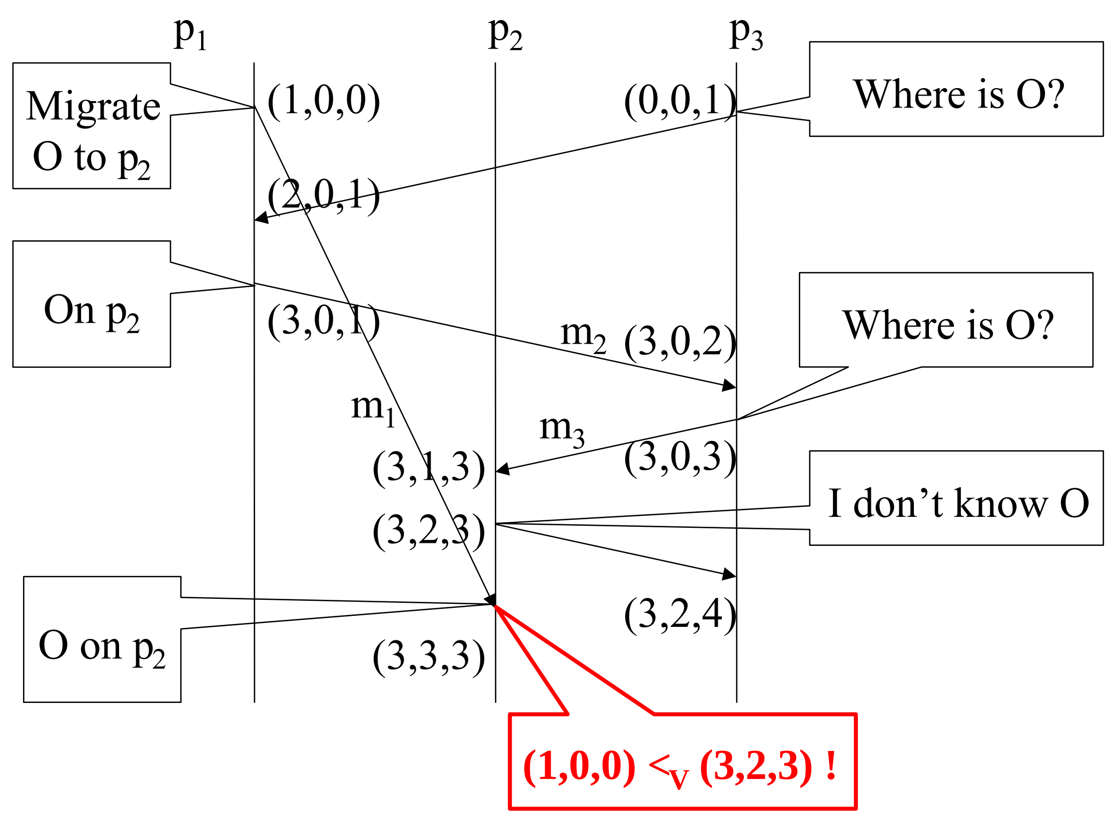
Vector Time Stamps (6)

- $e_1.VT = (5, 4, 1, 3)$
- $e_2.VT = (3, 6, 4, 2)$
- $e_3.VT = (0, 0, 1, 3)$
- e_1 and e_2 are concurrent (no path $e_1 \rightarrow e_2$ or $e_2 \rightarrow e_1$)
  - $e_1.VT[1] > e_2.VT[1]$, but $e_1.VT[2] < e_2.VT[2]$
- $e_3 <_H e_1$ (e_1 follows e_3):
  - $e_3.VT <_V e_1.VT$



Vector Time Stamps (7)

- Example (object *O* can migrate between processors)
- We still do not avoid causality violation, but we detect it
- To avoid it, we could e.g. buffer all messages that are not in order



Vector Stamp Implementation (1)

```
public class VectorStamp { // Implements vector stamp algorithm
  public static final int Nodes = 3;
  private static int myHost = MyHost.NameToNum(MyHost.Name());
  public static int [] myVector = Create(Nodes);
  public static int [] receivedVector = Create(Nodes);
  public static int [] Create (int N)  {
      int [] tsVector = new int [N+1];
      for (int i = 1; i <= N; i++) tsVector[i] = 0;
      tsVector[0] = myHost;                    // help information for traces
      return tsVector;
  } // Create
  public static void  Adapt ()   {
      for (int i = 1; i < myVector.length; i++)
              if (receivedVector[i] > myVector[i])
                  myVector[i] = receivedVector[i];
  } // Adapt

 public static void  Next ()   { myVector[myHost]++;  }
```

Vector Stamp Implementation (2)

```
public static boolean CausalError ()   {return Less(receivedVector, myVector); }

 public static boolean Equals (int [] v1, int [] v2)   {
        for (int i = 1; i < v1.length; i++) if (v1[i] != v2[i]) return false;
        return true;
 } // Equals
 public static boolean Less (int [] v1, int [] v2)   {
        if (Equals (v1, v2)) return false;
        for (int i = 1; i < v1.length; i++) if (v1[i] > v2[i]) return false;
        return true;
 } // Less
 public static String ToString (int [] vector)  {
        String t = "(";            for (int i = 1; i < (vector.length-1); i++) t = t + vector[i] + ",";
        t = t + vector[vector.length-1] + ")";
        return t;
 } // ToString

} // VectorStamp
```

Vector Stamp Producer

```
class Producer extends Thread  {
    private MessagePassing buf = null;
    private boolean stopped = false;
    private String target = null;
    Producer (MessagePassing sendBuf, String targetName)
        {  buf = sendBuf; target = targetName; }

    public void stopp () { stopped = true; }
    public void run ()  {
        while (!stopped)  {
            VectorStamp.Next();
            System.out.println("sends to " + target + " = " +
                                    VectorStamp.ToString(VectorStamp.myVector));
            try { buf.send(VectorStamp.myVector); }
            catch (java.rmi.RemoteException e)  {System.out.println("send-error" + e);}
            try { Thread.sleep(1000); } catch (java.lang.InterruptedException e) { }
}   }    } // while, run, Producer
```

Time Stamp Consumer

```
class Consumer extends Thread  {
   private MessagePassing buf = null; private boolean stopped = false;
   Consumer (MessagePassing recBuf) { buf = recBuf;}
   public void stopp () {stopped = true;}
   public void run ()  {
       while (!stopped)  {
          try  {      VectorStamp.receivedVector = (int []) buf.receive();
               System.out.println("received from " +
                    MyHost.NumToName(VectorStamp.receivedVector[0])     +
                         " = " + VectorStamp.ToString(VectorStamp.receivedVector));
               if (VectorStamp.CausalError ())
                    System.out.println("Causality Violation, my v. =" +
                              VectorStamp.ToString(VectorStamp.myVector));
               VectorStamp.Adapt (); VectorStamp.Next();
          }  catch (java.rmi.RemoteException e) {System.out.println("receive-err" + e); }
          try { Thread.sleep(1000);}  catch (java.lang.InterruptedException e) { }
}    }    } // while, run, Consumer
```

Causality Violation

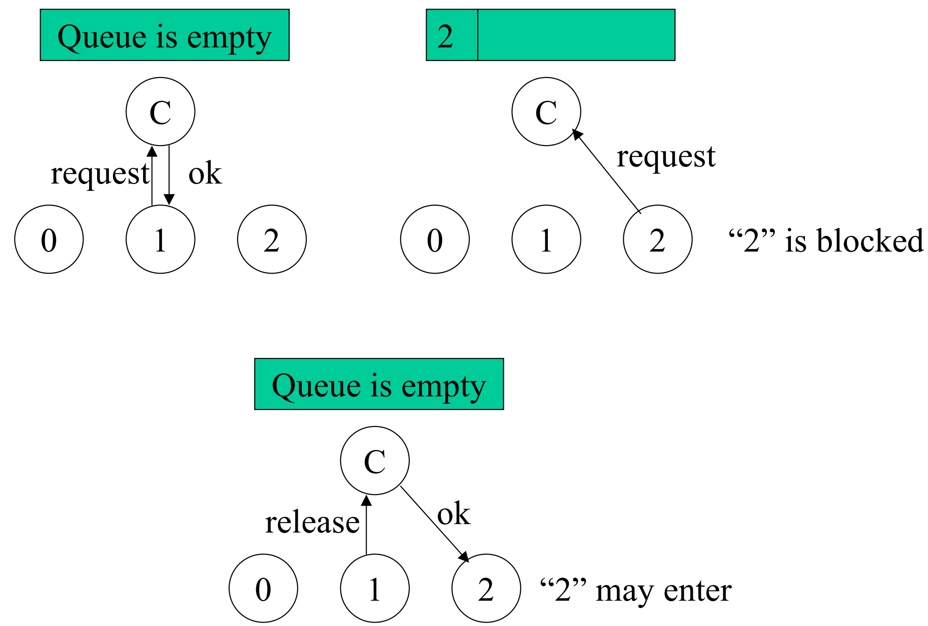
to lpc2 = (1,0,0)
to lpc3 = (2,0,0)
to lpc2 = (3,0,0)
to lpc3 = (4,0,0)
from lpc2 = (0,2,0)
from lpc2 = (1,4,0)
to lpc2 = (7,4,0)
from lpc2 = (3,6,0)
to lpc2 = (9,6,0)
from lpc3 = (0,0,1)
to lpc2 = (11,6,1)
from lpc2 = (7,8,0)
from lpc3 = (2,0,4)
from lpc3 = (9,10,8)
from lpc3 = (9,13,11)
from lpc3 = (9,15,14)
…
from lpc3 = (9,35,44)
to lpc2 = (27,35,44)
from lpc3 = (9,37,47)
from lpc3 = (9,39,49)
to lpc2 = (30,39,49)
from lpc3 = (11,41,52)
to lpc2 = (32,41,52)
from lpc2 = (9,11,0)
Causality Violation,
my vector = (32,41,52)

to lpc3 = (0,1,0)
to lpc1 = (0,2,0)
from lpc1 = (1,0,0)
to lpc1 = (1,4,0)
from lpc1 = (3,0,0)
to lpc1 = (3,6,0)
from lpc1 = (7,4,0)
to lpc1 = (7,8,0)
from lpc1 = (9,6,0)
to lpc3 = (9,10,0)
to lpc1 = (9,11,0)
from lpc3 = (0,0,2)
to lpc3 = (9,13,2)
from lpc3 = (2,1,6)
to lpc3 = (9,15,6)
from lpc3 = (9,10,9)
to lpc3 = (9,17,9)
from lpc3 = (9,13,12)
to lpc3 = (9,19,12)
from lpc3 = (9,15,15)
to lpc3 = (9,21,15)
from lpc3 = (9,17,18)
to lpc3 = (9,23,18)
…

to lpc1 = (0,0,1)
to lpc2 = (0,0,2)
from lpc1 = (2,0,0)
to lpc1 = (2,0,4)
from lpc2 = (0,1,0)
to lpc2 = (2,1,6)
from lpc2 = (9,10,0)
to lpc1 = (9,10,8)
to lpc2 = (9,10,9)
from lpc2 = (9,13,2)
to lpc1 = (9,13,11)
to lpc2 = (9,13,12)
from lpc2 = (9,15,6)
to lpc1 = (9,15,14)
to lpc2 = (9,15,15)
from lpc2 = (9,17,9)
to lpc1 = (9,17,17)
to lpc2 = (9,17,18)
from lpc2 = (9,19,12)
to lpc1 = (9,19,20)
to lpc2 = (9,19,21)
from lpc2 = (9,21,15)
to lpc1 = (9,21,23)
…

Distributed Mutual Exclusion (DME)

- Centralized algorithm
  - Simulates the one-processor algorithm
  - One process is elected (see later) as coordinator
  - If a process wants to enter the critical section it sends a message to the coordinator
  - If the critical section is free the coordinator sends a grant
  - If it is busy it may send a reject or simply block the sender in a FIFO queue and delays the grant until it may enter



Distributed DME Algorithm (1)

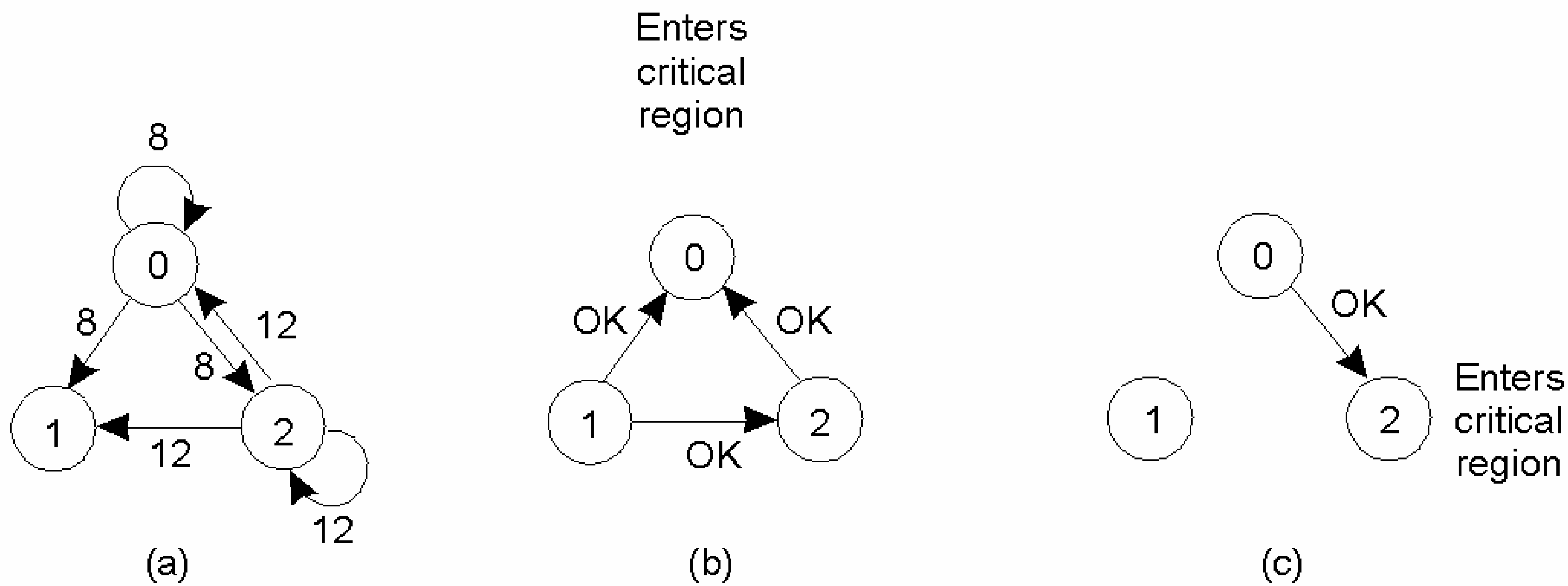
- Distributed Algorithm with Timestamp
  - Basic idea: the oldest requester wins (a “polite” protocol)
  - We assume that the communication is free of failures
  - All requests get a Lamport timestamp. As Lamport timestamps define a total order, it is always possible to agree which is the oldest request (lowest timestamp)
  - If a processor needs to enter a critical section it sends a request to all other processors
  - If a processor receives a request than it answers with its own timestamp, or with “youngest”, if it has no need for a CS
  - In this way, all processors can create the same priority queue
  - The processor, finding itself on the top (oldest request, smallest timestamp), may enter
  - When a processor exits the CS, it informs all others

Distributed DME Algorithm (2)

- The timestamp based algorithm is inefficient, because
  - We send more messages than necessary: 3(N – 1)
    - "Denial" messages (with higher timestamp) are waste, as the other processor must wait anyway
    - Even without denial messages: 2(N – 1) messages
  - We delay more than necessary
- With further improvements, it still remains in the O(N) message complexity class
- The centralized algorithm is more efficient and even more fault tolerant:

Algorithm	Messages per entry/exit	Min. delay bef. entry	Problems
Centralized	3	2 message time	Coordinator crash
Distributed	2 (N – 1)	2 (N – 1) m.t.	Any crash

Ricart & Agrawala DME Alg. (1)



a) Two processes (0 and 2) want to enter the same critical region at the same moment
b) Process 0 has the lowest timestamp, so it wins
c) When process 0 is done, it sends an OK also, so 2 can now enter the critical region

Ricart & Agrawala DME Alg. (2)

Data structures+algorithms are *in every node the same:* ***symmetric***

timestamp	current_time	Current Lamport time
timestamp	my_timestamp	
integer	reply_pending	No. of pending permissions
boolean	is_requesting	True if CS requested or used
boolean	reply_defferred[N]	True for younger requests

Request_CS()

```
my_timestamp = current_time
is_requesting = TRUE
reply_pending = N – 1
for every other processor j,
   send (j, REMOTE_REQUEST; my_timestamp)
wait until reply_pending = 0
```

Ricart & Agrawala DME Alg. (3)

```
Release_CS()
    is_requesting = FALSE
    for j = 1 to N
        if reply_defferred[j] = TRUE
            send(j, REPLY); reply_defferred[j] = FALSE
CS.Monitor()
    Wait until REMOTE_REQUEST or REPLY arrives
        REMOTE_REQUEST(sender; request_timestamp):
            if (not is_requesting or  my_timestamp > request_timestamp)
                send(sender, REPLY)
            else
                reply_defferred[sender] = TRUE
        REPLY (sender)
            reply_pending--
```

Naive Voting Algorithm for DME (1)

Basic idea: it is enough to have a majority of votes.

- If a process wants to enter a critical section it sends a request to all other processes
- If a processor gets a request and it does not want to enter then it sends a grant
- If a processor gets at least $\lceil (N + 1) / 2 \rceil$ (e.g. 3 of 4 or 5) votes then it may enter – no other processor may get as many votes
- If a processor leaves the CS it releases its vote
- Advantage
  - Much more fault tolerant than the timestamp algorithm
    - It tolerates that even half of the processors fail – except the lock holder

Naive Voting Algorithm for DME (2)

- Problems
  - The algorithm tends to deadlock
    - E.g. each of 3 processors get 1/3 of the votes
  - It is not substantially more efficient than the timestamp algorithm – still O(N)
- Improvements of the basic idea
  - Not all messages are the same important
    - E.g., two candidates are competing for $N = 2n + 1$ votes and both have already received n:
    - The last message decides
  - We may try to concentrate on "*swing*" *voters*

Maekawa's voting algorithm (1)

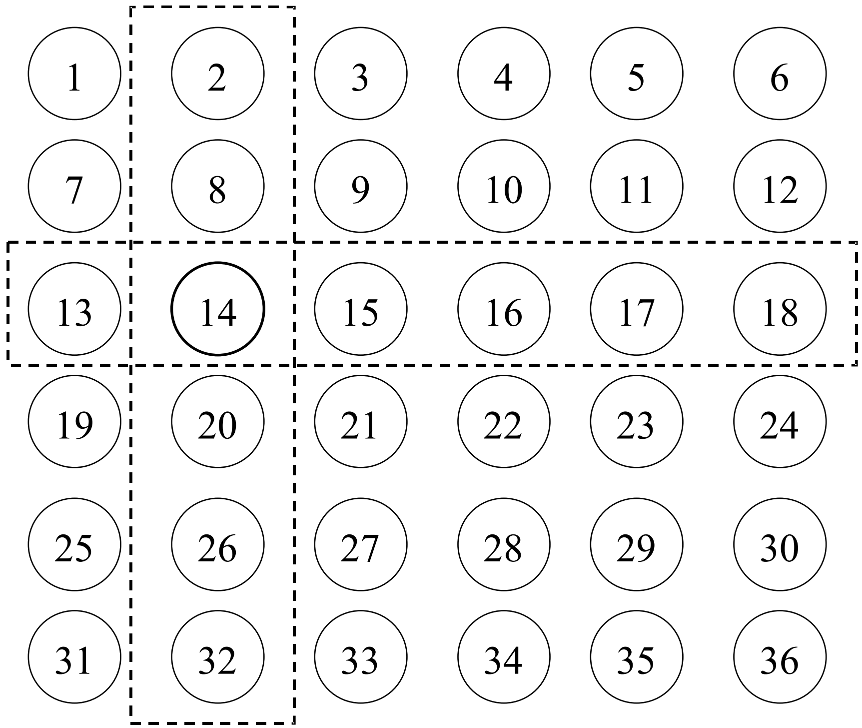
- Every processor *p* has a *voting district* $S_p \in \{S_1 , \ldots , S_N\}$
- The set $\{S_1 , \ldots , S_N\}$ is a *coterie*
- We assume that
  - S_p is fixed (some algorithms can handle dynamic districts)
  - Processor p must acquire the votes of all processors in S_p
- The districts must *not* be distinct ("swing voters"):

  $S_i \cap S_j \neq \emptyset, \quad \forall\, i, j: 1 \leq i, j \leq N$
- To be fair, the following should hold:
  - every voting district be about the same size ($|S_i| = K$)
  - every processor be ca. in the same number of districts (D)
- The smaller D and K are, the more efficient is the algorithm
- Let $N = n^2$ and label the processors (i, j) for $1 \leq i, j \leq N$
- Voting district for $p_{r,s}$ is: r^{th} row and s^{th} column
- $K = O(2\sqrt{N})$ – this is good, if N is fairly large

Maekawa's voting algorithm (2)

- The algorithm is similar to the naive algorithm
  - When a processor wants to enter a critical section, it sends a request to all members of its district
  - It may enter, if it gets a grant from all members
  - When a processor receives a request it answers with yes, if it has not already cast its vote
  - On exit it informs its district to enable a new voting
- Still deadlock danger – two polls may block each other
  - Assign each request a timestamp (Sanders)
  - The voters prefer the earliest candidate
    - Actually, we use ordering as deadlock prevention
  - If a processor V gave its vote for a processor B and then a processor C asks for V's vote with an earlier timestamp:
    - V tries to retrieve its vote by an inquire message
    - If it succeeds: C will win, if not: B has already won
    - One candidate can enter in any case → no deadlock

Maekawa's voting algorithm (3)



- A voting district consists of $2\sqrt{36} - 1 = 11$ processors ($n = 6$, $N = 36$)
- If processor p_{14} enters the critical section then no other processor may enter
- If e.g. $processor_6$ tries to enter, it will not get its vote from $processor_2$ and $processor_{18}$

Election

- We often need a *coordinator*
  - for centralized mutual exclusion
  - for a holder of the primary copy of replicated data
- The coordinator and the other *participants* form a *group*
- Election is similar to synchronization
  - Processors must come to an agreement
- Election is different from synchronization
  - All participants must know who is the leader
  - Fault tolerance is a central issue

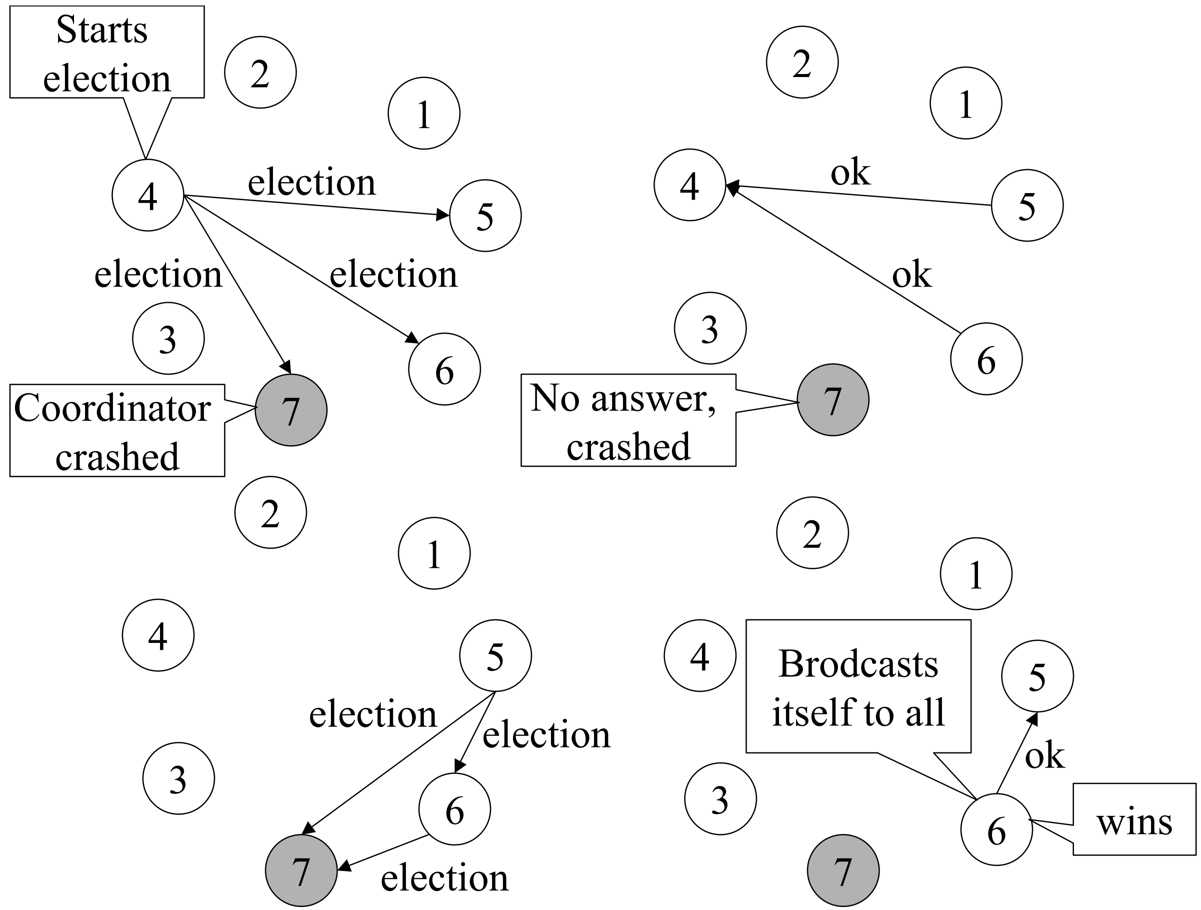
The Bully Algorithm for Election (1)

- Basic assumptions
  - Delivery time < T_m (all messages are delivered within T_m)
  - Message handling time < T_p (all nodes respond within T_p)
  - Reliable failure detection
    - If a node does not respond within $T = 2T_m + T_p$, it must have failed
    - All processors are able to detect a failure
    - The failed processor knows upon recovery that it failed
- A distributed system with such constraints is called *synchronous*
- Problem: Assumptions may not hold always
  - E.g. buffer overflows, temporary overloads etc.
  - The algorithm works in such a case incorrectly (e.g. 2 coordinators)
  - Or we have to work with too large time-outs
- Basic idea of the bully algorithm (Garcia-Molina)
  - The strongest processor (bull) wins
  - The processor with the highest number is the coordinator

The Bully Algorithm for Election (2)

Example

- The actual coordinator (process$_7$) crashes
- Process$_4$ notices this and starts an election
- If a process gets an *election* it sends an *ok* to the weaker ones and an *election* to the stronger ones
- If a process gets an *ok* then it knows that there is a stronger one and exits the election
- At last there is one single processor that does not get any *ok*
- It broadcasts that he is the new boss



Formal Correctness of the Bully Algorithm

- **Definitions**
  - Status $\forall$ p_i: one of {Down, Election, Normal, Reorganization}
    - Reorganization: after election, but not yet normal
  - Coordinator: Variable, containing the elected coordinator
  - Definition: State information for the actual task
- **Correctness Assertions**
  - $\forall$ p_i, p_j in a consistent state, *G*:
  - $(Status_i \in \{Normal, Reorganization\}) \wedge (Status_j \in \{Normal, Reorganization\}) \Rightarrow Coordinator_i = Coordinator_j$
  - $(Status_i = Normal) \wedge (Status_j = Normal) \Rightarrow Definition_i = Definition_j$
- **Liveness Assertions**

  Let be *R* the set of unfailed nodes. Eventually must hold in any run:
  - $\exists$ $p_i \in R$, such that $State_i = Normal \wedge Coordinator_i = p_i$
  - $\forall$ $p_j \in R$ such that $State_j = Normal \wedge Coordinator_j = p_i$

The Invitation Algorithm (1)

- Invitation Algorithm (Garcia-Molina)
- We cannot make safe assumptions about the timing of the events: *asynchronous system*
- The coordinator function makes only sense in relation to a certain *group*
- A group is identified by a unique group number
- Basic idea of the invitation algorithm:
  - Instead of electing a new coordinator, form a new group under the leadership of the new coordinator

The Invitation Algorithm (2)

- **Correctness Assertions**
  - $\forall$ p_i, p_j in a consistent state, *G*:
  - $(Status_i \in \{Normal, Reorganization\}) \wedge (Status_j \in \{Normal, Reorganization\}) \wedge (Group_i = Group_j) \Rightarrow Coordinator_i = Coordinator_j$
  - $(Status_i = Normal) \wedge (Status_j = Normal) \wedge (Group_i = Group_j) \Rightarrow Definition_i = Definition_j$
  - These are easy to satisfy
    - If a process *p* establishes itself as a coordinator then it creates a new, unique group number
    - Next, it suggests to the others to join the new group, with *p* as coordinator
    - Those, who join, accept its suggestion

The Invitation Algorithm (3)

- **Liveness Assertions**
  - Let be *R* the maximal set of nodes that can communicate in consistent state G_0.
  - Starting at G_0, eventually must hold:
    - $\exists\ p_i \in R$, such that $State_i$ = Normal $\wedge$ $Coordinator_i = p_i$
    - $\forall\ p_j \in R$ (unfailed p_j), $State_j$ = Normal $\wedge$ $Coordinator_j = p_i$
  - These are difficult to satisfy
  - Competing coordinators may repeatedly "steal" participants from each other, which may never end
  - Such groups can be *merged* into a global group – based on *invitation*

The Invitation Algorithm (3)

- Processor p_1 executes an *invitation* (a merge procedure) for joining the group “led” by it
- Processor p_2 accepts immediately
- Processor p_3 was a coordinator itself, so forwards the invitation to p_4 and p_5
- All send an *accept* and enter the *Reorganization* state
- Processor p_1 sends a ready message, taking the participants into the *Normal* state
- The coordinator may execute an invitation periodically
- Coordinators may have different priority
- The algorithm does not rely on error-free time-out