Distributed Synchronization: outline

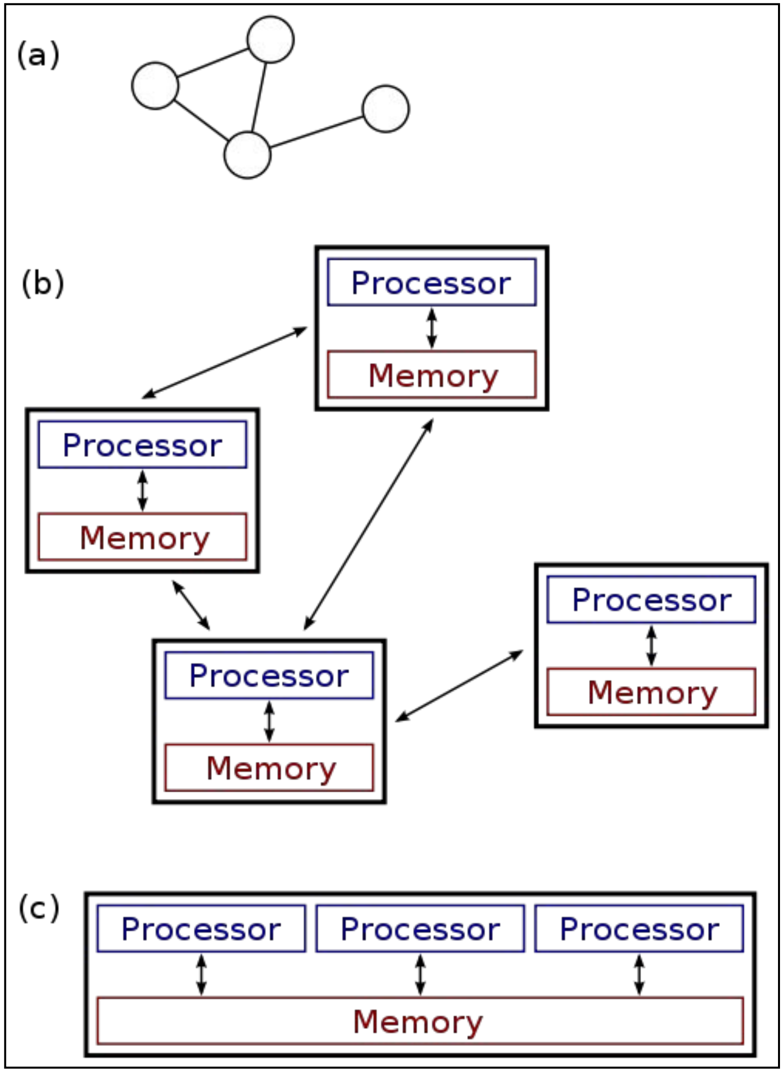
- Introduction
- Causality and time
  - Lamport timestamps
  - Vector timestamps
  - Causal communication
- Snapshots
- Distributed Mutual Exclusion

This presentation is based on the book: "Distributed operating-systems & algorithms" by Randy Chow and Theodore Johnson

Distributed systems

- A <u>distributed system</u> is a collection of independent computational nodes, communicating over a network, that is abstracted as a <u>single coherent system</u>
  - Grid computing
  - Cloud computing ("infrastructure as a service", "software as a service")
  - Peer-to-peer computing
  - Sensor networks
  - …

- A <u>distributed operating system</u> allows sharing of resources and coordination of distributed computation in a transparent manner



(a), (b) – a distributed system
(c) – a multiprocessor

Distributed synchronization

- Underlies distributed operating systems and algorithms
- Processes communicate via message passing (no shared memory)
- Inter-node coordination in distributed systems challenged by
  - Lack of global state
  - Lack of global clock
  - Communication links may fail
  - Messages may be delayed or lost
  - Nodes may fail
- Distributed synchronization supports correct coordination in distributed systems
  - May no longer use shared-memory based locks and semaphores

Distributed Synchronization: outline

- Introduction
- Causality and time
  - Lamport timestamps
  - Vector timestamps
  - Causal communication
- Snapshots
- Distributed Mutual Exclusion

Distributed computation model

- Events
  - Sending a message
  - Receiving a message
  - Timeout, internal interrupt
- Processors send control messages to each other
  - *send(destination, action; parameters)*
- Processes may declare that they are waiting for events:
  - *Wait for A_1, A_2, …., A_n*

    A_1(source; parameters)

    code to handle A_1

    .

    .

    A_n(source; parameters)

    code to handle A_n

Causality and events ordering

- A distributed system has no global state nor global clock
  $\Rightarrow$ no global order on all events may be determined

- <u>Each processor knows total orders on events occurring in it</u>

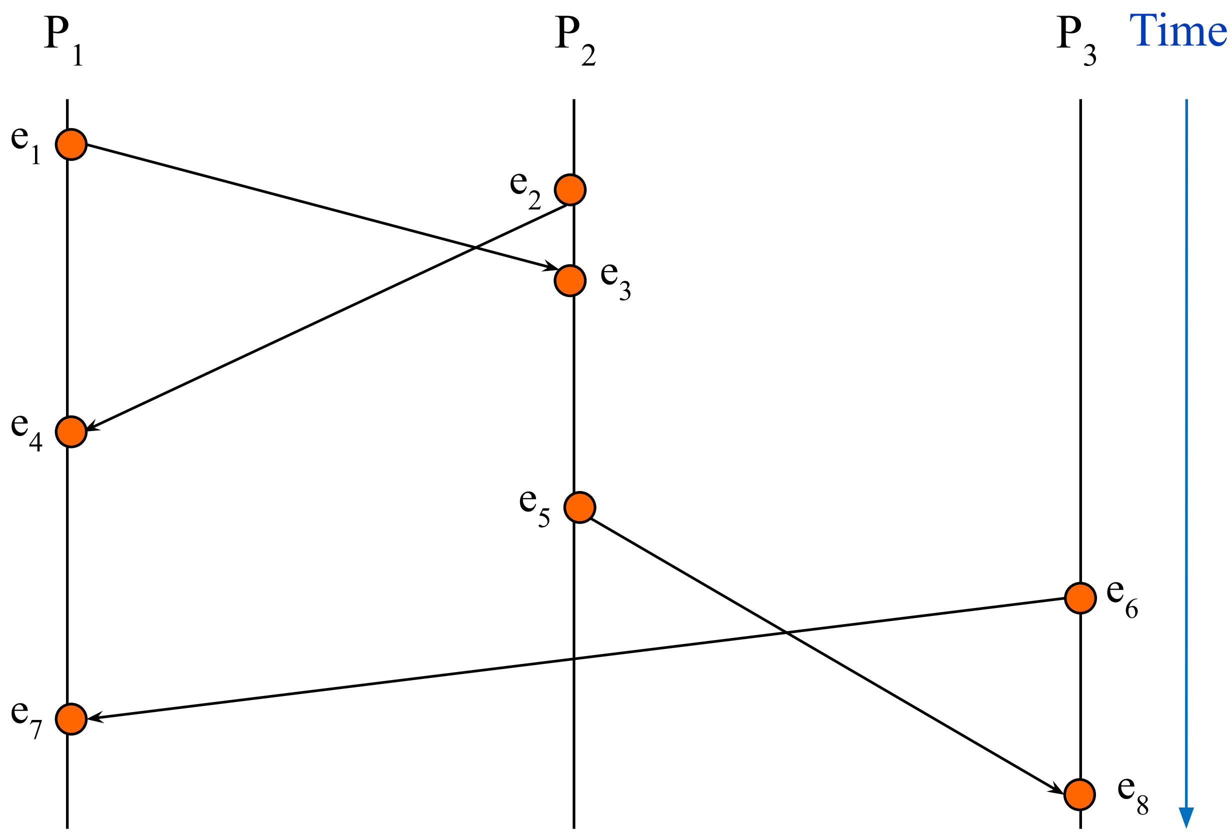
- <u>There is a causal relation between the sending of a message and its receipt</u>

<u>Lamport's *happened-before* relation H</u>

1. $e_1 <_p e_2 \Rightarrow e_1 <_H e_2$ (events within same processor are ordered)
2. $e_1 <_m e_2 \Rightarrow e_1 <_H e_2$ (each message m is sent before it is received)
3. $e_1 <_H e_2$ AND $e_2 <_H e_3 \Rightarrow e_1 <_H e_3$ (transitivity)

Leslie Lamport (1978): "Time, clocks, and the ordering of events in a distributed system"

Causality and events ordering (cont'd)



$e_1 <_H e_7$? Yes.

$e_1 <_H e_3$? Yes.

$e_1 <_H e_8$? Yes.

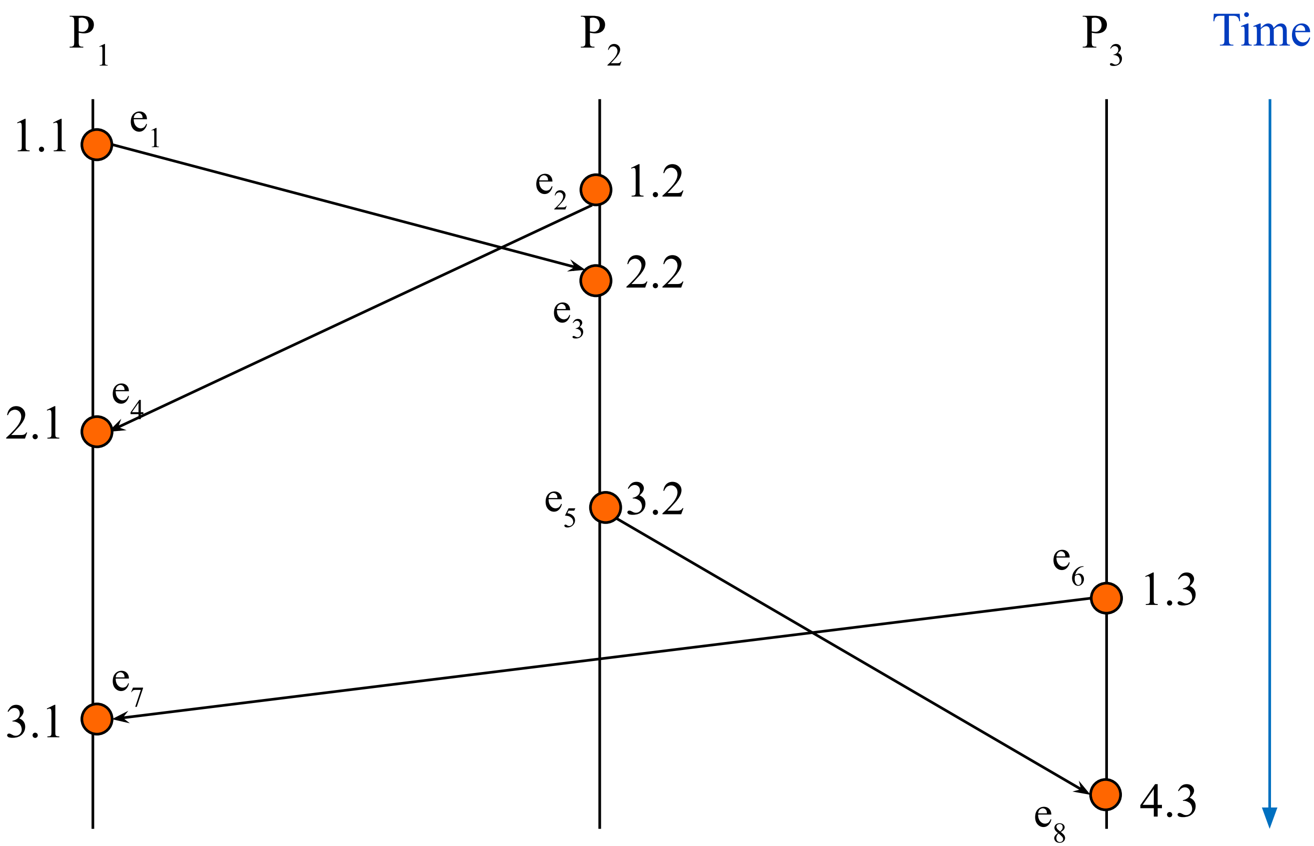
$e_5 <_H e_7$? No.

Lamport's timestamp algorithm

```
Initially my_TS=0

Upon event e,
   if e is the receipt of message m
      my_TS=max(m.TS, my_TS)
   my_TS++
   e.TS=my_TS
   If e is the sending of message m
      m.TS=my_TS
```

To create a total order, ties are broken by process ID

Lamport's timestamps (cont'd)



Distributed Synchronization: outline

- Introduction
- Causality and time
  - Lamport timestamps
  - Vector timestamps
  - Causal communication
- Snapshots
- Distributed Mutual Exclusion

Vector timestamps - motivation

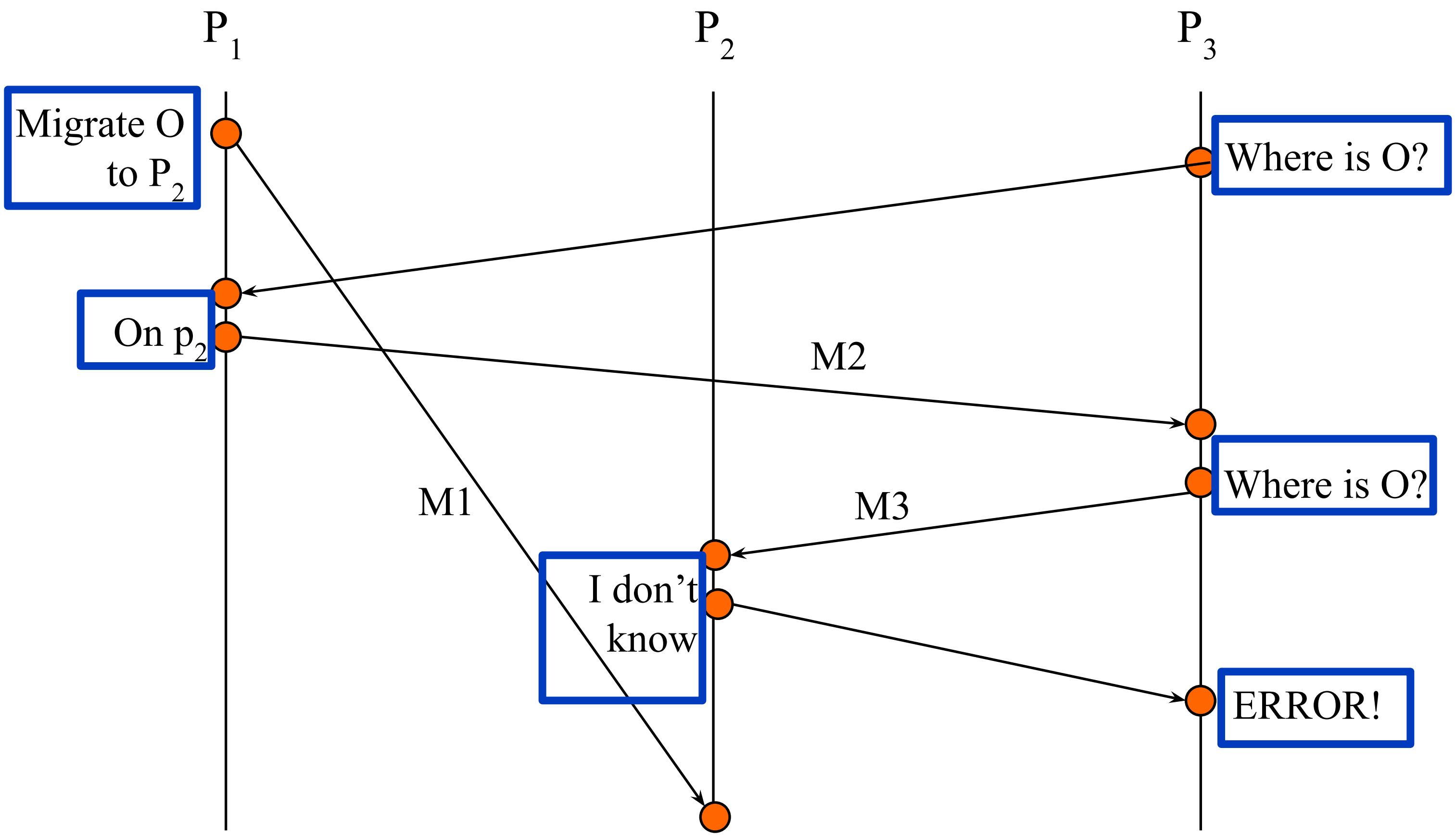
- Lamport's timestamps define a total order
  - $e_1 <_H e_2$ □ $e_1.TS < e_2.TS$
  - However, $e_1.TS < e_2.TS$ □ $e_1 <_H e_2$ does not hold, in general. (concurrent events ordered arbitrarily)

<u>Definition</u>: Message m_1 ***casually precedes*** message m_2 (written as $m_1 <_c m_2$) if $s(m_1) <_H s(m_2)$ *(sending* m_1 *happens before sending* m_2*)*

<u>Definition</u>: ***causality violation*** occurs if $m_1 <_c m_2$ *but* $r(m_2) <_p r(m_1)$.
In other words, m_1 is sent to processor p before m_2 but is received after it.

- Lamport's timestamps do not allow to detect (hence nor prevent) causality violations.

Causality violations – an example



Causality violation between…

M_1 and M_3.

Vector timestamps

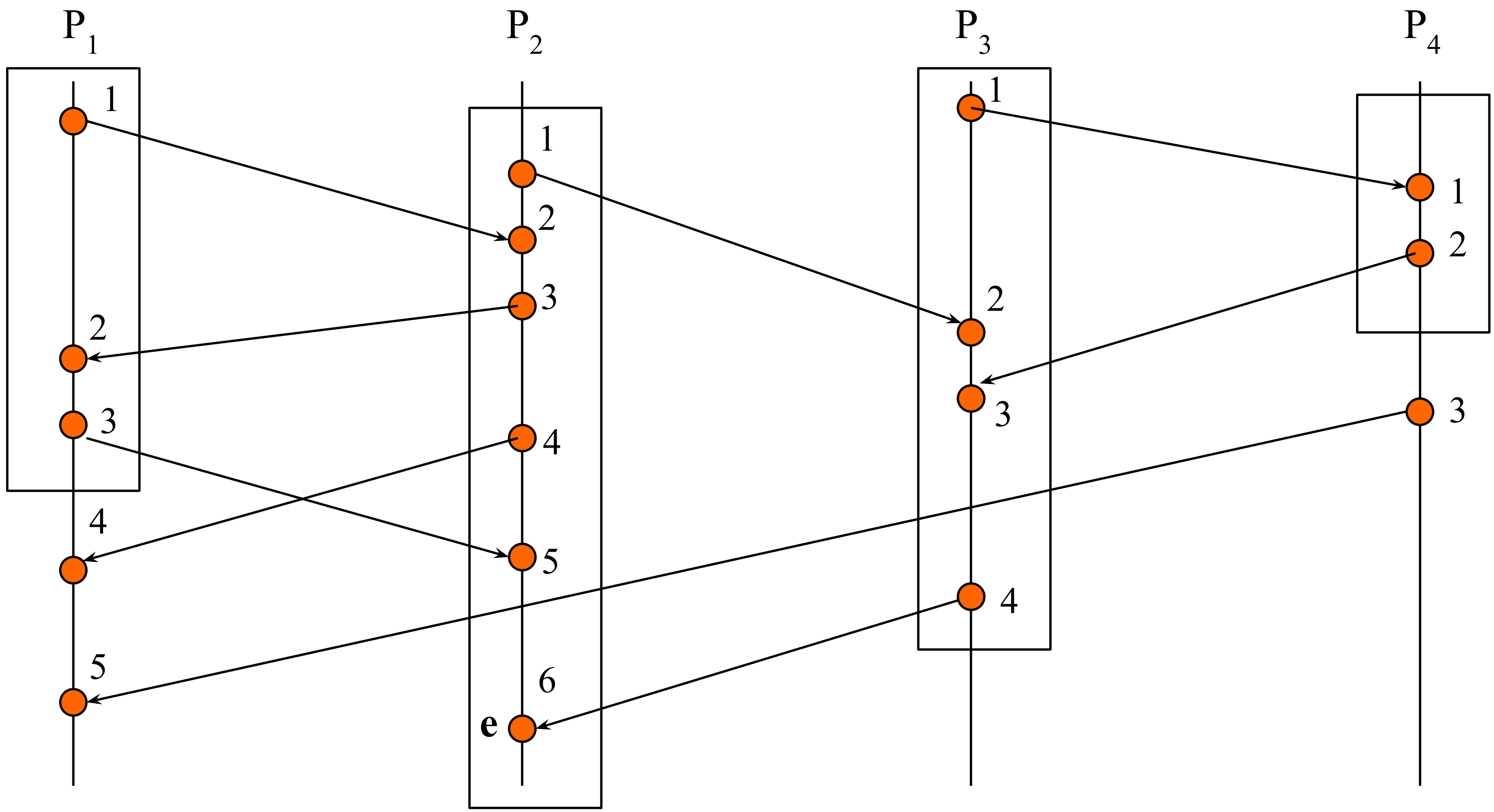
```
Initially my_VT=[0,…,0]

Upon event e,
  if e is the receipt of message m
    for i=1 to M
      my_VT[i]=max(m.VT[i], my_VT[i])
  My_VT[self]++
  e.VT=my_VT
  if e is the sending of message m
    m.VT=my_VT
```

$e_1.VT \leq_V e_2.VT$ *if and only if* $e_1.VT[i] \leq e_2.VT[i]$, *for every* $i=1,\ldots,M$

$e_1.VT <_V e_2.VT$ *if and only if* $e_1.VT \leq_V e_2.VT$ and $e_1.VT \neq e_2.VT$

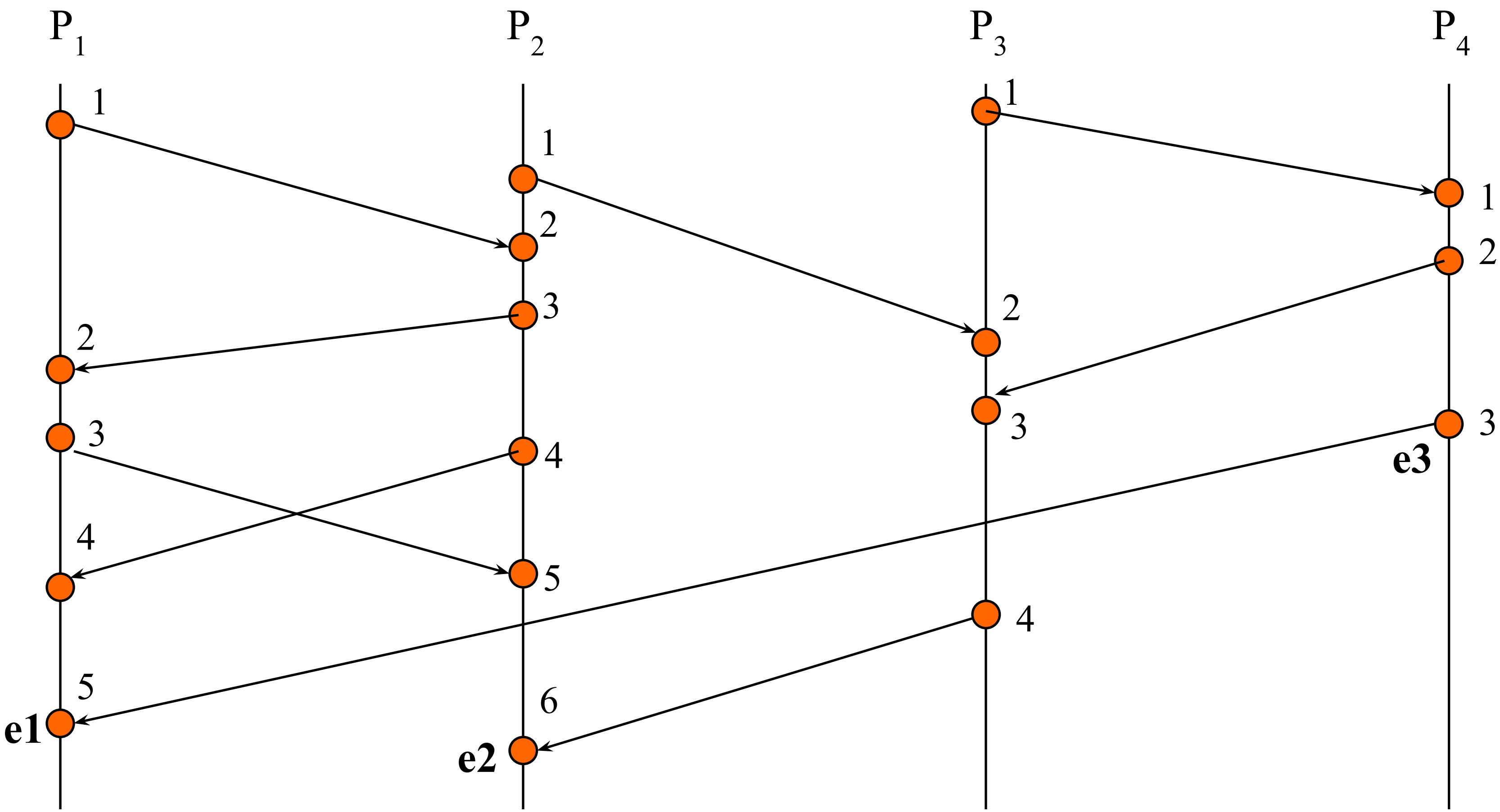
For vector timestamps it does hold that: $e_1 <_{VT} e_2 \leftrightarrow e_1 <_H e_2$

An example of a vector timestamp



e.VT=(3,6,4,2)

Comparison of vector timestamps

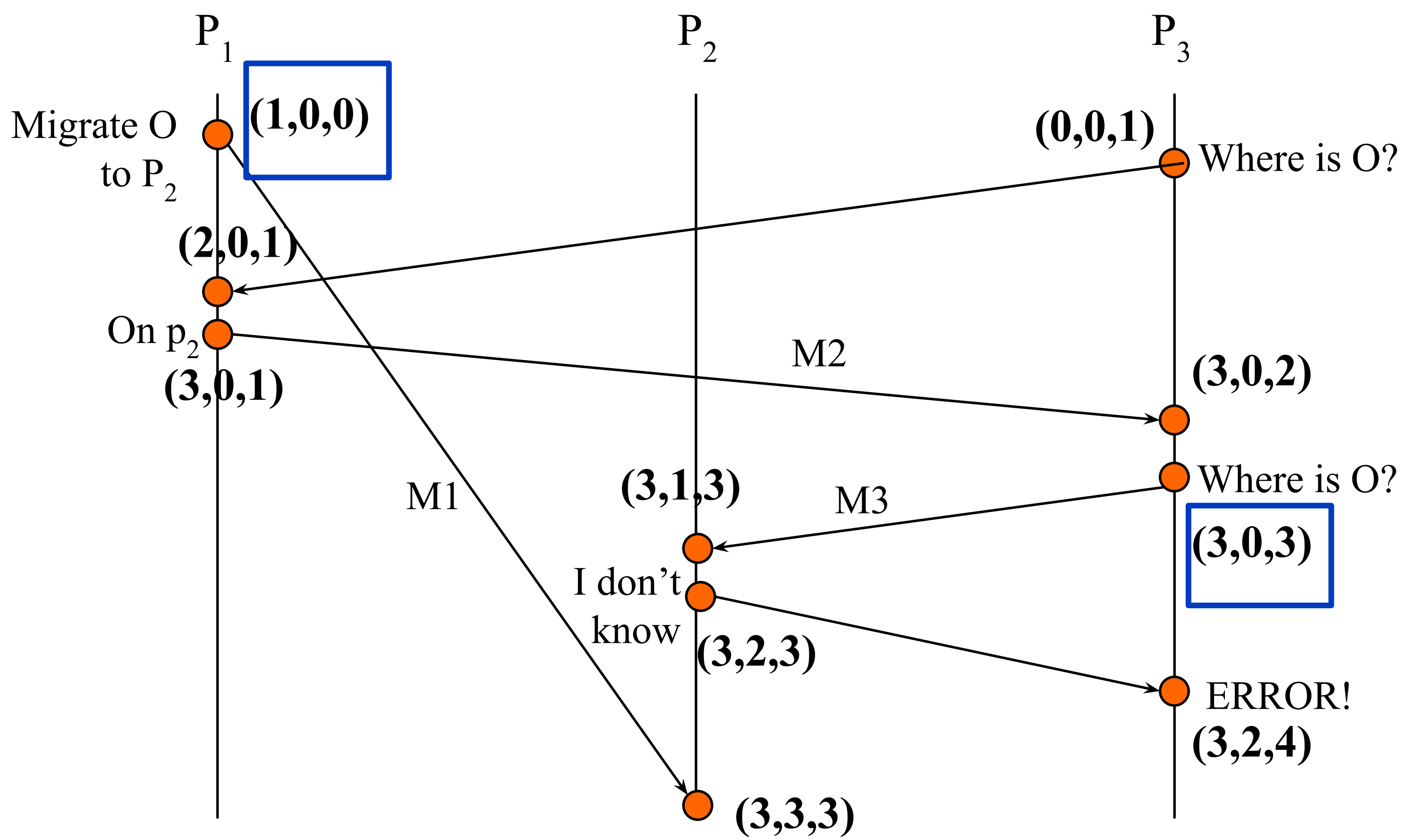


e1.VT=(5,4,1,3)

e2.VT=(3,6,4,2)

e3.VT=(0,0,1,3)

VTs can be used to detect causality violations



Causality violation between…

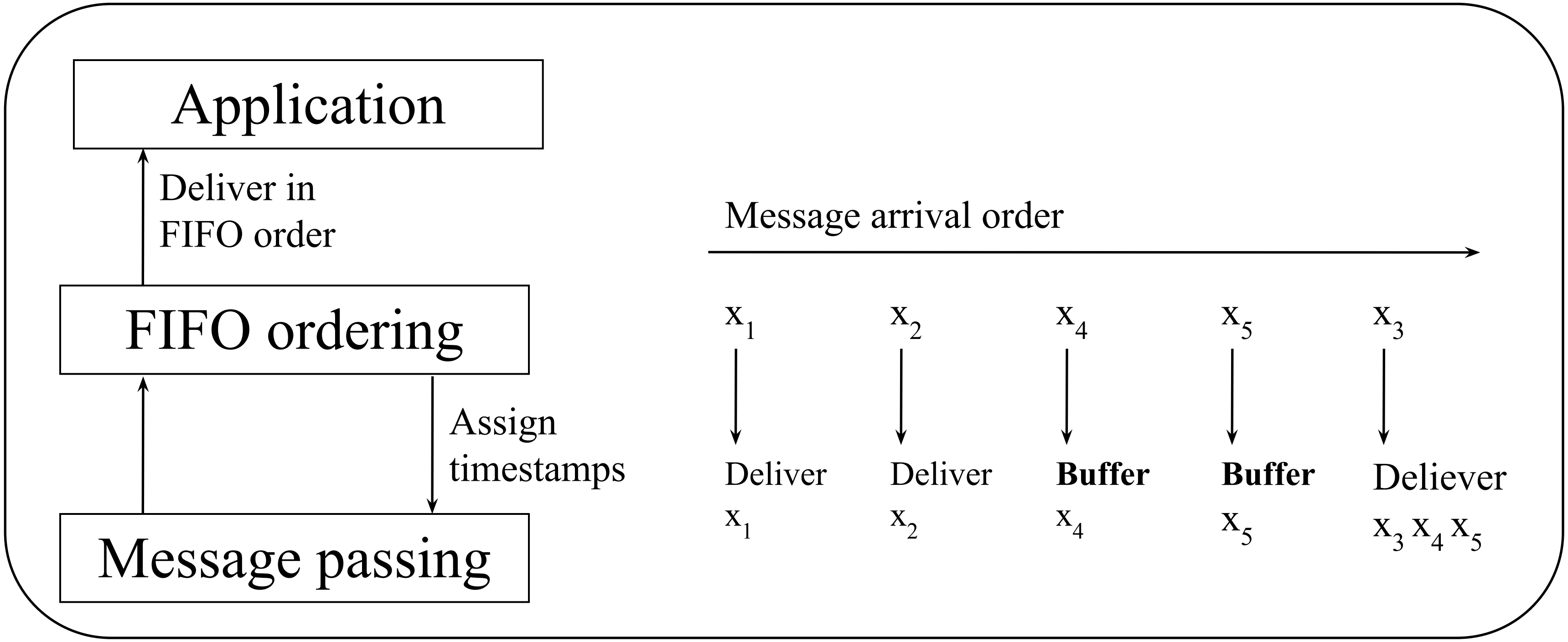
M_1 and M_3.

Distributed Synchronization: outline

- Introduction
- Causality and time
  - Lamport timestamps
  - Vector timestamps
  - Causal communication
- Snapshots
- Distributed Mutual Exclusion

Preventing causality violations

A processor cannot control the order in which it <u>receives</u> messages…
But it may control the order in which they are <u>delivered to applications</u>



Protocol for FIFO message delivery
(as in, e.g., TCP)

Preventing causality violations (cont'd)

- Senders attach a timestamp to each message
- Destination delays the delivery of out-of-order messages
- Hold back a message *m* until we are assured that no message $m' <_H m$ will be delivered
  - For every other process *p*, maintain the earliest timestamp of a message *m* that may be delivered from *p*
  - Do not deliver a message if an earlier message may still be delivered from another process

Algorithm for preventing causality violations

```
earliest[1..M] initially  [ <1,0,…0>, <0,1,…,0>, …, <0,0,…,1> ]
blocked[1…M] initially  [ {}, …, {} ]

Upon the receipt of message m from processor p
Delivery_list = {}
If (blocked[p] is empty)
    earliest[p]=m.timestamp
Add m to the tail of blocked[p]
While ( ∃ k such that blocked[k] is non-empty AND
            ∀ i ∈ {1,…,M} (except k and Self) not_earlier(earliest[i], earliest[k])
    remove the message at the head of blocked[k], put it in delivery_list
    if blocked[k] is non-empty
        earliest[k] □ m'.timestamp, where m' is at the head of blocked[k]
    else
        increment the k'th element of earliest[k]
End While
Deliver the messages in delivery_list, in causal order
```

For each processor p, the earliest timestamp with which a message from p may still be delivered

Algorithm for preventing causality violations

```
earliest[1..M] initially  [ <1,0,…0>, <0,1,…,0>, …, <0,0,…,1> ]
blocked[1…M] initially  [ {}, …, {} ]

Upon the receipt of message m from processor p
Delivery_list = {}
If (blocked[p] is empty)
   earliest[p]=m.timestamp
Add m to the tail of blocked[p]
While ( ∃  k such that blocked[k] is non-empty AND
            ∀ i ∈ {1,…,M} (except k and Self) not_earlier(earliest[i], earliest[k])
   remove the message at the head of blocked[k], put it in delivery_list
   if blocked[k] is non-empty
      earliest[k] □ m'.timestamp, where m' is at the head of blocked[k]
   else
      increment the k'th element of earliest[k]
End While
Deliver the messages in delivery_list, in causal order
```

For each process p, the messages from p that were received but were not delivered yet

Algorithm for preventing causality violations

```
earliest[1..M] initially  [ <1,0,…0>, <0,1,…,0>, …, <0,0,…,1> ]
blocked[1…M] initially  [ {}, …, {} ]

Upon the receipt of message m from processor p
Delivery_list = {}
If (blocked[p] is empty)
   earliest[p]=m.timestamp
Add m to the tail of blocked[p]
While ( ∃  k such that blocked[k] is non-empty AND
          ∀ i ∈ {1,…,M} (except k and Self) not_earlier(earliest[i], earliest[k])
   remove the message at the head of blocked[k], put it in delivery_list
   if blocked[k] is non-empty
      earliest[k] □ m'.timestamp, where m' is at the head of blocked[k]
   else
      increment the k'th element of earliest[k]
End While
Deliver the messages in delivery_list, in causal order
```

List of messages to be delivered as a result of *m*’s receipt

Algorithm for preventing causality violations

```
earliest[1..M] initially  [ <1,0,…0>, <0,1,…,0>, …, <0,0,…,1> ]
blocked[1…M] initially  [ {}, …, {} ]

Upon the receipt of message m from processor p
Delivery_list = {}
If (blocked[p] is empty)
    earliest[p]=m.timestamp
Add m to the tail of blocked[p]
While ( ∃ k such that blocked[k] is non-empty AND
            ∀ i ∈ {1,…,M} (except k and Self) not_earlier(earliest[i], earliest[k])
    remove the message at the head of blocked[k], put it in delivery_list
    if blocked[k] is non-empty
        earliest[k] □ m'.timestamp, where m' is at the head of blocked[k]
    else
        increment the k'th element of earliest[k]
End While
Deliver the messages in delivery_list, in causal order
```

If no blocked messages from *p*, update *earliest*[*p*]

Algorithm for preventing causality violations

```
earliest[1..M] initially  [ <1,0,…0>, <0,1,…,0>, …, <0,0,…,1> ]
blocked[1…M] initially  [ {}, …, {} ]

Upon the receipt of message m from processor p
Delivery_list = {}
If (blocked[p] is empty)
   earliest[p]=m.timestamp
Add m to the tail of blocked[p]
While ( ∃  k such that blocked[k] is non-empty AND
            ∀ i ∈ {1,…,M} (except k and Self) not_earlier(earliest[i], earliest[k])
   remove the message at the head of blocked[k], put it in delivery_list
   if blocked[k] is non-empty
      earliest[k] □ m'.timestamp, where m' is at the head of blocked[k]
   else
      increment the k'th element of earliest[k]
End While
Deliver the messages in delivery_list, in causal order
```

m is now the most recent message from *p* not yet delivered

Algorithm for preventing causality violations

```
earliest[1..M] initially  [ <1,0,…0>, <0,1,…,0>, …, <0,0,…,1> ]
blocked[1…M] initially  [ {}, …, {} ]

Upon the receipt of message m from processor p
Delivery_list = {}
If (blocked[p] is empty)
   earliest[p]=m.timestamp
Add m to the tail of blocked[p]
While ( ∃ k such that blocked[k] is non-empty AND
         ∀ i ∈ {1,…,M} (except k and Self) not_earlier(earliest[i], earliest[k])
   remove the message at the head of blocked[k], put it in delivery_list
   if blocked[k] is non-empty
      earliest[k] □ m'.timestamp, where m' is at the head of blocked[k]
   else
      increment the k'th element of earliest[k]
End While
Deliver the messages in delivery_list, in causal order
```

If there is a process *k* with delayed messages for which it is now safe to deliver its earliest message…

Algorithm for preventing causality violations

```
earliest[1..M] initially  [ <1,0,…0>, <0,1,…,0>, …, <0,0,…,1> ]
blocked[1…M] initially  [ {}, …, {} ]

Upon the receipt of message m from processor p
Delivery_list = {}
If (blocked[p] is empty)
   earliest[p]=m.timestamp
Add m to the tail of blocked[p]
While ( ∃  k such that blocked[k] is non-empty AND
            ∀ i ∈ {1,…,M} (except k and Self) not_earlier(earliest[i], earliest[k])
   remove the message at the head of blocked[k], put it in delivery_list
   if blocked[k] is non-empty
      earliest[k] □ m'.timestamp, where m' is at the head of blocked[k]
   else
      increment the k'th element of earliest[k]
End While
Deliver the messages in delivery_list, in causal order
```

Remove *k*'th earliest message from blocked queue, make sure it is delivered

Algorithm for preventing causality violations

```
earliest[1..M] initially [ <1,0,…0>, <0,1,…,0>, …, <0,0,…,1> ]
blocked[1…M] initially [ {}, …, {} ]

Upon the receipt of message m from processor p
Delivery_list = {}
If (blocked[p] is empty)
   earliest[p]=m.timestamp
Add m to the tail of blocked[p]
While ( ∃ k such that blocked[k] is non-empty AND
          ∀ i ∈ {1,…,M} (except k and Self) not_earlier(earliest[i], earliest[k])
   remove the message at the head of blocked[k], put it in delivery_list
   if blocked[k] is non-empty
      earliest[k] □ m'.timestamp, where m' is at the head of blocked[k]
   else
      increment the k'th element of earliest[k]
End While
Deliver the messages in delivery_list, in causal order
```

If there are additional blocked messages of *k*, update earliest[k] to be the timestamp of the earliest such message

Algorithm for preventing causality violations

```
earliest[1..M] initially  [ <1,0,…0>, <0,1,…,0>, …, <0,0,…,1> ]
blocked[1…M] initially  [ {}, …, {} ]

Upon the receipt of message m from processor p
Delivery_list = {}
If (blocked[p] is empty)
   earliest[p]=m.timestamp
Add m to the tail of blocked[p]
While ( ∃  k such that blocked[k] is non-empty AND
            ∀ i ∈ {1,…,M} (except k and Self) not_earlier(earliest[i], earliest[k])
   remove the message at the head of blocked[k], put it in delivery_list
   if blocked[k] is non-empty
      earliest[k] □ m'.timestamp, where m' is at the head of blocked[k]
   else
      increment the k'th element of earliest[k]
End While
Deliver the messages in delivery_list, in causal order
```

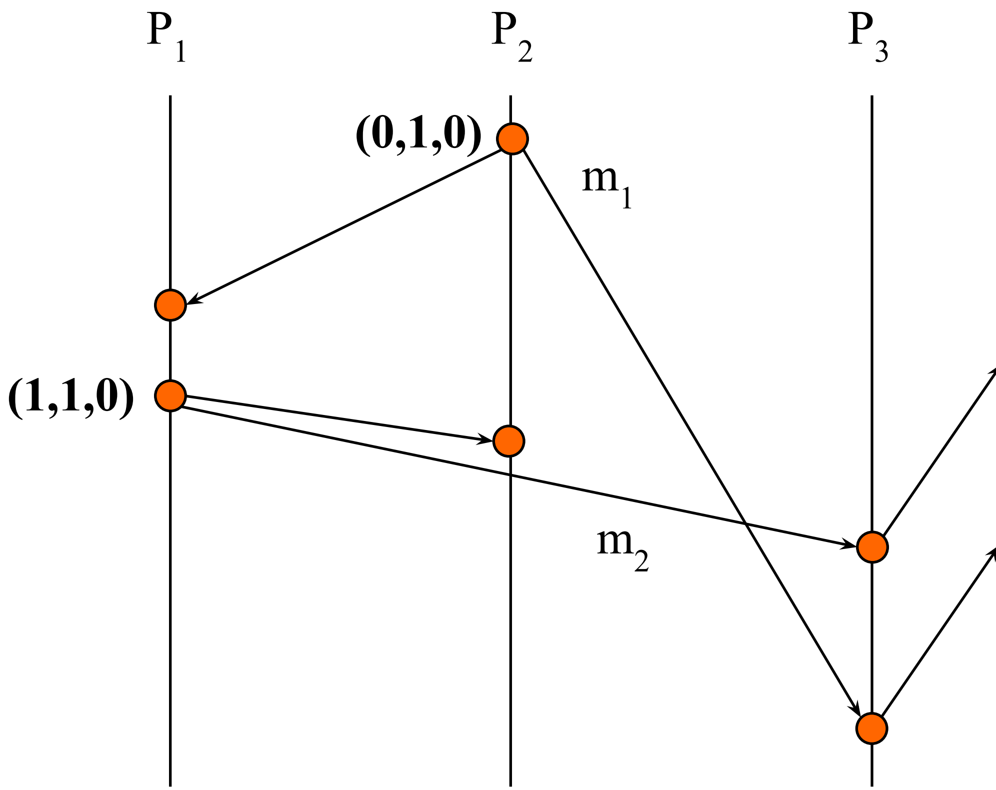
Otherwise the earliest message that may be delivered from *k* would have previous timestamp with the *k*'th component incremented

Algorithm for preventing causality violations

```
earliest[1..M] initially  [ <1,0,…0>, <0,1,…,0>, …, <0,0,…,1> ]
blocked[1…M] initially  [ {}, …, {} ]

Upon the receipt of message m from processor p
Delivery_list = {}
If (blocked[p] is empty)
   earliest[p]=m.timestamp
Add m to the tail of blocked[p]
While ( ∃  k such that blocked[k] is non-empty AND
            ∀ i ∈ {1,…,M} (except k and Self) not_earlier(earliest[i], earliest[k])
   remove the message at the head of blocked[k], put it in delivery_list
   if blocked[k] is non-empty
      earliest[k] □ m'.timestamp, where m' is at the head of blocked[k]
   else
      increment the k'th element of earliest[k]
End While
Deliver the messages in delivery_list, in causal order
```

Finally, deliver set of messages that will not cause causality violation (if there are any).

Execution of the algorithm as multicast



p_3 state

earliest[1]	earliest[2]	earliest[3]
(1,0,0)	**(0,1,0)**	**(0,0,1)**

Upon receipt of m_2

(1,1,0) m_2	**(0,1,0)**	**(0,0,1)**

Upon receipt of m_1

(1,1,0) m_2	**(0,1,0)** m_1	**(0,0,1)**
	deliver m_1	
(1,1,0) m_2	**(0,2,0)**	**(0,0,1)**
	deliver m_2	
(2,1,0)	**(0,2,0)**	**(0,0,1)**

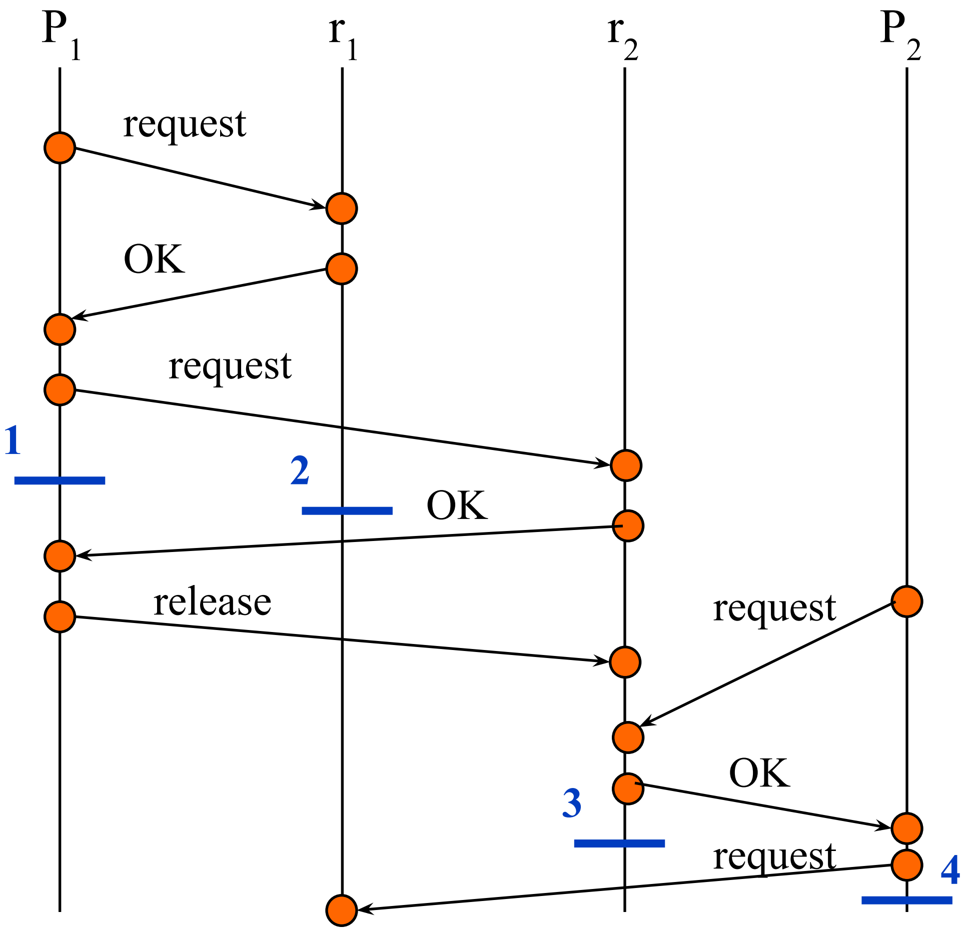
Since the algorithm is “interested” only in causal order of sending events, vector timestamp is incremented only upon send events.

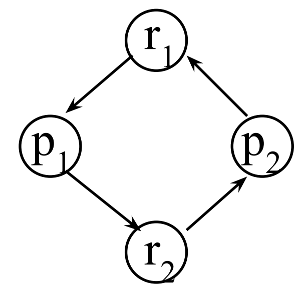
Distributed Synchronization: outline

- Introduction
- Causality and time
  - Lamport timestamps
  - Vector timestamps
  - Causal communication
- Snapshots
- Distributed Mutual Exclusion

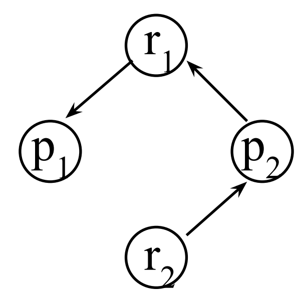
Snapshots motivation – phantom deadlock

- Assume we would like to implement a distributed ***deadlock-detector***
- We record process states to check if there is a *waits-for-cycle*





Observed waits-for graph



Actual waits-for graph

What is a snapshot?

- Global system state:
  - $S=(s_1, s_2, \ldots, s_M)$ – local processor states
  - The contents $L_{i,j} =(m_1, m_2, \ldots, m_k)$ of each communication channel $C_{i,j}$ (channels assumed to be FIFO)
    <u>These are messages sent but not yet received</u>

- Global state must be <u>consistent</u>
  - If we observe in state s_i that p_i received message *m* from p_k, then in observation s_k ,*k* must have sent *m.*
  - Each $L_{i,j}$ must contain exactly the set of messages sent by p_i but not yet received by p_j, as reflected by s_i, s_j.

- Snapshot state much be <u>consistent</u>, one that might have existed during the computation.
- Observations must be <u>mutually concurrent</u> that is, no observation casually precedes another observation (<u>a consistent cut</u>)

Snapshot algorithm – informal description

- Upon joining the algorithm, a process records its local state

- The process that initiates the snapshot sends *snapshot tokens* to its neighbors (before sending any other messages)
  - Neighbors send them to their neighbors – broadcast

- Upon receiving a snapshot token:
  - a process records its state prior to sending/receiving additional messages
  - Must then send tokens to all its other neighbors

- How shall we record sets $L_{p,q}$?
  - q receives token from p and that is the first time q receives token:
    $L_{p,q}$={ }
  - q receives token from p but q received token before:
    $L_{p,q}$={all messages received by q from p since q received token}

Snapshot algorithm – data structures

- Different snapshots distinguished by version number

<u>Per process variables</u>

integer *my_version* initially 0

integer *current_snap* initially 0

Integer *tokens_received*

processor_state S

channel_state [1..M]

The version number of my current snapshot

Snapshot algorithm – data structures

- Different snapshots distinguished by version number

<u>Per process variables</u>

integer *my_version* initially 0

integer *current_snap* initially 0

integer *tokens_received*

processor_state S

channel_state [1..M]

current_snap contains version number of snapshot

Snapshot algorithm – data structures

- Different snapshots distinguished by version number

<u>Per process variables</u>

integer *my_version* initially 0

integer *current_snap* initially 0

integer *tokens_received*

processor_state S

channel_state [1..M]

tokens_received contains the number of tokens received for the snapshot

Snapshot algorithm – data structures

- Different snapshots distinguished by version number

<u>Per process variables</u>

integer *my_version* initially 0

integer *current_snap* initially 0

integer *tokens_received*

processor_state S

channel_state [1..M]

processor_state contains the state recorded for the snapshot

Snapshot algorithm – data structures

- Different snapshots distinguished by version number

Per process variables

integer *my_version* initially 0

integer *current_snap* initially 0

integer *tokens_received*

processor_state S

channel_state [1..M]

channel_state[r] records the state of channel from q to current processor for the snapshot

Snapshot algorithm – pseudo-code

```
execute_snapshot()

Wait for a snapshot request or a token

Snapshot request:
    my_version++, current_snap=my_version
    S □ my_state
     for each outgoing channel q, send(q, TOEKEN, my_version)
     tokens_received=0

 TOKEN(q; version):
  If current_snap < version
     S □ my state
     current_snap=version
     L[q] □ empty, send token(version) on each outgoing channel
     tokens_received □ 1
  else
     tokens_received++
      L[q] □ all messages received from q since first receiving token(version)
      if tokens_received = #incoming channels, local snapshot for (version) is finished
```

Snapshot algorithm – pseudo-code

```
execute_snapshot()

Wait for a snapshot request or a token

Snapshot request:
    my_version++, current_snap=my_version
    S □ my_state
    for each outgoing channel q, send(q, TOEKEN, my_version)
    tokens_received=0

TOKEN(q; version):
    If current_snap < version
       S □ my state
       current_snap=version
       L[q] □ empty, send token( version) on each outgoing channel
       tokens_received □ 1
    else
       tokens_received++
       L[q] □ all messages received from q since first receiving token(version)
       if tokens_received = #incoming channels, local snapshot for (version) is finished
```

Snapshot algorithm – pseudo-code

Increment version number of this snapshot, record my local state

```
execute_snapshot()

Wait for a snapshot request or a token

Snapshot request:
1     my_version++, current_snap=my_version
2     S □ my_state
3     for each outgoing channel q, send(q, TOEKEN, my_version)
4     tokens_received=0

 TOKEN(q; version):
5.    If current_snap < version
6.      S □ my state
7.      current_snap=version
8.      L[q] □ empty, send token( version) on each outgoing channel
9.      tokens_received □ 1
10.   else
11.     tokens_received++
12.      L[q] □ all messages received from q since first receiving token(version)
13.      if tokens_received = #incoming channels, local snapshot for (version) is finished
```

Snapshot algorithm – pseudo-code

```
execute_snapshot()

Wait for a snapshot request or a token

Snapshot request:
    my_version++, current_snap=my_version
    S □ my_state
    for each outgoing channel q, send(q, TOEKEN; self, my_version)
    tokens_received=0

TOKEN(q; version):
    If current_snap < version
      S □ my state
      current_snap=version
      L[q] □ empty, send token( version) on each outgoing channel
      tokens_received □ 1
    else
      tokens_received++
       L[q] □ all messages received from q since first receiving token(version)
       if tokens_received = #incoming channels, local snapshot for (version) is finished
```

Send snapshot-token on all outgoing channels, initialize number of received tokens for my snapshot to 0

Snapshot algorithm – pseudo-code

```
execute_snapshot()

Wait for a snapshot request or a token

Snapshot request:
    my_version++, current_snap=my_version
    S □ my_state
     for each outgoing channel q, send(q, TOEKEN; self, my_version)
     tokens_received=0

 TOKEN(q; version):
    If current_snap < version
       S □ my state
       current_snap=version
       L[q] □ empty, send token( version) on each outgoing channel
       tokens_received □ 1
    else
       tokens_received++
        L[q] □ all messages received from q since first receiving token(version)
        if tokens_received = #incoming channels, local snapshot for (version) is finished
```

Upon receipt from q of TOKEN for snapshot (version)

Snapshot algorithm – pseudo-code

```
execute_snapshot()

Wait for a snapshot request or a token

Snapshot request:
    my_version++, current_snap=my_version
    S □ my_state
     for each outgoing channel q, send(q, TOEKEN; self, my_version)
     tokens_received=0

 TOKEN(q; version):
    If current_snap < version
        S □ my state
        current_snap=version
        L[q] □ empty, send token( version) on each outgoing channel
        tokens_received □ 1
    else
        tokens_received++
         L[q] □ all messages received from q since first receiving token(version)
         if tokens_received = #incoming channels, local snapshot for (version) is finished
```

If this is first token for snapshot (version)

Snapshot algorithm – pseudo-code

```
execute_snapshot()

Wait for a snapshot request or a token

Snapshot request:
    my_version++, current_snap=my_version
    S □ my_state
     for each outgoing channel q, send(q, TOEKEN; self, my_version)
     tokens_received=0

 TOKEN(q; version):
   If current_snap < version
      S □ my state
      current_snap=version
      L[q] □ empty, send token( version) on each outgoing channel
      tokens_received □ 1
   else
      tokens_received++
       L[q] □ all messages received from q since first receiving token(version)
       if tokens_received = #incoming channels, local snapshot for (version) is finished
```

Record local state for snapshot
Update version number

Snapshot algorithm – pseudo-code

```
execute_snapshot()

Wait for a snapshot request or a token

Snapshot request:
    my_version++, current_snap=my_version
    S □ my_state
     for each outgoing channel q, send(q, TOEKEN; self, my_version)
     tokens_received=0

 TOKEN(q; version):
    If current_snap < version
      S □ my state
      current_snap=version
      L[q] □ empty, send token( version) on each outgoing channel
      tokens_received □ 1
    else
      tokens_received++
       L[q] □ all messages received from q since first receiving token(version)
       if tokens_received = #incoming channels, local snapshot for (version) is finished
```

Set of messages on channel from q is empty
Send token on all outgoing channels
Initialize number of received tokens to 1

Snapshot algorithm – pseudo-code

```
execute_snapshot()

Wait for a snapshot request or a token

Snapshot request:
    my_version++, current_snap=my_version
    S □ my_state
    for each outgoing channel q, send(q, TOEKEN; self, my_version)
    tokens_received=0

TOKEN(q; version):                         Not the first token for (version)
    If current_snap < version
        S □ my state
        current_snap=version
        L[q] □ empty, send token( version) on each outgoing channel
        tokens_received □ 1
    else
        tokens_received++
        L[q] □ all messages received from q since first receiving token(version)
        if tokens_received = #incoming channels, local snapshot for (version) is finished
```

Snapshot algorithm – pseudo-code

```
execute_snapshot()

Wait for a snapshot request or a token

Snapshot request:
      my_version++, current_snap=my_version
      S □ my_state
       for each outgoing channel q, send(q, TOEKEN; self, my_version)
       tokens_received=0

TOKEN(q; version):
    If current_snap < version
       S □ my state
       current_snap=version
       L[q] □ empty, send token( version) on each outgoing channel
       tokens_received □ 1
    else
       tokens_received++
        L[q] □ all messages received from q since first receiving token(version)
        if tokens_received = #incoming channels, local snapshot for (version) is finished
```

Yet another token for snapshot (version)

Snapshot algorithm – pseudo-code

```
execute_snapshot()

Wait for a snapshot request or a token

Snapshot request:
    my_version++, current_snap=my_version
    S □ my_state
     for each outgoing channel q, send(q, TOEKEN; self, my_version)
     tokens_received=0

 TOKEN(q; version):
  If current_snap < version
     S □ my state
     current_snap=version
     L[q] □ empty, send token( version) on each outgoing channel
     tokens_received □ 1
  else
     tokens_received++
     L[q] □ all messages received from q since first receiving token(version)
     if tokens_received = #incoming channels, local snapshot for (version) is finished
```

These messages are the state of the channel from q for snapshot (version)

Snapshot algorithm – pseudo-code

```
execute_snapshot()

Wait for a snapshot request or a token

Snapshot request:
    my_version++, current_snap=my_version
    S □ my_state
    for each outgoing channel q, send(q, TOEKEN; self, my_version)
    tokens_received=0

TOKEN(q; version):
    If current_snap < version
        S □ my state
        current_snap=version
        L[q] □ empty, send token( version) on each outgoing channel
        tokens_received □ 1
    else
        tokens_received++
        L[q] □ all messages received from q since first receiving token(version)
        if tokens_received = #incoming channels, local snapshot for (version) is finished
```

If all tokens of snapshot version arrived, snapshot computation is over

Snapshot algorithm – data structures

- The algorithm supports multiple ongoing snapshots – one per process
- Different snapshots from same process distinguished by version number

Per process variables

integer *my_version* initially 0
integer *current_snap*[1..M] initially [0,…,0]
integer *tokens_received*[1..M]
processor_state S[1…M]
channel_state [1..M][1..M]

The version number of my current snapshot

Snapshot algorithm – data structures

- The algorithm supports multiple ongoing snapshots – one per process
- Different snapshots from same process distinguished by version number

<u>Per process variables</u>

integer *my_version* initially 0
integer *current_snap*[1..M] initially [0,…,0]
integer *tokens_received*[1..M]
processor_state S[1…M]
channel_state [1..M][1..M]

current_snap[r] contains version number of snapshot initiated by processor r

Snapshot algorithm – data structures

- The algorithm supports multiple ongoing snapshots – one per process
- Different snapshots from same process distinguished by version number

<u>Per process variables</u>

integer *my_version* initially 0
integer *current_snap*[1..M] initially [0,…,0]
integer *tokens_received*[1..M]
processor_state S[1…M]
channel_state [1..M][1..M]

tokens_received[r] contains the number of tokens received for the snapshot initiated by processor r

Snapshot algorithm – data structures

- The algorithm supports multiple ongoing snapshots – one per process
- Different snapshots from same process distinguished by version number

<u>Per process variables</u>

integer *my_version* initially 0

integer *current_snap*[1..M] initially [0,…,0]

integer *tokens_received*[1..M]

processor_state S[1…M]

channel_state [1..M][1..M]

processor_state[r] contains the state recorded for the snapshot of processor r

Snapshot algorithm – data structures

- The algorithm supports multiple ongoing snapshots – one per process
- Different snapshots from same process distinguished by version number

<u>Per process variables</u>

integer *my_version* initially 0

integer *current_snap*[1..M] initially [0,…,0]

integer *tokens_received*[1..M]

processor_state S[1…M]

channel_state [1..M][1..M]

channel_state[r][q] records the state of channel from q to current processor for the snapshot initiated by processor r

Snapshot algorithm – pseudo-code

```
execute_snapshot()

Wait for a snapshot request or a token

Snapshot request:
    my_version++, current_snap[self]=my_version
    S[self] □ my_state
     for each outgoing channel q, send(q, TOEKEN, my_version)
     tokens_received[self]=0

TOKEN(q; r,version):
    If current_snap[r] < version
      S[r] □ my state
      current_snap[r]=version
      L[r][q] □ empty, send token(r, version) on each outgoing channel
      tokens_received[r] □ 1
    else
      tokens_received[r]++
       L[r][q] □ all messages received from q since first receiving token(r,version)
       if tokens_received = #incoming channels, local snapshot for (r,version) is finished
```

Snapshot algorithm – pseudo-code

```
execute_snapshot()

Wait for a snapshot request or a token

Snapshot request:
    my_version++, current_snap[self]=my_version
    S[self] □ my_state
     for each outgoing channel q, send(q, TOEKEN, my_version)
     tokens_received[self]=0

 TOKEN(q; r,version):
   If current_snap[r] < version
     S[r] □ my state
     current_snap[r]=version
     L[r][q] □ empty, send token(r, version) on each outgoing channel
     tokens_received[r] □ 1
   else
     tokens_received[r]++
      L[r][q] □ all messages received from q since first receiving token(r,version)
      if tokens_received = #incoming channels, local snapshot for (r,version) is finished
```

Snapshot algorithm – pseudo-code

Increment version number of this snapshot, record my local state

execute_snapshot()

<u>*Wait for a snapshot request or a token*</u>

<u>Snapshot request</u>:

1 *my_version*++, current_snap[self]=my_version

2 *S[self]* □ my_state

3 *for each outgoing channel q, send(q, TOEKEN, my_version)*

4 *tokens_received[self]=0*

<u>TOKEN(q; r,version)</u>:

5. If current_snap[r] < version
6. S[r] □ my state
7. current_snap[r]=version
8. L[r][q] □ empty, send token(r, version) on each outgoing channel
9. tokens_received[r] □ 1
10. else
11. tokens_received[r]++
12. L[r][q] □ all messages received from q since first receiving token(r,version)
13. if tokens_received = #incoming channels, local snapshot for (r,version) is finished

Snapshot algorithm – pseudo-code

execute_snapshot()

<u>*Wait for a snapshot request or a token*</u>

Send snapshot-token on all outgoing channels, initialize number of received tokens for my snapshot to 0

<u>Snapshot request</u>:

1 *my_version*++, current_snap[self]=my_version

2 *S[self]* □ my_state

3 *for each outgoing channel q, send(q, TOEKEN; self, my_version)*

4 *tokens_received[self]=0*

<u>TOKEN(q; r,version)</u>:

5. If current_snap[r] < version
6. S[r] □ my state
7. current_snap[r]=version
8. L[r][q] □ empty, send token(r, version) on each outgoing channel
9. tokens_received[r] □ 1
10. else
11. tokens_received[r]++
12. L[r][q] □ all messages received from q since first receiving token(r,version)
13. if tokens_received = #incoming channels, local snapshot for (r,version) is finished

Snapshot algorithm – pseudo-code

```
execute_snapshot()

Wait for a snapshot request or a token

Snapshot request:
      my_version++, current_snap[self]=my_version
      S[self] □ my_state
      for each outgoing channel q, send(q, TOEKEN; self, my_version)
      tokens_received[self]=0

TOKEN(q; r,version):
      If current_snap[r] < version
        S[r] □ my state
        current_snap[r]=version
        L[r][q] □ empty, send token(r, version) on each outgoing channel
        tokens_received[r] □ 1
      else
        tokens_received[r]++
        L[r][q] □ all messages received from q since first receiving token(r,version)
        if tokens_received = #incoming channels, local snapshot for (r,version) is finished
```

Upon receipt from q of TOKEN for snapshot (r,version)

Snapshot algorithm – pseudo-code

```
execute_snapshot()

Wait for a snapshot request or a token

Snapshot request:
    my_version++, current_snap[self]=my_version
    S[self] □ my_state
     for each outgoing channel q, send(q, TOEKEN; self, my_version)
     tokens_received[self]=0

 TOKEN(q; r,version):
    If current_snap[r] < version
      S[r] □ my state
      current_snap[r]=version
      L[r][q] □ empty, send token(r, version) on each outgoing channel
      tokens_received[r] □ 1
    else
      tokens_received[r]++
       L[r][q] □ all messages received from q since first receiving token(r,version)
       if tokens_received = #incoming channels, local snapshot for (r,version) is finished
```

If this is first token for snapshot (r,version)

Snapshot algorithm – pseudo-code

```
execute_snapshot()

Wait for a snapshot request or a token

Snapshot request:
1     my_version++, current_snap[self]=my_versio
2     S[self] □ my_state
3     for each outgoing channel q, send(q, TOEKEN; self, my_version)
4     tokens_received[self]=0

TOKEN(q; r,version):
5.  If current_snap[r] < version
6.    S[r] □ my state
7.    current_snap[r]=version
8.    L[r][q] □ empty, send token(r, version) on each outgoing channel
9.    tokens_received[r] □ 1
10. else
11.   tokens_received[r]++
12.   L[r][q] □ all messages received from q since first receiving token(r,version)
13.   if tokens_received = #incoming channels, local snapshot for (r,version) is finished
```

Record local state for r'th snapshot
Update version number of r'th snapshot

Snapshot algorithm – pseudo-code

```
execute_snapshot()

Wait for a snapshot request or a token

Snapshot request:
    my_version++, current_snap[self]=my_version
    S[self] □ my_state
     for each outgoing channel q, send(q, TOEKEN; self, my_version)
     tokens_received[self]=0

 TOKEN(q; r,version):
    If current_snap[r] < version
       S[r] □ my state
       current_snap[r]=version
       L[r][q] □ empty, send token(r, version) on each outgoing channel
       tokens_received[r] □ 1
    else
       tokens_received[r]++
        L[r][q] □ all messages received from q since first receiving token(r,version)
        if tokens_received = #incoming channels, local snapshot for (r,version) is finished
```

```
Set of messages on channel from q is empty
Send token on all outgoing channels
Initialize number of received tokens to 1
```

Snapshot algorithm – pseudo-code

```
execute_snapshot()

Wait for a snapshot request or a token

Snapshot request:
    my_version++, current_snap[self]=my_version
    S[self] □ my_state
     for each outgoing channel q, send(q, TOEKEN; self, my_version)
     tokens_received[self]=0

 TOKEN(q; r,version):
    If current_snap[r] < version
       S[r] □ my state
       current_snap[r]=version
       L[r][q] □ empty, send token(r, version) on each outgoing channel
       tokens_received[r] □ 1
    else
       tokens_received[r]++
        L[r][q] □ all messages received from q since first receiving token(r,version)
        if tokens_received = #incoming channels, local snapshot for (r,version) is finished
```

Not the first token for (r,version)

Snapshot algorithm – pseudo-code

```
execute_snapshot()

Wait for a snapshot request or a token

Snapshot request:
    my_version++, current_snap[self]=my_version
    S[self] □ my_state
    for each outgoing channel q, send(q, TOEKEN; self, my_version)
    tokens_received[self]=0

TOKEN(q; r,version):                          Yet another token for snapshot (r,version)
    If current_snap[r] < version
        S[r] □ my state
        current_snap[r]=version
        L[r][q] □ empty, send token(r, version) on each outgoing channel
        tokens_received[r] □ 1
    else
        tokens_received[r]++
        L[r][q] □ all messages received from q since first receiving token(r,version)
        if tokens_received = #incoming channels, local snapshot for (r,version) is finished
```

Snapshot algorithm – pseudo-code

```
execute_snapshot()

Wait for a snapshot request or a token

Snapshot request:
    my_version++, current_snap[self]=my_version
    S[self] □ my_state
    for each outgoing channel q, send(q, TOEKEN; self, my_version)
    tokens_received[self]=0

TOKEN(q; r,version):
    If current_snap[r] < version
        S[r] □ my state
        current_snap[r]=version
        L[r][q] □ empty, send token(r, version) on each outgoing channel
        tokens_received[r] □ 1
    else
        tokens_received[r]++
        L[r][q] □ all messages received from q since first receiving token(r,version)
        if tokens_received = #incoming channels, local snapshot for (r,version) is finished
```

These messages are the state of the channel from q for snapshot (r,version)

Snapshot algorithm – pseudo-code

```
execute_snapshot()

Wait for a snapshot request or a token

Snapshot request:
    my_version++, current_snap[self]=my_version
    S[self] □ my_state
     for each outgoing channel q, send(q, TOEKEN; self, my_version)
     tokens_received[self]=0

 TOKEN(q; r,version):
  If current_snap[r] < version
     S[r] □ my state
     current_snap[r]=version
     L[r][q] □ empty, send token(r, version) on each outgoing channel
     tokens_received[r] □ 1
  else
     tokens_received[r]++
      L[r][q] □ all messages received from q since first receiving token(r,version)
      if tokens_received = #incoming channels, local snapshot for (r,version) is finished
```

If all tokens of snapshot (r,version) arrived, snapshot computation is over

Distributed Synchronization: outline

- Introduction
- Causality and time
  - Lamport timestamps
  - Vector timestamps
  - Causal communication
- Snapshots
- Distributed Mutual Exclusion
  - Ricart and Agrawala's algorithm
  - Raymond's algorithm
  - The MCS algorithm

Distributed mutual exclusion: introduction

- Distributed mutual exclusion required (e.g.) for transaction processing on replicated data

- We assume there are no failures
  - Processors do not fail
  - Communication links do not fail

- It is easy to implement mutual exclusion using totally-ordered timestamps
  - The first algorithm we show may use (e.g.) Lamport's timestamps

Ricart and Agrawal's algorithm: high-level ideas

- When you want to enter your CS
  - Record your timestamp
  - Ask everyone else whether they “permit”

- When asked for a permission
  - Halt response if in CS
  - Halt response if in entry code with a smaller timestamp (<u>we use total order between timestamps</u>)
  - Otherwise, “permit”

- Upon exit from CS
  - Send halted responses (if any)

Ricart and Agrawal's algorithm: data-structures

<u>Per processor variables</u>

timestamp *current_time*

Timestamp *my_timestamp*

integer *reply_pending*

boolean *isRequesting*

boolean *reply_deferred*[M]

Processor's current Lamport timestamp

Ricart and Agrawal's algorithm: data-structures

<u>Per processor variables</u>

timestamp *current_time*

Timestamp *my_timestamp*

integer *reply_pending*

boolean *isRequesting*

boolean *reply_deferred*[M]

The timestamp of the processor's current request

Ricart and Agrawal's algorithm: data-structures

<u>Per processor variables</u>

timestamp *current_time*

Timestamp *my_timestamp*

integer *reply_pending*

boolean *isRequesting*

boolean *reply_deferred*[M]

The number of permissions that the processor still need to collect before entering the CS

Ricart and Agrawal's algorithm: data-structures

Per processor variables

timestamp *current_time*

Timestamp *my_timestamp*

integer *reply_pending*

boolean *isRequesting*

boolean *reply_deferred*[M]

True iff this processor is requesting or using the CS

Ricart and Agrawal's algorithm: data-structures

<u>Per processor variables</u>

timestamp *current_time*

Timestamp *my_timestamp*

integer *reply_pending*

boolean *isRequesting*

boolean *reply_deferred*[M]

Entry j is true iff this processor deferred replying to processor j's request

Ricart and Agrawal's algorithm: entry-code

```
Request_CS:
my_timstamp ← current_time
isRequesting ← TRUE
Reply_pending ← M-1
for every other processor j
    send(REMOTE_REQUEST; my_timestamp)
wait until reply_pending = 0
```

Set the (Lamport) timestamp of my request

Ricart and Agrawal's algorithm: entry-code

```
Request_CS:
my_timstamp ← current_time
isRequesting ← TRUE
reply_pending ← M-1
for every other processor j
   send(REMOTE_REQUEST; my_timestamp)
wait until reply_pending = 0
```

Mark that this processor is requesting entry to CS

Ricart and Agrawal's algorithm: entry-code

```
Request_CS:
my_timstamp ← current_time
isRequesting ← TRUE
reply_pending ← M-1
for every other processor j
    send(REMOTE_REQUEST; my_timestamp)
wait until reply_pending = 0
```

Need to receive replies from all other processors

Ricart and Agrawal's algorithm: entry-code

```
Request_CS:
my_timstamp □ current_time
isRequesting □ TRUE
reply_pending □ M-1
for every other processor j
    send(REMOTE_REQUEST; my_timestamp)
wait until reply_pending = 0
```

Request permission from all other processors

Ricart and Agrawal's algorithm: entry code

```
Request_CS:
my_timstamp ← current_time
isRequesting ← TRUE
reply_pending ← M-1
for every other processor j
    send(REMOTE_REQUEST; my_timestamp)
wait until reply_pending = 0
```

When all other processors reply – may enter the CS

Ricart and Agrawal's algorithm: monitoring

CS_monitoring:
Wait until a REMOTE_REUQUEST or REPLY message is received

REMOTE_REQUEST(sender; request_time)
1. Let j be the sender of the REMOTE_REQUEST message
2. if (not *is_requesting* or *my_timestamp* > *request_time*)
3. send(*j*, REPLY)
4. else
5. reply_deferred[j]=TRUE

REPLY
4. *reply_pending* □ *reply_pending*-1

Listener thread to respond to protocol messages at all times

Ricart and Agrawal's algorithm: monitoring

<u>CS_monitoring</u>:

Wait until a REMOTE_REUQUEST or REPLY message is received

<u>REMOTE_REQUEST(sender; request_time)</u>

1. Let j be the sender of the REMOTE_REQUEST message
2. if (not *is_requesting* or *my_timestamp* > *request_time)*
3. send(*j*, REPLY)
4. else
5. reply_deferred[j]=TRUE

Upon receipt of remote request

<u>REPLY</u>

4. *reply_pending* □ *reply_pending*-1

Ricart and Agrawal's algorithm: monitoring

CS_monitoring:

Wait until a REMOTE_REUQUEST or REPLY message is received

REMOTE_REQUEST(sender; request_time)

1. Let j be the sender of the REMOTE_REQUEST message
2. if (not *is_requesting* or *my_timestamp* > *request_time)*
3. send(*j*, REPLY)
4. else
5. reply_deferred[j]=TRUE

If should grant processor j'th request

REPLY

4. *reply_pending* □ *reply_pending*-1

Ricart and Agrawal's algorithm: monitoring

CS_monitoring:

Wait until a REMOTE_REUQUEST or REPLY message is received

REMOTE_REQUEST(sender; request_time)

1. Let j be the sender of the REMOTE_REQUEST message
2. if (not *is_requesting* or *my_timestamp* > *request_time)*
3. send(*j*, REPLY)
4. else
5. reply_deferred[j]=TRUE

Send reply to processor j

REPLY

4. *reply_pending* □ *reply_pending*-1

Ricart and Agrawal's algorithm: monitoring

<u>CS_monitoring</u>:

Wait until a REMOTE_REUQUEST or REPLY message is received

<u>REMOTE_REQUEST(sender; request_time)</u>

1. Let j be the sender of the REMOTE_REQUEST message
2. if (not *is_requesting* or *my_timestamp* > *request_time)*
3.   send(*j*, REPLY)
4. else
5.   reply_deferred[j]=TRUE

<u>REPLY</u>

4. *reply_pending* □ *reply_pending*-1

Otherwise, defer replying to this request

Ricart and Agrawal's algorithm: monitoring

CS_monitoring:

Wait until a REMOTE_REUQUEST or REPLY message is received

REMOTE_REQUEST(sender; request_time)

1. Let j be the sender of the REMOTE_REQUEST message
2. if (not *is_requesting* or *my_timestamp* > *request_time)*
3.   send(*j*, REPLY)
4. else
5.   reply_deferred[j]=TRUE

REPLY

4. *reply_pending* □ *reply_pending*-1

Upon receiving a reply, decrement *reply_pending*

Ricart and Agrawal's algorithm: exit section

Release_CS:

1. *is_requesting* false
2. For j=1 through M (other than this processor's ID)
3.   if *reply_deferred*[i]=TRUE
4.   send(j, REPLY)
5.   reply_deferred[j]=FALSE

No longer requesting CS

Ricart and Agrawal's algorithm: exit section

<u>Release_CS_monitoring</u>:

```
is_requesting □ false
For j=1 through M (other than this processor's ID)
    if reply_deferred[i]=TRUE
        send(j, REPLY)
        reply_deferred[j]=FALSE
```

For each processor awaiting a reply from this processor, send reply and mark that there are no more deferred replies.

Ricart and Agrawal's algorithm: comments

Why is mutual exclusion satisfied?

Because Lamport timestamps maintain total order and causality

What is the number of messages required for each *passage* through the critical section?

2(M-1) messages.

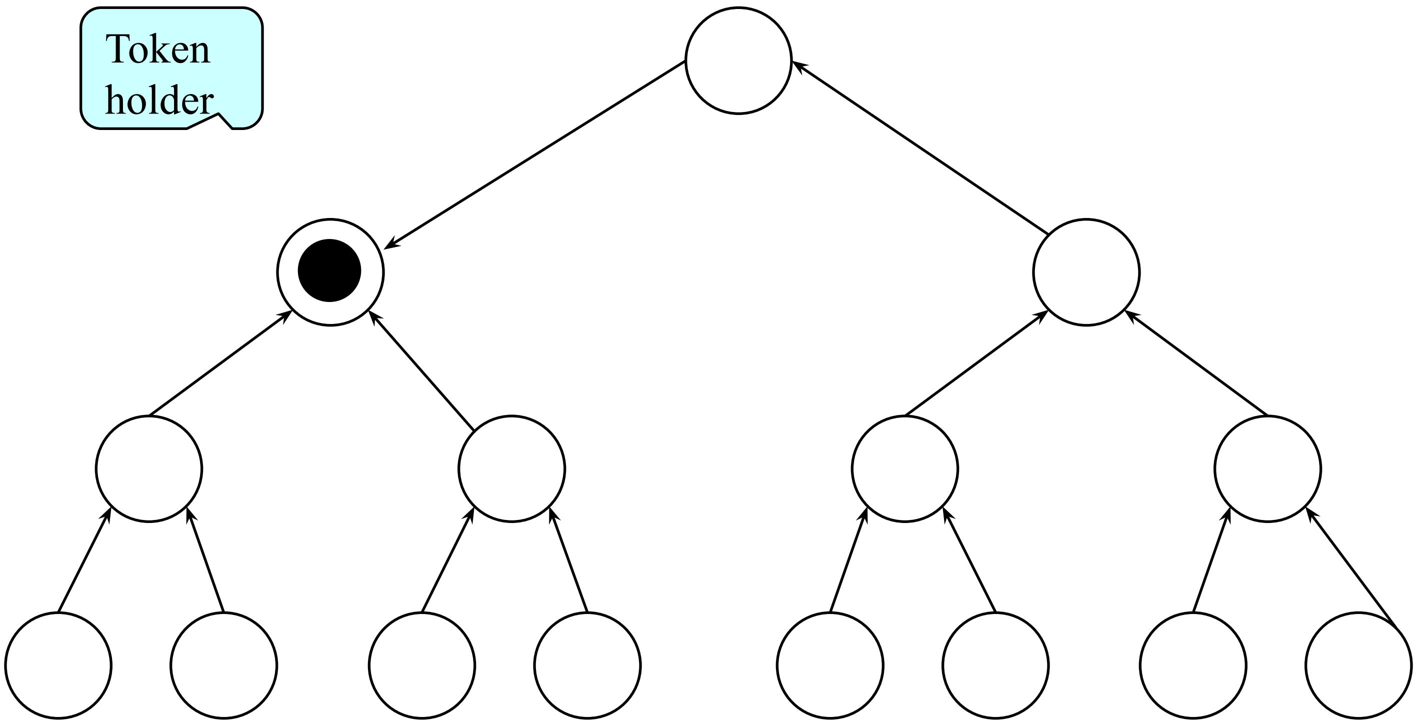
Next, we'll see a more message-efficient algorithm…

Raymond's algorithm: high-level ideas

- There is a single token in the system
  - Only the holder of the token may enter the CS
- Processors communicate by using a static tree structure
  - Requests for the token are sent along tree edges
  - The token itself is sent when available and requested
- Processors maintain FIFO request queues to prevent starvation
- At most a logarithmic number of messages per entry

Raymond's algorithm: high-level ideas (cont'd)

- Algorithm invariant: tree is always oriented towards token holder



Raymond's algorithm: data-structures

<u>Per processor variables</u>

Boolean *token_holder*

Boolean *inCS*

current_dir

requests_queue

True iff this processor currently holds the token

Raymond's algorithm: data-structures

Per processor variables

Boolean *token_holder*

Boolean *inCS*

current_dir

requests_queue

True iff this processor is currently in the critical section

Raymond's algorithm: data-structures

Per processor variables

Boolean *token_holder*

Boolean *inCS*

current_dir

requests_queue

The neighbor that is in the direction of the token (or *self* if this processor holds the token)

Raymond's algorithm: data-structures

Per processor variables

Boolean *token_holder*

Boolean i*nCS*

current_dir

requests_queue

FIFO queue holding IDs of neighbors from which requests for the token arrived (may also contain *self*)

Raymond's algorithm: entry and exit code

Request_CS:

```
If not token_holder
    if requests_queue.isEmpty( )
        send(current_dir, REQUEST)
    requests_queue.enqueue(self)
    wait until token_holder is true
inCS □ true
```

Release_CS:

```
inCS □ false
If not requests_queue.isEmpty( )
    current_dir □ requests_queue.dequeue( )
    send(current_dir, TOKEN)
    token_holder □ false
    if not requests_queue.isEmpty( )
        send(current_dir, REQUEST)
```

If this processor currently holds the token it immediately enters CS. Otherwise…

Raymond's algorithm: entry and exit code

```
Request_CS:
If not token_holder
   if requests_queue.isEmpty( )
      send(current_dir, REQUEST)
   requests_queue.enqueue(self)
   wait until token_holder is true
inCS ← true

Release_CS:
inCS ← false
If not requests_queue.isEmpty( )
   current_dir ← requests_queue.dequeue( )
   send(current_dir, TOKEN)
   token_holder ← false
   if not requests_queue.isEmpty( )
      send(current_dir, REQUEST)
```

If requests queue is empty, send a request for the token. (If queue is non-empty, a request for the token was already sent.)

Raymond's algorithm: entry and exit code

```
Request_CS:
If not token_holder
    if requests_queue.isEmpty( )
        send(current_dir, REQUEST)
    requests_queue.enqueue(self)
    wait until token_holder is true
inCS ← true

Release_CS:
inCS ← false
If not requests_queue.isEmpty( )
    current_dir ← requests_queue.dequeue( )
    send(current_dir, TOKEN)
    token_holder ← false
    if not requests_queue.isEmpty( )
        send(current_dir, REQUEST)
```

Enqueue ‘self’ to requests queue since this request is on behalf of this processor

Raymond's algorithm: entry and exit code

```
Request_CS:
If not token_holder
   if requests_queue.isEmpty( )
      send(current_dir, REQUEST)
   requests_queue.enqueue(self)
   wait until token_holder is true
inCS □ true

Release_CS:
inCS □ false
If not requests_queue.isEmpty( )
   current_dir □ requests_queue.dequeue( )
   send(current_dir, TOKEN)
   token_holder □ false
   if not requests_queue.isEmpty( )
      send(current_dir, REQUEST)
```

When *token_holder* is set, this processor has the token and may enter the CS

Raymond's algorithm: entry and exit code

```
Request_CS:
If not token_holder
   if requests_queue.isEmpty( )
      send(current_dir, REQUEST)
   requests_queue.enqueue(self)
   wait until token_holder is true
inCS □ true

Release_CS:
inCS □ false
If not requests_queue.isEmpty( )
   current_dir □ requests_queue.dequeue( )
   send(current_dir, TOKEN)
   token_holder □ false
   if not requests_queue.isEmpty( )
      send(current_dir, REQUEST)
```

No longer in critical section

Raymond's algorithm: entry and exit code

```
Request_CS:
If not token_holder
   if requests_queue.isEmpty( )
      send(current_dir, REQUEST)
   requests_queue.enqueue(self)
   wait until token_holder is true
inCS ← true

Release_CS:
inCS ← false
If not requests_queue.isEmpty( )
   current_dir ← requests_queue.dequeue( )
   send(current_dir, TOKEN)
   token_holder ← false
   if not requests_queue.isEmpty( )
      send(current_dir, REQUEST)
```

If requests are waiting…

Raymond's algorithm: entry and exit code

```
Request_CS:
If not token_holder
    if requests_queue.isEmpty( )
        send(current_dir, REQUEST)
    requests_queue.enqueue(self)
    wait until token_holder is true
inCS ← true

Release_CS:
inCS ← false
If not requests_queue.isEmpty( )
    current_dir ← requests_queue.dequeue( )
    send(current_dir, TOKEN)
    token_holder ← false
    if not requests_queue.isEmpty( )
        send(current_dir, REQUEST)
```

Dequeue the next hop for the earliest request and send the TOKEN to it. Also, update orientation of the token.

Raymond's algorithm: entry and exit code

```
Request_CS:
If not token_holder
    if requests_queue.isEmpty( )
        send(current_dir, REQUEST)
    requests_queue.enqueue(self)
    wait until token_holder is true
inCS □ true

Release_CS:
inCS □ false
If not requests_queue.isEmpty( )
    current_dir □ requests_queue.dequeue( )
    send(current_dir, TOKEN)
    token_holder □ false
    if not requests_queue.isEmpty( )
        send(current_dir, REQUEST)
```

This processor no longer holds token

Raymond's algorithm: entry and exit code

```
Request_CS:
If not token_holder
    if requests_queue.isEmpty( )
        send(current_dir, REQUEST)
    requests_queue.enqueue(self)
    wait until token_holder is true
inCS ← true

Release_CS:
inCS ← false
If not requests_queue.isEmpty( )
    current_dir ← requests_queue.dequeue( )
    send(current_dir, TOKEN)
    token_holder ← false
    if not requests_queue.isEmpty( )
        send(current_dir, REQUEST)
```

If there are more requests in this processor’s queue, send another request for the token

Raymond's algorithm: monitoring

```
Monitor_CS:
while (true)
  wait for a REQUEST or a TOKEN message
    REQUEST
      if token_holder
        if inCS
          requests_queue.enqueue(sender)
        else
          current_dir □ sender
          send(current_dir, TOKEN)
          token_holder □ false
      else
        if requests_queue.isEmpty()
          send(current_dir,REQUEST)
        requests_queue.enqueue(sender)
  TOKEN
      current_dir □ requests_queue.dequeue( )
      if current_dir = self
        token_holder □ true
      else
        send(current_dir, TOKEN)
        if not requests_queue.isEmpty( )
          send(current_dir, REQUEST)
```

Listener thread to respond to protocol messages at all times

Raymond's algorithm: monitoring

```
Monitor_CS:
while (true)
  wait for a REQUEST or a TOKEN message
  REQUEST
    if token_holder
      if inCS
        requests_queue.enqueue(sender)
      else
        current_dir ← sender
        send(current_dir, TOKEN)
        token_holder ← false
    else
      if requests_queue.isEmpty()
        send(current_dir,REQUEST)
      requests_queue.enqueue(sender)
  TOKEN
    current_dir ← requests_queue.dequeue( )
    if current_dir = self
      token_holder ← true
    else
      send(current_dir, TOKEN)
      if not requests_queue.isEmpty( )
        send(current_dir, REQUEST)
```

Upon a request.
If current processor holds token…

Raymond's algorithm: monitoring

```
Monitor_CS:
while (true)
  wait for a REQUEST or a TOKEN message
  REQUEST
   if token_holder
     if inCS
       requests_queue.enqueue(sender)
     else
       current_dir □ sender
       send(current_dir, TOKEN)
       token_holder □ false
   else
     if requests_queue.isEmpty()
       send(current_dir,REQUEST)
     requests_queue.enqueue(sender)
TOKEN
   current_dir □ requests_queue.dequeue( )
   if current_dir = self
     token_holder □ true
   else
     send(current_dir, TOKEN)
     if not requests_queue.isEmpty( )
       send(current_dir, REQUEST)
```

If current processor in CS then request must wait, enqueue the direction of requesting processor

Raymond's algorithm: monitoring

```
Monitor_CS:
while (true)
  wait for a REQUEST or a TOKEN message
  REQUEST
  if token_holder
      if inCS
          requests_queue.enqueue(sender)
      else
          current_dir □ sender
          send(current_dir, TOKEN)
          token_holder □ false
  else
      if requests_queue.isEmpty()
          send(current_dir,REQUEST)
      requests_queue.enqueue(sender)
TOKEN
  current_dir □ requests_queue.dequeue( )
  if current_dir = self
      token_holder □ true
  else
      send(current_dir, TOKEN)
      if not requests_queue.isEmpty( )
          send(current_dir, REQUEST)
```

Otherwise current processor holds the token but is not in CS, hence requests queue is empty.

Send token to where the request came from, mark that current processor no longer holds token, and the new orientation of the token…

Raymond's algorithm: monitoring

```
Monitor_CS:
  while (true)
    wait for a REQUEST or a TOKEN message
    REQUEST
     if token_holder
        if inCS
           requests_queue.enqueue(sender)
        else
           current_dir ← sender
           send(current_dir, TOKEN)
           token_holder ← false
     else
        if requests_queue.isEmpty()
           send(current_dir,REQUEST)
        requests_queue.enqueue(sender)
  TOKEN
     current_dir ← requests_queue.dequeue( )
     if current_dir = self
        token_holder ← true
     else
        send(current_dir, TOKEN)
        if not requests_queue.isEmpty( )
           send(current_dir, REQUEST)
```

Otherwise current processor does not hold the token…

Raymond's algorithm: monitoring

```
Monitor_CS:
  while (true)
    wait for a REQUEST or a TOKEN message
    REQUEST
      if token_holder
        if inCS
          requests_queue.enqueue(sender)
        else
          current_dir ← sender
          send(current_dir, TOKEN)
          token_holder ← false
      else
        if requests_queue.isEmpty()
          send(current_dir,REQUEST)
        requests_queue.enqueue(sender)
    TOKEN
      current_dir ← requests_queue.dequeue( )
      if current_dir = self
        token_holder ← true
      else
        send(current_dir, TOKEN)
        if not requests_queue.isEmpty( )
          send(current_dir, REQUEST)
```

If requests queue is empty, send request in the direction of the token…

Raymond's algorithm: monitoring

```
Monitor_CS:
while (true)
  wait for a REQUEST or a TOKEN message
  REQUEST
   if token_holder
      if inCS
         requests_queue.enqueue(sender)
      else
          current_dir □ sender
          send(current_dir, TOKEN)
          token_holder □ false
   else
      if requests_queue.isEmpty()
         send(current_dir,REQUEST)
      requests_queue.enqueue(sender)
TOKEN
   current_dir □ requests_queue.dequeue( )
    if current_dir = self
      token_holder □ true
     else
      send(current_dir, TOKEN)
      if not requests_queue.isEmpty( )
          send(current_dir, REQUEST)
```

Enqueue the direction of this request…

Raymond's algorithm: monitoring

```
Monitor_CS:
while (true)
  wait for a REQUEST or a TOKEN message
  REQUEST
   if token_holder
      if inCS
         requests_queue.enqueue(sender)
      else
         current_dir □ sender
         send(current_dir, TOKEN)
         token_holder □ false
   else
      if requests_queue.isEmpty()
         send(current_dir,REQUEST)
      requests_queue.enqueue(sender)
 TOKEN
   current_dir □ requests_queue.dequeue( )
    if current_dir = self
       token_holder □ true
    else
       send(current_dir, TOKEN)
       if not requests_queue.isEmpty( )
          send(current_dir, REQUEST)
```

Upon the arrival of the token…

Raymond's algorithm: monitoring

```
Monitor_CS:
while (true)
  wait for a REQUEST or a TOKEN message
  REQUEST
   if token_holder
       if inCS
          requests_queue.enqueue(sender)
       else
          current_dir □ sender
          send(current_dir, TOKEN)
          token_holder □ false
   else
       if requests_queue.isEmpty()
          send(current_dir,REQUEST)
       requests_queue.enqueue(sender)
TOKEN
   current_dir □ requests_queue.dequeue( )
    if current_dir = self
        token_holder □ true
     else
        send(current_dir, TOKEN)
        if not requests_queue.isEmpty( )
           send(current_dir, REQUEST)
```

Dequeue oldest request and set new orientation of the token to its direction

Raymond's algorithm: monitoring

```
Monitor_CS:
while (true)
  wait for a REQUEST or a TOKEN message
  REQUEST
   if token_holder
      if inCS
         requests_queue.enqueue(sender)
      else
         current_dir ← sender
         send(current_dir, TOKEN)
         token_holder ← false
   else
      if requests_queue.isEmpty()
         send(current_dir,REQUEST)
      requests_queue.enqueue(sender)
 TOKEN
   current_dir ← requests_queue.dequeue( )
   if current_dir = self
      token_holder ← true
   else
      send(current_dir, TOKEN)
      if not requests_queue.isEmpty( )
         send(current_dir, REQUEST)
```

If request was by this processor, mark that it currently has the token and may enter the CS

Raymond's algorithm: monitoring

```
Monitor_CS:
while (true)
  wait for a REQUEST or a TOKEN message
  REQUEST
   if token_holder
      if inCS
         requests_queue.enqueue(sender)
      else
         current_dir □ sender
         send(current_dir, TOKEN)
         token_holder □ false
   else
      if requests_queue.isEmpty()
         send(current_dir,REQUEST)
      requests_queue.enqueue(sender)
TOKEN
   current_dir □ requests_queue.dequeue( )
    if current_dir = self
       token_holder □ true
    else
       send(current_dir, TOKEN)
       if not requests_queue.isEmpty( )
          send(current_dir, REQUEST)
```

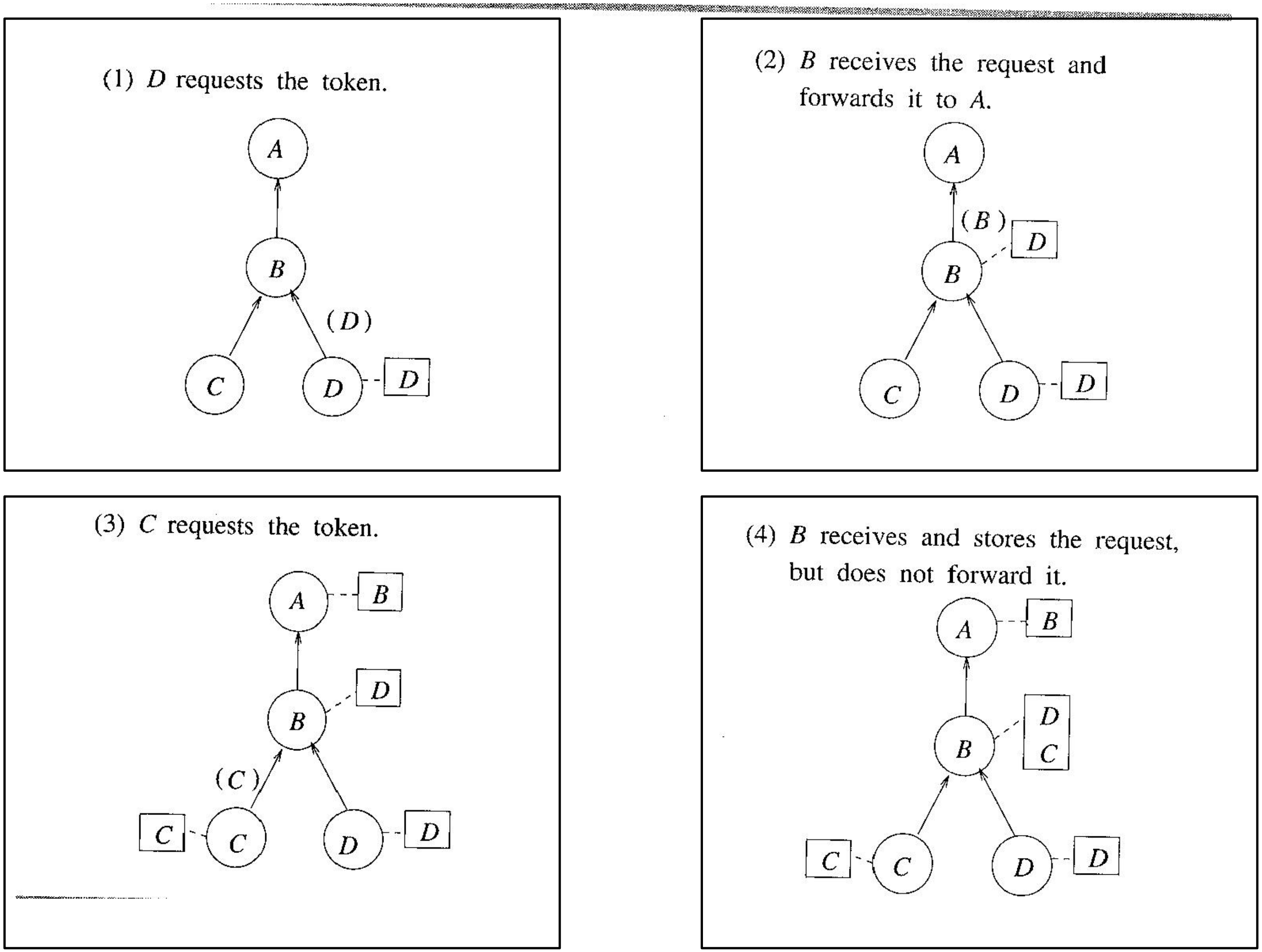
Otherwise, send the token in the direction of the request

Raymond's algorithm: monitoring

```
Monitor_CS:
while (true)
  wait for a REQUEST or a TOKEN message
  REQUEST
    if token_holder
        if inCS
            requests_queue.enqueue(sender)
        else
            current_dir □ sender
            send(current_dir, TOKEN)
            token_holder □ false
    else
        if requests_queue.isEmpty()
            send(current_dir,REQUEST)
        requests_queue.enqueue(sender)
TOKEN
    current_dir □ requests_queue.dequeue( )
    if current_dir = self
        token_holder □ true
    else
        send(current_dir, TOKEN)
        if not requests_queue.isEmpty( )
            send(current_dir, REQUEST)
```

If the queue is non-empty, send another request for the token

Raymond's algorithm: execution scenario

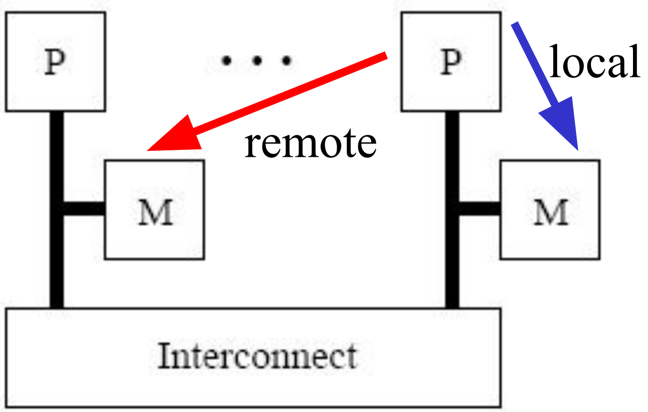


Distributed Synchronization: outline

- Introduction
- Causality and time
  - Lamport timestamps
  - Vector timestamps
  - Causal communication
- Snapshots
- Distributed Mutual Exclusion
  - Ricart and Agrawala's algorithm
  - Raymond's algorithm
  - The MCS algorithm

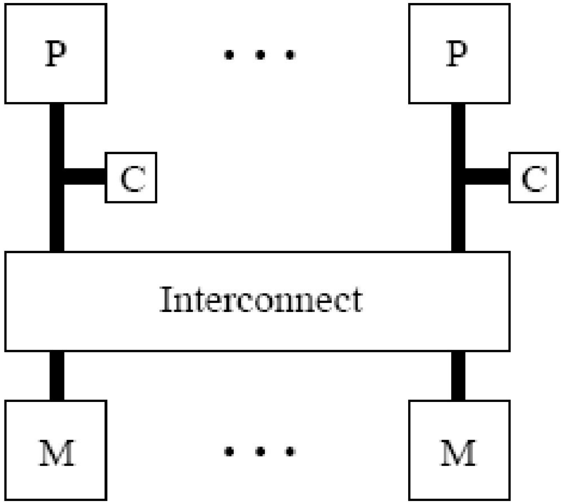
Remote and local memory references

In a Distributed Shared-memory (DSM) system:



In a Cache-coherent system:

An access of v by p is <u>remote</u> if it is the first access of v <u>or</u> if v has been written by another process since p's last access of it.



Local spin algorithms

- In a <u>local-spin algorithm</u>, all busy waiting ('await') is done by read-only loops of local-accesses, that do not cause interconnect traffic.
- The same algorithm may be local-spin on one architecture (DSM or CC) and non-local spin on the other.

For local-spin algorithms, the complexity metric is the <u>worst-case number of Remote Memory References (**RMR**s)</u>

Peterson's 2-process algorithm

Program for process 0

1. b[0]:=true
2. turn:=0
3. await (b[1]=false or turn=1)
4. CS
5. b[0]:=false

Program for process 1

1. b[1]:=true
2. turn:=1
3. await (b[0]=false or turn=0)
4. CS
5. b[1]:=false

Is this algorithm local-spin on a DSM machine? No

Is this algorithm local-spin on a CC machine? Yes

The MCS queue-based algorithm

- Mellor-Crummey and Scott (1991)
- Uses Read, Write, Swap, and Compare-And-Swap (CAS) operations
- Provides starvation-freedom and FIFO
- O(1) RMRs per passage in both CC/DSM
- Widely used in practice (also in Linux)

Swap & compare-and-swap

```
Swap(w, new)
do atomically
   prev:=w
   w:=new
   return prev
```

```
Compare-and-swap(w, old, new)
do atomically
   if w = old
      w:=new
      return true
   else
      return false
```

The MCS algorithm

Qnode: structure {bit locked, Qnode *next}
shared Qnode *myNode, Qnode *tail initially null

Program for process i

```
myNode->next := null; prepare to be last in queue
pred=swap(&tail, myNode ) ;tail now points to myNode
if (pred ≠ null) ;I need to wait for a predecessor
  myNode->locked := true ;prepare to wait
  pred->next := myNode  ;let my predecessor know it has to unlock me
  await myNode->locked := false
CS
if (myNode->next = null) ; if not sure there is a successor
  if (compare-and-swap(&tail, myNode, null) = false) ; if there is a successor
    await (myNode->next ≠ null) ; spin until successor lets me know its identity
    successor := myNode->next ; get a pointer to my successor
    successor->locked := false ; unlock my successor
else ; for sure, I have a successor
    successor := myNode->next ; get a pointer to my successor
    successor->locked := false ; unlock my successor
```

The MCS algorithm

Qnode: structure {bit locked, Qnode *next}
shared Qnode *myNode, Qnode *tail initially null

Program for process i

```
myNode->next := null; prepare to be last in queue
pred=swap(&tail, myNode ) ;tail now points to myNode
if (pred ≠ null) ;I need to wait for a predecessor
  myNode->locked := true ;prepare to wait
  pred->next := myNode  ;let my predecessor know it has to unlock me
  await myNode->locked := false
CS
if (myNode->next = null) ; if not sure there is a successor
  if (compare-and-swap(&tail, myNode, null) = false) ; if there is a successor
    await (myNode->next ≠ null) ; spin until successor lets me know its identity
    successor := myNode->next ; get a pointer to my successor
    successor->locked := false ; unlock my successor
else ; for sure, I have a successor
    successor := myNode->next ; get a pointer to my successor
    successor->locked := false ; unlock my successor
```

The MCS algorithm

Qnode: structure {bit locked, Qnode *next}
shared Qnode *myNode, Qnode *tail initially null

Program for process i

```
myNode->next := null; prepare to be last in queue
pred=swap(&tail, myNode ) ;tail now points to myNode
if (pred ≠ null) ;I need to wait for a predecessor
  myNode->locked := true ;prepare to wait
  pred->next := myNode  ;let my predecessor know it has to unlock me
  await myNode->locked := false
CS
if (myNode->next = null) ; if not sure there is a successor
  if (compare-and-swap(&tail, myNode, null) = false) ; if there is a successor
    await (myNode->next ≠ null) ; spin until successor lets me know its identity
    successor := myNode->next ; get a pointer to my successor
    successor->locked := false ; unlock my successor
else ; for sure, I have a successor
    successor := myNode->next ; get a pointer to my successor
    successor->locked := false ; unlock my successor
```

The MCS algorithm

Qnode: structure {bit locked, Qnode *next}
shared Qnode *myNode, Qnode *tail initially null

Program for process i

```
myNode->next := null; prepare to be last in queue
pred=swap(&tail, myNode ) ;tail now points to myNode
if (pred ≠ null) ;I need to wait for a predecessor
   myNode->locked := true ;prepare to wait
   pred->next := myNode  ;let my predecessor know it has to unlock me
   await myNode->locked := false
CS
if (myNode->next = null) ; if not sure there is a successor
   if (compare-and-swap(&tail, myNode, null) = false) ; if there is a successor
      await (myNode->next ≠ null) ; spin until successor lets me know its identity
      successor := myNode->next ; get a pointer to my successor
      successor->locked := false ; unlock my successor
else ; for sure, I have a successor
      successor := myNode->next ; get a pointer to my successor
      successor->locked := false ; unlock my successor
```

The MCS algorithm

Qnode: structure {bit locked, Qnode *next}
shared Qnode *myNode, Qnode *tail initially null

Program for process i

```
myNode->next := null; prepare to be last in queue
pred=swap(&tail, myNode ) ;tail now points to myNode
if (pred ≠ null) ;I need to wait for a predecessor
  | myNode->locked := true ;prepare to wait
  | pred->next := myNode  ;let my predecessor know it has to unlock me
  | await myNode->locked := false
CS
if (myNode->next = null) ; if not sure there is a successor
  | if (compare-and-swap(&tail, myNode, null) = false) ; if there is a successor
  |    await (myNode->next ≠ null) ; spin until successor lets me know its identity
  |    successor := myNode->next ; get a pointer to my successor
  |    successor->locked := false ; unlock my successor
else ; for sure, I have a successor
     successor := myNode->next ; get a pointer to my successor
     successor->locked := false ; unlock my successor
```

MCS: execution scenario

Figure 6: An example execution of ALGORITHM MCS.