Distributed Systems

Synchronization

Dynamic and Distributed Information Systems

Today's Agenda

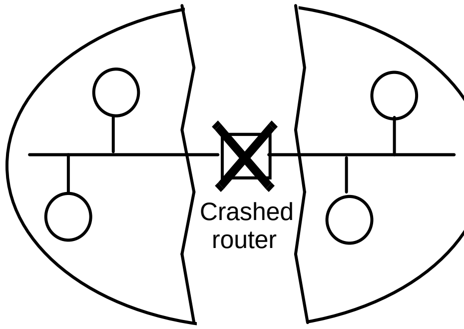
- Introduction
- Time Synchronization
  - Clock Synchronization
  - Logical Clocks
- Mutual Exclusion
- Leader Election
- The Multicast Problem

Need for Synchronization

- Being able to communicate is not enough
- Nodes also need to coordinate & synchronize for various tasks
  - Synchronize with respect to time
  - Not access a resource (e.g., a printer, or some memory location) simultaneously
  - Agree on an ordering of (distributed) events
  - Appoint a coordinator

Assumptions & Algorithms

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



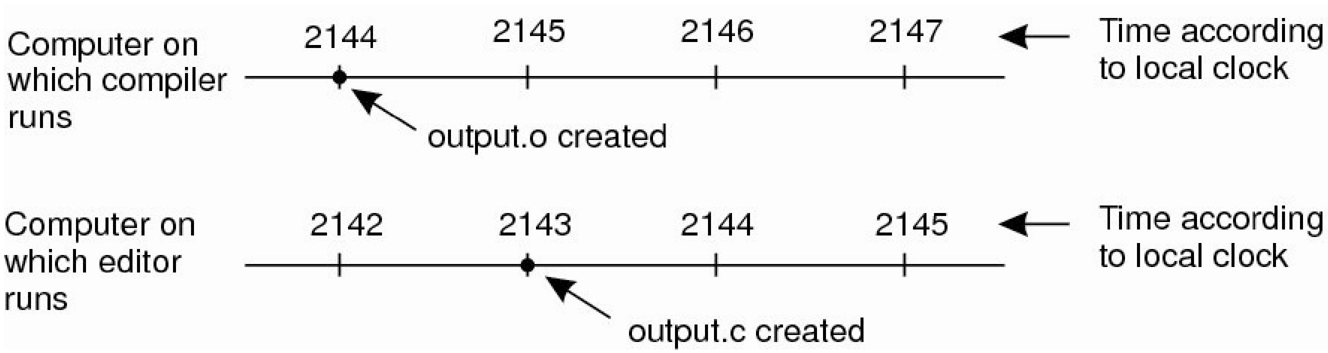
- Algorithms
  - Distributed Mutual exclusion
  - Elections
  - Multicast Communication

Figure 12.1

Time Synchronization

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

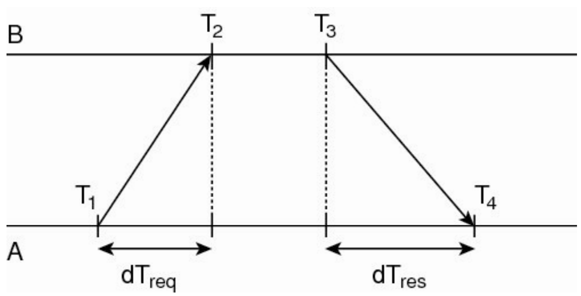


Synchronization with a Time Server

- A *time server* has very accurate time (e.g., atomic clock, GPS receiver, etc.)

- But how can a client synchronize with a time server?
  - Problem: messages do not travel instantly

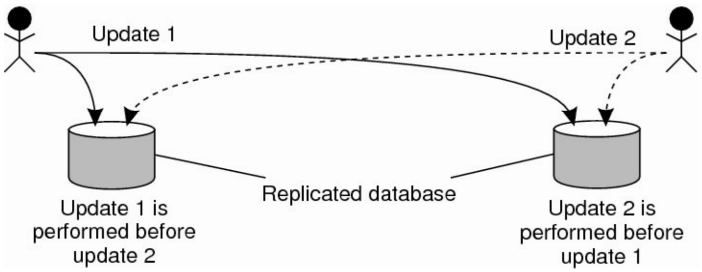
- Cristian's algorithm:
  - Estimate the transmission delay to the server: $((T_4-T_1) - (T_3-T_2)) / 2$



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

Logical Clocks

- In many cases, absolute time synchronization is not needed
- We only 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 has €1,000 and he adds €100
  - 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 or €1,111 ?



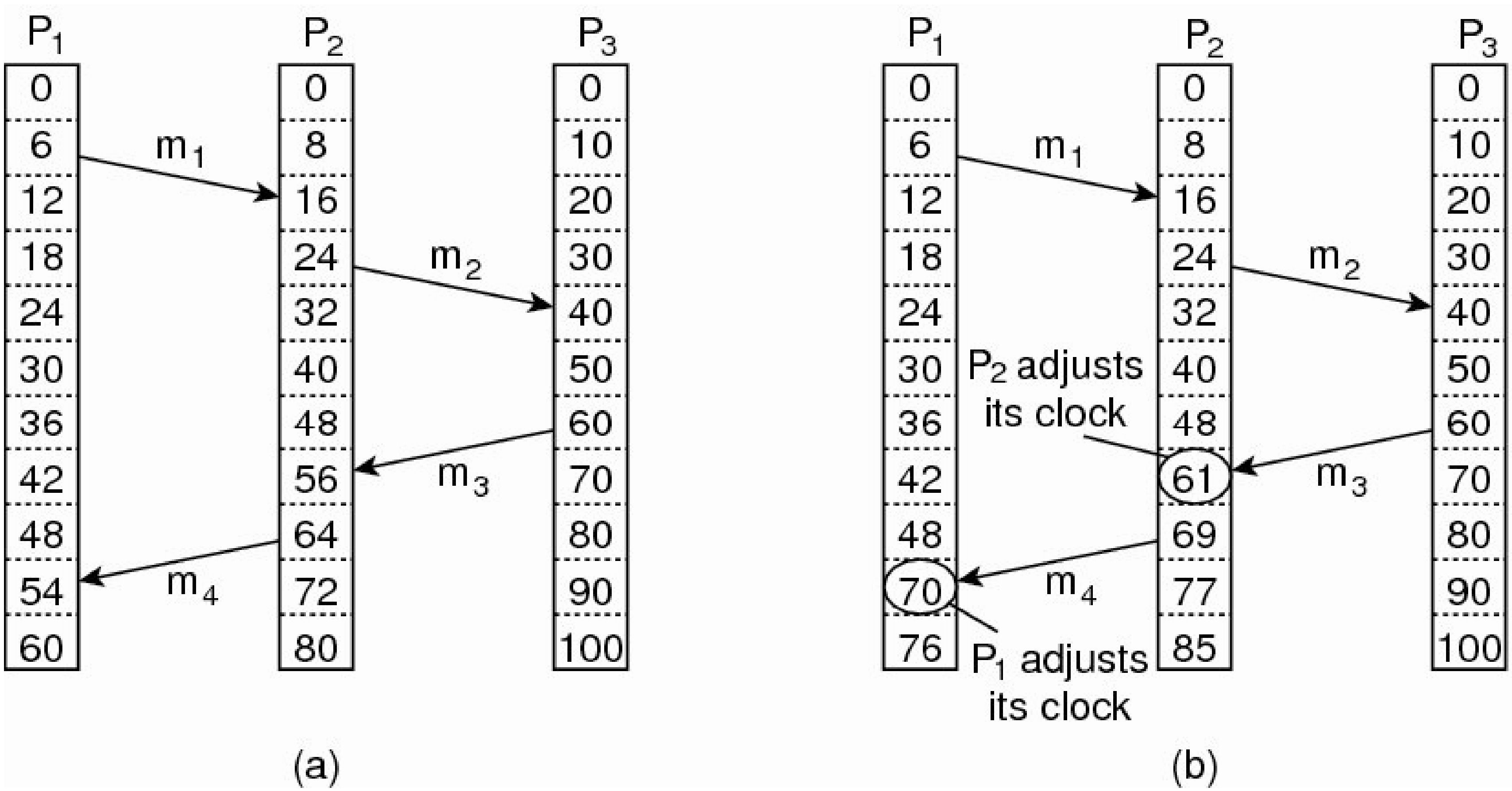
Lamport Timestamps

- In a classic paper in 1978, Leslie Lamport defined the fundamental rules to have 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 Timestamps example



Mutual Exclusion

Dynamic and Distributed Information Systems

The Mutual Exclusion Problem

- Application level protocol
  1. Enter *Critical Section*
  2. Use resource exclusively
  3. 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

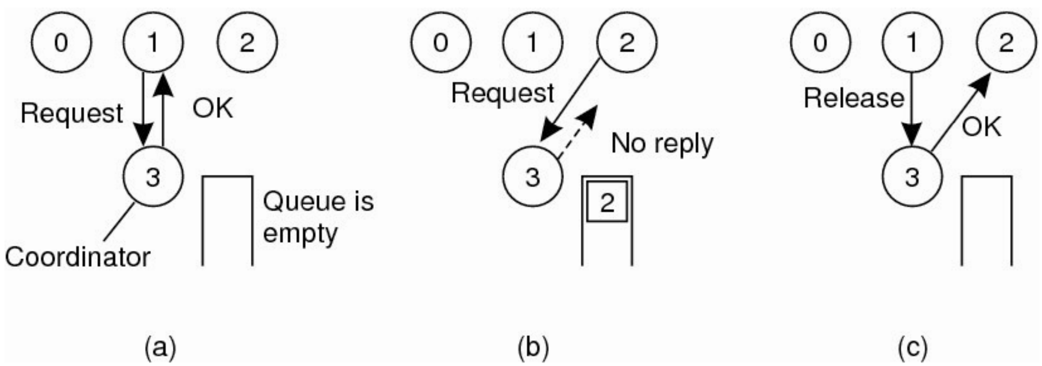
- Evaluation criteria
  - Bandwidth (number of messages)
  - Client waiting time to enter Critical Section
  - Vulnerabilities

The Mutual Exclusion Problem

- We will see three approaches:
  - Centralized Approach
  - Distributed Approach
  - Token-Ring Approach

Centralized Approach

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



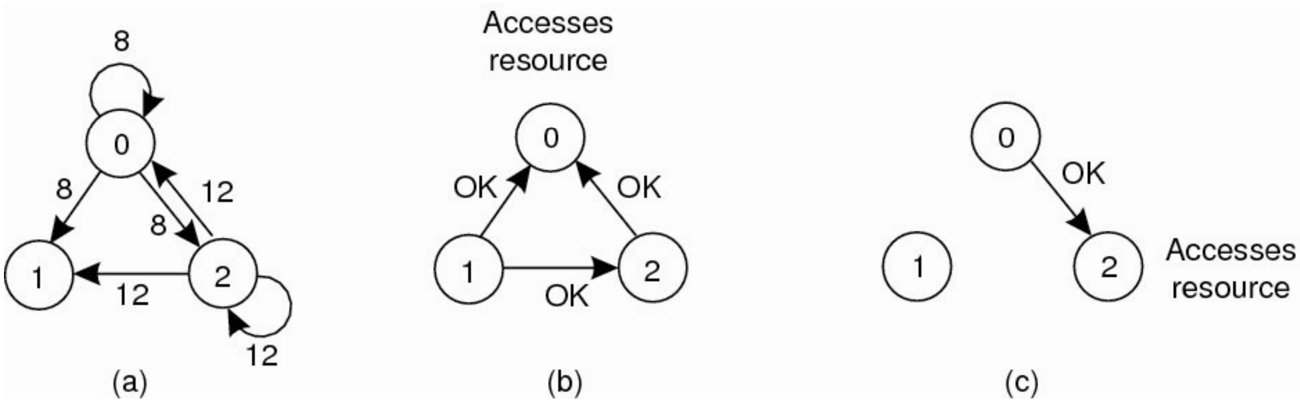
- +: Easy to implement, few messages (3 per CS: *Request*, *OK*, *Release*), fair (First-In-First-Out), no starvation

- - : Single point of failure, processes cannot distinguish between dead coordinator or busy resource

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 the 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 but *all* other nodes
  - 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 1, so node 1 enters the CS
- When node 1 exits the CS, it sends OK to node 2, who enters the CS then

Ricart & Agrawala's algorithm

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

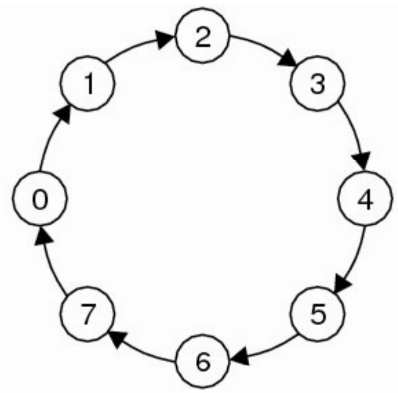
On receipt of a request $<T_i, p_i>$ *at* p_j $(i \neq j)$
    if (*state* = HELD or (*state* = WANTED *and* $(T, p_j) < (T_i, p_i)$))
    then
        queue *request* from p_i without replying;
    else
        reply immediately to p_i;
    end if

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

Distributed Approach

- 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*
- 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 if the token is lost
  - Long delay might mean that the token is lost, or that someone is using it

Comparison

Algorithm	Messages per entry/exit	Waiting time to enter CS	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

Leader Election

Dynamic and Distributed
Information Systems

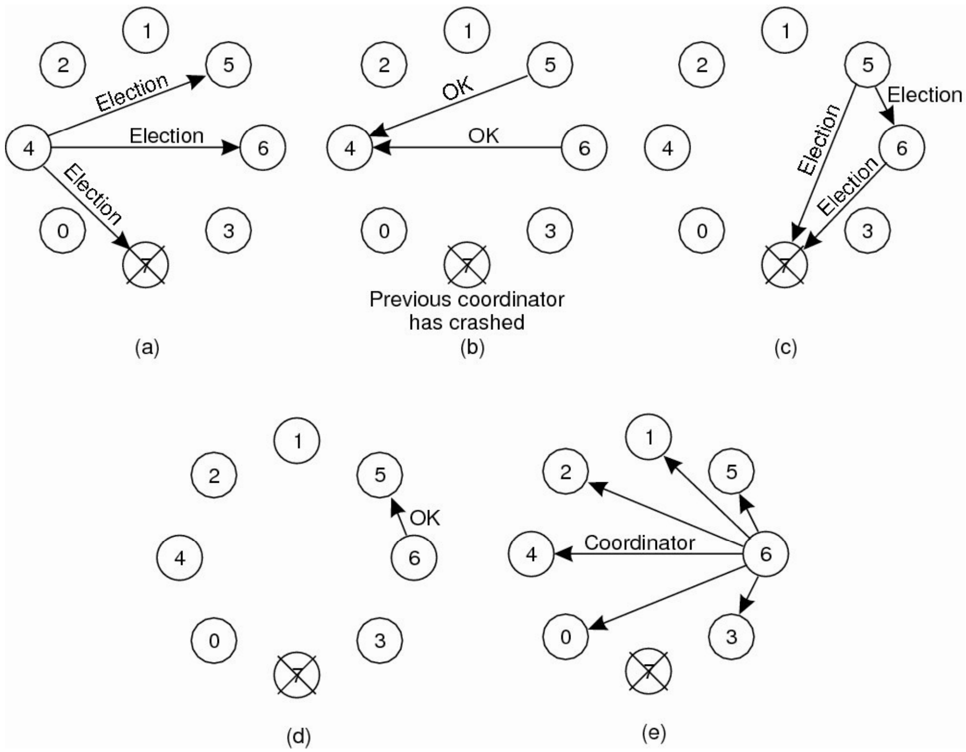
The Leader Election Problem

- Choice of one node among a selection of participants
  - 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

The bully algorithm

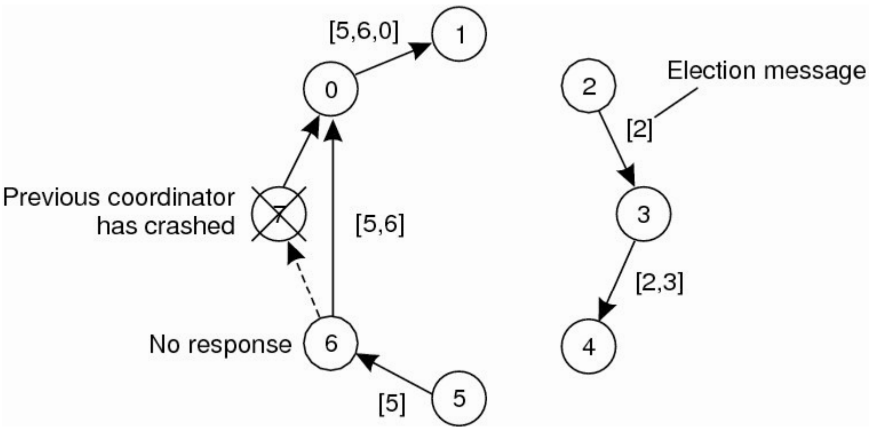
- Assumptions
  - Synchronous messages
  - Timeouts
- Message types:
  - *election* (announcement)
  - *ok* (response)
  - *coordinator* (result)
- Election procedure
  - When a node notices that the coordinator is not responding, it starts the election process
  - Sends *election* message to all processes with a higher number; if no response, then it is elected
  - If one gets an *election* message and has higher ID, he replies *ok* and starts election
  - Process that knows it has the highest ID elects itself by sending a *coordinator* message to all others



- In this example, 7 was the coordinator, but it fails
- 4 notices it first, and starts election (notifies higher nodes)
- Eventually 6 prevails and becomes the new coordinator

The ring algorithm

- Assumptions
  - Synchronous messages
  - Timeouts
  - Nodes are organized in a ring
- Message types:
  - election: <list of IDs>
- Election procedure
  - When a node notices that the coordinator is not responding, it starts the election process
  - Sends election message to its successor, with a list containing only its own ID
  - 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
  - When one gets back its own election message, it picks the highest ID as the leader and announces it to everyone



- In this example, 7 was the coordinator, but it fails
- Nodes 2 and 5 notice it has crashed, and they both start the election procedure in parallel
- Eventually 6 prevails and becomes the new coordinator (both 2 and 5 reach the same conclusion)

The Multicast Problem

The 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
- Groups are called closed iff only members can send messages
- 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

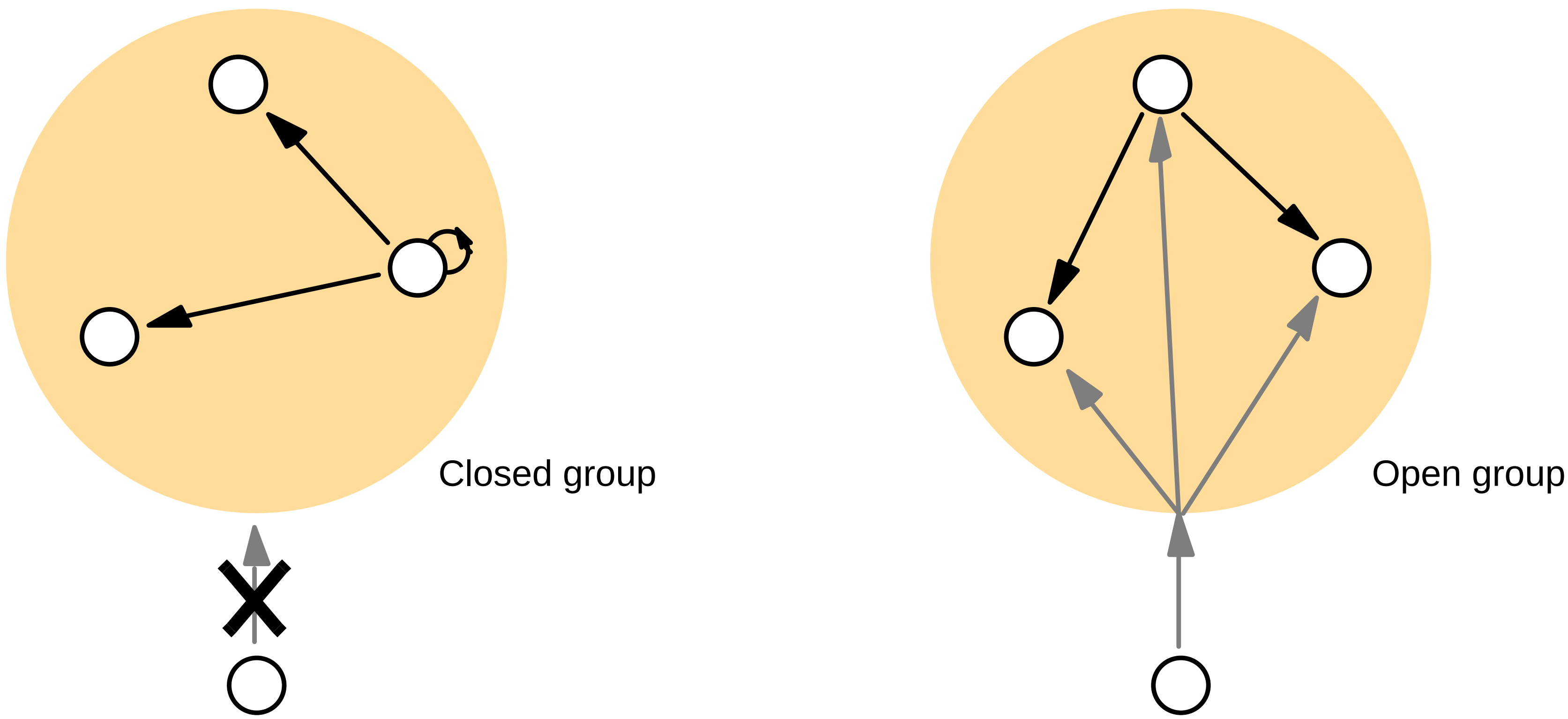


Figure 12.9

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
  $B\text{-}multicast(g, m);$ // $p \in g$ is included as a destination

On B-deliver(m) at process q with g = group(m)
  $if\ (m \notin Received)$
  then
    $Received := Received \cup \{m\};$
    $if\ (q \neq p)\ then\ B\text{-}multicast(g, m);\ end\ if$
    R-deliver m;
  end if

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