More Vector Clocks And Leader Election

Lec9

Agenda

- Last time:
  - Vector clocks
  - Totally Ordered Multicast using lamport timestamps
  - State Machine Replication
- Today:
- Causal ordering
- Vector clock refinements
  - Direct-dependency clocks
  - Reducing message size
- Applications of Vector Clocks
  - Version vectors
  - VCs for race detection
- Leader election algorithms

Why VC

- A → B IFF Timestamp(A) < Timestamp(B)
- Can look at logical timestamps to determine the order of events in the happened before model:
  - If TS(A) < TS(B), then A→B
  - If TS(B) < TS(A), then B→A
  - Else, A and B are concurrent

Recap: Vector Clock Update Rules

1. Initially all clock values are set to the smallest value (e.g., 0).
2. The local clock value is incremented at least once before each primitive event in a process i.e., $v_i[i] = v_i[i] + 1$
3. The current value of the entire logical clock vector is delivered to the receiver for every outgoing message.
4. Values in the timestamp vectors are never decremented.
5. Upon receiving a message, the receiver sets the value of each entry in its local timestamp vector to the maximum of the two corresponding values in the local vector and in the remote vector received.

- Let v_q be piggybacked on the message sent by process *q* to process p; We then have:
  - For i = 1 to num-processes {

    $v_p[i] = \mathbf{max}(v_p[i], v_q[i])$

    }

    $v_p[p] = v_p[p] + 1;$

Class Exercise

- A, B, C, and D are planning to meet next week for dinner.

1. A suggests they meet on Wednesday by broadcasting the message
2. Later, D discuss alternatives with C, and they decide on Thursday instead.
3. D also exchanges email with B, and they decide on Tuesday.
4. When A pings everyone again to find out whether they still agree with her Wednesday suggestion, she gets mixed messages: C claims to have settled on Thursday with D, and B claims to have settled on Tuesday with D.
5. D can't be reached, and so no one is able to determine the order in which these communications happened, and so none of A, B, and C know whether Tuesday or Thursday is the correct choice.

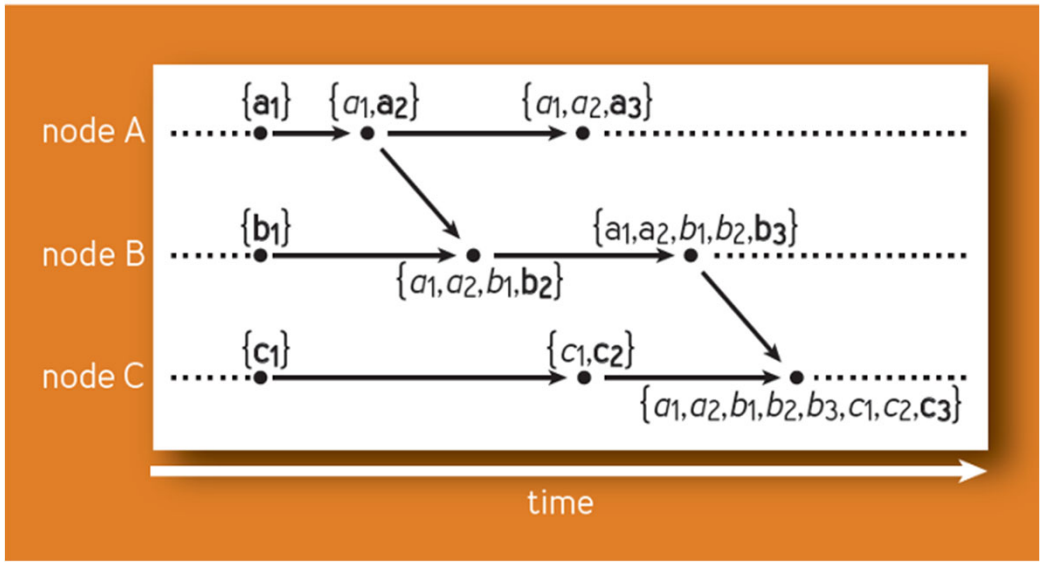
Simulate this scenario with message-passing and vector clocks. Can you resolve the conflict? At the end of the exercise, each group should prepare and submit the space-time diagram.

Causal Order Broadcast

Causality

- Want to capture cause-and-effect relations between events/actions
- External causality: messages exchanged "outside" the system
- Cannot be detected by the system and is usually approximated by physical time
- Vector clocks compression of causal histories:
  - {a1,a2,a3,b1,b2} : a:3, b:2

FIGURE 2: CAUSAL HISTORIES



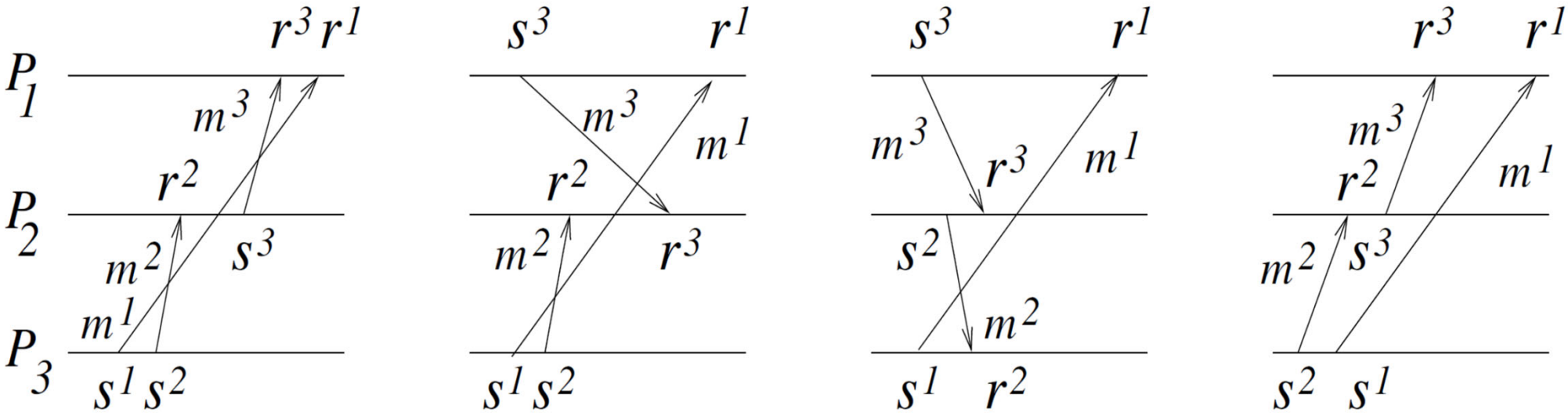
Vector Timestamps and Causality

- We have looked at *total order* of messages where all messages are processed in the same order at each process.
- *Causal order* is used when a message received by a process can potentially affect any subsequent message sent by that process. Those messages should be received in that order at all processes. Unrelated messages may be delivered in any order.

$$S_1 \rightarrow S_2 \Rightarrow \neg(r_2 < r_1)$$

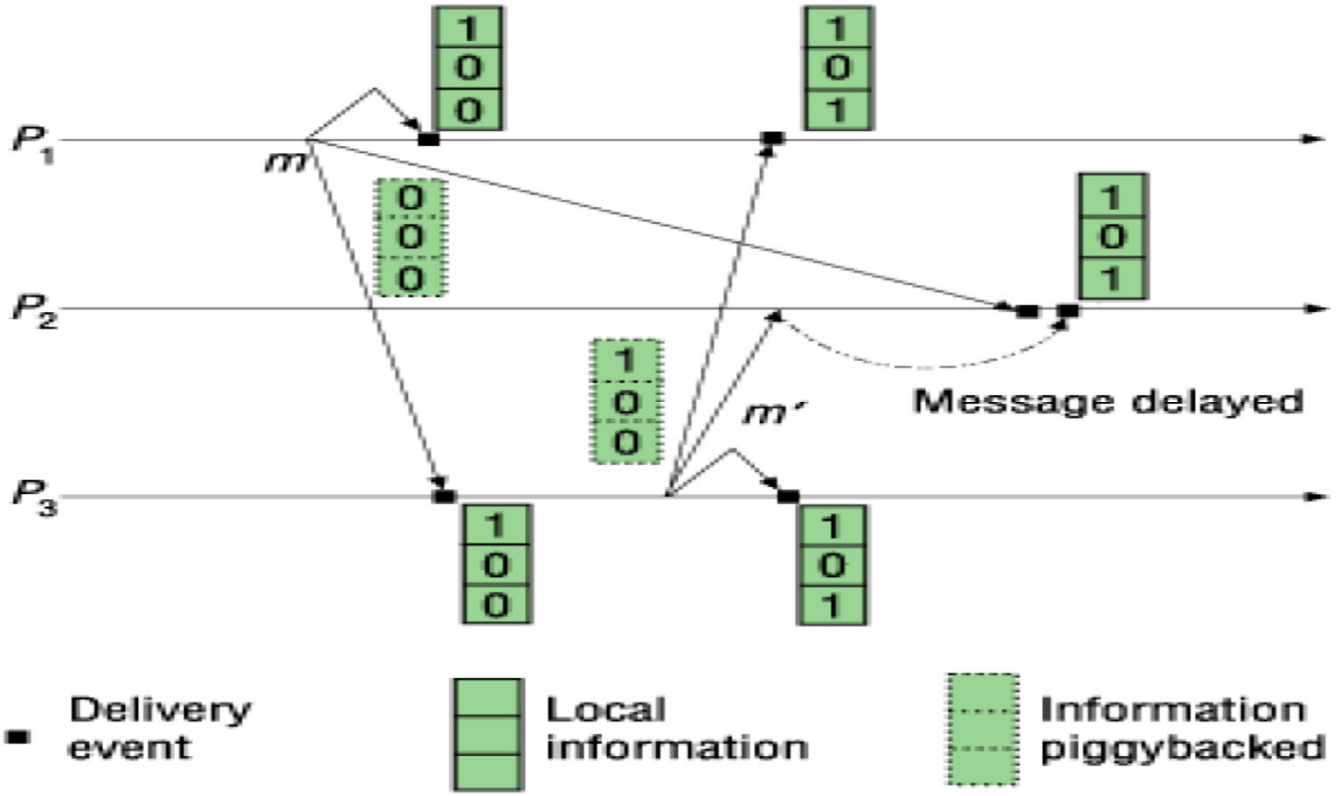
Locally preceedes

Causal Order Examples



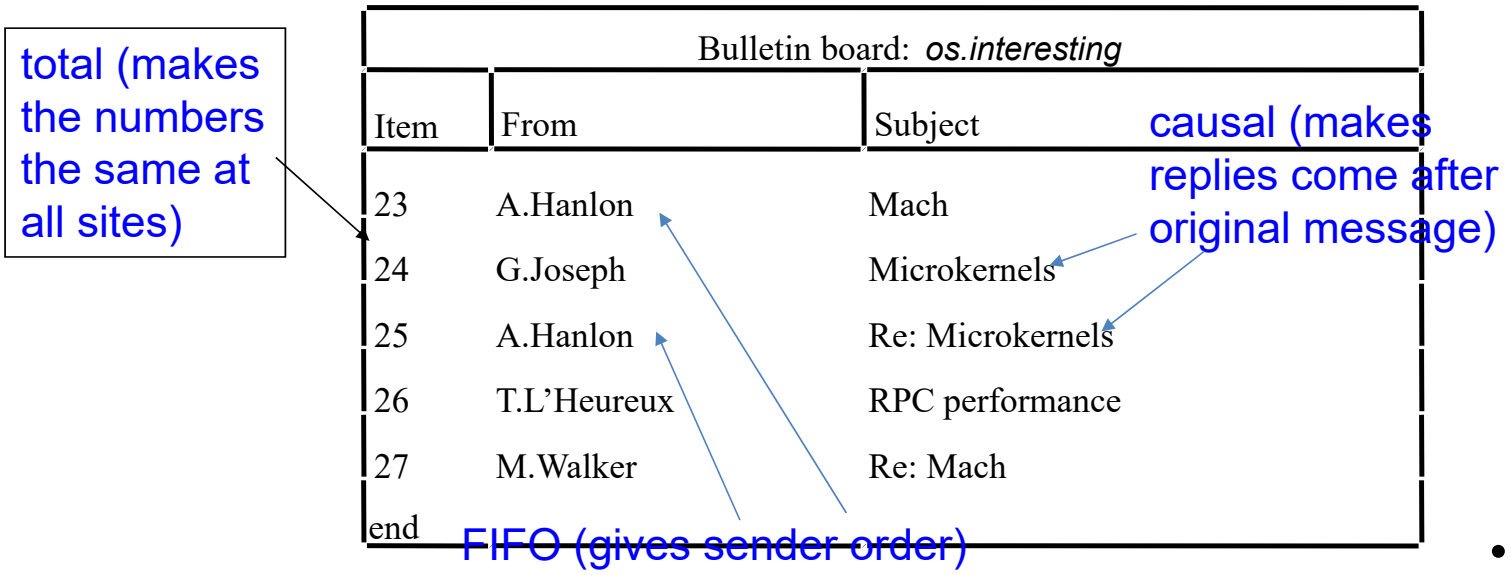
Causal Message Delivery

- Basic idea: Delay delivering a message until all messages in the causal past of m have been received



Display from a Bulletin Board Program

- Users run bulletin board applications which multicast messages
- One multicast group per topic (e.g. *os.interesting*)
- Require reliable multicast - so that all members receive messages
- Ordering:

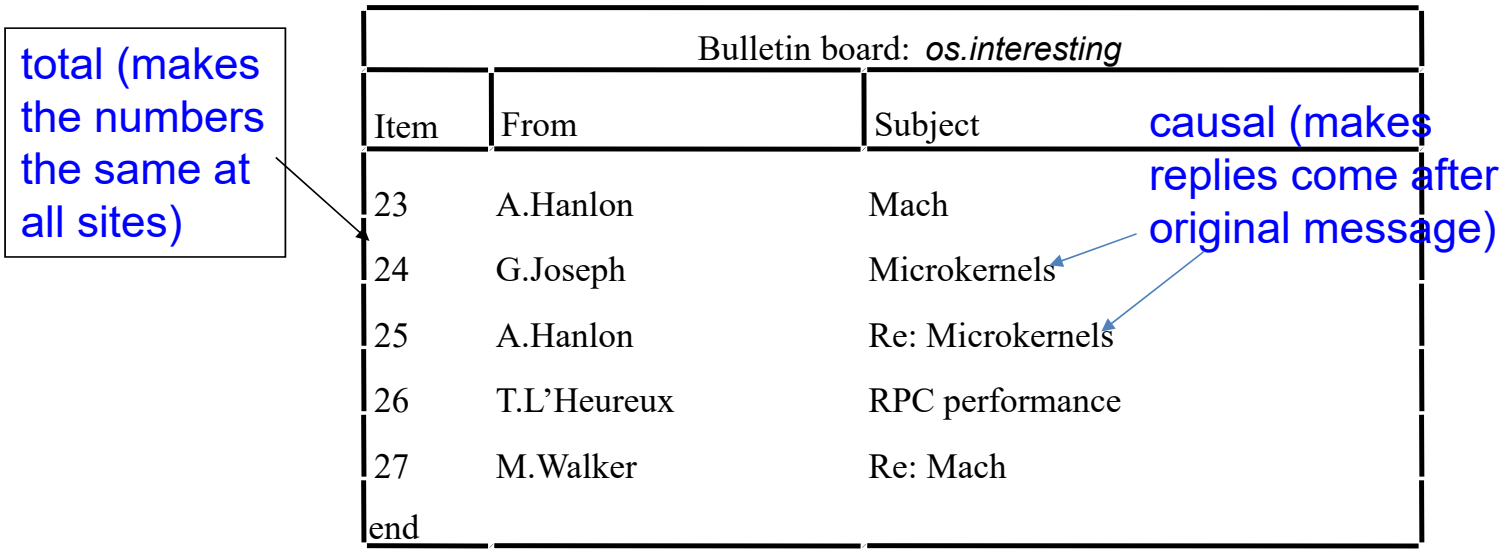


Example Application: Bulletin Board

- A totally-ordered multicasting scheme does not imply that if message B is delivered after message A, that B is a reaction to A.
- Totally-ordered multicasting is too strong in this case.
- The receipt of an article causally precedes the posting of a reaction. The receipt of the reaction to an article should always follow the receipt of the article.

Example Application: Bulletin Board

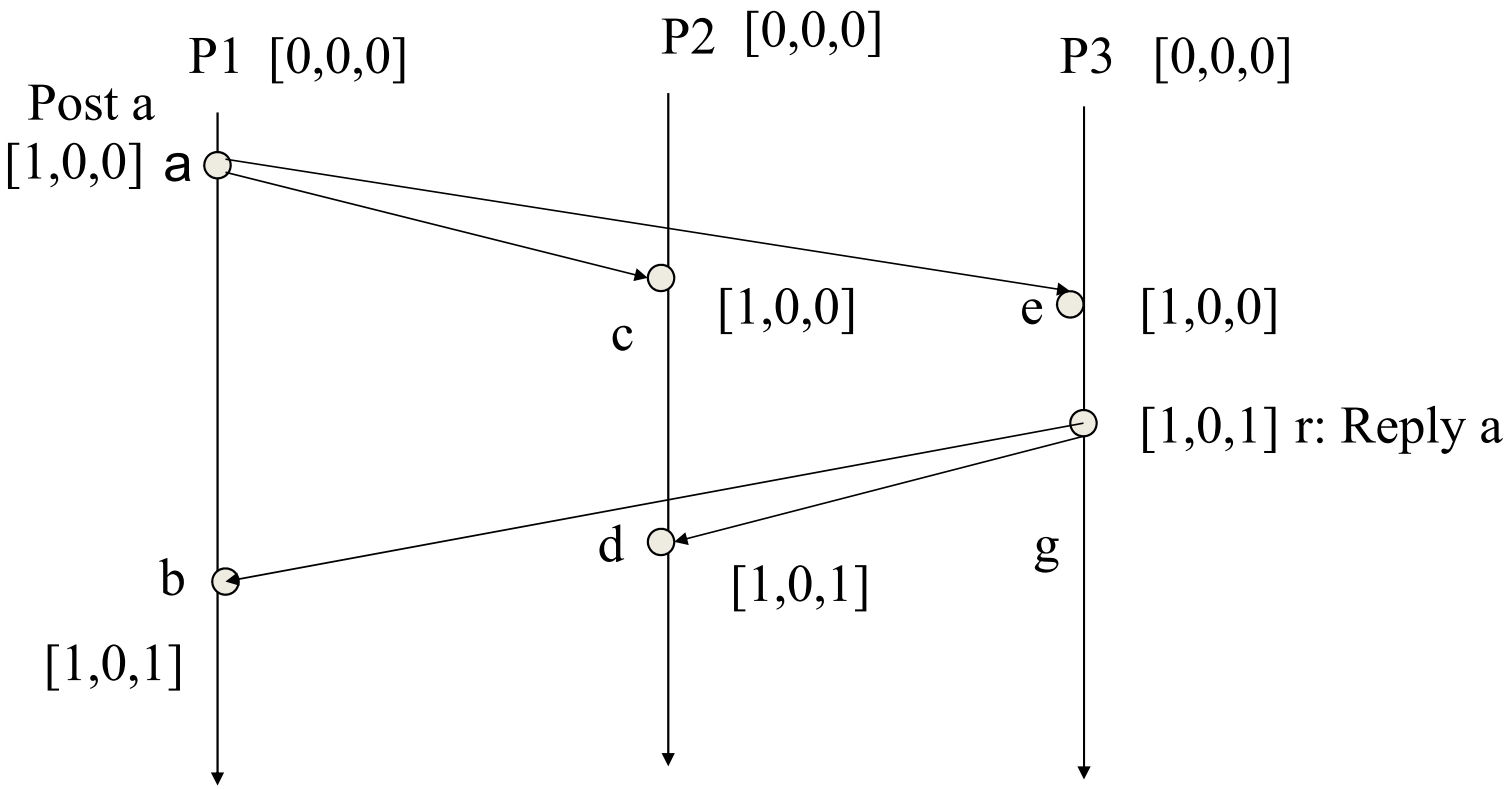
- If we look at the bulletin board example, it is allowed to have items 26 and 27 in different order at different sites.
- Items 25 and 26 may be in different order at different sites.



Bulletin board: *os.interesting*

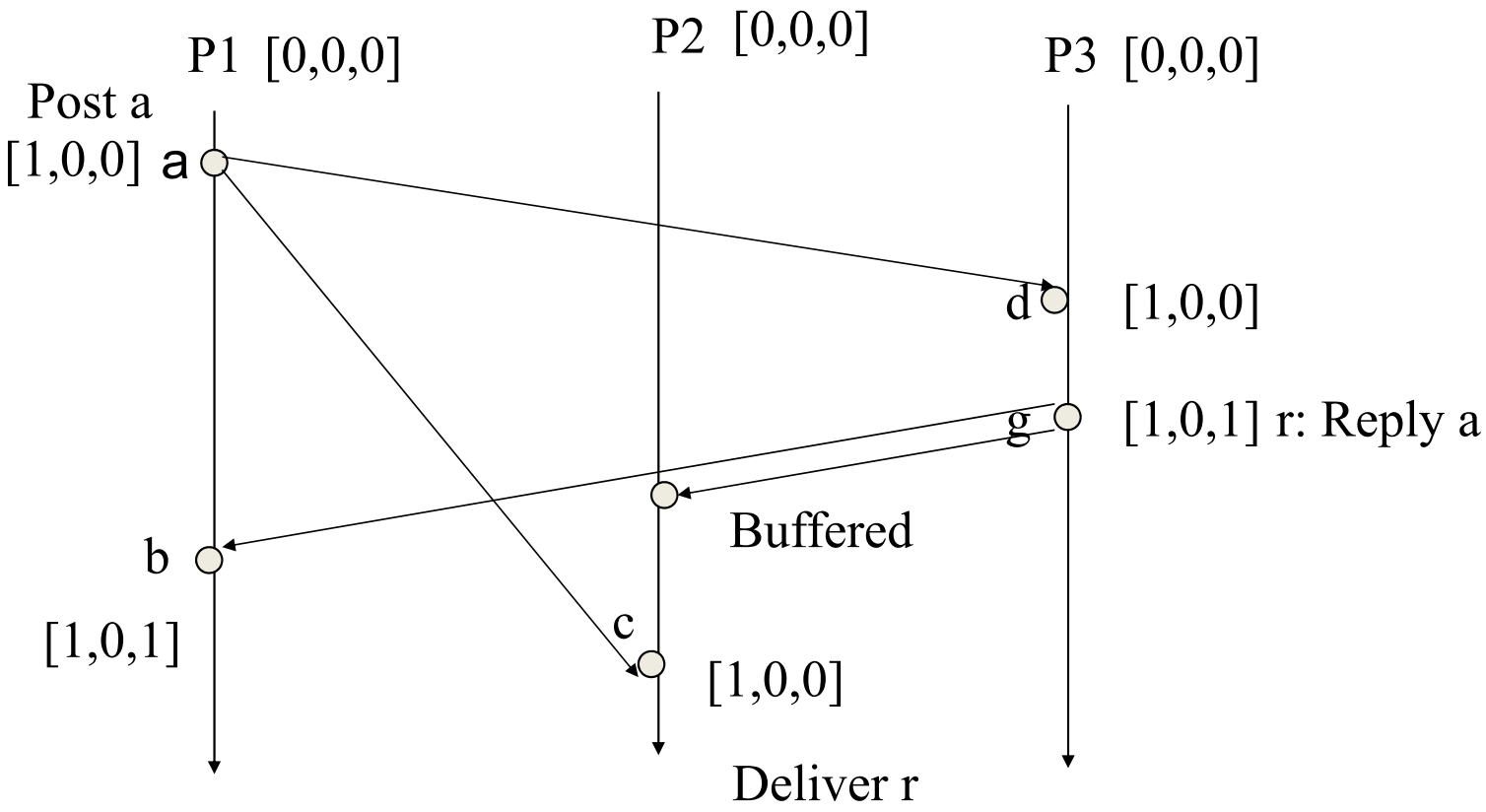
Item	From	Subject
23	A.Hanlon	Mach
24	G.Joseph	Microkernels
25	A.Hanlon	Re: Microkernels
26	T.L'Heureux	RPC performance
27	M.Walker	Re: Mach
end		

Example Application: Bulletin Board



What if message *a* arrives at P2 before the reply *r* from P3 does

Example Application: Bulletin Board



The message *a* arrives at P2 after the reply from P3; The reply is not delivered right away.

Vector Clock Refinements

Direct Dependency Clocks

- Simplification of vector clocks.
- Each process still maintains vector timestamps
- But, only sends it's logical timestamp component with messages (i.e., process-i sends v[i], instead of entire vector v)
- On receive, process-j updates only two VC components:
  - It's own v[j], is simply incremented
  - The sender's component, v[i], is updated using max of earlier value and value sent in the message
- Direct dependency clocks can capture "directly happened before" relationships
  - Message path contains at most one message.

Singhal-Kshemkalyani Optimization

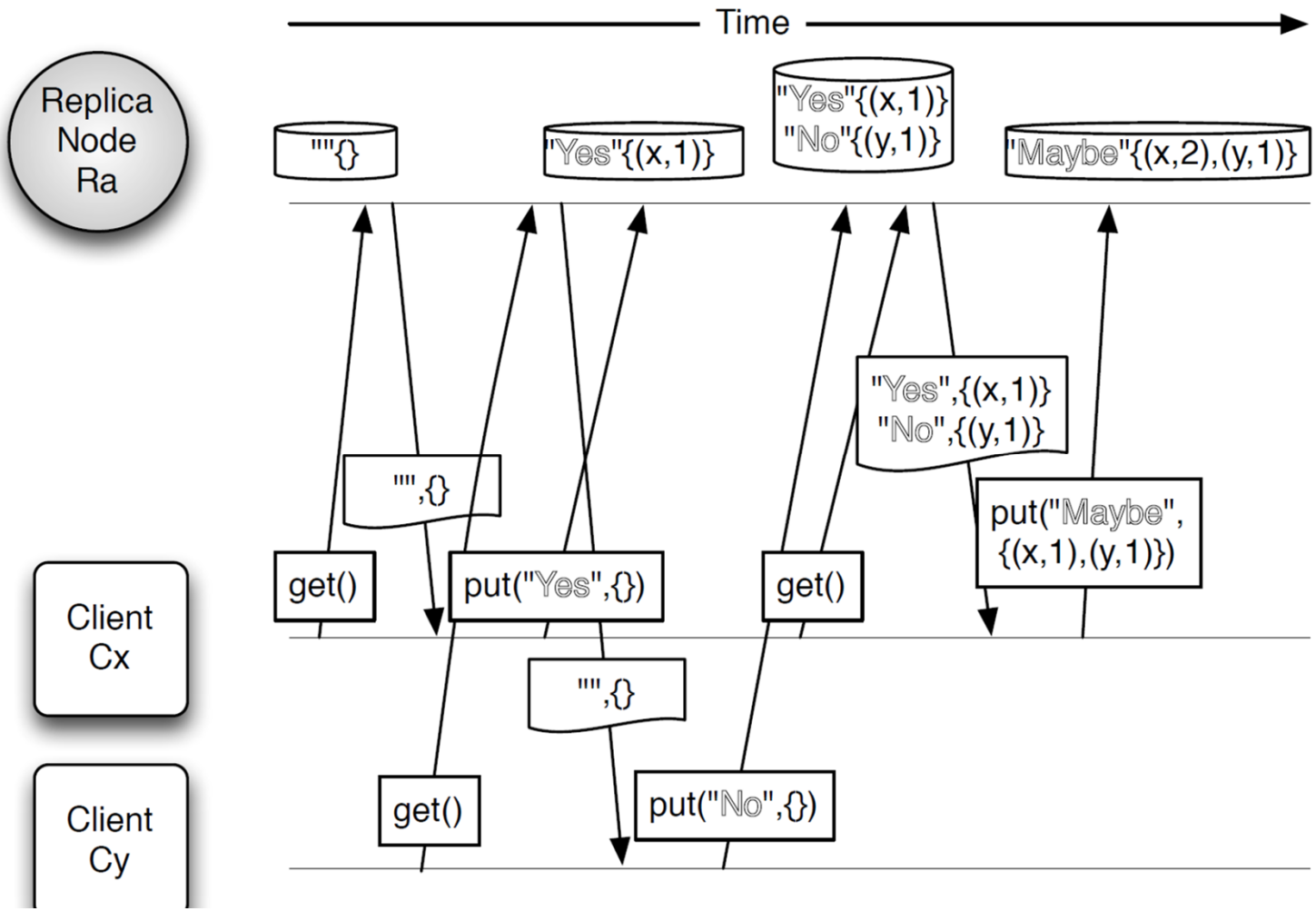
- Goal: reduce the size of VC sent along with messages
- Main idea: Send compressed timestamps (only those components that have changed since the last message to the same recipient)
- BUT: Each process must remember the VC they last sent to other processes.
  - O(n^2) storage overhead
- Singhal-Kshemkalyani technique cuts this overhead down to O(n)
  - Maintain Last-sent and Last-updated vectors, which contain only single timestamps last sent/updated to/from each process, instead of entire vectors

Version Vectors

- Convential VC: One entry for every process, including clients!
  - Can be in the millions
- Version vectors: maintained by each server replica for each object
- N replicas for an object, then $V_j[i]$ is number of updates generated at replica i that is known by replica-j
- Can be expanded/pruned as replicas leave and join
  - Zero-valued entries add no distinguishing information

- Dynamic Version Vector Maintenance. David Ratner, Peter Reiher, and Gerald Popek

Version Vector Example



Resettable VC

- Bounded: At phase boundaries, a process can reset its VC

Resettable Vector Clocks *

This paper appears in PODC 2000

Anish Arora †
Department of Computer and Information Science
The Ohio State University
Columbus, Ohio 43210 USA
anish@cis.ohio-state.edu

Sandeep Kulkarni
Department of Computer Science and Engineering
Michigan State University
East Lansing, Michigan 48824 USA
sandeep@cse.msu.edu

Murat Demirbas
Department of Computer and Information Science
The Ohio State University
Columbus, Ohio 43210 USA
demirbas@cis.ohio-state.edu

ABSTRACT

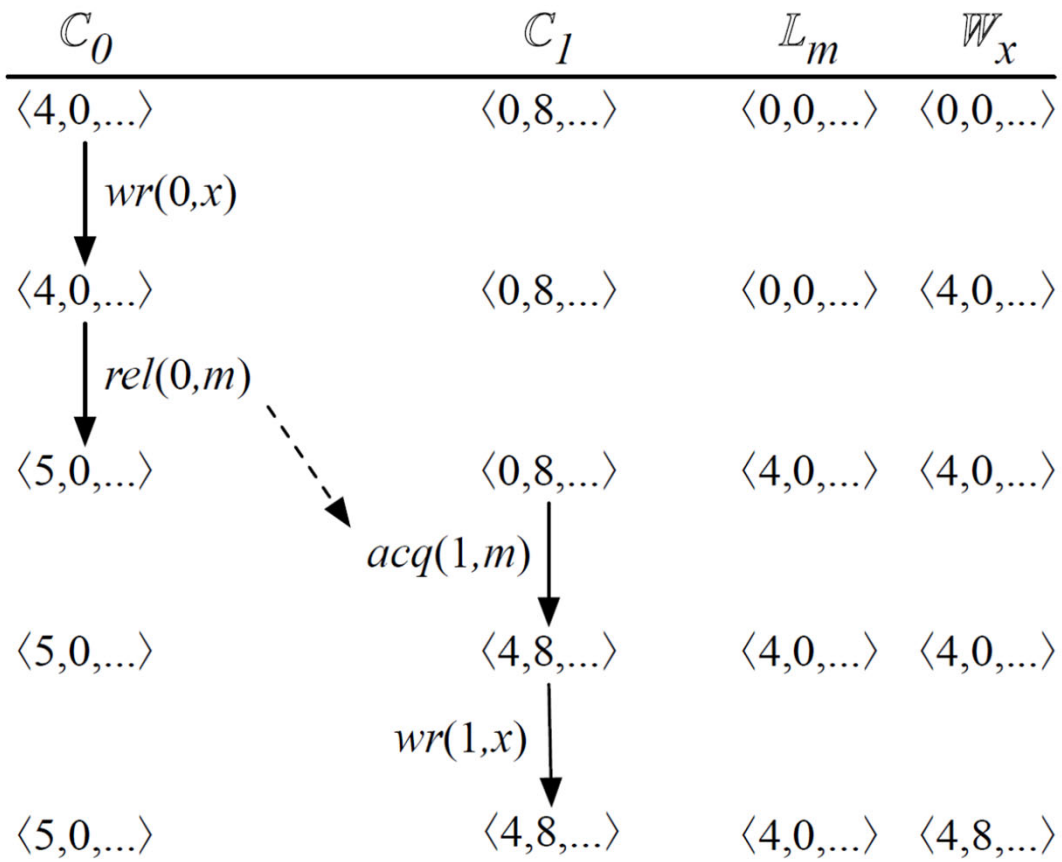
Vector clocks (VC) are an inherent component of a rich class of distributed applications. In this paper, we consider the problem of realistic —more specifically, bounded-space and fault-tolerant— implementation of these client applications. To this end, we generalize the notion of VC to resettable vector clocks (RVC), and provide a realistic implementation

ship among events is a powerful concept that provides one such abstraction for reasoning about events in a distributed computation. Fidge [7] and Mattern [11] independently proposed vector clocks that capture the causality relation between events. Vector clocks (VC) are extensively used in distributed applications, such as, distributed debugging [8], checkpointing and recovery, and causal communication [15].

VCs for Data Race Detection

- Maintain VCs for each thread u C_u , and each lock m L_m
- Thread u releases lock m, update L_m to C_u
- Thread u acquires lock m, C_u updated with max(C_u, L_m)
- For identifying conflicting accesses, R_x and W_x VCs maintained for each variable x
- $R_x(t), W_x(t)$ records the clock of last read and write to x by thread t
- Race-free condition: $W_x \leq C_u$ and $R_x \leq C_u$

Race Detection Example



Leader Election

Why Leader Election

- Given a group of processes, we want to elect a leader that is a "special" designated process for certain tasks
  - Who is the primary replica?
  - Useful for implementing centralized algorithms, since leader can broadcast messages to keep replicas in sync

- All processes must agree on who the leader is
- Any process can call for an election at any time
- A process can call for only one election at a time
- Multiple processes can call for an election simultaneously
- Result of the election should not depend on which process calls for it

Chang-Roberts Leader Election

• Processes arranged in a ring, first phase:

1. To start an election, send your id clockwise as part of "election" message
2. If received id is greater than your own, send the id clockwise
3. If received id is smaller, send your id clockwise
4. If received id is equal, then you are the leader (we assume unique id's)

Second phase:

1. Leader sends an "elected" message along with id
2. Other processes forward it and can leave the election phase

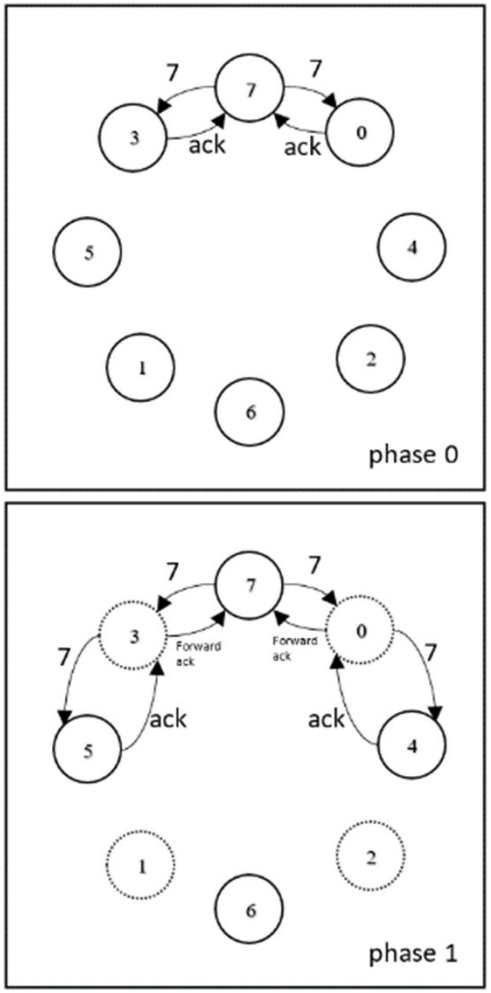
Analysis

- Worst-case: 3N-1 messages
- N-1 messages for everyone to circulate their value
- N messages for election candidate to be confirmed
- N 'elected' messages to announce the winner

Binary Search based Leader Election

- Election proceeds in phases, and processes know which phase they are in
- In phase k, processes send their id in both directions to travel distance 2^k and return back to it
- If both messages return, then process continues to phase k+1
- When a process receives an outgoing id, it compares against its own
  - If received id is smaller, then that message is discarded and not forwarded
  - If greater, then forwards to the next hop,
    - Or bounces the message back to the original sender, if message reached hop limit
  - If equal, then process declares itself the leader

HS algorithm example



END