SIMON FRASER UNIVERSITY
ENGAGING THE WORLD

CMPT 431 Distributed Systems

Fall 2019

Logical Time

https://www.cs.sfu.ca/~keval/teaching/cmpt431/fall19/

Instructor: Keval Vora

Distributed Systems

- No global clock
- Causality important to analyze distributed system
- How to capture the fundamental monotonicity property associated with causality?

- Logical time!
- Distributed computations make progress in spurts
  - They are asynchronous
- Logical time captures this progress across processes

Logical Time

- Captures causality in distributed systems

- Scalar Time
- Vector Time
- Implementations of vector clocks

Reading

- [DC] Chapter 3
  - Upto 3.5

System of Logical Clocks

- Time domain T and Logical clock C

- Elements of T form a partially ordered set over a relation <
  - Causal precedence or happens before relationship
- Intuitively, analogous to the "earlier than" relation provided by physical time

- C is a function that maps an event e to an element in T
  - Timestamp of e, denoted as C(e)
  - For two events e_i and e_j: $e_i \rightarrow e_j \Rightarrow C(e_i) < C(e_j)$

System of Logical Clocks

- C is a function that maps an event e to an element in T
  - Timestamp of e, denoted as C(e)
  - For two events e_i and e_j: $e_i \rightarrow e_j \Rightarrow C(e_i) < C(e_j)$

- Monotonicity property is called the clock consistency condition
- The system of clocks is strongly consistent when, for any two events e_i and e_j :

$$e_i \rightarrow e_j \Leftrightarrow C(e_i) < C(e_j)$$

Implementing Logical Clocks

- Requires two things:
  - Data structures local to every process to represent logical time
  - A protocol to update the data structures in a way that ensures the clock consistency condition

- Logical clock implementations differ in their representation of logical time, and also in their protocol to update the logical clocks

Logical Clocks: Data Structures

- Each process p_i maintains:
  - A logical local clock that helps p_i measure its own progress
    - Denoted by lc_i
  - A logical global clock which represents p_i's local view of the logical global time
    - Denoted by gc_i

- Typically, lc_i is a part of gc_i

Logical Clocks: Protocol

- The protocol ensures that a process's logical clock, and thus its view of the global time, is managed consistently

- The protocol consists of following two rules:
  - R1: This rule governs how the local logical clock is updated by a process when it executes an event
  - R2: This rule governs how a process updates its global logical clock to update its view of the global time and global progress

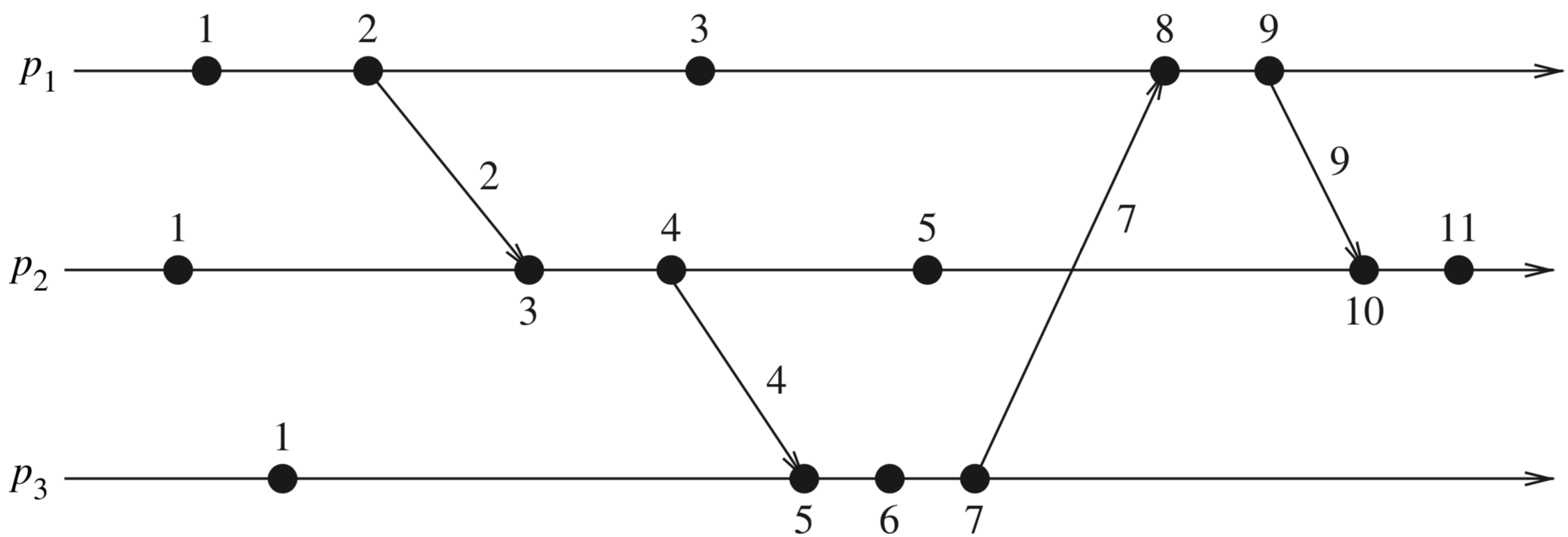
Scalar Time

- Proposed by Lamport to totally order events
- Time domain is the set of non-negative integers
- The logical local clock of a process p_i and its local view of the global time are squashed into one integer variable C_i

Scalar Time

- R1: Before executing an event (send, receive, or internal), process p_i executes: $C_i := C_i + 1$
- R2: Each message piggybacks the clock value of its sender at sending time
- When a process p_i receives a message with timestamp C_{msg}, it does the following:
  - $C_i := \max(C_i, C_{msg})$
  - Execute R1
  - Deliver the message (i.e., continue forward)

Scalar Time

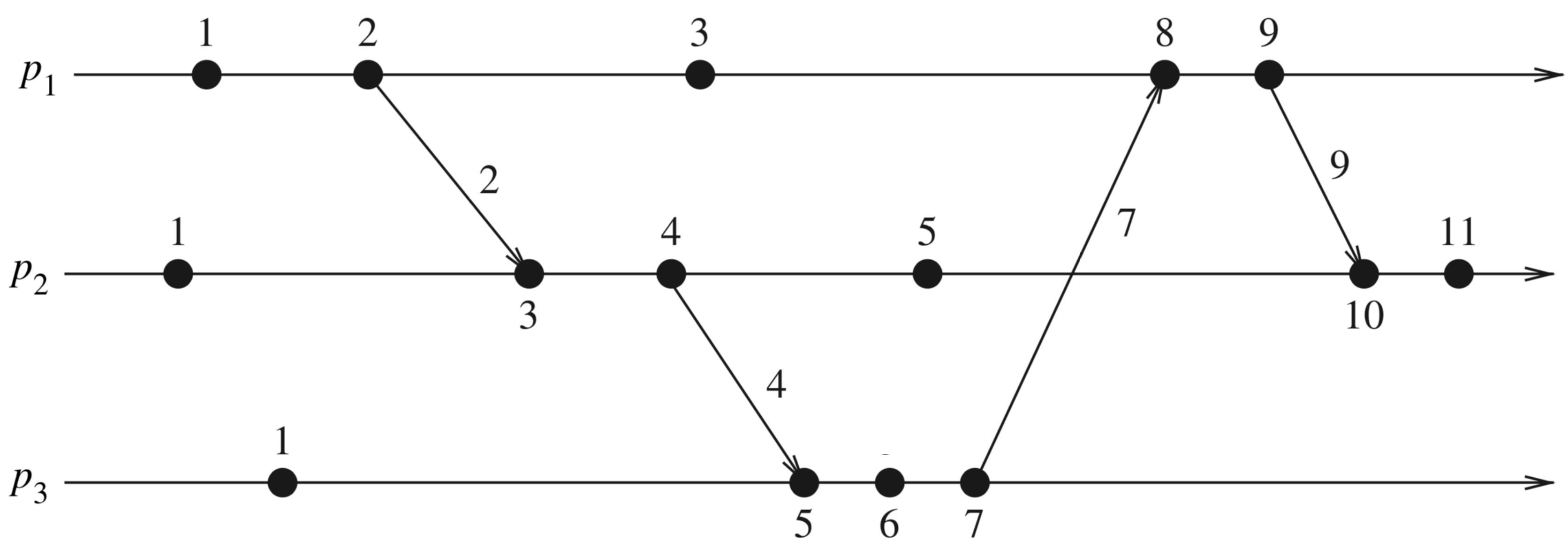


Scalar Time: Consistency

- Scalar clocks satisfy the monotonicity, and hence the consistency property
- For two events e_i and e_j : $e_i \rightarrow e_j \Rightarrow C(e_i) < C(e_j)$

Scalar Time: Total Ordering

- Does scalar clock provide total ordering?
- Two or more events at different processes may have identical timestamp



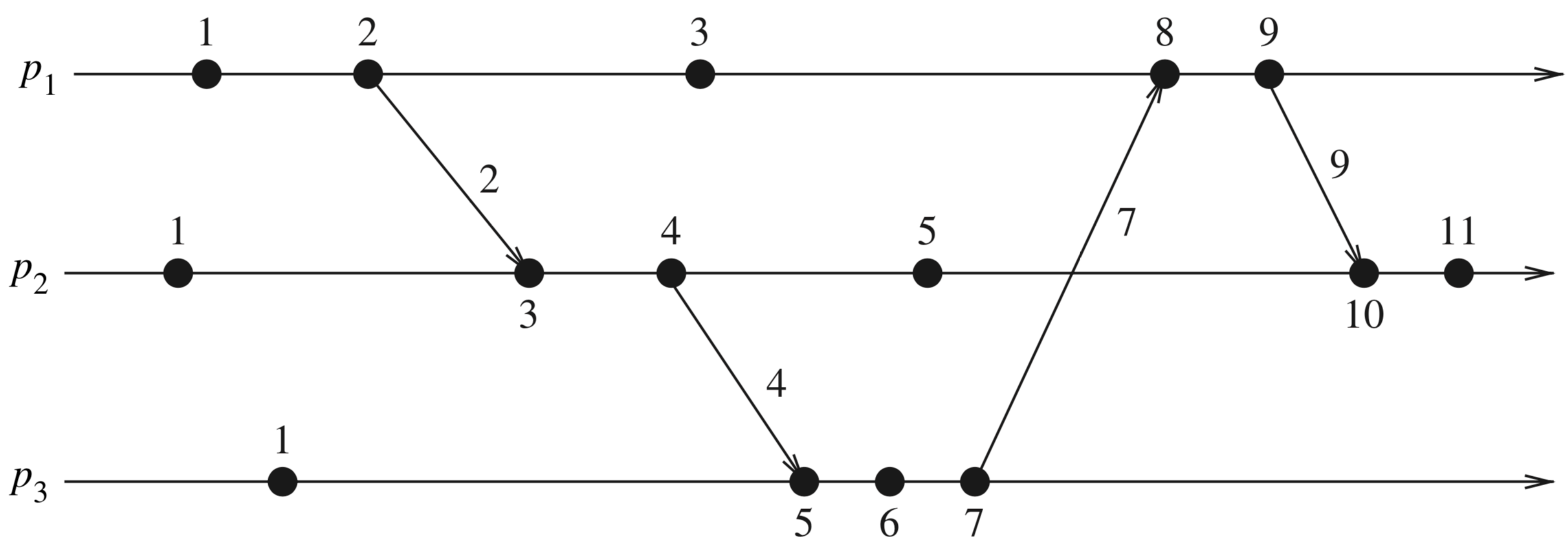
Scalar Time: Total Ordering

- Tie-breaking: Process identifiers are linearly ordered based on which ties among events with identical scalar timestamps are broken
- Lower process identifier means higher priority
- Timestamp is denoted by a tuple (t, i) where t is its time and i is the process id
- The total order relation $\prec$ on two events x and y with timestamps (h,i) and (k,j), respectively, is defined as:

$$x \prec y \Leftrightarrow (h < k \text{ or } (h = k \text{ and } i < j))$$

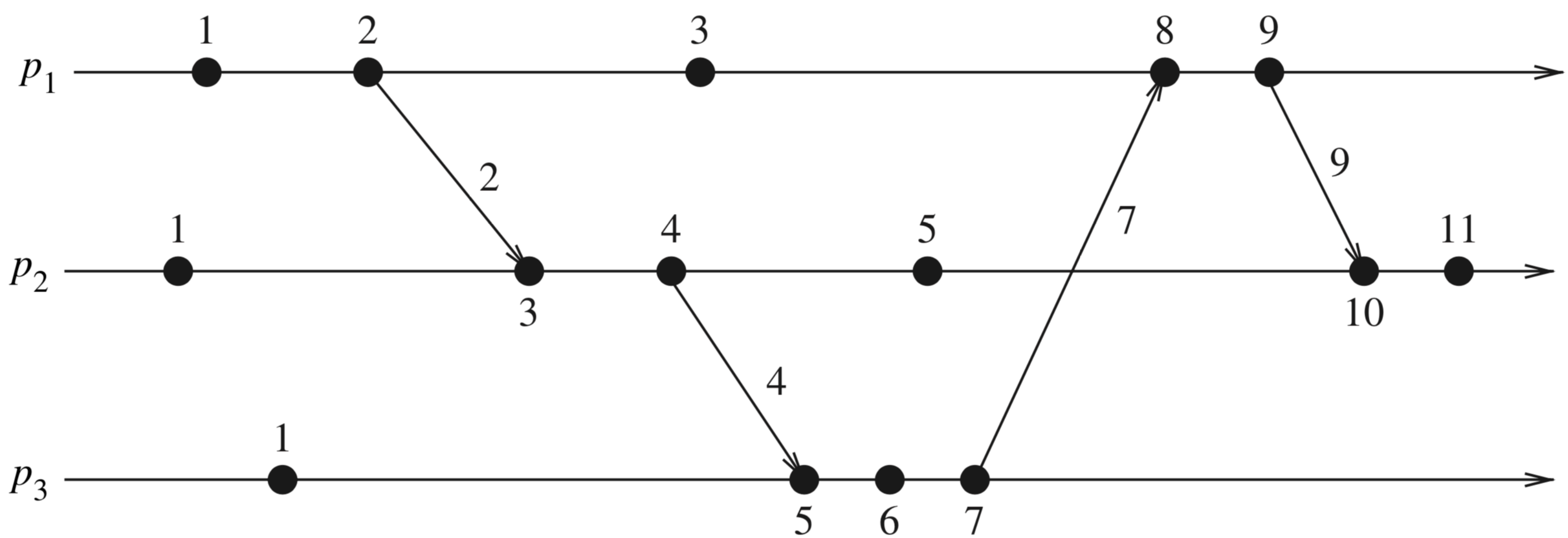
Scalar Time: Event Counting

- If event e has a timestamp h, what can we say about when e happened (in terms of h)?
- h-1 represents the minimum logical duration (counted in units of events) required before producing event e
- h-1 events have happened sequentially before the event e



Scalar Time: Strong Consistency

- Are scalar clocks strongly consistent?
- No. For two events e_i and e_j : $C(e_i) < C(e_j) \not\Rightarrow e_i \rightarrow e_j$



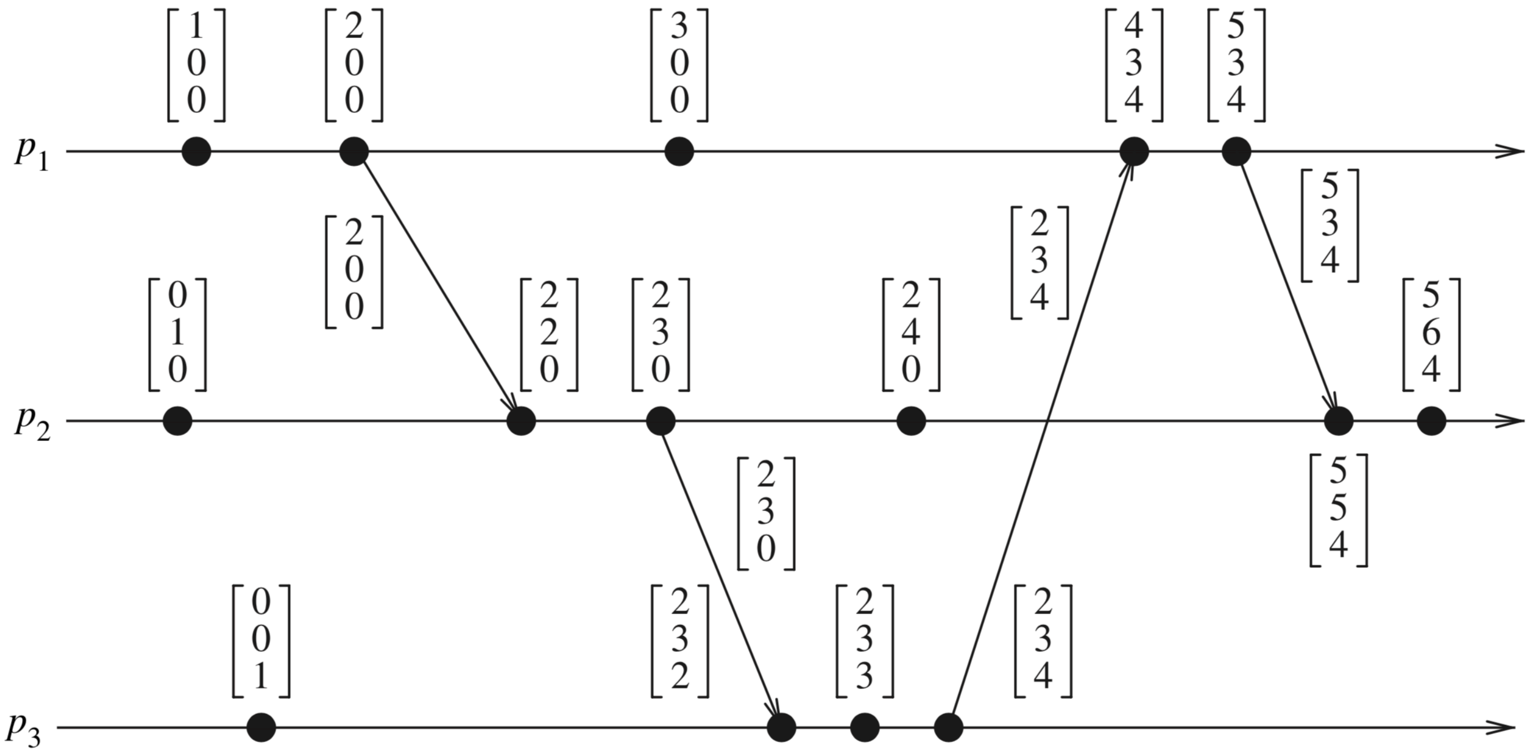
Scalar Time: Strong Consistency

- Are scalar clocks strongly consistent?
- No. For two events e_i and e_j : $C(e_i) < C(e_j) \not\Rightarrow e_i \rightarrow e_j$

- Logical local clock and logical global clock of a process are squashed into one, resulting in the loss of causal dependency information among events at different processes

Vector Time

- Time domain is represented by a set of n-dimensional non-negative integer vectors



Vector Time

- Each process p_i maintains a vector $vt_i[1..n]$
  - $vt_i[i]$ is the local logical clock of p_i and describes the logical time progress at process p_i
  - $vt_i[j]$ is p_i's latest knowledge of p_j's local time
- If $vt_i[j]=x$, then p_i knows that local time at p_j has progressed till x

- The entire vector vt_i constitutes p_i's view of the global logical time, and is used to timestamp events

Vector Time

- R1: Before executing an event, process p_i updates its local logical time as follows: $vt_i[i] := vt_i[i] + 1$

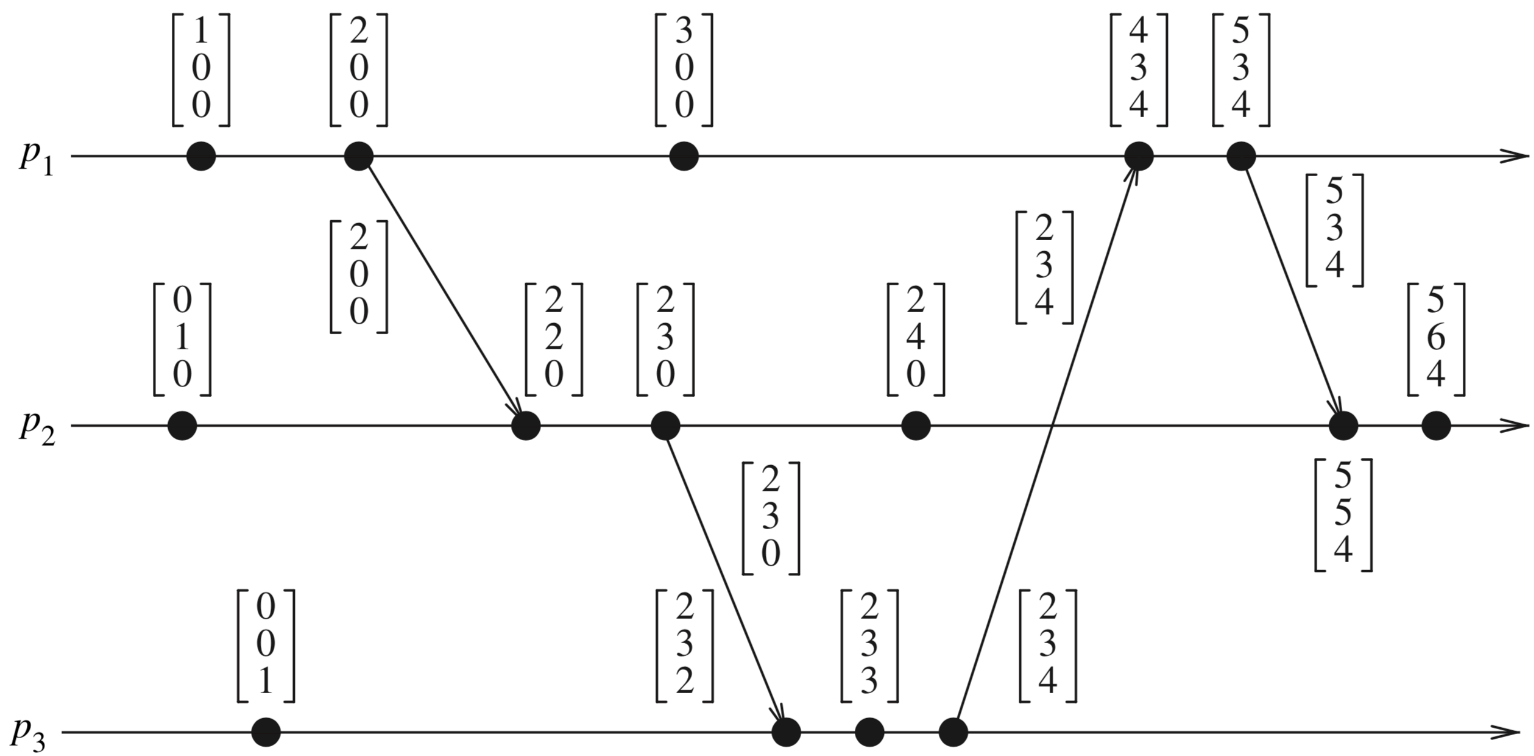
- R2: Each message m is piggybacked with the vector clock vt of the sender process at sending time
- On the receipt of such a message (m, vt), process p_i does the following:
  - Update its global logical time as:

$$1 \leq k \leq n : vt_i[k] := \max(vt_i[k], vt[k])$$

  - Execute R1
  - Deliver the message m

Vector Time

- The timestamp of an event is the value of the vector clock of its process when the event is executed



Comparing Vector Timestamps

$$vh = vk \Leftrightarrow \forall x : vh[x] = vk[x]$$

$$vh \leq vk \Leftrightarrow \forall x : vh[x] \leq vk[x]$$

$$vh < vk \Leftrightarrow vh \leq vk \text{ and } \exists x : vh[x] < vk[x]$$

$$vh \,||\, vk \Leftrightarrow \neg(vh < vk) \wedge \neg(vk < vh)$$

- If events x and y respectively occurred at processes p_i and p_j , and are assigned timestamps vh and vk respectively:

$$x \rightarrow y \Leftrightarrow vh[i] \leq vk[i]$$

$$x \,||\, y \Leftrightarrow vh[i] > vk[i] \wedge vh[j] < vk[j]$$

Vector Time: Isomorphism

- There is an isomorphism between the set of partially ordered events produced by a distributed computation and their vector timestamps

- If two events x and y have timestamps vh and vk:

$$x \rightarrow y \Leftrightarrow vh < vk$$

$$x \mid\mid y \Leftrightarrow vh \mid\mid vk$$

Vector Time: Strong Consistency

- Are vector clocks strongly consistent?
  - Yes: $x \rightarrow y \Leftrightarrow vh < vk$ (previous slide)

- We can find if events are causally related by simply examining their vector timestamps
- The dimension of vector clocks cannot be less than n for this property to hold (n = the total number of processes)

Vector Time: Event Counting

- $vt_i[i]$ denotes the number of events that have occurred at p_i until that instant

- If an event e has timestamp vh, vh[j] denotes the number of events executed by process p_j that causally precede e

- $\sum vh[j] - 1$ represents the total number of events that causally precede e

Implementing Vector Clocks

- If number of processes is large, vector clocks will require piggybacking of huge amount of information in messages
  - The message overhead grows linearly with the number of processes in the system
  - Message size becomes huge even if there are only a few events

- To have strong consistency property, vector timestamps must be at least of size n
- Optimizations are possible! Any thoughts?

Differential Vector Clocks

- Singhal-Kshemkalyani's differential technique
- Key observation: between successive message sends to the same process, only a few entries of the vector clock at the sender process are likely to change
- Send only the differences in vector clocks!

Differential Vector Clocks

- When p_i sends a message to p_j, it piggybacks only those entries of its vector clock that differ since the last message sent to p_j

- If entries $i_1, i_2, \ldots, i_{n1}$ of the vector clock at p_i have changed to $v_1, v_2, \ldots, v_{n1}$ respectively, since the last message sent to p_j , then p_i piggybacks a compressed timestamp to the next message to p_j :

$$\{(i_1, v_1), (i_2, v_2), \ldots, (i_{n1}, v_{n1})\}$$

Differential Vector Clocks

- When p_j receives this message, it updates its vector clock:

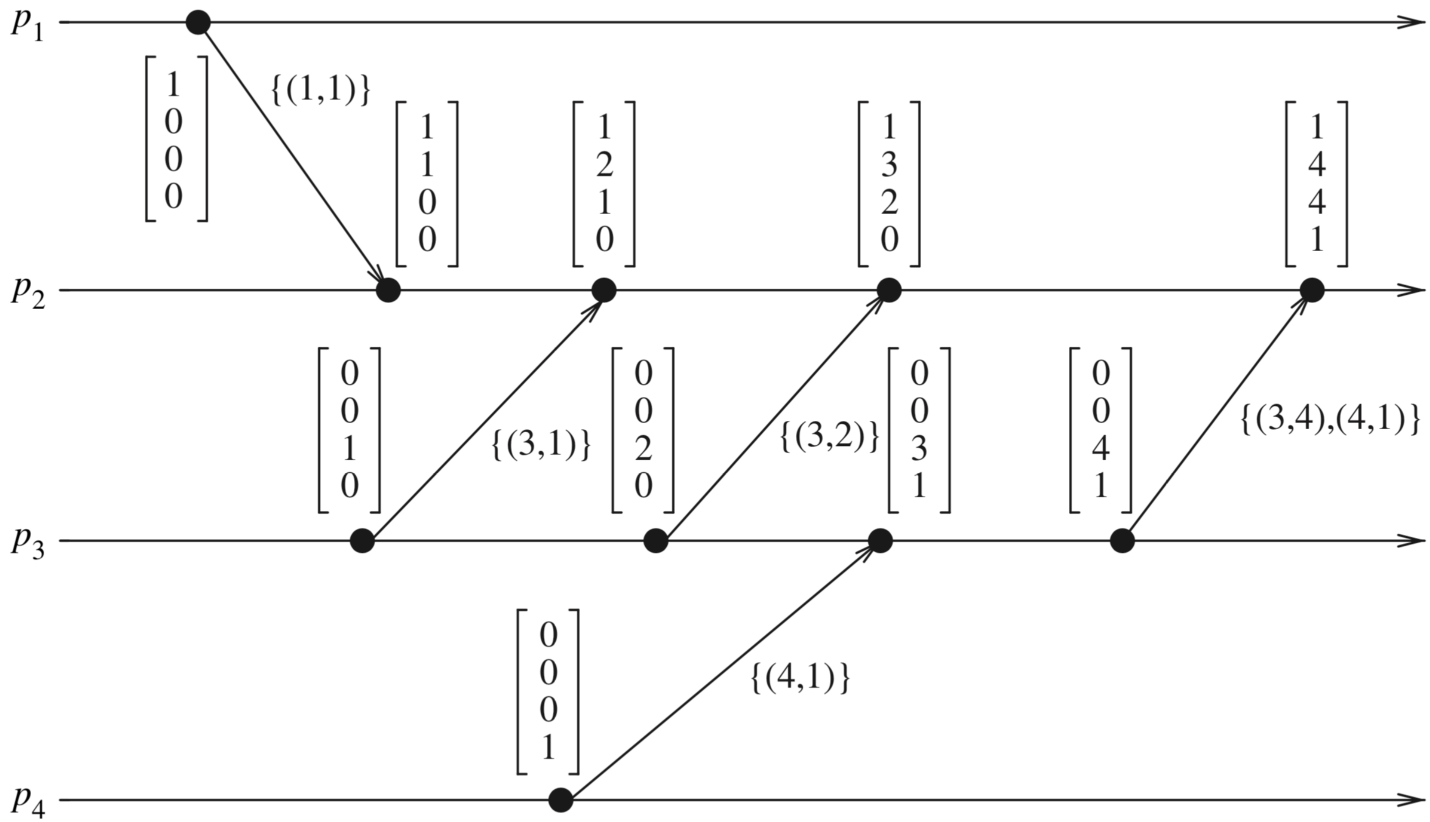
$$vt_i[i_k] = \max(vt_i[i_k], v_k) \text{ for } k = 1, 2, \ldots, n_1$$

- Cuts down the message size, communication bandwidth, and buffer (to store messages) requirements

- Worst case: every element of the vector clock gets updated
- On an average, the size of timestamp on a message will be less than n

Differential Vector Clocks

Assumption: Communication channels are FIFO



Differential Vector Clocks

- Naïve implementation: each process remembers the vector timestamp last sent to every other process
  - What is the storage requirement at each process?
  - $O(n^2)$ storage at each process

- Clever implementation to reduce overhead to O(n)
- Process p_i maintains:
  - $LS_i[1..n]$ ('Last Sent'): $LS_i[j]$ indicates the value of $vt_i[i]$ when process p_i last sent a message to process p_j
  - $LU_i[1..n]$ ('Last Update'): $LU_i[j]$ indicates the value of $vt_i[i]$ when process p_i last updated the entry $vt_i[j]$

Differential Vector Clocks

- $LS_i[j]$: when process p_i last sent a message to process p_j
- $LU_i[j]$: when process p_i last updated the entry $vt_i[j]$

- $LU_i[i] = vt_i[i]$ at all times
- $LU_i[j]$ is updated only upon receipt of a message which causes p_i to update entry $vt_i[j]$
- $LS_i[j]$ is updated only when p_i sends a message to p_j

- How to find out the differences?

Differential Vector Clocks

- Since the last communication from p_i to p_j, the changed elements of $vt_i[k]$ are ones for which $LS_i[j] < LU_i[k]$

- Vector timestamp sent from p_i to p_j

$$\{(x, vt_i[x]) \mid LS_i[j] < LU_i[x]\}$$

Direct Dependency Technique

- Fowler-Zwaenepoel’s direct dependency technique
- Reduces size of timestamp transmission to scalar values
    - No vector clocks maintained on the fly
- Key idea
    - Only track direct dependency information between processes
    - Vector representing transitive dependencies is constructed offline using recursive search on direct dependencies
- Originally developed for causal distributed breakpoints

Direct Dependency Technique

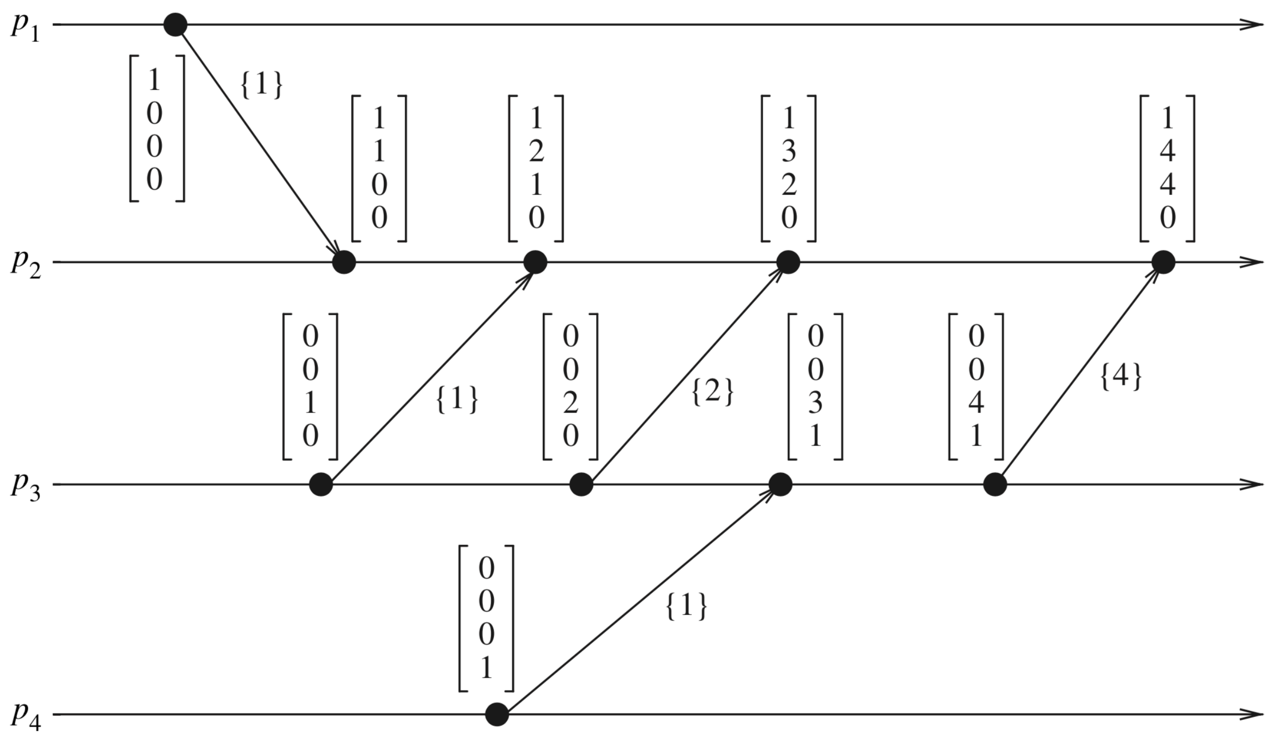
- Process p_i maintains dependency vector D_i

- When event occurs at p_i, $D_i[i] := D_i[i] + 1$
- When p_i sends message, piggyback $D_i[i]$
- When pi receives a message with piggybacked value d,

$$D_i[j] := \max\{D_i[j], d\}$$

Direct Dependency Technique

- Only direct dependencies are maintained
- p_2 is never informed about its indirect dependency on p_4



Direct Dependency Technique

- $D_i[j]$ denotes the sequence number of the latest event on p_j that directly affects the current state of p_i

DependencyTrack$(i : process,\ \sigma : event\ index)$
* Casual distributed breakpoint for σ_i *\
* *DTV* holds the result *\
for all $k \neq i$ **do**
    $DTV[k]$=0
end for
$DTV[i]$=σ
end DependencyTrack

VisitEvent$(j : process,\ e : event\ index)$
* Place dependencies of τ into *DTV* *\
for all $k \neq j$ **do**
    $\alpha = D_j^e[k]$
    if $\alpha > DTV[k]$ **then**
        $DTV[k]$=α
        VisitEvent$(k,\ \alpha)$
    end if
end for
end VisitEvent

Reading

- [DC] Chapter 3
  - Upto 3.5