# Outline for January 23, 2001

1. Greetings and felicitations!
   a. No class on January 25 or January 30; no office hours on Wednesday, January 24 or Monday, January 29
   b. Extra office hour: Friday January 26: 11 AM–1 PM
2. Distributed system?
   a. What is it?
   b. Why use it?
3. System Architectures
   a. minicomputer mode
   b. workstation model
   c. processor pool
4. Issues
   a. global knowledge
   b. naming
   c. scalability
   d. compatibility
   e. process synchronization, communication
   f. security
   g. structure
5. Networks
   a. goals
   b. message, packet, subnet, session
   c. switching: circuit, store-and-forward, message, packet, virtual circuit, dynamic routing
   d. OSI model: PDUs, layering
      i. physical: ethernet, aloha, *etc*.
      ii. data link layer: frames, parity checks, link encryption
      iii. network layer: virtual circult vs. datagram, routing via flooding, static routes, dynamic routes, centralized routing vs. distributed routing; congestion solutions (packet discarding, isarithmic, choke packets)
      iv. transport: services provided (UDP vs. TCP), functions to higher layers, addressing schemes (flat, DNS, etc.), gateway fragmentation and reassembly
      v. session: adds session characteristics like authentication
      vi. presentation: compression, end-to-end encryption, virtual terminal
      vii. application: user-level programs
6. Clocks
   a. happened-before relation
   b. Lamport's distributed clocks: $a \rightarrow b$ means $C(a) < C(b)$
   c. Example where $C(a) < C(b)$ does *not* mean $a \rightarrow b$
   d. Vector clocks and causal relation
   e. ordering of messages so you receive them in the order sent
      i. why
      ii. for broadcast (ISIS): Birman-Schiper-Stephenson
      iii. for point to point: Schiper-Eggli-Sandoz
7. Global state
   a. Show problem of slicing state when something is in transit
   b. Define local state; $send(m_{ij}) \in LS_i$ iff time of $send(m_{ij})$ < current time of $LS_i$; similar for receive
   c. transit($LS_i$, $LS_j$); inconsistent($LS_i$, $LS_j$); consistent state is one with inconsistent set empty for all pairs $LS_i$, $LS_j$
   d. Consistent global state: Chandry-Lamport
8. Termination detection
   a. Haung

# Lamport's Clocks

## Introduction

Lamport's clocks keep a virtual time among distributed systems. The goal is to provide an ordering upon events within the system.

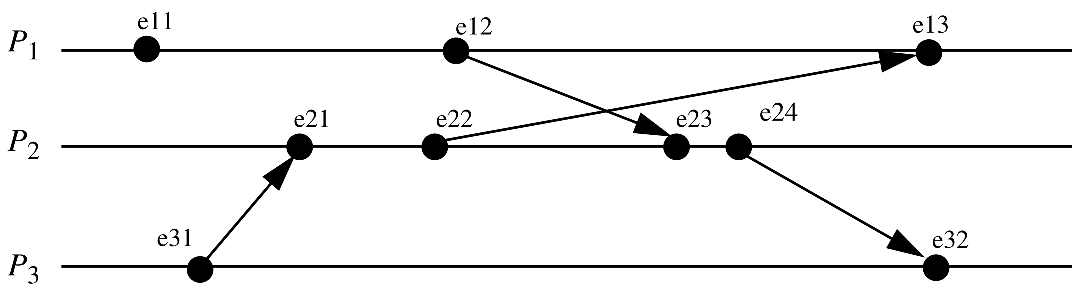
## Notation

- $P_i$ process
- $C_i$. clock associated with process $P_i$

## Protocol

1. Increment clock $C_i$ between any two successive events in process $P_i$: $C_i \leftarrow C_i + d$ ($d > 0$)
2. Let event $a$ be the sending of a message by process $P_i$; the timestamp is $t^a = C_i(a)$ *after* the clock is incremented.
   Let $b$ be the receipt of that message by $P_j$. Then when $P_j$ receives the message, $C_j \leftarrow \max(C_j, t^a + d)$ ($d > 0$)

## Example




Assume all clocks start at 0, and $d$ is 1 (that is, each event incrememts the clock by 1). The events and clocks are:
$e_{11}$: $C_1 \leftarrow 1$

$e_{31}$: $C_3 \leftarrow 1$; timestamp $t^{3,1}$ of message is 1

$e_{21}$: $C_2 \leftarrow 2$ as $t^{3,1} =$ 1, after the increment $C_2 \leftarrow 1$, and $C_2 \leftarrow \max(C_2, t^{3,1} + 1) = \max(1, 1 + 1) = \max(1, 2) = 2$

$e_{22}$: $C_2 \leftarrow 3$; timestamp $t^{2,2}$ of message is 3

$e_{12}$: $C_1 \leftarrow 2$; timestamp $t^{1,2}$ of message is 2

$e_{23}$: $C_2 \leftarrow 4$ as $t^{1,2} =$ 2, after the increment $C_2 \leftarrow 4$, and $C_2 \leftarrow \max(C_2, t^{1,2} + 1) = \max(4, 2 + 1) = \max(4, 2) = 4$

$e_{24}$: $C_2 \leftarrow 5$; timestamp $t^{2,4}$ of message is 5

$e_{13}$: $C_1 \leftarrow 4$ as $t^{2,2} =$ 3, after the increment $C_1 \leftarrow 3$, and $C_1 \leftarrow \max(C_1, t^{2,2} + 1) = \max(3, 3 + 1) = \max(3, 4) = 4$

$e_{32}$: $C_3 \leftarrow 6$ as $t^{2,4} =$ 5, after the increment $C_3 \leftarrow 2$, and $C_3 \leftarrow \max(C_3, t^{2,4} + 1) = \max(2, 5 + 1) = \max(2, 6) = 6$

## Problem

Clearly, if a $\rightarrow$ b, then $C(a) < C(b)$. But if $C(a) < C(b)$, does $a \rightarrow b$?
The answer, surprisingly, is not necessarily. In the above example, $C_3(e_{31}) = 1 < 2 = C_1(e_{12})$. But $e_{31}$ and $e_{12}$ are causally unrelated; that is, $e_{31} \not\rightarrow e_{12}$. However, $C_1(e_{11}) = 1 < 6 = C_3(e_{32})$, and clearly $e_{11} \rightarrow e_{32}$. Hence one cannot say one way or the other.

# Vector Clocks

## Introduction

This is based upon Lamport's clocks, but each process keeps track of what is believes the other processes' interrnal clocks are (hence the name, vector clocks). The goal is to provide an ordering upon events within the system.

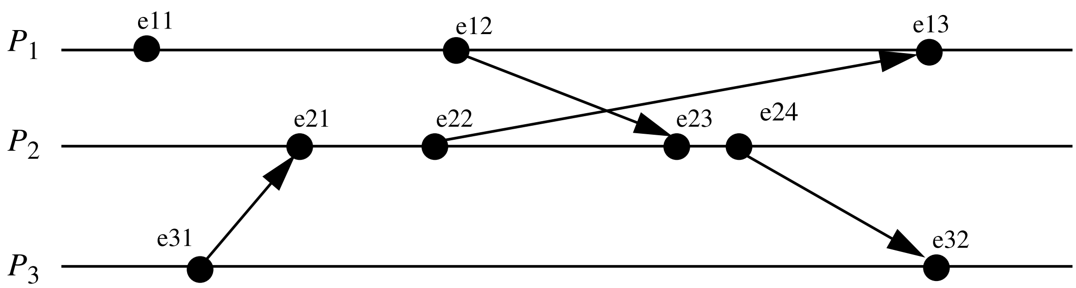
## Notation

- $n$ processes
- $P_i$ process
- $C_i$. vector clock associated with process $P_i$; $j$th element is $C_i[j]$ and contains $P_i$'s latest value for the current time in process $P_k$.

## Protocol

1. Increment clock $C_i$ between any two successive events in process $P_i$: $C_i[i] \leftarrow C_i[i] + d$ ($d > 0$)
2. Let event $a$ be the sending of a message by process $P_i$; its vector timestamp is $t^a = C_i(a)$ *after* the clock is incremented.. Let $b$ be the receipt of that message by $P_j$. Then when $P_j$ receives the message, it updates its vector clock for all $k = 1, \ldots, n$: $C_j[k] \leftarrow \max(C_j[k], t^a[k])$

## Example




Here is the progression of time for the three processes:

$e_{11}$: $C_1 \leftarrow (1, 0, 0)$

$e_{31}$: $C_3 \leftarrow (0, 0, 1)$; timestamp $t^{3,1}$ of message is $(0, 0, 1)$

$e_{21}$: $C_2 \leftarrow (0, 1, 1)$ as $t^{3,1} = (0, 0, 1)$, $C_2 \leftarrow (0, 1, 0)$ after the increment, $C_2[0] \leftarrow \max(C_2[0], t^{3,1}[0]) = \max(0, 0) = 0$, $C_2[1] \leftarrow \max(C_2[1], t^{3,1}[1]) = \max(1, 0) = 1$, and $C_2[2] \leftarrow \max(C_2[2], t^{3,1}[2]) = \max(0, 1) = 1$

$e_{22}$: $C_2 \leftarrow (0, 2, 1)$; timestamp $t^{2,1}$ of message is $(0, 2, 1)$

$e_{12}$: $C_1 \leftarrow (2, 0, 0)$; timestamp $t^{1,1}$ of message is $(2, 0, 0)$

$e_{23}$: $C_2 \leftarrow (2, 3, 1)$ as $t^{1,1} = (2, 0, 0)$, $C_2 \leftarrow (0, 3, 1)$ after the increment, $C_2[0] \leftarrow \max(C_2[0], t^{1,1}[0]) = \max(0, 2) = 2$, $C_2[1] \leftarrow \max(C_2[1], t^{1,1}[1]) = \max(3, 0) = 3$, and $C_2[2] \leftarrow \max(C_2[2], t^{1,1}[2]) = \max(0, 1) = 1$

$e_{24}$: $C_2 \leftarrow (2, 3, 1)$; timestamp $t^{2,2}$ of message is $(2, 3, 1)$

$e_{13}$: $C_1 \leftarrow (3, 2, 1)$ as $t^{2,1} = (0, 2, 1)$, $C_1 \leftarrow (3, 0, 0)$ after the increment, $C_1[0] \leftarrow \max(C_1[0], t^{2,1}[0]) = \max(3, 0) = 3$, $C_1[1] \leftarrow \max(C_1[1], t^{2,1}[1]) = \max(2, 0) = 2$, and $C_1[2] \leftarrow \max(C_1[2], t^{2,1}[2]) = \max(1, 0) = 1$

$e_{32}$: $C_3 \leftarrow (2, 3, 2)$ as $t^{2,2} = (2, 3, 1)$, $C_3 \leftarrow (0, 0, 2)$ after the increment, $C_3[0] \leftarrow \max(C_3[0], t^{2,2}[0]) = \max(2, 0) = 2$, $C_3[1] \leftarrow \max(C_3[1], t^{2,2}[1]) = \max(3, 0) = 3$, and $C_3[2] \leftarrow \max(C_3[2], t^{2,2}[2]) = \max(1, 2) = 2$

Notice that $C_1(e_{11}) < C_3(e_{32})$, so $e_{11} \rightarrow e_{32}$, but $C_1(e_{11})$ and $C_3(e_{31})$ are incomparable, so $e_{11}$ and $e_{31}$ are concurrent.

# Birman-Schiper-Stephenson Protocol

## Introduction

The goal of this protocol is to preserve ordering in the sending of messages. For example, if *send*($m_1$) → send($m_2$), then for all processes that receive both $m_1$ and $m_2$, *receive*($m_1$)→ *receive*($m_2$). The basic idea is that $m_2$ is not given to the process until $m_1$ is given. This means a buffer is needed for pending deliveries. Also, each message has an associated vector that contains information for the recipient to determine if another message preceded it. Also, we shall assume all messages are broadcast. Clocks are updated only when messages are sent.

## Notation

- $n$ processes
- $P_i$ process
- $C_i$. vector clock associated with process $P_i$; $j$th element is $C_i[j]$ and contains $P_i$'s latest value for the current time in process $P_k$.
- $t^m$ vector timestamp for message $m$ (stamped *after* local clock is incremented)

## Protocol

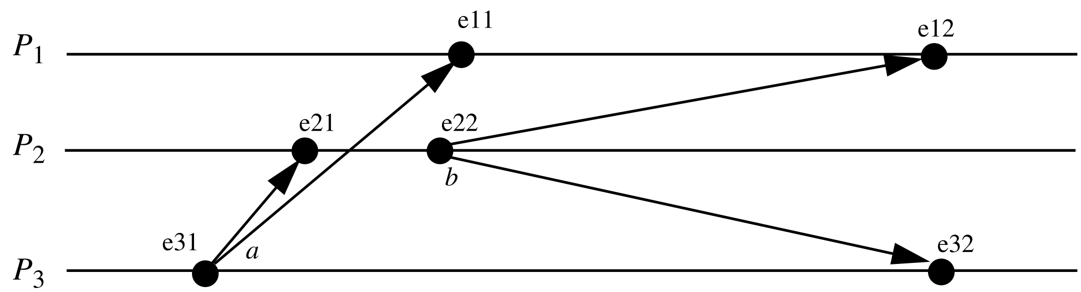
***$P_i$* broadcasts a message**

1. $P_i$ increments $C_i[i]$ and sets the timestamp $t^m = C_i$ for message $m$.

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

1. When $P_j$, $j \neq i$, receives $m$ with timestamp $t^m$, it delays the message's delivery until *both*:
   a. $C_j[i] = t^m[i] - 1$; and
   b. for all $k \leq n$ and $k \neq i$, $C_j[k] \geq t^m[k]$.
2. When the message is delivered to $P_j$, update $P_j$'s vector clock for all $k = 1, \ldots, n$: $C_j[k] \leftarrow \max(C_j[k], t^a[k])$
3. Check buffered messages to see if any can be delivered.

## Example




Here is the protocol applied to the above situation:

$e_{31}$: $P_3$ sends message $a$; $C_3 = (0, 0, 1)$; $t^a = (0, 0, 1)$

$e_{21}$: $P_2$ receives message $a$. As $C_2 = (0, 0, 0)$, $C_2[3] = t^a[3] - 1 = 1 - 1 = 0$ and $C_2[1] \geq t^a[1]$ and $C_2[2] \geq t^a[2] = 0$. So the message is accepted, and $C_2$ is set to (0, 0, 1)

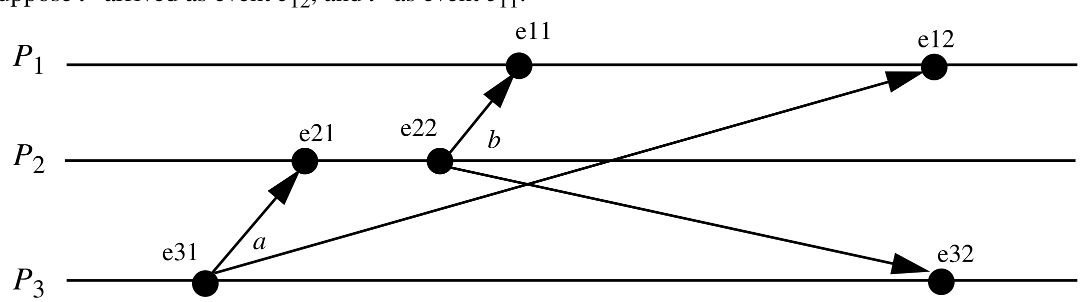
$e_{22}$: $P_2$ sends message $b$; $C_2 = (0, 1, 1)$; $t^b = (0, 1, 1)$

$e_{11}$: $P_1$ receives message $a$. As $C_1 = (0, 0, 0)$, $C_1[3] = t^a[3] - 1 = 1 - 1 = 0$ and $C_1[1] \geq t^a[1]$ and $C_1[2] \geq t^a[2] = 0$. So the message is accepted, and $C_1$ is set to (0, 0, 1)

$e_{12}$: $P_1$ receives message $b$. As $C_1 = (0, 0, 1)$, $C_1[2] = t^b[2] - 1 = 1 - 1 = 0$ and $C_1[1] \geq t^b[1]$ and $C_1[3] \geq t^b[2] = 0$. So the message is accepted, and $C_1$ is set to (0, 1, 1)

$e_{32}$: $P_3$ receives message $b$. As $C_3 = (0, 0, 1)$, $C_3[2] = t^b[2] - 1 = 1 - 1 = 1$ and $C_1[1] \geq t^b[1]$ and $C_1[3] \geq t^b[2] = 0$. So the message is accepted, and $C_3$ is set to (0, 1, 1)

Now, suppose $t^a$ arrived as event $e_{12}$, and $t^b$ as event $e_{11}$:




Then the progression of time in $P_1$ goes like this:

$e_{31}$: $P_3$ sends message $a$; $C_3 = (0, 0, 1)$; $t^a = (0, 0, 1)$

$e_{21}$: $P_2$ receives message $a$. As $C_2 = (0, 0, 0)$, $C_2[3] = t^a[3] - 1 = 1 - 1 = 0$ and $C_2[1] \geq t^a[1]$ and $C_2[2] \geq t^a[2] = 0$. So the message is accepted, and $C_2$ is set to (0, 0, 1)

$e_{22}$: $P_2$ sends message $b$; $C_2 = (0, 1, 1)$; $t^b = (0, 1, 1)$

$e_{11}$: $P_1$ receives message $b$. As $C_1 = (0, 0, 0)$, $C_1[2] = t^b[2] - 1 = 1 - 1 = 0$ and $C_1[1] \geq t^b[1]$, but $C_1[3] < t^b[3]$, so the message is held until another message arrives. The vector clock updating algorithm is not run.

$e_{12}$: $P_1$ receives message $a$. As $C_1 = (0, 0, 0)$, $C_1[3] = t^a[3] - 1 = 1 - 1 = 0$, $C_1[1] \geq t^a[1]$, and $C_1[2] \geq t^a[2]$. The message is accepted and $C_1$ is set to (0, 0, 1). Now the queue is checked. As $C_1[2] = t^b[2] - 1 = 1 - 1 = 0$, $C_1[1] \geq t^b[1]$, and $C_1[3] \geq t^b[3]$, that message is accepted and $C_1$ is set to (0, 1, 1).

$e_{32}$: $P_3$ receives message $b$. As $C_3 = (0, 0, 1)$, $C_3[2] = t^b[2] - 1 = 1 - 1 = 1$ and $C_1[1] \geq t^b[1]$ and $C_1[3] \geq t^b[2] = 0$. So the message is accepted, and $C_3$ is set to (0, 1, 1)

# Schiper-Eggli-Sandoz Protocol

## Introduction

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

## Notation

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

## Protocol

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

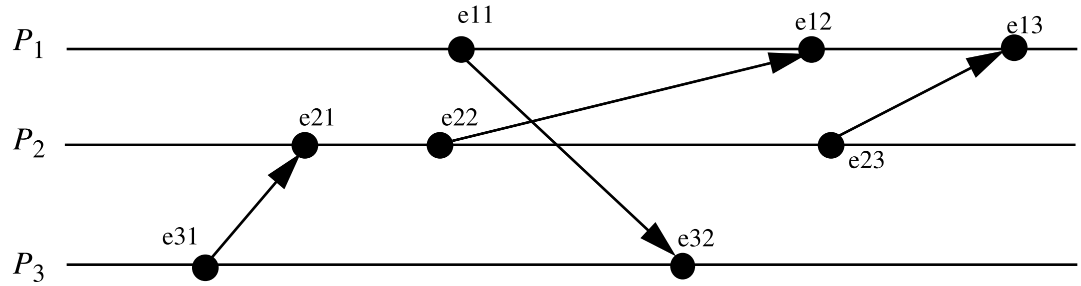
1. $P_i$ sends message $m$, timestamped $t^m$, and $V_i$, to process $P_j$ .
2. $P_i$ sets $V_i[j] \leftarrow t^m$.

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

1. When $P_j$, $j \neq i$, receives $m$, it delays the message's delivery if *both*:
   a. $V^m[j]$ is set; and
   b. $V^m[j] < t^j$

   Otherwise it is queued for later delivery.
2. If the message can be delivered to $P_j$, the following three actions occur:
   a. Update all set elements of $V_j$ with the corresponding elements of $V^m$, except for $V_j[j]$, as follows:
      i. If $V_j[k]$ and $V^m[k]$ are uninitialized, do nothing.
      ii. If $V_j[k]$ is uninitialized and $V^m[k]$ is initialized, set $V_j[k] \leftarrow V^m[k]$.
      iii. If both $V_j[k]$ and $V^m[k]$ are initialized, $V_j[k][k'] \leftarrow \max(V_j[k][k'], V^m[k][k'])$ for all $k' = 1, \ldots, n$
3. Update $P_j$'s vector clock for all $k = 1, \ldots, n$: $C_j[k] \leftarrow \max(C_j[k], t^m[k])$
4. Check buffered messages to see if any can be delivered.

## Example




Here is the protocol applied to the above situation. [ ... ] and { ... } are like ( ... ) but used when too many parentheses would be confusing (to me, at any rate!):

$e_{31}$: $P_3$ sends message $m_{3,1}$ to $P_2$. $C_3 = (0, 0, 1)$; $t^{3,1} \leftarrow (0, 0, 1)$, $V^{3,1} \leftarrow (?, ?, ?)$; $V_3 \leftarrow [\ ?, (0, 0, 1), ?\ ]$

$e_{21}$: $P_2$ receives message $m_{3,1}$ from $P_3$. As $V^{3,1}[2] = (?, ?, ?)[2]$ is uninitialized, the message is accepted.
$V_2 \leftarrow [\ ?, ?, ?\ ]$ and $C_2 \leftarrow max[(0, 0, 0), (0, 0, 1)] = (0, 0, 1)$

$e_{22}$: $P_2$ sends message $m_{2,1}$ to $P_1$. $C_2 \leftarrow (0, 1, 1)$; $t^{2,1} \leftarrow (0, 1, 1)$, $V^{2,1} \leftarrow [\ ?, ?, ?\ ]$; $V_2 \leftarrow [\ (0, 1, 1), ?, ?\ ]$

$e_{11}$: $P_1$ sends message $m_{1,1}$ to $P_3$. $C_1 \leftarrow (1, 0, 0)$; $t^{1,1} \leftarrow (1, 0, 0)$, $V^{1,1} \leftarrow (\ ?, ?, ?\ )$; $V_1 \leftarrow [\ ?, ?, (1, 0, 0)\ ]$

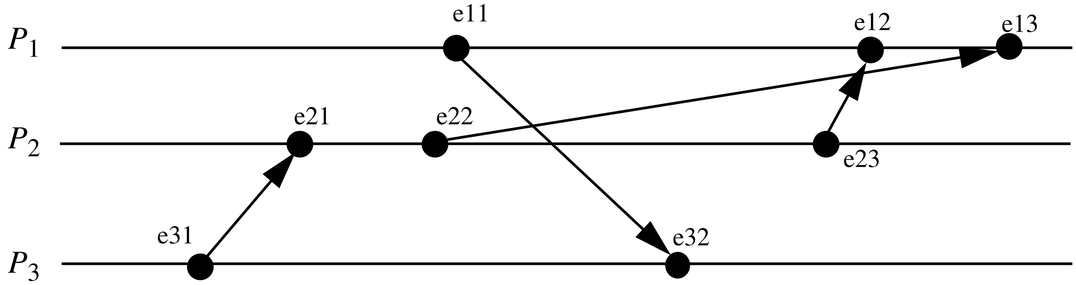
$e_{32}$: $P_3$ receives message $m_{1,1}$ from $P_1$. As $V^{1,1}[3] = (\ ?, ?, ?\ )[3]$ is uninitialized, the message is accepted.
$V_3 \leftarrow [\ ?, (0, 0, 1), ?\ ]$ and $C_3 \leftarrow max[(0, 0, 1), (1, 0, 0)] = (1, 0, 1)$.

$e_{12}$: $P_1$ receives message $m_{2,1}$ from $P_2$. As $V^{2,1}[1] = [\ ?, ?, ?\ ][1]$ is uninitialized, the message is accepted.
$V_1 \leftarrow [\ ?, ?, (1, 0, 0)\ ]$ and $C_1 \leftarrow max[(1, 0, 0), (0, 1, 1)] = (1, 1, 1)$

$e_{23}$: $P_2$ sends message $m_{2,2}$ to $P_1$. $C_2 \leftarrow (0, 2, 1)$; $t^{2,2} \leftarrow (0, 2, 1)$, $V^{2,2} \leftarrow [\ (0, 1, 1), ?, ?\ ]$; $V_2 \leftarrow [\ (0, 2, 1), ?, ?\ ]$

$e_{13}$: $P_1$ receives message $m_{2,2}$ from $P_2$. *As* $V^{2,2}[1] = (0, 1, 1) < (1, 1, 1) = C_1$, the message is accepted.
$V_1 \leftarrow [\ ?, ?, (1, 0, 0)\ ]$ and $C_1 \leftarrow max[(0, 2, 1), (1, 1, 1)] = (1, 2, 1)$

Now, suppose $m_{2,1}$ arrived as event $e_{13}$, and $m_{2,2}$ as event $e_{12}$:




$e_{31}$: $P_3$ sends message $m_{3,1}$ to $P_2$. $C_3 = (0, 0, 1)$; $t^{3,1} \leftarrow (0, 0, 1)$, $V^{3,1} \leftarrow (?, ?, ?)$; $V_3 \leftarrow [\ ?, (0, 0, 1), ?\ ]$

$e_{21}$: $P_2$ receives message $m_{3,1}$ from $P_3$. As $V^{3,1}[2] = (?, ?, ?)[2]$ is uninitialized, the message is accepted.
$V_2 \leftarrow [\ ?, ?, ?\ ]$ and $C_2 \leftarrow max[(0, 0, 0), (0, 0, 1)] = (0, 0, 1)$

$e_{22}$: $P_2$ sends message $m_{2,1}$ to $P_1$. $C_2 \leftarrow (0, 1, 1)$; $t^{2,1} \leftarrow (0, 1, 1)$, $V^{2,1} \leftarrow [\ ?, ?, ?\ ]$; $V_2 \leftarrow [\ (0, 1, 1), ?, ?\ ]$

$e_{11}$: $P_1$ sends message $m_{1,1}$ to $P_3$. $C_1 \leftarrow (1, 0, 0)$; $t^{1,1} \leftarrow (1, 0, 0)$, $V^{1,1} \leftarrow (\ ?, ?, ?\ )$; $V_1 \leftarrow [\ ?, ?, (1, 0, 0)\ ]$

$e_{32}$: $P_3$ receives message $m_{1,1}$ from $P_1$. As $V^{1,1}[3] = (\ ?, ?, ?\ )[3]$ is uninitialized, the message is accepted.
$V_3 \leftarrow [\ ?, (0, 0, 1), ?\ ]$ and $C_3 \leftarrow max[(0, 0, 1), (1, 0, 0)] = (1, 0, 1)$.

$e_{23}$: $P_2$ sends message $m_{2,2}$ to $P_1$. $C_2 \leftarrow (0, 2, 1)$; $t^{2,2} \leftarrow (0, 2, 1)$, $V^{2,2} \leftarrow [\ (0, 1, 1), ?, ?\ ]$; $V_2 \leftarrow [\ (0, 2, 1), ?, ?\ ]$

$e_{12}$: $P_1$ receives message $m_{2,2}$ from $P_2$. But $V^{2,2}[1] = (0, 1, 1) \nless (1, 0, 0) = C_1$, so the message is queued.

$e_{13}$: $P_1$ receives message $m_{2,1}$ from $P_2$. As $V^{2,1}[1] = [\ ?, ?, ?\ ][1]$ is uninitialized, the message is accepted.
$V_1 \leftarrow [\ ?, ?, (1, 0, 0)\ ]$ and $C_1 \leftarrow max[(1, 0, 0), (0, 1, 1)] = (1, 1, 1)$.
The message on the queue is now checked. As $V^{2,2}[1] = (0, 1, 1) < (1, 1, 1) = C_1$, the message is now accepted.
$V_1 \leftarrow [\ ?, ?, (1, 0, 0)\ ]$ and $C_1$ is set to $(1, 2, 1)$.

# Chandy-Lamport Global State Recording Protocol

## Introduction

The goal of this distributed algorithm is to capture a consistent global state. It assumes all communication channels are FIFO. It uses a distinguished message called a *marker* to start the algorithm.

## Protocol

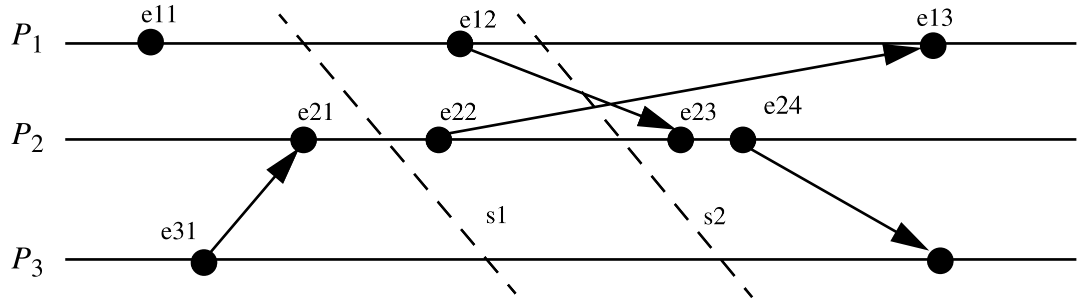
**$P_i$ sends marker**

1. *Pi* records its local state $LS_i$
2. For each $C_{ij}$ on which $P_i$ has not already sent a marker, $P_i$ sends a marker *before* sending other messages.

**$P_i$ receives marker from $P_j$**

1. If $P_i$ has *not* recorded its state:
   a. Record the state of $C_{ji}$ as empty
   b. Send the marker as described above
2. If $P_i$ has recorded its state $LS_i$
   a. Record the state of $C_{ji}$ to be the sequence of messages received between the computation of $LS_i$ and the marker from $C_{ji}$.

## Example




Here, all processes are connected by communications channels $C_{ij}$. Messages being sent over the channels are represented by arrows between the processes.

Snapshot $s_1$:

$P_1$ records $LS_1$, sends markers on $C_{12}$ and $C_{13}$

$P_2$ receives marker from $P_1$ on $C_{12}$; it records its state $LS_2$, records state of $C_{12}$ as empty, and sends marker on $C_{21}$ and $C_{23}$

$P_3$ receives marker from $P_1$ on $C_{13}$; it records its state $LS_3$, records state of $C_{13}$ as empty, and sends markers on $C_{31}$ and $C_{32}$.

$P_1$ receives marker from $P_2$ on $C_{21}$; as $LS_1$ is recorded, it records the state of $C_{21}$ as empty.

$P_1$ receives marker from $P_3$ on $C_{31}$; as $LS_1$ is recorded, it records the state of $C_{31}$ as empty.

$P_2$ receives marker from $P_3$ on $C_{32}$; as $LS_2$ is recorded, it records the state of $C_{32}$ as empty.

$P_3$ receives marker from $P_2$ on $C_{23}$; as $LS_3$ is recorded, it records the state of $C_{23}$ as empty.

Snapshot $s_2$: now messages are in transit on $C_{12}$ and $C_{21}$.

$P_1$ records $LS_1$, sends markers on $C_{12}$ and $C_{13}$

$P_2$ receives marker from $P_1$ on $C_{12}$ after the message from $P_1$ arrives; it records its state $LS_2$, records state of $C_{12}$ as empty, and sends marker on $C_{21}$ and $C_{23}$

$P_3$ receives marker from $P_1$ on $C_{13}$; it records its state $LS_3$, records state of $C_{13}$ as empty, and sends markers on $C_{31}$ and $C_{32}$.

$P_1$ receives marker from $P_2$ on $C_{21}$; as $LS_1$ is recorded, and a message has arrived since $LS_1$ was recorded, it records the state of $C_{21}$ as containing that message.

$P_1$ receives marker from $P_3$ on $C_{31}$; as $LS_1$ is recorded, it records the state of $C_{31}$ as empty.

$P_2$ receives marker from $P_3$ on $C_{32}$; as $LS_2$ is recorded, it records the state of $C_{32}$ as empty.

$P_3$ receives marker from $P_2$ on $C_{23}$; as $LS_3$ is recorded, it records the state of $C_{23}$ as empty.

# Huang's Termination Detection Protocol

## Introduction

The goal of this protocol is to detect when a distributed computation terminates.

## Notation

- $n$ processes
- $P_i$ process; without loss of generality, let $P_0$ be the *controlling agent*
- $W_i$. weight of process $P_i$; initially, $W_0 = 1$ and for all other $i$, $W_i = 0$.
- $B(W)$ computation message with assigned weight $W$
- $C(W)$ control message sent from process to controlling agent with assigned weight $W$

## Protocol

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

1. Set $W_i$' and $W_j$ to values such that $W_i' + W_j = W_i$, $W_i > 0$, $W_j > 0$. ($W_i$' is the new weight of $P_i$.)
2. Send $B(W_j)$ to $P_j$

**$P_j$ receives a computation message $B(W)$ from $P_i$**

1. $W_j = W_j + W$
2. If $P_j$ is idle, $P_j$ becomes active

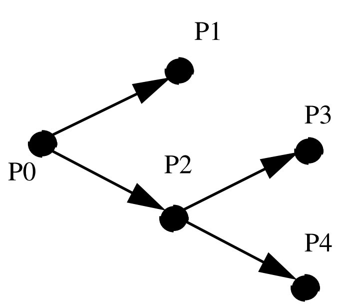
**$P_i$ becomes idle:**

1. Send $C(W_i)$ to $P_0$
2. $W_i = 0$
3. $P_i$ becomes idle

**$P_i$ receives a control message $C(W)$:**

1. $W_i = W_i + W$
2. If $W_i = 1$, the computation has completed.

## Example




The picture shows a process $P_0$, designated the *controlling agent*, with $W_0 = 1$. It asks $P_1$ and $P_2$ to do some computation. It sets $W_1$ to 0.2, $W_2$ to 0.3, and $W_3$ to 0.5. $P_2$ in turn asks $P_3$ and $P_4$ to do some computations. It sets $W_3$ to 0.1 and $W_4$ to 0.1.

When $P_3$ terminates, it sends C($W_3$) = C(0.1) to $P_2$, which changes $W_2$ to 0.1 + 0.1 = 0.2.

When $P_2$ terminates, it sends C($W_2$) = C(0.2) to $P_0$, which changes $W_0$ to 0.5 + 0.2 = 0.7.

When $P_4$ terminates, it sends C($W_4$) = C(0.1) to $P_0$, which changes $W_0$ to 0.7 + 0.1 = 0.8.

When $P_1$ terminates, it sends C($W_1$) = C(0.2) to $P_0$, which changes $W_0$ to 0.8 + 0.2 = 1.

$P_0$ thereupon concludes that the computation is finished.

Total number of messages passed: 8 (one to start each computation, one to return the weight).