Chapter 13: Synchronization in Distributed Systems

Synchronization in Distributed Systems

- Objectives
  - To develop an understanding of distributed view on time
  - To learn about alternative clock designs and implementation
  - To control sequentialized, distributed processes operations
  - To select a unique “master” in distributed settings
  - To combine multicast and group communications
- Topics
  - Time Synchronization
  - Clock Synchronization
  - Logical and Vector Clocks
  - Mutual Exclusion
  - Leader Election
  - The Multicast Problem

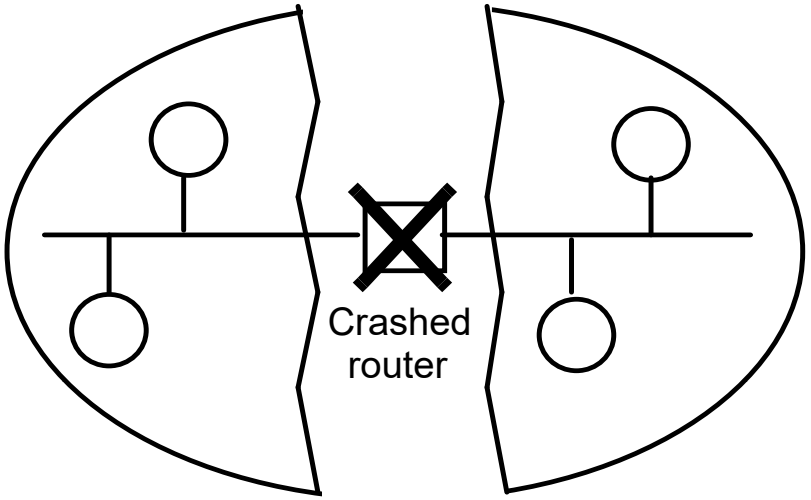
Need for Synchronization

- Being able to communicate as such is not sufficient

- Communicating nodes also need to coordinate and synchronize for various tasks
  - To synchronize with respect to time
  - Not to access a resource (*e.g.*, a printer, or some memory location) simultaneously
  - To agree on an ordering of (distributed) events
  - To appoint a coordinator

Assumptions and Algorithms

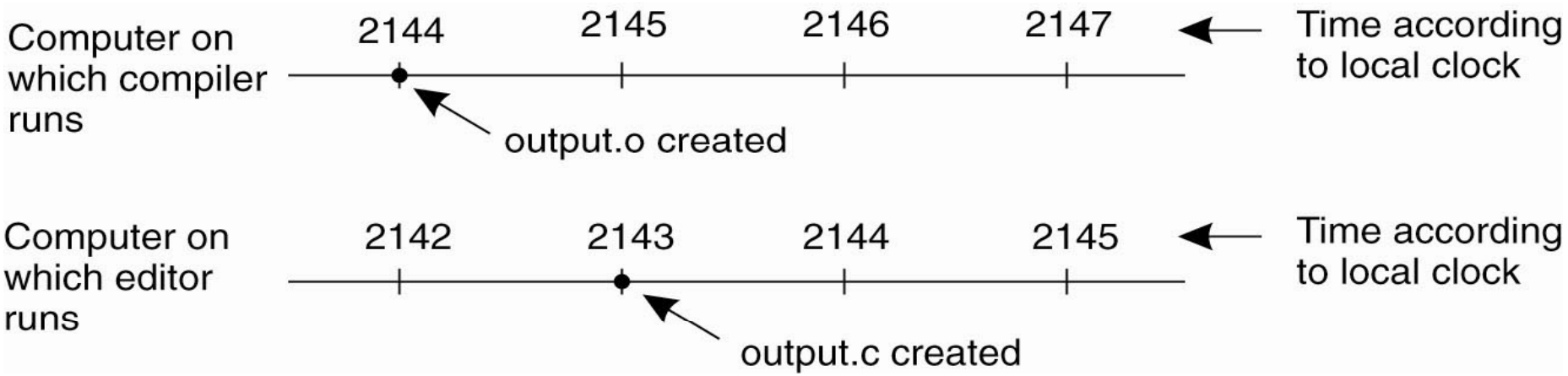
- Assumptions
  - Communication is reliable (but may incur delays)
    - Network partitioning might occur
  - Detecting failure is difficult
  - Time-out is not reliable



- Algorithms
  - Distributed mutual exclusion
  - Elections
  - Multicast communications

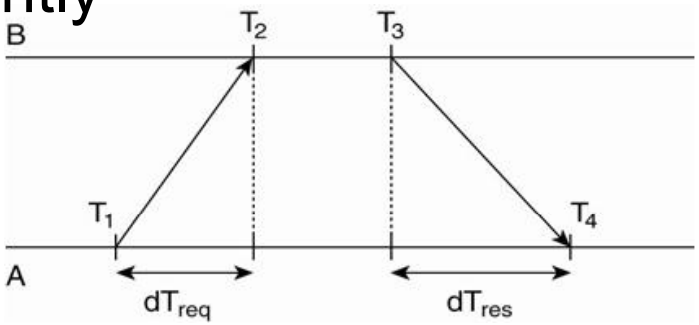
Clock Synchronization

- Time consistency is not an issue for a single computer
  - Time never runs backward
    - A later reading of the clock returns a later time
- In distributed environments it can be a real challenge
  - Think of how **`make`** works



Synchronization with a Time Server

- A time server has very accurate time
  - *E.g.*, atomic clock or GPS receiver
- How can a client synchronize with a time server?
  - Problem: messages do not travel instantly
- Cristian's algorithm
  - Estimate the transmission delay to the server: $((T_4-T_1) - (T_3-T_2)) / 2$

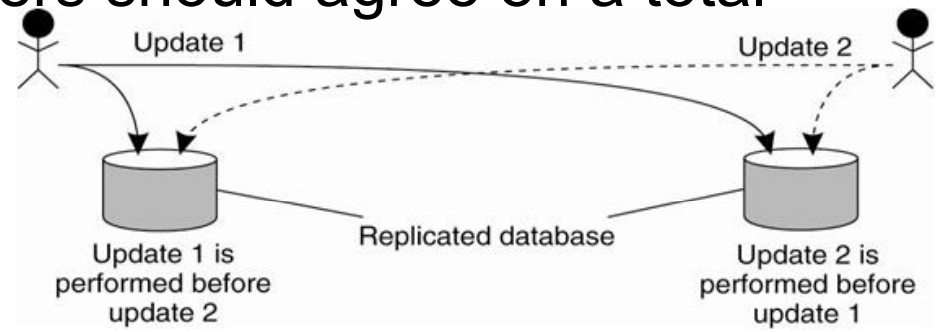


- Used in the Network Time Protocol (NTP)
- Cristian's algorithm is run multiple times
  - Outlier values are ignored to rule out packets delayed due to congestion or longer paths

GPS: Global Positioning System

Logical Clocks

- Absolute time synchronization is not always needed
- Need to ensure that the order in which events happen is preserved across all computers
  - More specifically: All computers should agree on a total ordering of events



- Example
  - A person's account holds 1,000 CHF and he adds 100 CHF
  - At the same time, an accountant invokes a command that gives 1% interest to each account
  - Does that person's account end up with 1,110 CHF or 1,111 CHF instead?

Lamport's Timestamps

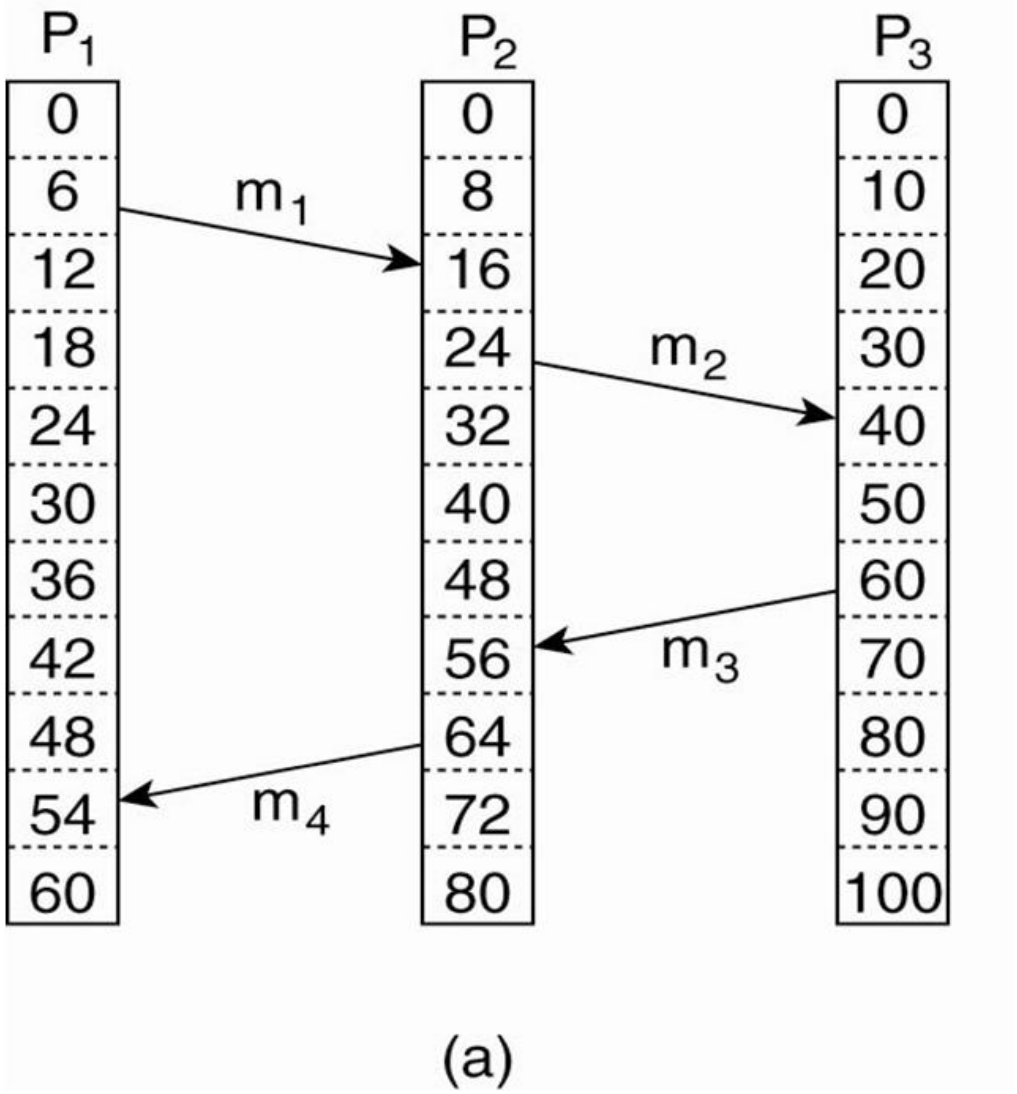
- In the classic paper as of 1978 Leslie Lamport defined the fundamental rules to reach consistent timestamps on events:

1. If *a* and *b* are events on the same process, then, if *a* occurs before *b*, CLOCK(*a*) < CLOCK(*b*)
2. If *a* and *b* correspond to the events of a message being sent from the source process and received by the destination process, respectively, then CLOCK(*a*) < CLOCK(*b*), because a message cannot be received before it is sent

Lamport's Timestamps Example

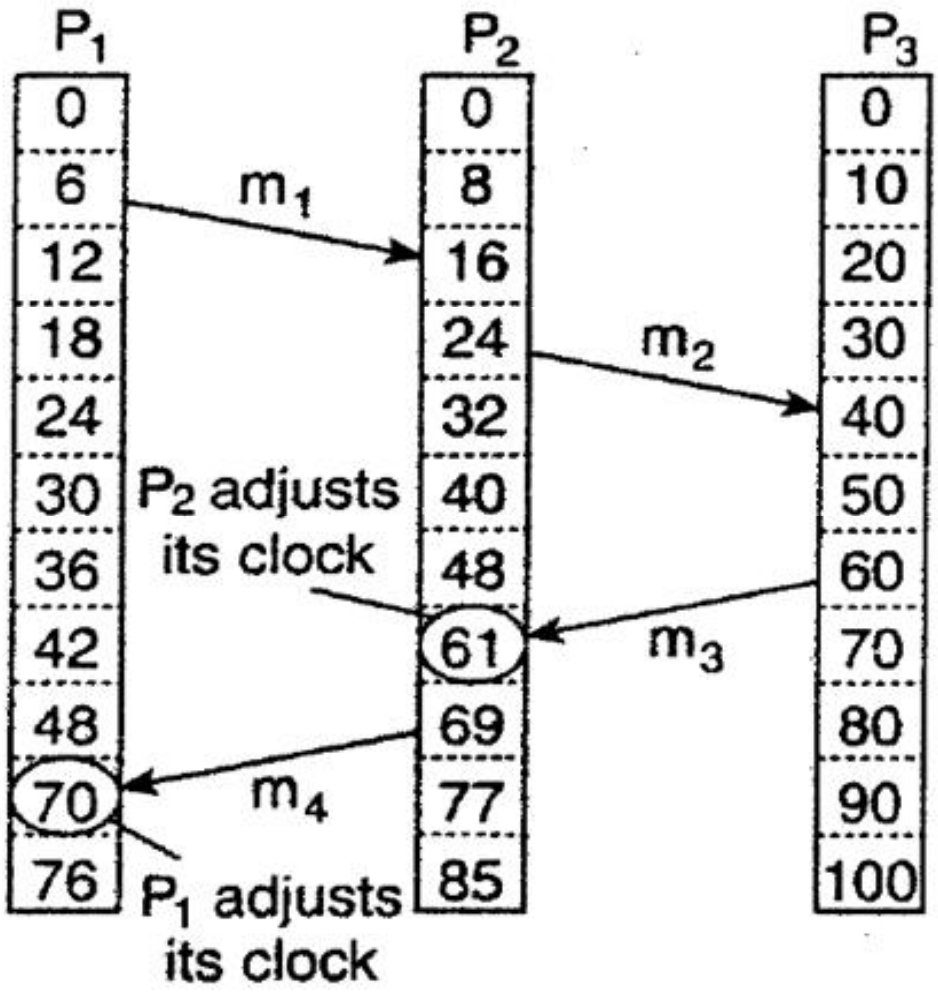
Each process bears its own logical clock, a monotonically increasing software counter.

Time stamps are applied to events and messages are sent to synchronize.



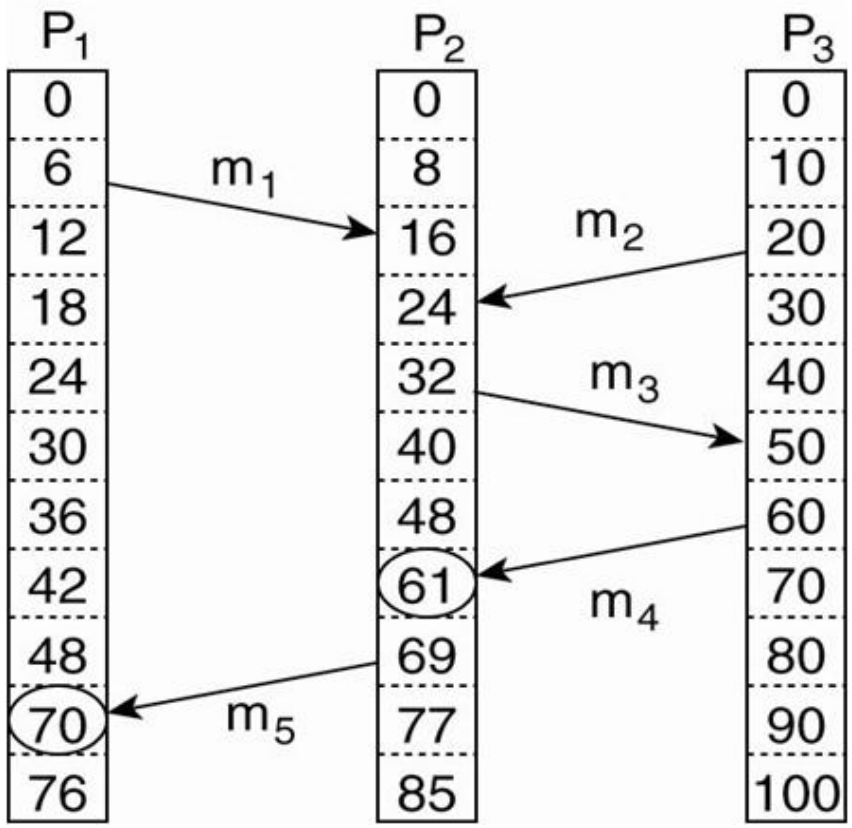
Vector (Logical) Clocks

- Lamport's Timestamps give total ordering of events
  - Notion of causality (dependencies between events) is lost
  - Total ordering is often too strict



Vector Clock Example

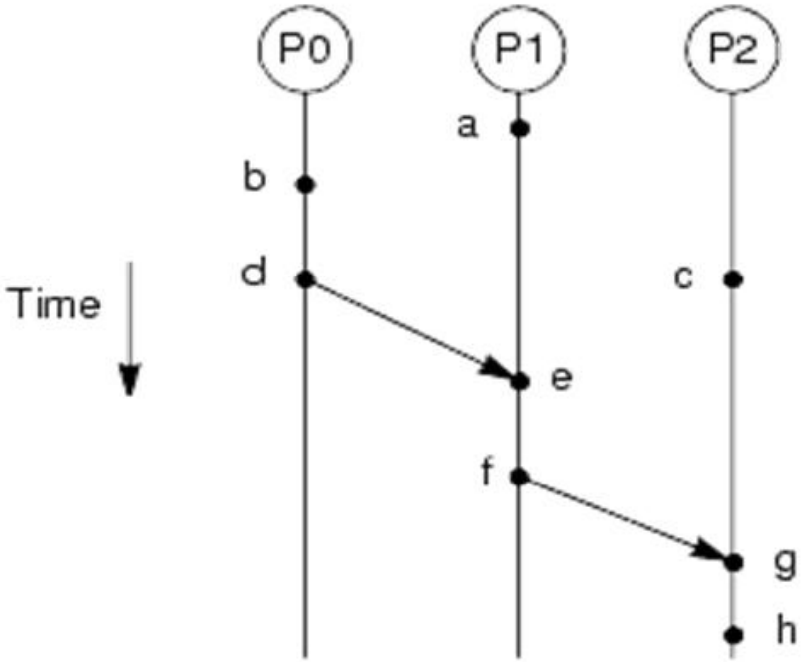
- The reception of m_3 (50) could depend on the reception of m_2 (24) and m_1 (16)
  - That's correct
- The sending of m_2 (20) seems to depend on the reception of m_1 (16)
  - But is it?
    No!



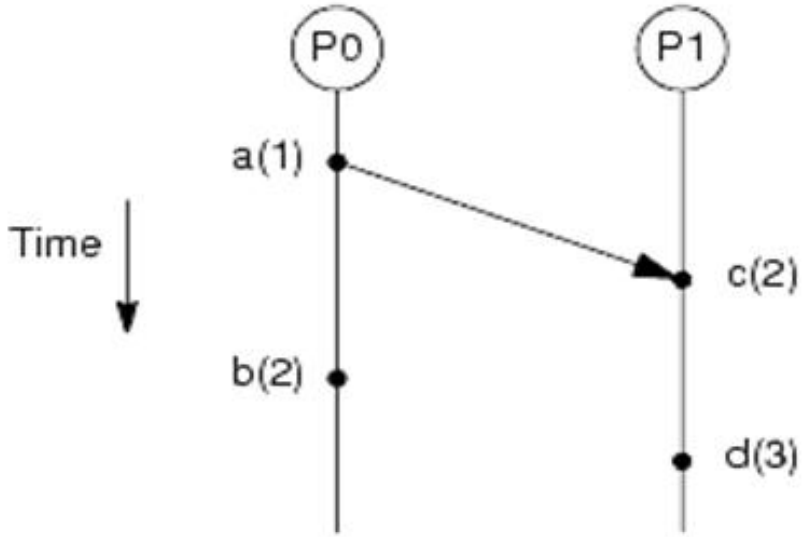
Causality

- Generally, two events can …
  - … be linked by a dependency

    a → b, which means a happens before b
    - *E.g.*, b → d, d → e, b → e, b → h
  - … be independent (concurrent)
    - *E.g.*, a and c, or a and b



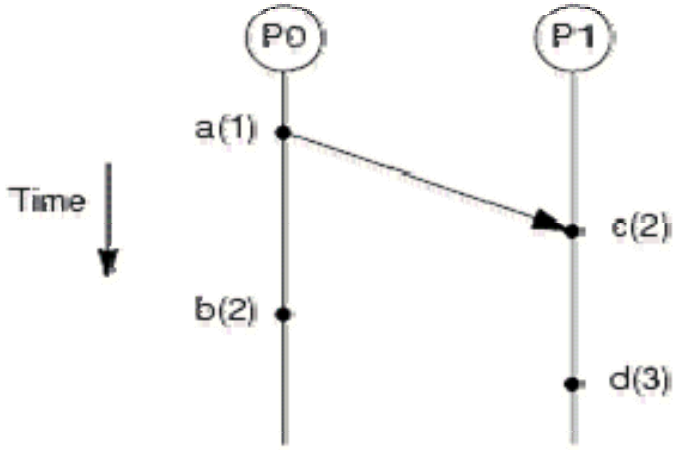
Inefficiency of Logical Clocks



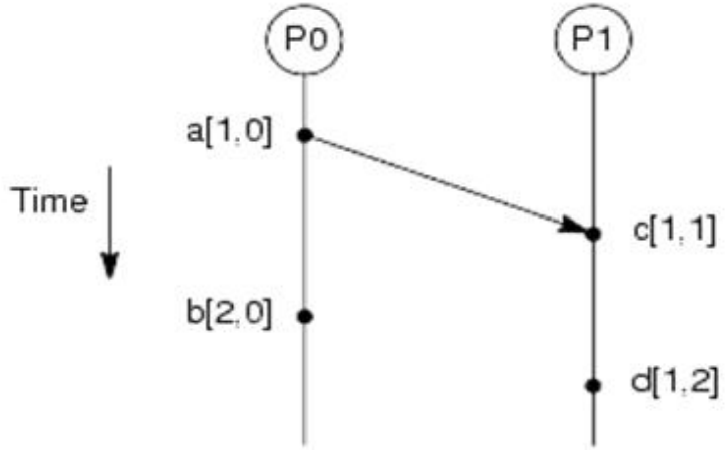
- To observe causality with Logical Clocks (a.k.a. Lamport Timestamps), the approach may fail
  - *d* consistently has a later timestamp than *b*, so one would (wrongly) assume that *b* → *d*

Logical and Vector Clocks

- With Vector Clocks
  - a → c, because [1,0] < [1,1]
  - a → d, because [1,0] < [1,2]
  - Same for a → b, c → d
  - But b and d are independent (concurrent), because there is no clear order between [2,0] and [1,2]



Logical Clocks



Vector Clocks

Vector Clocks (1)

- Assuming N nodes
  - Each node maintains a vector of *N* logical clocks
  - One logical clock as its own
  - The rest *N-1* logical clocks are estimations for other nodes

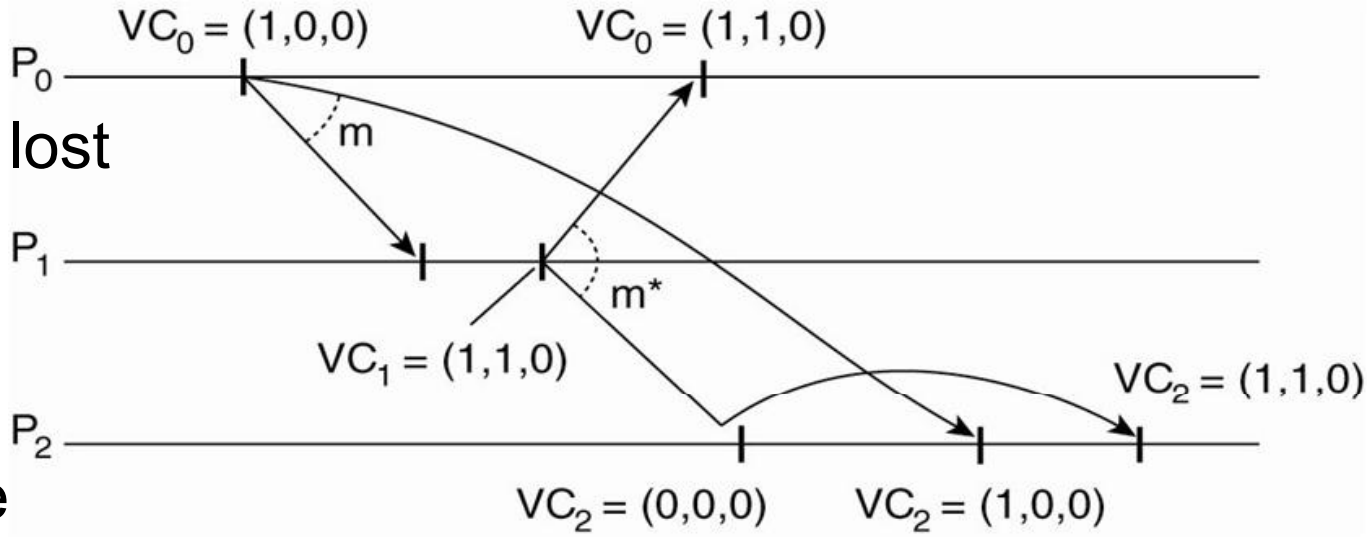
- Alternatively, logical clocks are managed as follows
  - They are all initialized with zero
  - When an event happens in a node, it increases its own logical clock in the vector by one
  - When a node sends a message, it includes its whole vector
  - When a node receives a message, it updates each element in its vector by taking the maximum of the value in its own vector clock and the value in the vector in the received message (for every element)

Vector Clocks (2)

- An event *a* is considered to happen before event *b*, only if all elements of the VC of *a* are less than or equal than the respective elements of the VC of *b*
  - In fact, at least one element has to be lower

Causal Communication

- Vector Clocks can be used to enforce causal communication
  - Do not deliver a packet until all causally earlier packets have been delivered
- Assumptions
  - No packets are lost
  - Clocks are increased only when sending a new message
  - A packet is delivered when only the sender's logical clock is increased



Causal Communication's "Location" (1)

- Included in middleware layer
  - Advantage: Generic approach
  - Drawback
    - Potential (but not definite) causality is captured
      - Even messages that are not related, but happen to occur in a given order, are assumed dependent: this makes it "heavier" than necessary
    - Some causality may not be captured
      - Alice posts a message, and then calls Bob and informs him about that message. Bob may take some other action that depends on the information he got from Alice, even before receiving the official message of Alice. This causality is not captured by the middleware.
      - External communication can mess up the assumptions of the middleware

Causal Communication's "Location" (2)

- Partial restriction: Application-specific
  - Advantages
    - Can be tuned to be more lightweight
    - Can be tuned to be more accurate
  - Drawback
    - Puts the burden of causality checking on the application developer

Mutual Exclusion Problem (1)

- Application level protocol
  - Enter Critical Section
  - Use resource exclusively
  - Exit Critical Section

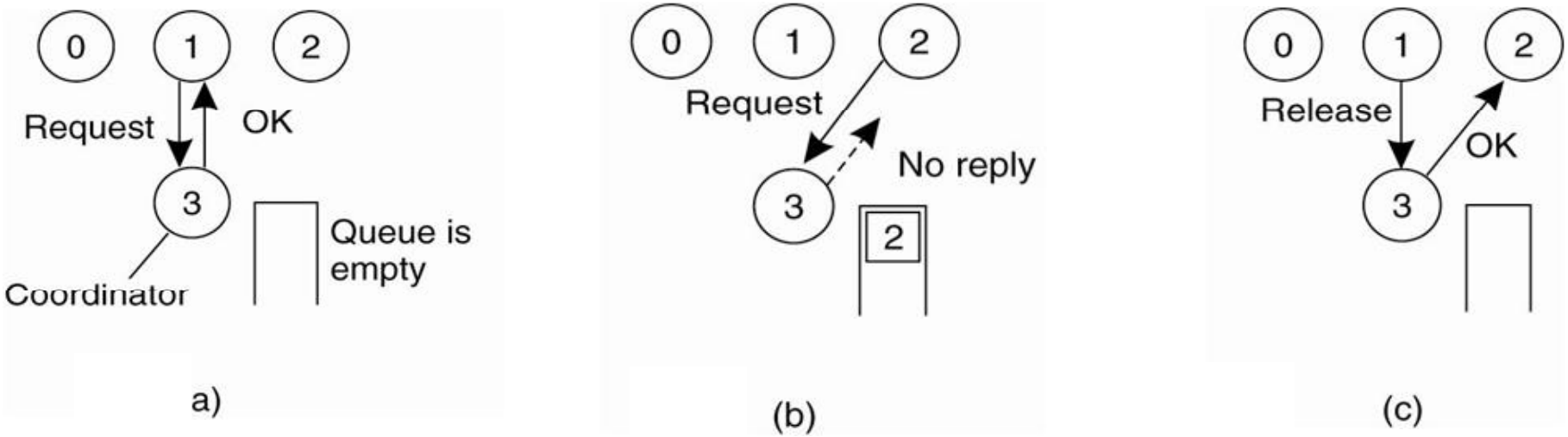
- Requirements
  - Safety: At most one process may execute in Critical Section at once
  - Liveness: Requests to enter and exit the critical section should eventually succeed (no deadlocks or livelocks should occur, and fairness should be enforced)
  - Ordering: Requests are handled in order of appearance

Mutual Exclusion Problem (2)

- Evaluation criteria
  - Bandwidth (number of messages)
  - Client waiting time to enter Critical Section
  - Vulnerabilities
- Alternative approaches
  - Centralized Approach
  - Distributed Approach
  - Token-Ring Approach

Centralized Approach

- Simplest algorithm to achieve Mutual Exclusion
  - Simulate what happens in a single processor



  - Easy to implement, few messages (3 per CS: Request, OK, Release), fair (First-In-First-Out), no starvation
  - Single point of failure, processes cannot distinguish between dead coordinator or busy resource

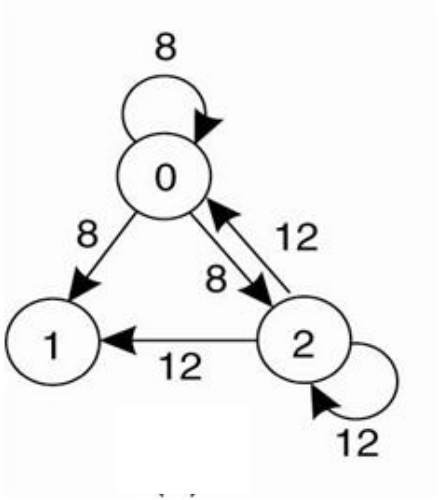
CS: Critical Section

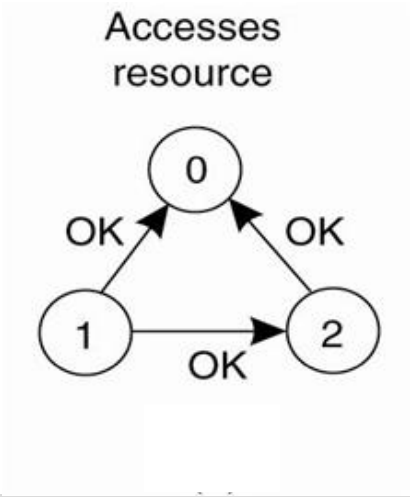
Distributed Approach

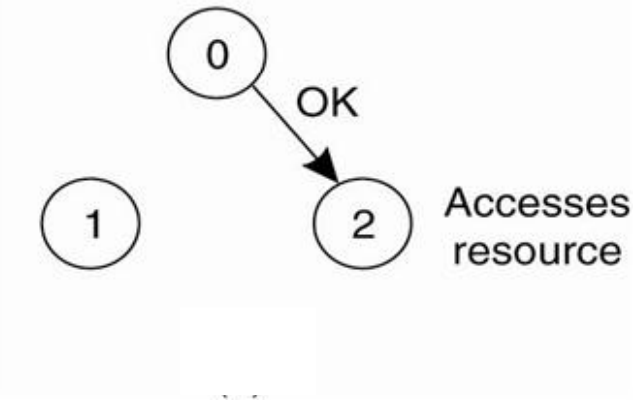
- Ricart and Agrawala's algorithm
  - Nodes use logical clocks: all events are in total order
  - When a node wants to enter a CS (Critical Section) it sends a message with its (logical) time and CS name to all other nodes
  - When a node receives such a request
    - If it is not interested in this CS, it replies OK immediately
    - If it is interested in this CS:
      - If its message's timestamp was older, then replies OK,
      - Else, it puts the sender in a queue and doesn't reply anything (yet)
    - If it is already in the CS, it puts the sender in a queue and doesn't reply anything (yet)
  - A node enters the CS when it received OK
  - A node that exits the CS, sends immediately OK to all nodes that it may have placed in the queue

Example

- Nodes 0 and 2 express interest in the CS almost simultaneously
- Node 0's message has an earlier timestamp, so it wins
- Node 1 (not interested) and node 2 (interested, but higher timestamp) send OK to node 0, so node 0 enters the CS
- When node 0 exits the CS, it sends OK to node 2, who enters the CS then







Ricart and Agrawala's Algorithm

On initialization
    state := RELEASED;
To enter the section
    state := WANTED;
    Multicast *request* to all processes;
    T := request's timestamp;
    Wait until (number of replies received = (*N* – 1));
    state := HELD;

request processing deferred here

On receipt of a request $<T_i, p_i>$ *at* p_j $(i \neq j)$
    if (*state* = HELD or (*state* = WANTED *and* $(T, p_j) < (T_i, p_i)$))
    then
        queue *request* from p_i without replying;
    else
        reply immediately to p_i;
    end if

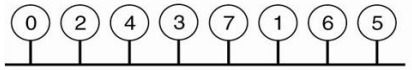
To exit the critical section
    state := RELEASED;
    reply to any queued requests;

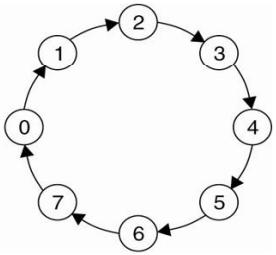
Distributed Approach Evaluation

- Problems
  - More messages: 2*(n-1)
  - No single point of failure ... but *n* points of failure
    - A failure on any one of n processes brings the system down
- Some improvements have been proposed
  - Maekawa's algorithm
    - Don't wait for approval from all, but from the majority only
- Moral conclusion
  - Distributed Algorithms are not always more robust to failures!

Token-Ring Approach

- Nodes are organized in a ring
  - A token goes around
    - Each one passes it to its successor





- If a node wants to enter a CS, it can do so when it gets the token
  - It is guaranteed it is the only one holding the token
  - When it exits the CS, it passes the token to the next node
- Very simple, fair, no starvation
- Messages per entry/exit: 1 to infinite
- Problem upon a token lost
  - Long delay might mean that the token is lost or someone is using it

Comparison

Algorithm	Messages per entry/exit	Waiting time to enter CS (best case)	Problems
Centralized	3	2	Crash of coordinator
Distributed	2*(n-1)	2*(n-1)	Crash of any node
Token ring	1 to infinite	0 to n-1	Lost token, crash of any node

CS: Critical Sections

Leader Election Problem

- Choice of one node among a selection of participants determines the leader
  - Each process gets a number (no two have the same)
  - For each process p_i:
    - There is a variable $elected_i$
  - Initialize:
    - Set all $elected_i$ = NONE
- Requirements
  - Safety: Participant p_i has $elected_i$ = NONE or p, where p is the number of the elected process
  - Liveness: All participating processes p_i eventually have $elected_i = p$ or crash

Bully Algorithm

- Assumptions
  - Synchronous messages
  - Timeouts

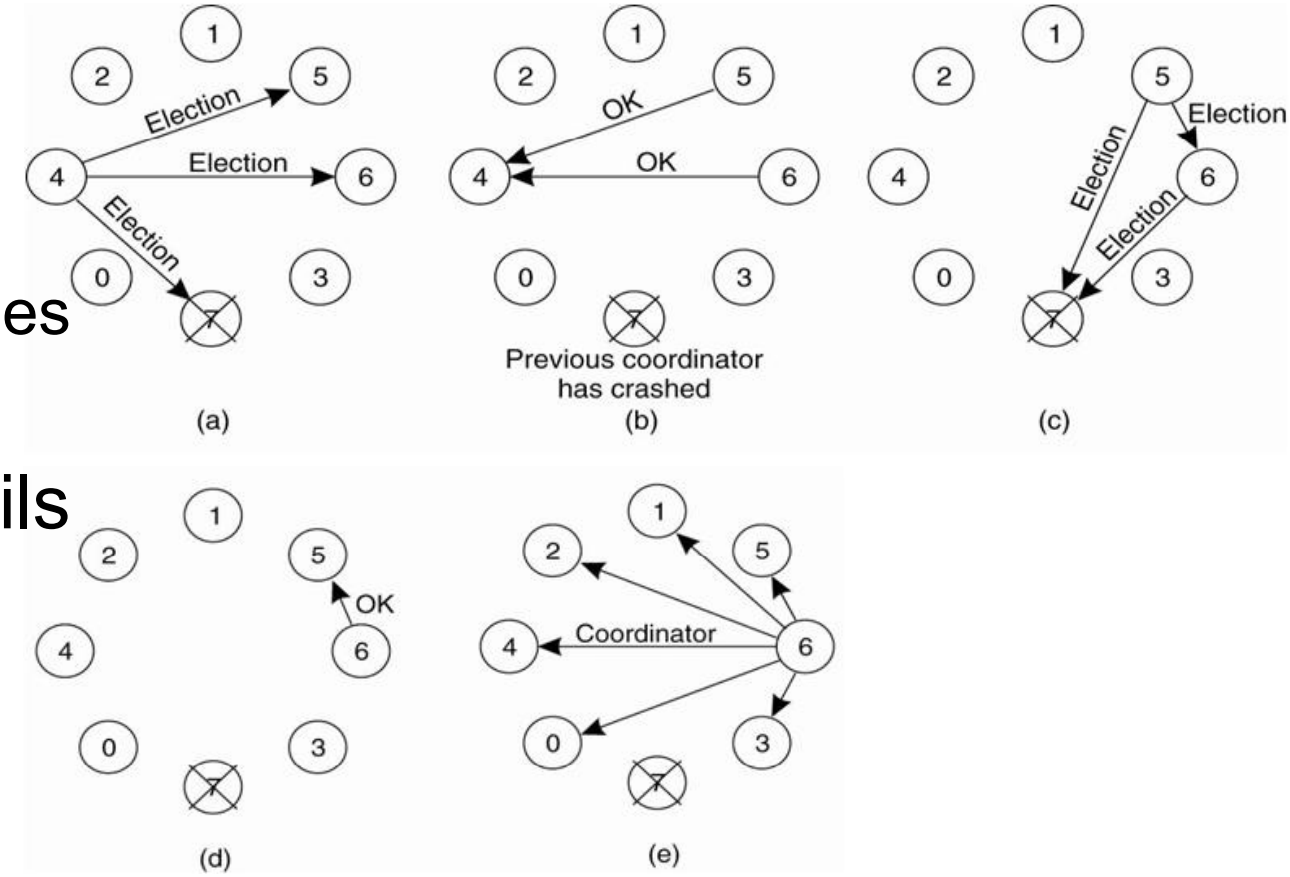
- Message types
  - Election (announcement)
  - Ok (response)
  - Coordinator (result)

Bully Algorithm Election Procedure

1. When a node notices that the coordinator is not responding, it starts the election process
2. Sends election message to all processes with a higher number;
   if no response, then it is elected
3. If one gets an election message and has higher ID, he replies ok and starts election
4. Process that knows it has the highest ID elects itself by sending a coordinator message to all others

Bully Algorithm Example

- Remarks
  - 7 was the coordinator, but it failed
- 4 notices it first and starts election
  - Notifies higher nodes
- Eventually 6 prevails and becomes the new coordinator



Ring Algorithm

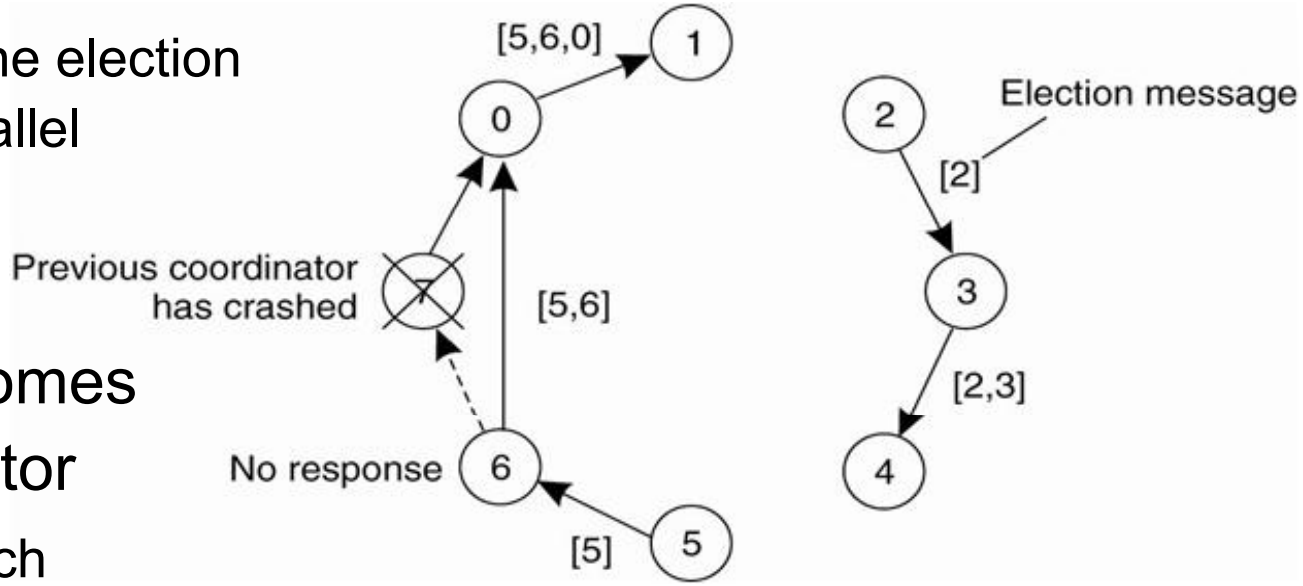
- Assumptions
  - Synchronous messages
  - Timeouts
  - Nodes are organized in a ring
- Message type
  - Election: <list of IDs>

Ring Algorithm Election Procedure

1. When a node notices that the coordinator is not responding, it starts the election process
2. Sends election message to its successor, with a list containing only its own ID
3. When one gets an election message that originated at a different node, it appends its ID to the list, and forwards the message to its successor
4. When one gets back its own election message, it picks the highest ID as the leader and announces it to everyone

Ring Algorithm Example

- Remarks
  - 7 was the coordinator, but it failed
  - Nodes 2 and 5 notice it has crashed
    - They both start the election procedure in parallel
  - Eventually 6 prevails and becomes the new coordinator
    - Both 2 and 5 reach the same conclusion



Multicast

- Multicast messages are useful in distributed systems
  - Multiple receivers can receive the same message

- Characteristics
  - Fault tolerance based on replicated servers
    - Requests sent to all servers, all service may not fail at the same time
  - Discovering services in spontaneous networking
    - Locating available services, *e.g.*, in order to register interfaces
  - Better performance through replicated data
    - Data are replicated, new values are multicast to all processes
  - Propagation of event notifications
    - Groups may be notified on happenings

Group Communications

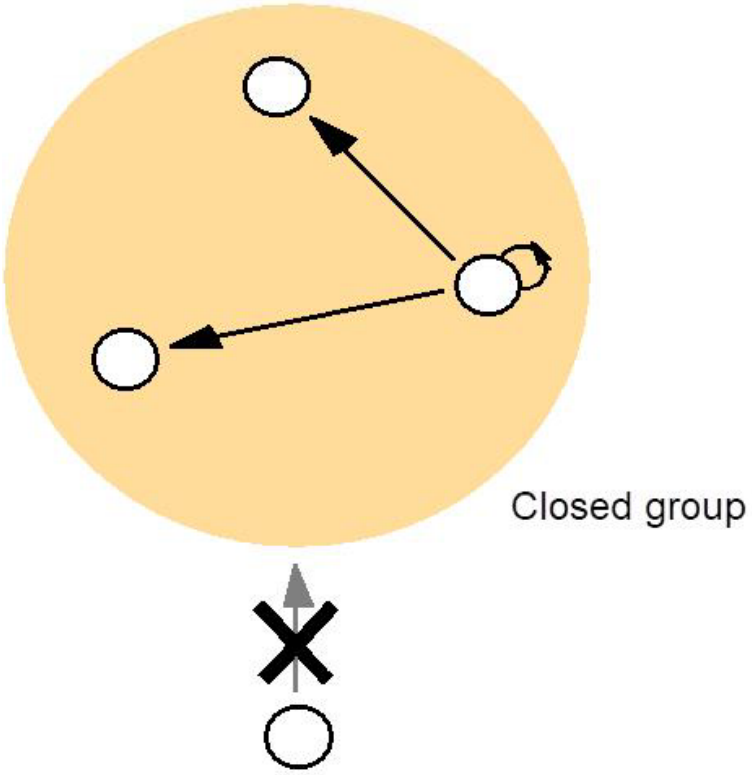
- Group communications are an important building block in distributed systems, particularly for reliable ones
- Characteristics and application areas
  - Reliable dissemination of information to many clients
    - Financial industry
  - Events distributed to multiple users to preserve a common user view
    - Collaborative applications, multi-user games
  - Fault tolerance strategies
    - Consistent updates of replicated data, *e.g.*, backups for machines
  - System monitoring and management
    - Data and configuration collection and distribution

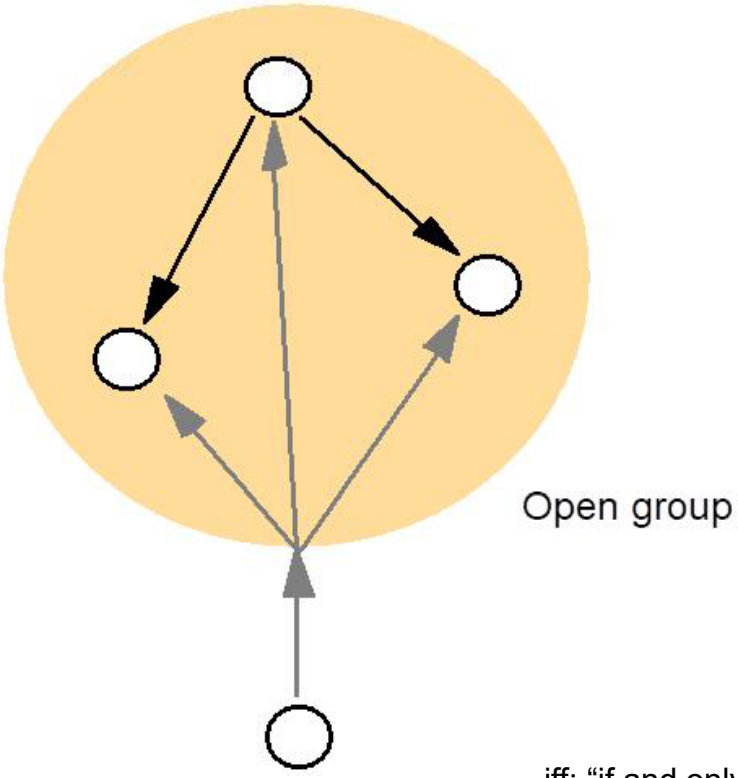
Multicast Problem

- Process sends a single send operation
  - Efficiency
  - Delivery guarantees
- System model
  - *multicast(m, g)* → sends message m to all members of group *g*
  - *deliver(m)* → delivers the message to the receiving process
- Properties
  - Integrity: Each message is delivered at most once
  - Validity: If *multicast(m,g)* and *p* in *g* → eventually *p.deliver(m)*
  - Agreement: If a message of *multicast(m,g)* is delivered to *p*, it should be delivered also to all other processes in *g*

Open and Closed Groups

- A group is closed iff only members can send messages
- A group is open iff any outside process can send to it





iff: “if and only if”

Basic Multicast

- Basic multicast
  - *B-multicast(m, g)*
    - For each *p* in *g*, do *send(p,m)*
  - On *receive(m)* at *p*: *B-deliver(m)* at *p*

- Problems
  - Implosion of acknowledgements
  - Not reliable

Reliable Multicast Algorithm

```
On initialization
    Received := {};

For process p to R-multicast message m to group g
    B-multicast(g, m);          // p ∈ g is included as a destination

On B-deliver(m) at process q with g = group(m)
    if (m ∉ Received)
    then
                Received := Received ∪ {m};
                if (q ≠ p) then B-multicast(g, m); end if
                R-deliver m;
    end if
```

- Problems
  - Inefficient: $O(|g|^2)$ messages
  - Implosion of acknowledgements