Event Ordering

Greg Bilodeau

CS 5204

November 3, 2009

Virginia Tech

Fault Tolerance

- How do we prepare for rollback and recovery in a distributed system?

- How do we ensure the proper processing order of communications between distributed processes?

Time

- No shared clock
- All specifications of a system must be given in terms of events observable within that system
- Can we construct a concept of “time” that would be useful from events of a distributed system?

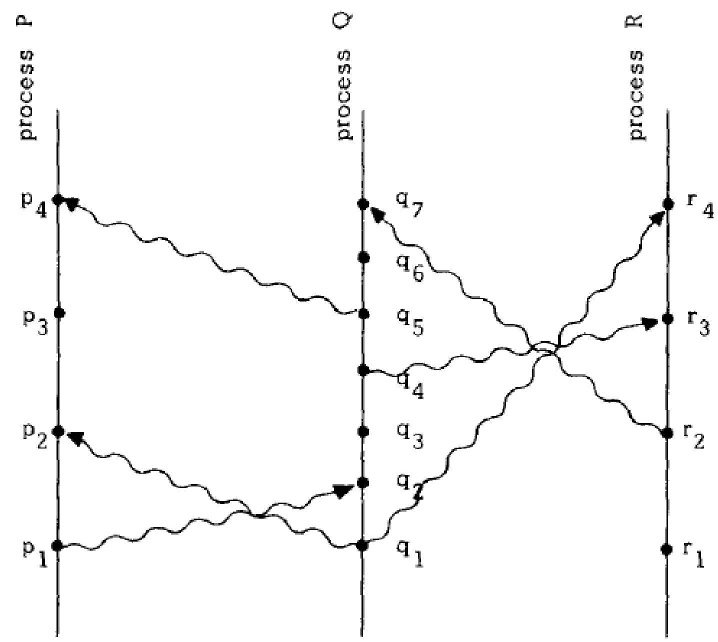
Events

- An event is just an event of interest – example: a communication between processes
- Single process defined as totally ordered sequence of events

Events

- "Happened before" relation →:
  - If a and b from same process and a comes before b
  - a is a send and b a receive from different processes
  - If a → b and b → c, then a → c
  - Events a and b concurrent if !a → b and !b → a

Fig. 1.



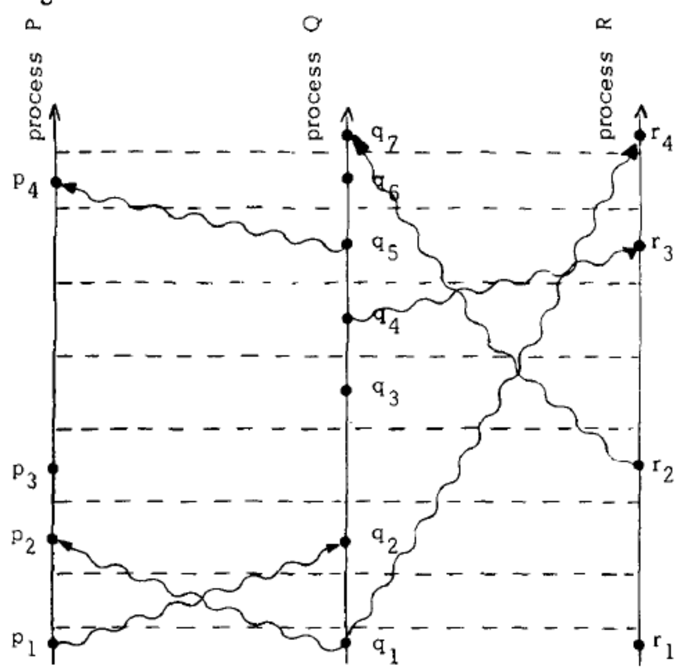
Events

- Another definition: events causally affect each other
- a $\rightarrow$ b means it is possible for a to causally affect b
- a and b are concurrent if they cannot causally affect each other

Logical Clocks

- Assigns a number to an event
- Simple counter
- Clock Condition:
  - For a, b: if a → b then C(a) < C(b)
  - $C(p_1) < C(p_2)$
  - $C(p_1) < C(q_2)$
  - C1: Line between local events
  - C2: Line between send and receive

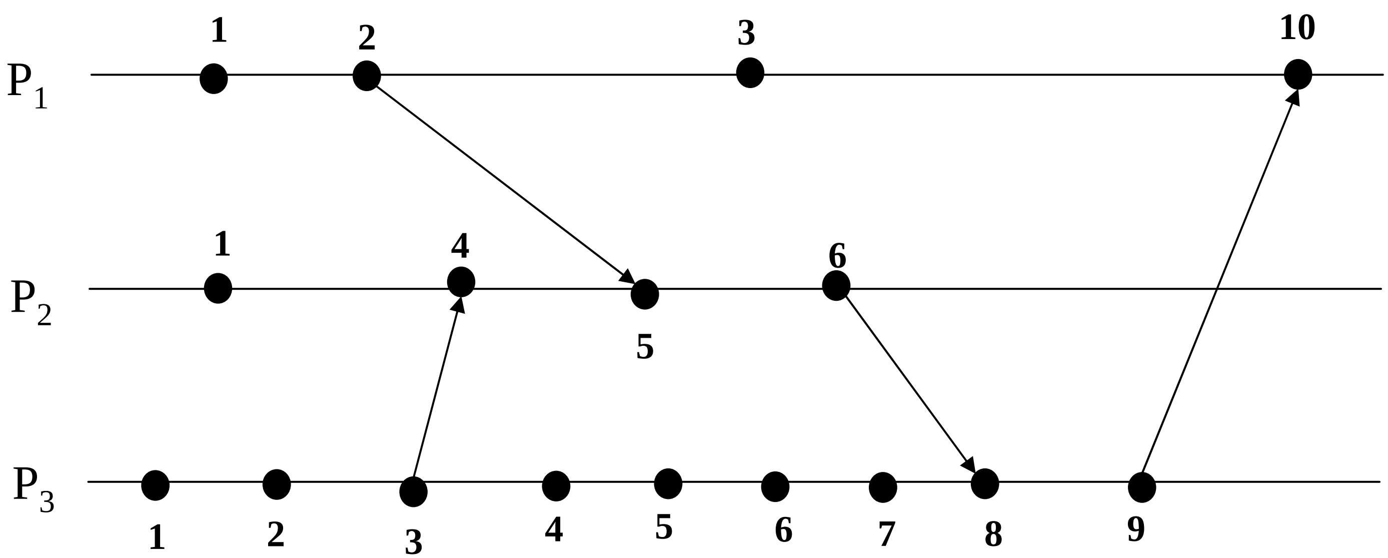
Fig. 3.



Logical Clocks

- How we meet these conditions:
- C1:
  - Each process increments its clock between successive events
- C2:
  - Requires each message to include a timestamp equal to time the message was sent
  - Receiver sets its own clock to a value greater than or equal to its own value and greater than the timestamp from the message – cannot move its clock backward

Example of Lamport's Algorithm



Lamport's Approach

- Just order events according to "times" at which they occur
- If "times" are equal, choose one to proceed
- Example: mutual exclusion problem
  - Assume all messages received in order
  - Assume all messages eventually received
  - Each process has own request queue
  - Conditions we must achieve:
    - Process with resource must release before used by others
    - Requests must be granted in order made
    - Every request must eventually be granted

Lamport's Mutual Exclusion Example

- Process P_i sends T_m:P_i message to all others, adds message to own request queue
- Process P_j adds resource request to its queue, sends a time stamped acknowledgement
- When finished, P_i removes the message from its queue, sends a time stamped removal to all others
- Process P_j removes the resource request from the queue
- P_i can use the resource when:
  - It's own request is ordered before any others in its queue
  - It has received a message from all others stamped later than T_m

Limits of Lamport

- Clock times cannot guarantee causal relationship
  - We can say if a → b then C(a) < C(b)
  - CANNOT say if C(a) < C(b) then a → b
  - Concept of "time" is exclusive to each process, i.e. causality only in same process

- We can provides this through:
  - Using physical clocks
  - Using vector clocks

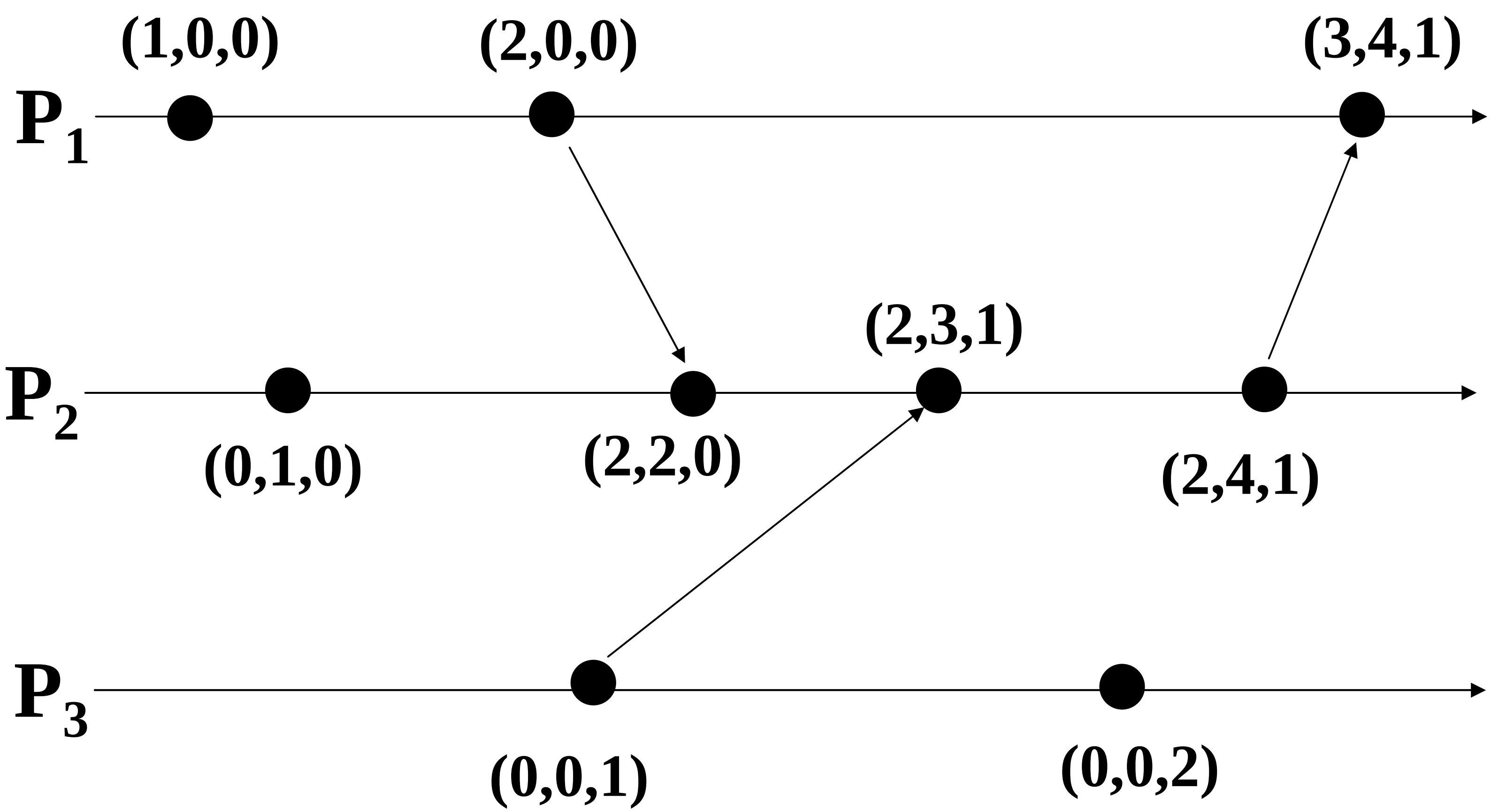
Vector Time

- The vector time for p_i, $VT(p_i)$:
  - Length n, where n is number of processes in group
  - Initialized to all zeros
  - p_i increments $VT(p_i)[i]$ when sending m
  - Each message sent in time-stamped with $VT(p_i)$
  - Receiving processes in the group modify their vector clock:

$$\forall k \in 1 \cdots n : VT(p_j)[k] = \max(VT(p_j)[k], VT(m)[k]).$$

  - Vector time-stamp of m counts the number of messages that causally precede m on a per-sender basis

Vector Clocks



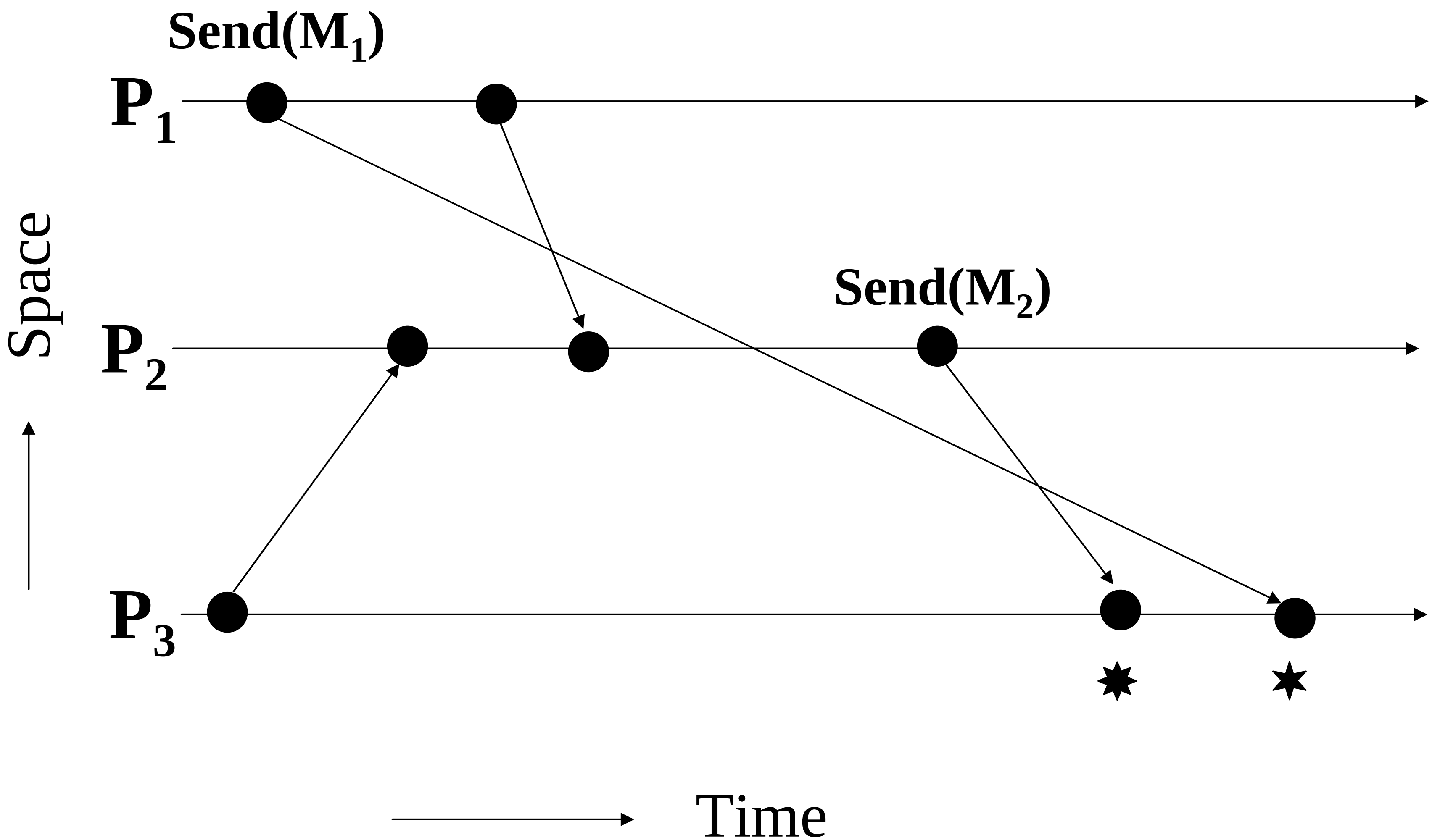
Birman-Schiper-Stephenson

- ISIS toolkit – tools for building software in loosely coupled distributed environments
- CBCAST – multicast primitive
  - Fault-tolerant, causally ordered message delivery
  - Asynchronous
- ABCAST
  - Extension allowing total ordering
  - Synchronous
- Group communication
- Imposes overhead proportional to group size

Birman-Schiper-Stephenson

- Cooperative processes form groups
- Processes *multicast* to all members of their group(s)
- Delivery times are uncertain…possible to receive messages out of causal sequence
- Message processing mechanism must provide lossless, uncorrupted and *sequenced delivery*
- Distinction between "receiving" and "delivering"
  - Allows delay of delivery until some condition satisfied – i.e. causal order maintained

Causal Ordering of Messages



Vector Clocks in BSS

- Values in vector clock indicate how many multicasts preceded message by each process; must process same number from each before same state is reached
- Recipient will delay delivery of the message using a delay queue until corresponding number of messages have been received

$$\forall k : 1 \cdots n \begin{cases} VT(m)[k] = VT(p_j)[k] + 1 & \text{if } k = i \\ VT(m)[k] \le VT(p_j)[k] & \text{otherwise} \end{cases}$$

Conclusions

- Causal relationships between events of processes in a distributed environment are critical when discussing fault-tolerance and rollback/recovery
- Achieving total ordering of events is difficult in the absence of a shared clock
- Mechanisms to provide shared logical clocks use simple counters but can enforce causal orderings

Questions?