SS ZG 526: Distributed Computing (CS2)

Logical Time

Prof. Geetha
Associate Professor, Dept. of Computer Sc. & Information Systems
BITS Pilani Hyderabad Campus
geetha@hyderabad.bits-pilani.ac.in

BITS Pilani
Hyderabad Campus

Introduction

- concept of causality between events → important for design and analysis of parallel and distributed computing and operating systems
- causality is tracked using physical time
- But in distributed systems
  - not possible to have global physical time
  - possible to realize only an approximation of physical time
- way of measuring time
  - asynchronous distributed computations progress in spurts
  - logical time also advances in jumps
  - sufficient to capture the fundamental monotonicity property associated with causality in distributed systems

Introduction

- **monotonicity property: if event *a* causally affects event *b*, then timestamp of *a* is smaller than timestamp of *b***
- Ways to represent logical time:
  - scalar time
  - vector time

Introduction

- **Causality / Causal Precedence Relation**
- described among events in a distributed system

- useful in reasoning, analyzing, and drawing inferences about a computation

- helps to solve several problems in distributed systems
  - distributed algorithms design
  - tracking of dependent events
  - knowledge about progress
  - concurrency measure

Introduction

- **Distributed algorithms design**

knowledge of the causal precedence relation among events helps to:

- ensure liveness and fairness in mutual exclusion algorithms
- maintain consistency in replicated databases
- design correct deadlock detection algorithms

- **Tracking of dependent events**
  - helps construct a consistent state for resuming re-execution
  - in failure recovery, helps build a checkpoint
  - in replicated databases, aids in detection of file inconsistencies

Introduction

- **Knowledge about progress - knowledge of causal dependency among events**
  - •helps measure progress of processes in distributed computation
  - •helps to discard obsolete information, garbage collection, and termination detection
- **Concurrency measure - knowledge regarding how many events are causally dependent**
  - •useful in measuring amount of concurrency
  - •events not causally related are concurrent
  - •analysis of causality tells about concurrency in program

Capturing Causality between Events

- in distributed systems
  - rate of event occurrence is of high magnitude
  - event execution time is of low magnitude
- physical clocks must be precisely synchronized
- in distributed computation
  - clocks accurate to a few tens of milliseconds are not sufficient
  - progress occurs in spurts
  - interaction between processes occurs in spurts
- logical clocks can accurately capture
  - causality relation between events produced by a program execution
  - fundamental monotonicity property

Logical Clocks

- every process has a logical clock
- logical clock is advanced using a set of rules
- each event is assigned a timestamp
- causality relation between events can be inferred from their timestamps
- timestamps follow the monotonicity property

Representing Logical Clocks

1. Lamport's scalar clocks
    - time is represented by non-negative integers
2. Vector clocks (Fidge, Mattern and Schmuck)
    - time is represented using a vector of non-negative integers

Real world applications of Logical clocks

- Scalar clocks => Amazon S3
- Vector clocks => Voldemort (Linkedin) Amazon Dynamo, Version control systems etc.

Self assessment Question:

- Why Lamport’s clock cannot be used in blockchain technology?

Definition of Logical Clocks

- system of logical clocks consists of
  - time domain *T*
  - logical clock *C*
- elements of *T* form a partially ordered set over a relation <
- <:
  - happened before or causal precedence relation
  - analogous to ‘earlier than’ relation provided by physical time

Definition of Logical Clocks

- logical clock *C*
  - is a function that maps an event *e* in a distributed system to an element in the time domain *T*
  - denoted as C(*e*)
  - C(*e*): timestamp of *e*
  - *C is defined as:* $C : H \mapsto T$
  - satisfies the following property –
    - for any 2 events e_i and e_j, $e_i \rightarrow e_j \implies C(e_i) < C(e_j)$.
    - monotonicity property
    - known as *clock consistency condition*

Definition of Logical Clocks

- system of clocks is *strongly consistent* if T and C satisfy the following:
    - for 2 events e_i and e_j,

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

Implementing Logical Clocks

- Two issues need to be addressed
  - data structures local to every process to represent logical time
  - a protocol to update the data structures to ensure the consistency condition

Implementing Logical Clocks

Data Structures

- Each process p_i maintains two data structures
- A *local logical clock*
  - denoted by lc_i
  - helps process p_i measure its own progress

- A *logical global clock*
  - denoted by gc_i
  - represents p_i's local view of the logical global time
  - allows p_i to assign consistent timestamps to its local events
  - *lc_i is a part of gc_i*

Implementing Logical Clocks

Protocol - ensures consistent management of:

- a process's logical clock
- process's view of global time

- consists of the following 2 rules:
- **R1:** governs how the local logical clock is updated by a process when it executes an event (send, receive, or internal)
- **R2:**
  - governs how a process updates its global logical clock to update its view of the global time and global progress
  - dictates what information about logical time is piggybacked in a message
  - how this information is used by the receiving process to update its view of the global time

Implementing Logical Clocks

- systems of logical clocks differ in
  - representation of logical time
  - the protocol to update the logical clocks
- all logical clock systems
  - implement rules **R1** and **R2**
  - ensure the fundamental monotonicity property associated with causality
  - provide users with some additional properties

Scalar Time - Definition

- representation was proposed by Lamport in 1978 to totally order events in distributed system

- in this representation, time domain is the set of non-negative integers

- C_i:
  - integer variable
  - denotes logical local clock of p_i and its local view of global time

Scalar Time - Definition

Rules for updating clock:

- **R1: Before executing an event (send, receive, or internal), process p_i executes** $$C_i := C_i + d \qquad (d > 0)$$

For each execution of **R1**,

- *d can have different value*
- value of *d* may be application-dependent

Usually *d* = 1

- helps to identify the time of each event uniquely at a process
- keeps rate of increase of *d* to lowest level

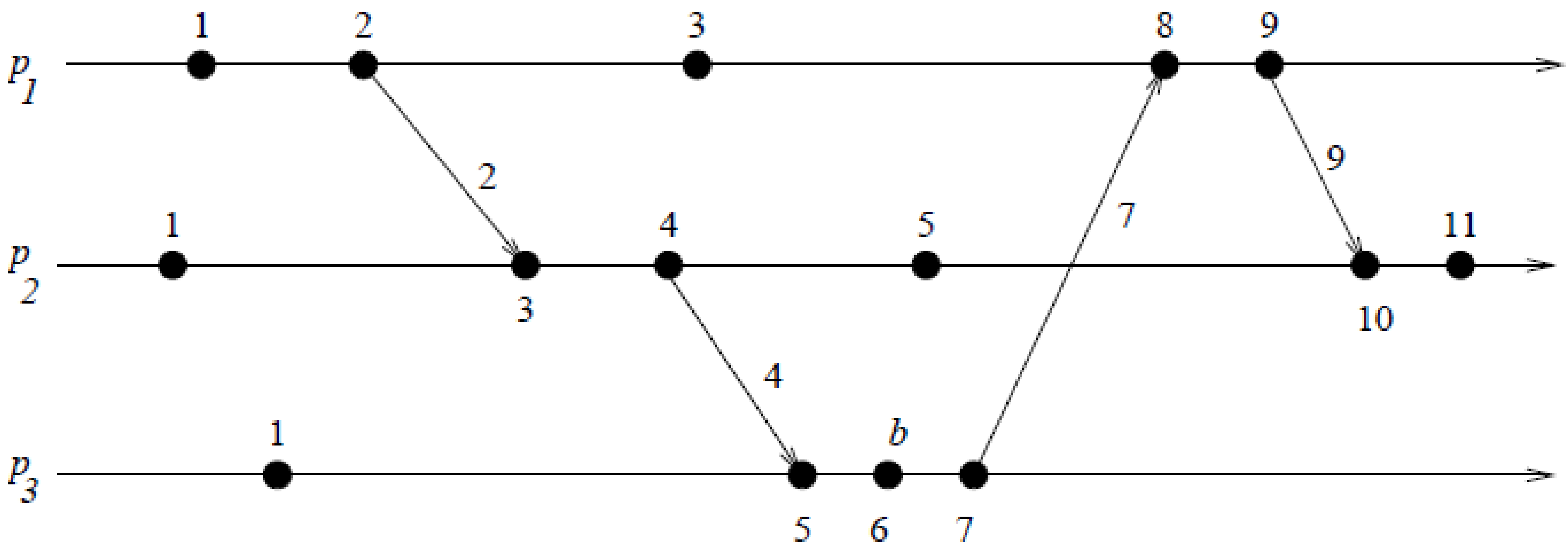
Scalar Time - Definition

Rules for updating clock (contd. from previous slide):

R2: When process p_i receives a message with timestamp C_{msg}, it executes the following actions:

1. $C_i = max\ (C_i, C_{msg})$;
2. execute **R1;**
3. deliver the message.

Evolution of Scalar Time



Space-time diagram of a distributed system

Scalar Time – Basic Properties

- Consistency
- Total Ordering
- Event Counting
- No Strong Consistency

Scalar Time – Basic Properties

Consistency Property

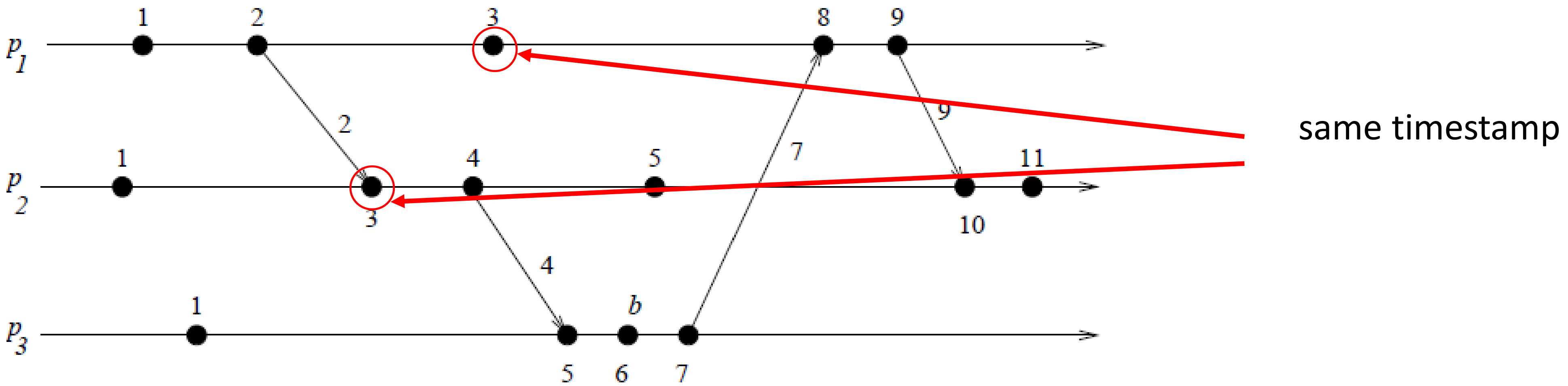
- for two events e_i and e_j, $e_i \rightarrow e_j \Longrightarrow C(e_i) < C(e_j)$
- scalar clocks satisfy monotonicity property
- hence, they also satisfy consistency property

Scalar Time – Basic Properties

Total Ordering

- Scalar clocks are used to totally order events in a distributed system
- Problem in totally ordering events – 2 or more events at different processes may have same timestamp

  for 2 events e_1 and e_2, $C(e_1) = C(e_2) \implies e_1 \parallel e_2$



Scalar Time – Basic Properties

Total Ordering

- Tie breaking mechanism is needed
- Tie breaking procedure:
  - process identifiers are linearly ordered
  - break tie among events with identical scalar timestamps on the basis of their process identifiers
  - lower process identifier implies higher priority
  - timestamp of an event is denoted by a tuple *(t, i)*
    - *t* → *time of occurrence*
    - *i* → *identity of the process where it occurred*

Scalar Time – Basic Properties

Total Ordering

- Total order relation $\prec$ on 2 events *x* and *y* with timestamps *(h, i)* and *(k, j)* respectively, is defined:

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

- events that occur at the same logical scalar time are independent (i.e., they are not causally related)
- such events can be ordered using any arbitrary criterion without violating the causality relation $\rightarrow$
- so, total order is consistent with the causality relation "$\rightarrow$"
- Note:- $x \prec y \Rightarrow x \rightarrow y \vee x \,||\, y$

Scalar Time – Basic Properties

Total Ordering

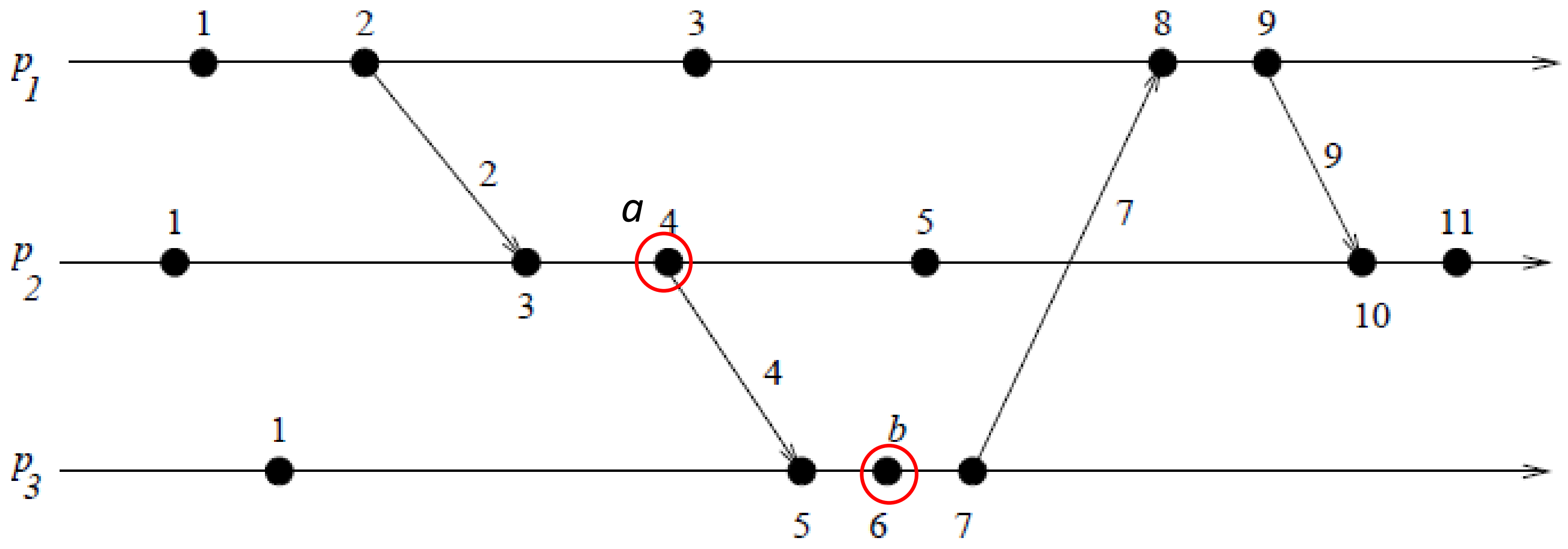
- Utility
  - total order is generally used to ensure liveness properties in distributed algorithms
    - Liveness property:
      - something good eventually happens
      - system makes progress, no starvation, programs terminate
  - requests are timestamped and served according to the total order based on these timestamps

Scalar Time – Basic Properties

Event Counting

- If the increment value *d* = 1, scalar time has the following property:
  - if event *e* has a timestamp *h*, then *h* – 1
    - is the minimum logical duration required before producing event *e*
    - counted in units of events
    - called the ***height*** of event *e*
  - implies that *h* - 1 events have been produced sequentially before event *e,* irrespective of the processes that produced these events

Scalar Time – Basic Properties

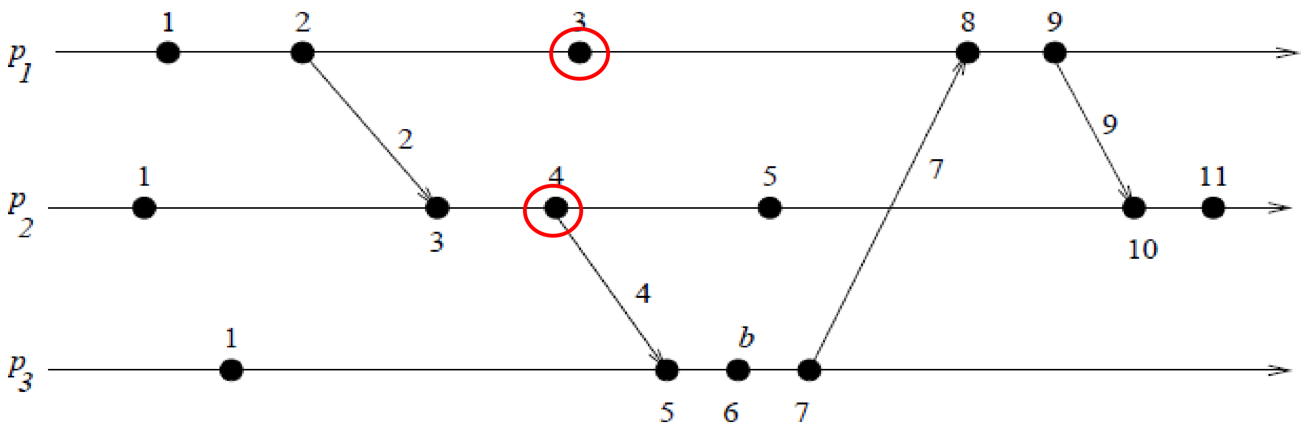


- height of event *a* = 4 – 1 = 3, i.e., 3 events precede *a* on the longest causal path ending at *a*
- height of event *b* = 6 – 1 = 5, i.e., 5 events precede *b* on the longest causal path ending at *b*

Scalar Time – Basic Properties

No strong consistency

- system of scalar clocks is not strongly consistent
- for 2 events e_i and e_j, $C(e_i) < C(e_j) \not\Rightarrow e_i \rightarrow e_j$

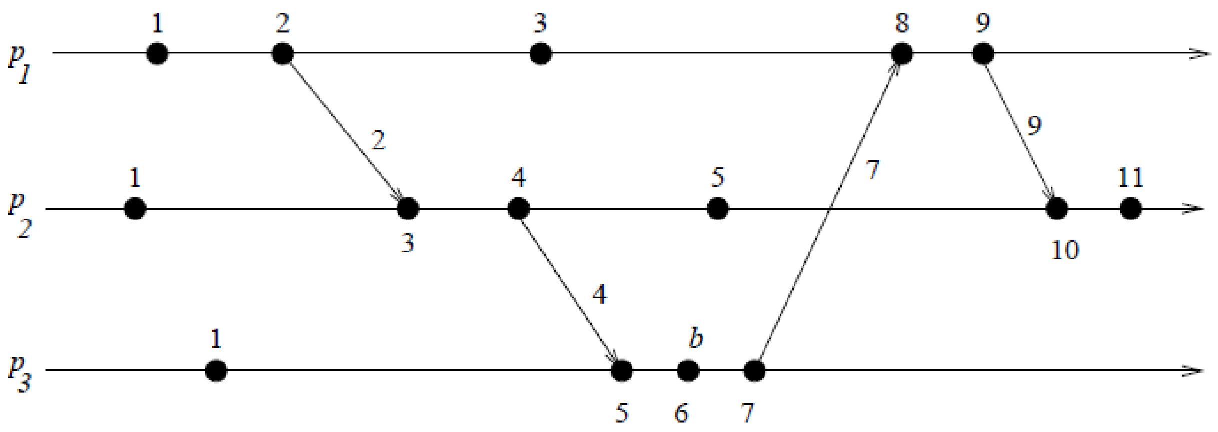


- scalar timestamp of 3rd event of p_1 < scalar timestamp of 3rd event of p_2
- but the former did not happen before the latter

Scalar Time – Basic Properties

No strong consistency

- scalar clocks are not strongly consistent because logical local clock and logical global clock of a process are combined into one
- results in the loss causal dependency information among events at different processes



- when p_2 receives the 1st message from p_1, p_2 updates its clock to 3
- forgets that the timestamp of the latest event at p_1 on which it depends is 2

Vector Time - Definition

- time domain is represented by a set of *n*-dimensional non-negative integer vectors
- each process p_i maintains a vector $vt_i[1..n]$
  - $vt_i[i]$ *: local logical clock of* p_i
  - describes progress of logical time at p_i
  - $vt_i[j]$*: represents process* p_i*'s latest knowledge of process* p_j*'s local time*
  - if $vt_i[j] = x$
    - p_i *knows that local time at* p_j *has progressed till* x
- vector vt_i constitutes p_i's view of the global logical time
- vt_i *is used to timestamp events*

Vector Time - Definition

Rules used by p_i for clock updating

- **R1: Before executing an event, process p_i updates its local logical time as follows:**

$$vt_i[i] = vt_i[i] + d, \quad d > 0$$

Vector Time - Definition

Rules used by p_i for clock updating

- **R2:**
  - each message *m* is piggybacked with the vector clock *vt* of the sender process at sending time
  - when p_i receives message (*m, vt*), it executes the following sequence of actions:
    1. update its global logical time as:

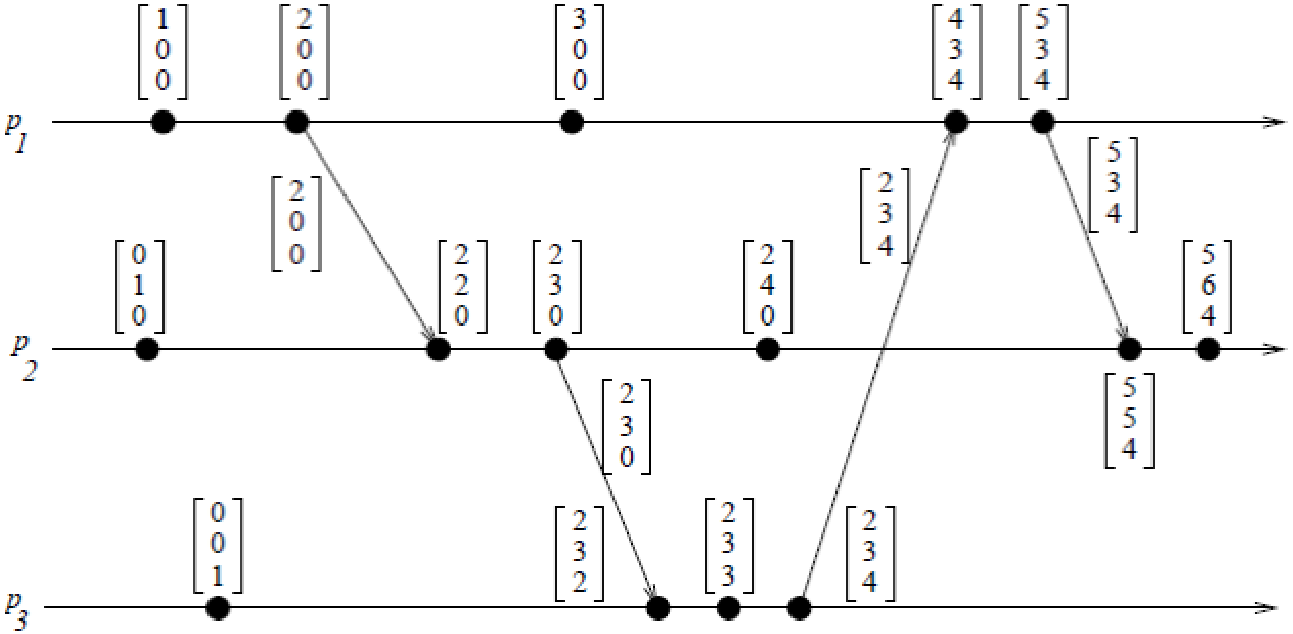
       $$1 \le k \le n: \; vt_i[k] = max(vt_i[k], vt[k]);$$

    2. execute R1;
    3. deliver the message *m*.

Vector Time - Definition

Rules used by p_i for clock updating

- **R2:**
  - timestamp associated with an event is the value of the vector clock of its process when the event is executed
  - initially, a vector clock is [0, 0, 0, ……, 0]



vector clock's progress with d = 1

Vector Time - Definition

Relations for comparing 2 vector timestamps *vh* and *vk*

- $vh = vk \Leftrightarrow \forall\, x : vh[x] = vk[x]$
- $vh \leq vk \Leftrightarrow \forall\, x : vh[x] \leq vk[x]$
- $vh < vk \Leftrightarrow vh \leq vk \text{ and } \exists\, x : vh[x] < vk[x]$
- $vh \,||\, vk \Leftrightarrow \neg(vh < vk) \wedge \neg(vk < vh)$

Vector Time – Basic Properties

- Isomorphism
- Strong Consistency
- Event Counting

Vector Time – Basic Properties

Isomorphism

- if events in a distributed system are timestamped using a system of vector clocks, following property holds:
  - if 2 events *x* and *y* have timestamps *vh* and *vk*, respectively, then

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

$$x \parallel y \Leftrightarrow vh \parallel vk$$

- there is an isomorphism between the set of partially ordered events produced by a distributed computation and their vector timestamps
- very powerful, useful, and interesting property of vector clocks

Vector Time – Basic Properties

Isomorphism

- if the process at which an event occurred is known, test for comparing 2 timestamps can be simplified as:
  - if events *x* and *y* respectively occurred at processes p_i and p_j and have timestamps *vh* and *vk* respectively, then

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

Vector Time – Basic Properties

Strong Consistency

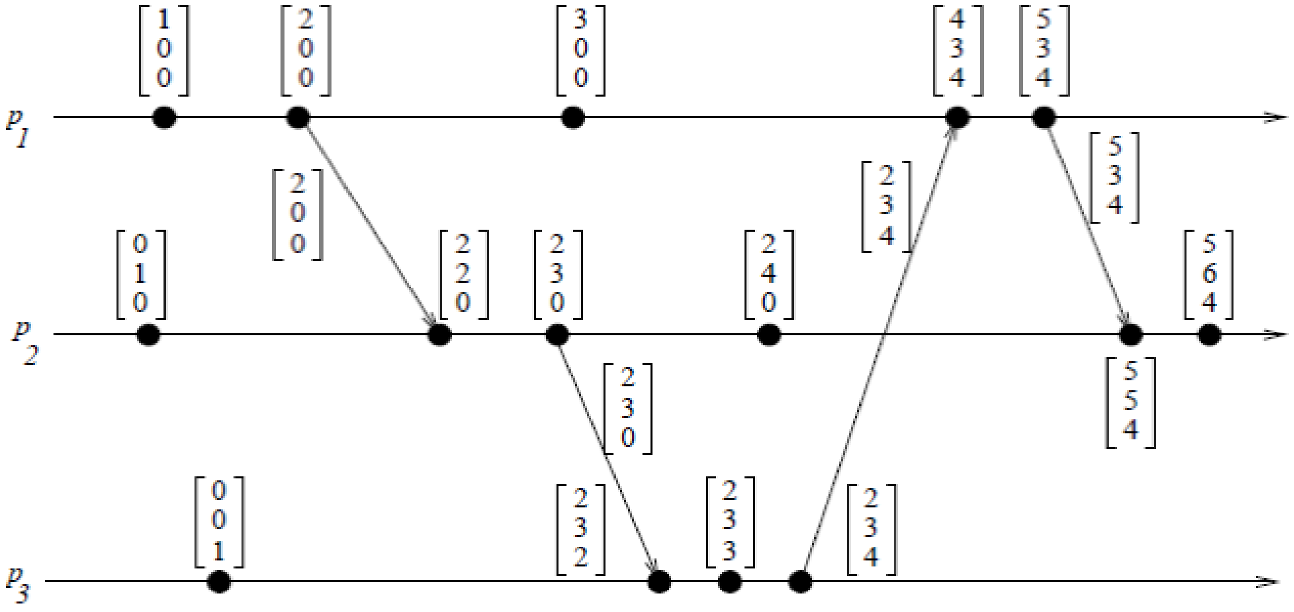
- system of vector clocks is strongly consistent
- examination of vector timestamps of 2 events can determine if the events are causally related
- dimension of vector clocks can't be less than *n* for this property to hold

  (*n* = total no. of processes in the distributed computation)

Vector Time – Basic Properties

Event counting

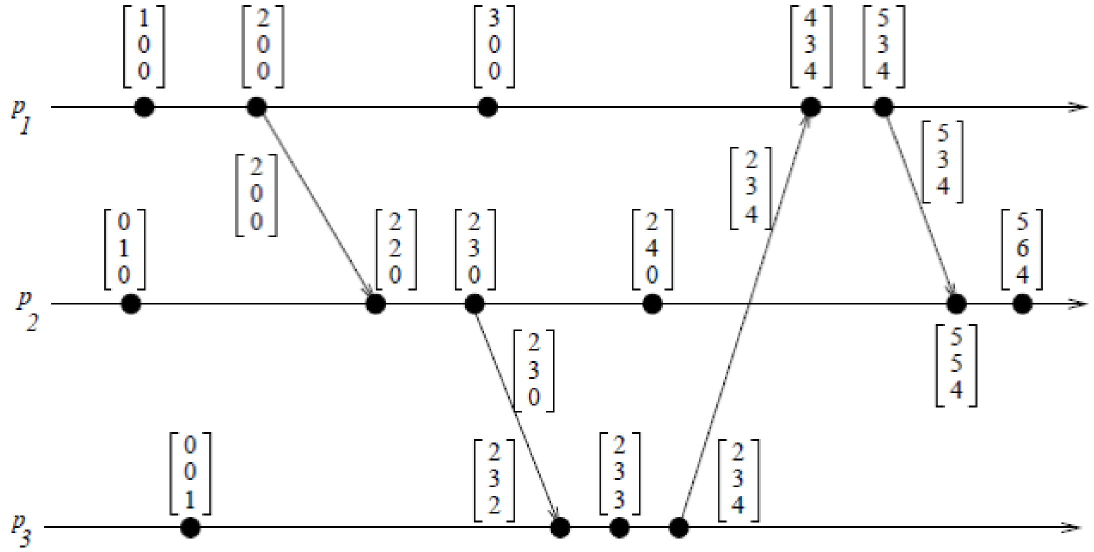
- if d is always 1 in rule **R1** and $vt_i[i]$ is the i^{th} component of vector clock at process p_i
  - then $vt_i[i]$ = no. of events that have occurred at p_i until that instant



Vector Time – Basic Properties

Event counting

- if an event *e* has timestamp *vh*
  - *vh*[*j*] = number of events executed by process p_j that causally precede *e*
  - Σ *vh*[*j*] −1 : total no. of events that causally precede *e* in the distributed computation



Efficient Implementation of Vector Clocks

Why is efficient implementation necessary?

- when no. of processes in a distributed computation is large
    - vector clocks will require piggybacking of huge amount of information in messages
    - messages are required for disseminating time progress and updating clocks
    - message overhead grows linearly with the no. of processes
    - message overhead is not affected by the no. of events occurring at the processors

Singhal–Kshemkalyani's Differential Technique

- based on the 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
- this is more likely when the number of processes is large
  - reason: only a few of them will interact frequently by passing messages
- fundamental idea behind the technique
  - when process p_i sends a message to process p_j, it piggybacks only those entries of its vector clock that differ since the last message sent to p_j

Singhal–Kshemkalyani's Differential Technique

- if entries $i_1, i_2, \ldots, i_{n_1}$ of the vector clock at p_i have changed to $v_1, v_2, \ldots, v_{n_1}$, respectively, since the last message sent to p_j ,
    - then p_i piggybacks a compressed timestamp of the form

$$\{(i_1, v_1), (i_2, v_2), \ldots, (i_{n_1}, v_{n_1})\}$$

to the next message to p_j

- on receiving this message, p_j updates its vector clock as follows:

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

Singhal–Kshemkalyani's Differential Technique

- **Benefit:**
  - reduces message size, communication bandwidth and buffer (to store messages) requirements
- **Worst case:**
  - every element of the vector clock has been updated at p_i since the last message sent to p_j
  - next message from p_i to p_j will need to carry the entire vector timestamp of size n
- **Average case: size of the timestamp on a message will be less than n**

Singhal–Kshemkalyani's Differential Technique

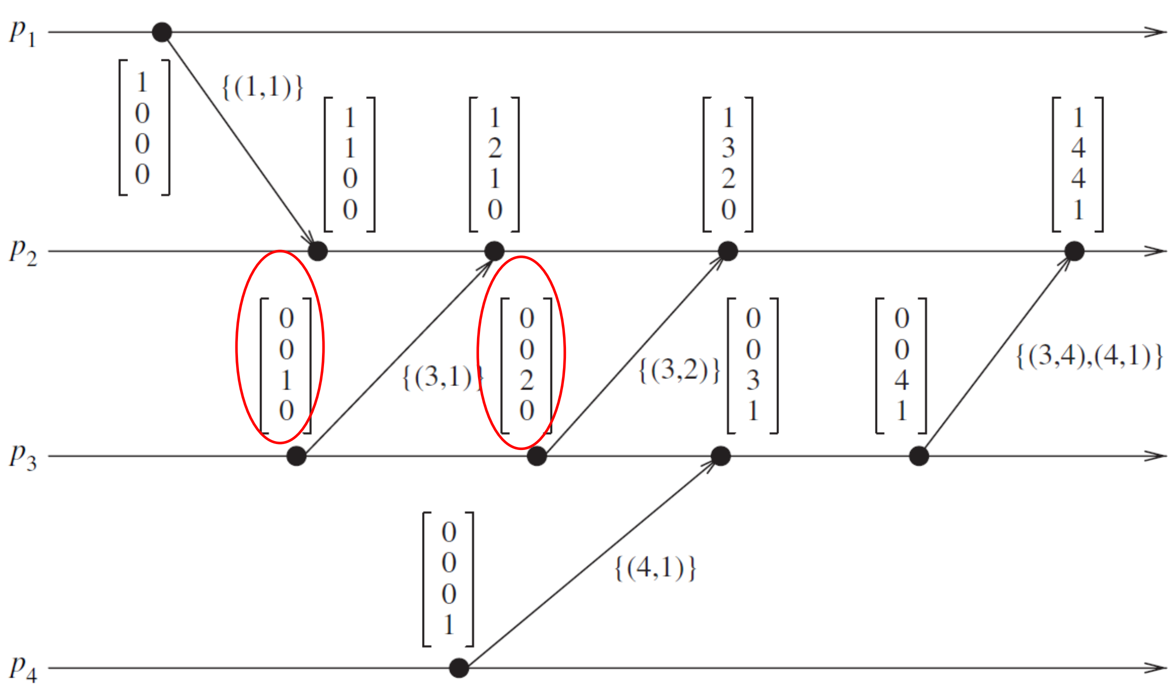
- **Requirements for implementation:**
  - each process must remember the vector timestamp in the message last sent to every other process
  - communication channels follow FIFO discipline for delivery of messages

Singhal–Kshemkalyani's Differential Technique

- in general, the timestamp is of the form:

  $\{(p_1, latest\_value), (p_2, latest\_value), \ldots\}$

  p_i *indicates* p_i^{th} *component of the vector clock has changed*



- 2nd message from p_3 to p_2 contains a timestamp {(3, 2)}
- informs p_2 that the 3rd component of the vector clock has been modified and the new value is 2

Singhal–Kshemkalyani's Differential Technique

Benefits:

- cost of maintaining vector clocks in large systems can be substantially reduced
- especially if the process interactions exhibit temporal or spatial localities
- useful in applications
  - causal distributed shared memories
  - distributed deadlock detection
  - enforcement of mutual exclusion and localized communications

Fowler–Zwaenepoel's Direct-Dependency Technique

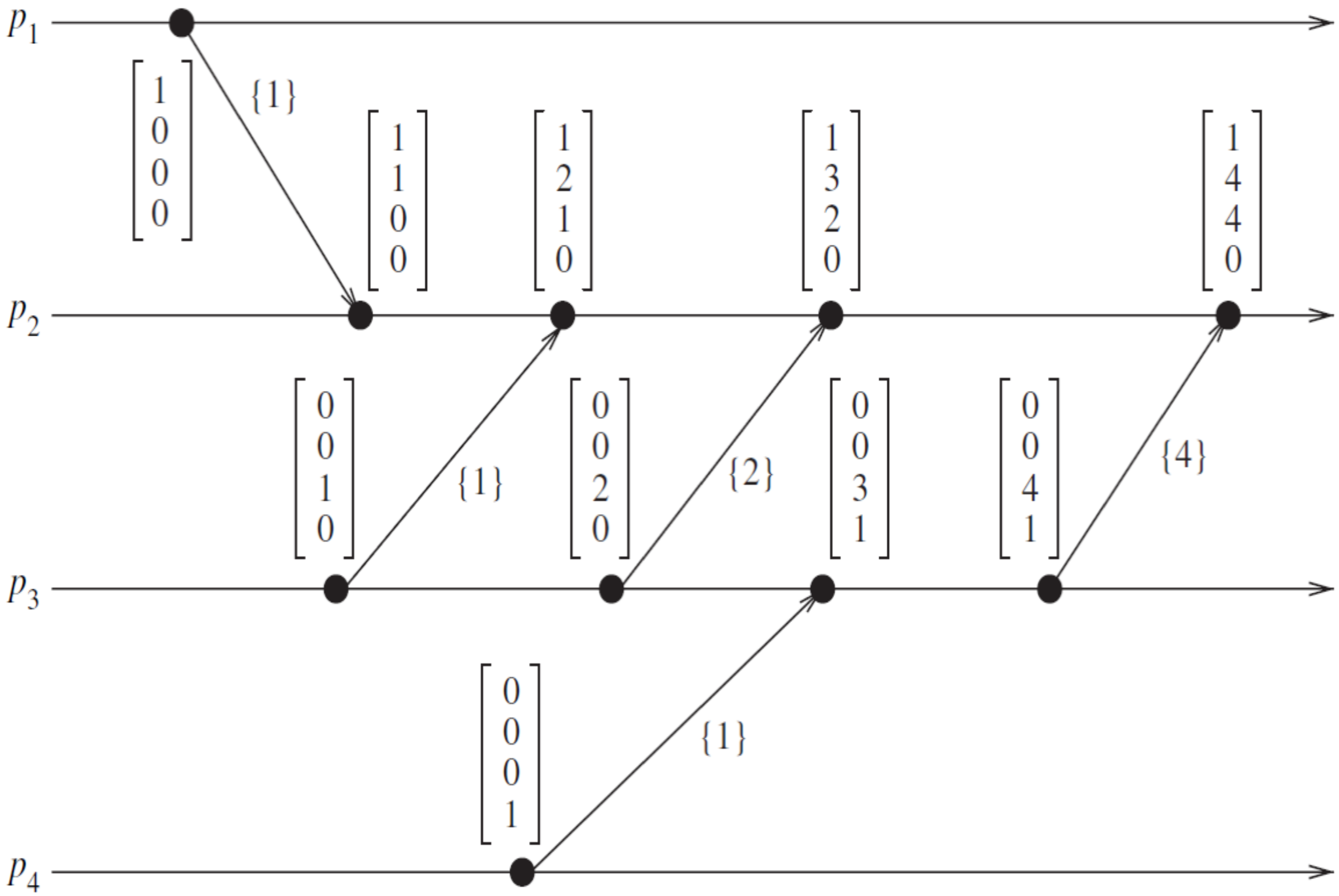
- reduces the size of messages by transmitting only a scalar value
- no vector clocks are maintained on-the-fly
- process only maintains information regarding direct dependencies on other processes
- vector time for an event represents transitive dependencies on other processes
- vector time for event is constructed off-line from a recursive search of the direct dependency information at processes

Fowler–Zwaenepoel's Direct-Dependency Technique

- p_i *maintains a dependency vector* D_i
- initially, $D_i[j] = 0$ for $j = 1,\ldots..,n$
- D_i *is updated using the following rules:*

1. When an event occurs at p_i, $D_i[i] = D_i[i] + 1$ (i.e., increment the vector component corresponding to its own local time by one)
2. When p_i sends a message to p_j , it piggybacks the updated value of $D_i[i]$ in the message
3. When p_i receives a message from p_j with piggybacked value d, p_i updates its dependency vector as follows: $D_i[j] = max\{D_i[j], d\}$

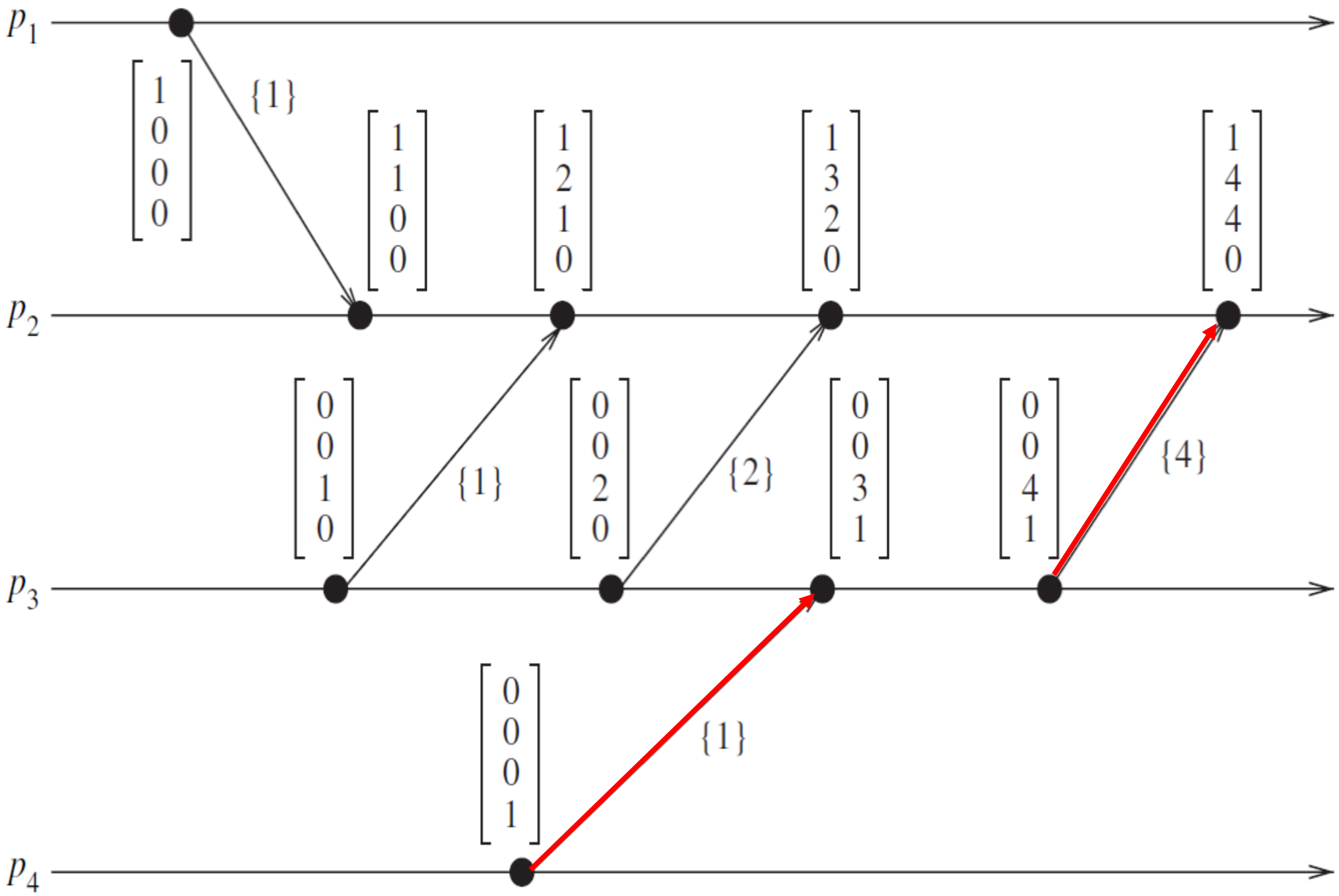
Fowler–Zwaenepoel's Direct-Dependency Technique



Fowler–Zwaenepoel's Direct-Dependency Technique

- *D_i reflects only direct dependencies*
- at any instant, $D_i[j]$ denotes the sequence no. of the latest event on p_j that ***directly*** affects the current state
- this event may precede the latest event at p_j that *causally* affects the current state

Fowler–Zwaenepoel's Direct-Dependency Technique



- when p_4 sends a message to p_3, it piggybacks a scalar that indicates the direct dependency of p_3 on p_4 because of this message
- then p_3 sends a message to p_2 piggybacking a scalar to indicate the direct dependency of p_2 on p_3 because of this message
- *p_2 is indirectly dependent on p_4 since p_3 is dependent on p_4*

Fowler–Zwaenepoel's Direct-Dependency Technique

- transitive (indirect) dependencies are not maintained by this method
- transitive dependencies
  - can be obtained only by recursively tracing the direct dependency vectors of the events off-line
  - involves computational overhead and latencies
- this method is ideal only for those applications
  - that do not require computation of transitive dependencies on the fly
  - eg. applications: causal breakpoints, asynchronous checkpoint recovery

Fowler–Zwaenepoel's Direct-Dependency Technique

- saves cost considerably
- not suitable for applications that require on-the-fly computation of vector timestamps

Physical Clock Synchronization

- no global clock or common memory
- each processor has its own internal clock and its own notion of time
- clocks can drift apart by several seconds per day, accumulating significant errors over time
- clock rates are different, may not remain always synchronized
- for most applications and algorithms that run in a distributed system, need to know time in one or more contexts:
  - time of the day at which an event happened on a specific machine in the network
  - time interval between two events that happened on different machines in network
  - relative ordering of events that happened on different machines in network

Physical Clock Synchronization

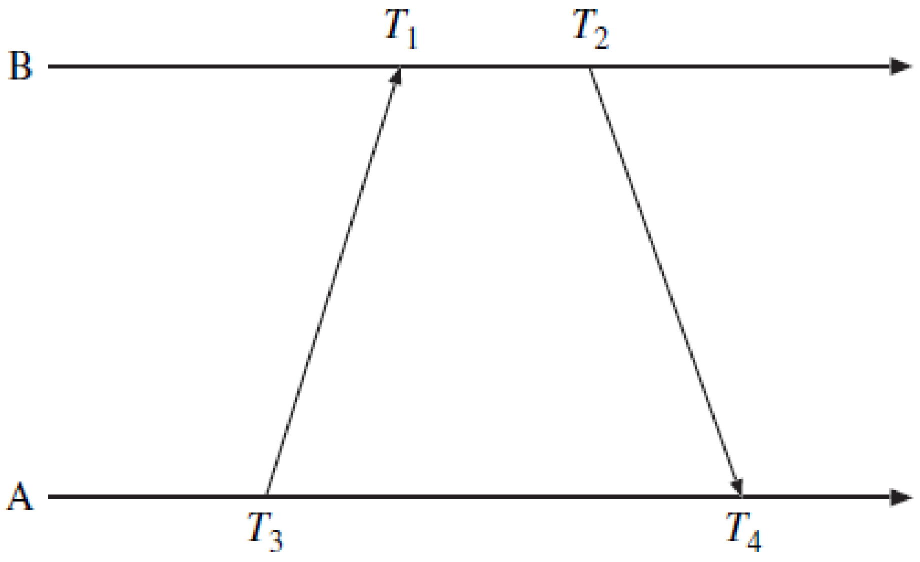
- Clock synchronization –
  - ensuring that physically distributed processors have a common notion of time
  - performed to correct clock skew in distributed systems
  - synchronized to an accurate real-time standard like UTC (Universal Coordinated Time)

- Offset –
  - difference between the time reported by a clock and the real time
  - offset of the clock C_a is given by $C_a(t) - t$ ($C_a(t)$ – time of clock)
  - offset of clock C_a relative to C_b at time $t \geq 0$ is given by $C_a(t) - C_b(t)$

Network Time Protocol

- uses offset delay estimation method
- involves a hierarchical tree of time servers
- primary server at the root synchronizes with the UTC
- next level contains secondary servers, which act as a backup to the primary server
- at the lowest level is the synchronization subnet which has the clients

Network Time Protocol



- Each NTP message includes the latest three timestamps T_1, T_2, and T_3
- T_4 is determined upon arrival
- Peers A and B can independently calculate delay and offset using a bidirectional message stream

- T_1, T_2, T_3, T_4 - values of the 4 most recent timestamps as shown
- assume that clocks A and B are stable and running at the same speed
- $a = T_1 - T_3$, $b = T_2 - T_4$
- if the network delay difference from A to B and from B to A, called differential delay, is small, the clock offset θ and roundtrip delay δ of B relative to A at time T_4

$\theta = (a+b)/2$

$\delta = a - b$

Network Time Protocol

- A pair of servers exchange pairs of timing messages.
- A store of data is then built up about the relationship between the two
- Servers (pairs of offset and delay).
- Each peer maintains pairs (O_i, D_i), where:
  - O_i – measure of offset (θ)
  - D_i – transmission delay of two messages (δ)
- Offset corresponding to the minimum delay is chosen
- Assume that message m takes time t to transfer and m' takes t' to transfer
- Offset between A's clock and B's clock is O
- If A's local clock time is A(t) and B's local clock time is B(t)
  - $A(t) = B(t)+O$
- Then, $T_{i-2} = T_{i-3} + t + O$, $T_i = T_{i-1} - O + t$

Network Time Protocol

- Assume t = t’
- Offset O_i can be estimated as $O_i = (T_{i-2} - T_{i-3} + T_{i-1} - Ti) / 2$
- Round-trip delay is estimated as $D_i = (T_i - T_{i-3}) - (T_{i-1} - T_{i-2})$
- 8 most recent pairs of (O_i, D_i) are retained
- Value of O_i that corresponds to minimum D_i is chosen to estimate O

Recap Quiz

1. If an event e has a timestamp h in scalar clock, then the minimum logical duration required before producing event e is
(a) h (b) h+1 (c) h – 1 (d) 1/h
2. If the height of an event b = 5, then how many events would have preceded b on the longest causal path ending at b if we use a scalar clock?
(a)4 (b) 5 (c) 3 (d) 2
3. In the tie breaking mechanism used in total ordering in scalar clocks, a lower process identifier implies ___ priority
(a) Equal (b) lower (c) higher (d) none of the above
4. Which of the following is not a mechanism to represent logical time in distributed computing systems?
(a) Scalar clock (b) vector clock (c) matrix clock (d)list clock
5. Consider the event counting in vector clocks. if an event e has timestamp vh, then the umber of events executed by process pj that causally precede e will be
(a) vh[j] (b)vh[j-1] (c) vh[j-1] (d)vh[0]
6. Let the total number of processes n = 8, in a distributed environment. Then the dimension of the vector clocks can't be ______ 8 for strong consistency
(a) < (b) > (c)<= (d) >=
7. Which of the following would ensure the efficient implementation of vector clocks if the process interactions exhibit temporal or spatial localities in a distributed system?
(a)Singhal-Kshemkalyani (b)Fowler-Zwaenepoel (c) physical clock synchronization (d) NTP
8. While implementing NTP in distributed computing systems, if A's local clock time is A(t) and B's local clock time is B(t), then the Offset O is
(a) A(t) + B(t) (b) A(t) * B(t) (c) A(t)/B(t) (d) A(t) – B(t)
9. Fowler-Zwaenepoel approach does not maintain ____ dependencies
(a) Commutative (b) associative (c) transitive (d) none of the above
10. In distributed computing systems, the clock skew can be corrected by ______ clock synchronization
(a) Scalar (b) vector (c) matrix (d) physical

Recap Quiz - key

Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10
c	b	c	d	a	a	a	d	c	d

Major References

- Ajay D. Kshemkalyani, and Mukesh Singhal, Chapter 3, “Distributed Computing: Principles, Algorithms, and Systems”, Cambridge University Press, 2008 (Reprint 2013).