Distributed Systems (CS543)

Coordination & Agreement

Dongman Lee

KAIST

Class Overview

- Introduction
- Distributed Mutual Exclusion
  - Centralized
  - Distributed
- Election Algorithm
  - Ring-based algorithm
  - Bully algorithm
- Group Communication Model
  - Membership synchronization
  - Request ordering

Introduction

- In a distributed system,
  - resources are shared by multiple entities whose activities need to be synchronized
  - some of entities play the role of server
- Key technologies are
  - *mutual exclusion* is required to prevent interference and ensure consistency
  - *election* is required to choose which of entities will play the role of server

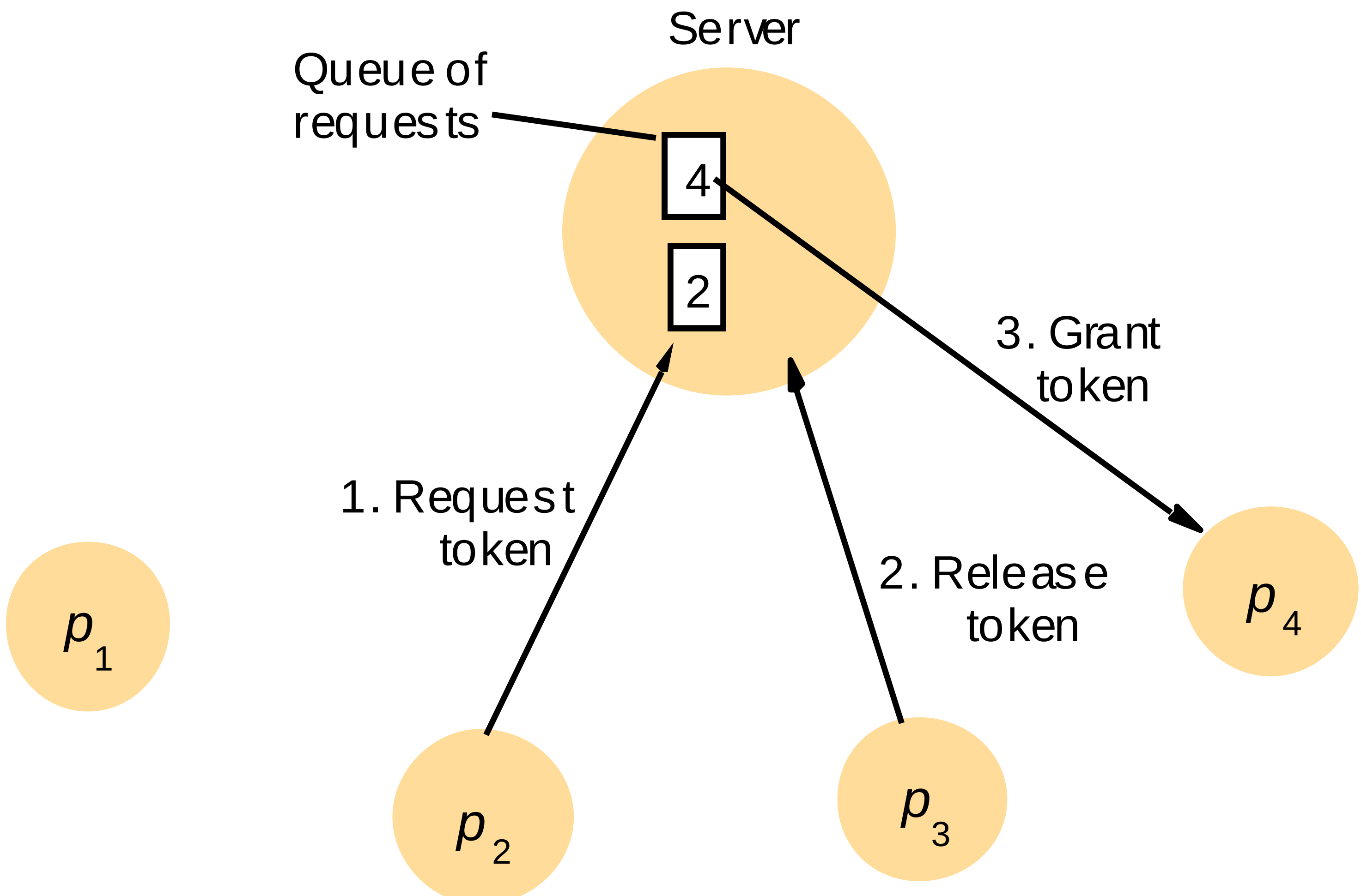
Distributed Mutual Exclusion

- Essential requirements for distributed mutual exclusion
  - Assumptions
    - the system is asynchronous
    - processes do not fail
    - message delivery is reliable
  - ME1: (safety)
    - at most one process may execute in the critical section (CS) at a time
  - ME2: (liveness)
    - a process requesting entry to the CS is eventually granted it
      - algorithms should guarantee freedom from deadlock and starvation
  - ME3: (ordering)
    - entry to the CS should be granted in happened-before order
      - fairness provision
  - approaches
    - centralized
    - distributed

ME: Centralized Solution

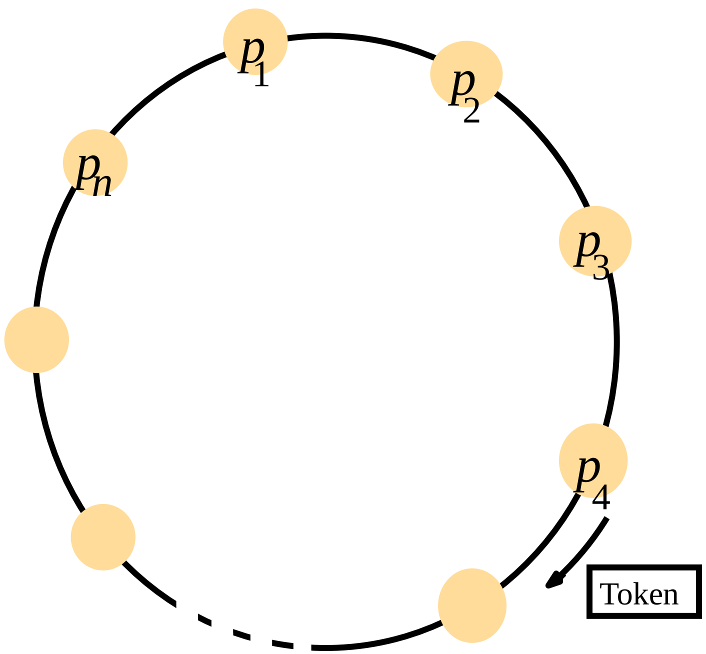
- A server process coordinates mutual exclusion
- Algorithm
  - Clients
    - before entering the CS,
      - a process sends a request message to the server and waits for a reply from it
    - when leaving the CS,
      - a process sends a release message to the server
  - Server
    - on receipt of request
      - if no process exists in the CS and the queue is empty, send a reply message; otherwise, queue the request
    - on receipt of release
      - remove the next request from the queue and send a reply
- Analysis
  - a single point of failure
  - synchronization delay: round-trip

ME: Centralized Solution (cont.)



ME: Ring-based Algorithm

- Algorithm
  - processes form a ring and a token message is circulated around it
  - possession of a token implies right to enter CS
  - after leaving CS, pass a token to its neighbor
- Analysis
  - 1 to (n-1) messages are taken to get a token and round trip time for sync
  - a token is not necessarily obtained in a happened-before order
  - if one process fails, need reconfiguration
    - process assumed to be failed may inject the old token



ME: Ricart and Agrawala Algorithm

- Overview
  - based on distributed agreement using multicast and logical clocks
  - Assumptions
    - processes p1, …, pn know one another’s address
    - all messages sent are eventually delivered
    - each process pi keeps a logical clock conforming to LC1 & LC2
    - token is being used to represent the state of a process
      - RELEASE
      - WANTED
      - HELD

ME: Ricart and Agrawala Algorithm(cont.)

- Algorithm

On initilization:

state := RELEASED

To obtain token:

state := WANTED

T = T + 1

multicast request to (n-1) processes

wait until reply from all (n-1) processes

state := HELD

On receipt of request $<T_i, p_i>$ at p_k:

if (state = HELD or (state = WANTED and $(T, p_k) < (T_i, p_i)$))

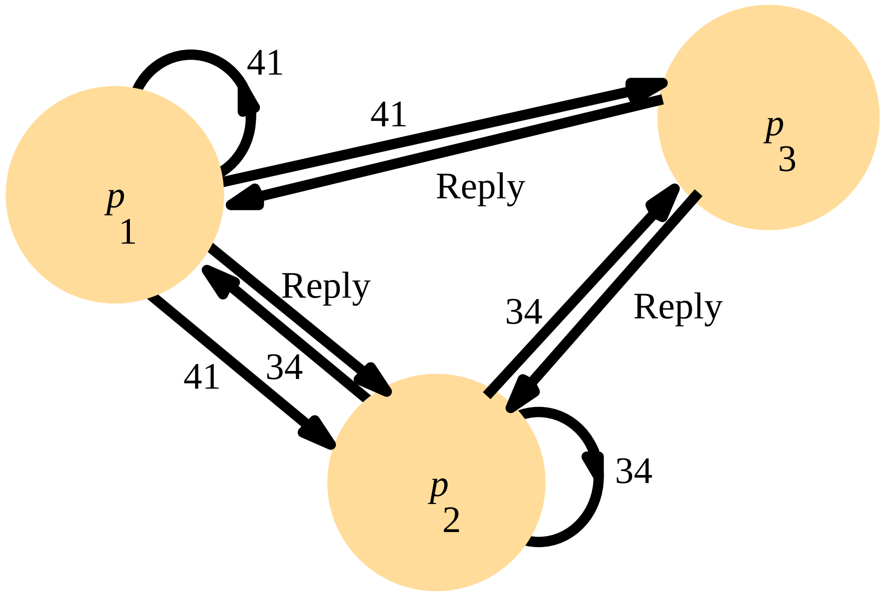
then queue request from p_i

else reply immediately to p_i

To release token:

state := HELD

reply to any queued requests



P1 & P2 but P3 are interested in entering CS

ME: Ricart and Agrawala Algorithm(cont.)

- Analysis
  - 2(n-1) messages are required to access CS
    - reduced to n when multicast is supported
  - synchronization delay is one message transmission time

ME: Maekawa's Voting Algorithm

- Overview
  - determine a candidate by getting permission from *subsets* (voting sets) of its peers which used by any two processes overlap
  - *voting sets* V_i are chosen for all i, j = 1, 2 …, N
    - $p_i \in V_i$
    - $V_i \cap V_j \neq$ NULL - there is at least one common member of any two voting sets
    - $|V_i| = K$ – to be fair, each process has a voting set of the same size
    - Each process p_j is contained in M of the voting sets V_i
  - optimal solution: $K \sim \sqrt{N}$ and $M = K$

ME: Voting Algorithm (cont.)

- Algorithm

On initialization

state := RELEASED;
voted := FALSE;

For p_i to enter the critical section

state := WANTED;
Multicast *request* to all processes in $V_i - \{p_i\}$;
Wait until (number of replies received = $(K - 1)$);
state := HELD;

On receipt of a request from p_i at p_j $(i \neq j)$

if (*state* = HELD *or voted* = TRUE)
then
    queue *request* from p_i without replying;
else
    send *reply* to p_i;
    voted := TRUE;
end if

For p_i to exit the critical section

state := RELEASED;
Multicast *release* to all processes in $V_i - \{p_i\}$;

On receipt of a release from p_i at p_j $(i \neq j)$

if (queue of requests is non-empty)
then
    remove head of queue – from p_k, say;
    send *reply* to p_k;
    voted := TRUE;
else
    voted := FALSE;
end if

- Analysis
  - bandwidth utilization: $3\sqrt{N}$
    - $2\sqrt{N}$ messages per entry to the critical section + $\sqrt{N}$ messages per exit
  - client delay is the same as Ricart and Agrawala's but synchronization delay is worse due to message round-trip time

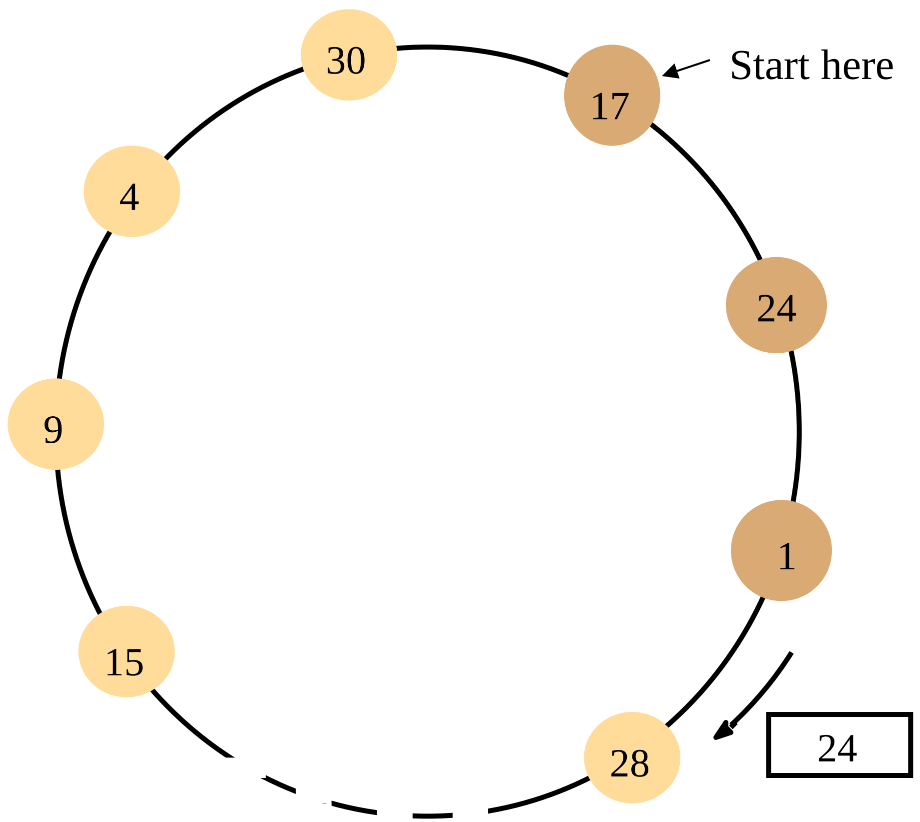
Elections

- Purpose
  - to choose a unique process to play a particular role
  - examples
    - select a new master in Berkeley clock synchronization algorithm
    - select a new member generating a token in ring-based distributed synchronization
- Requirements
  - Assumptions
    - each process is uniquely identified
    - the elected process be chosen as the one with largest id
  - E1 (safety):
    - a participant process p_i is notified of the elected one which will be chosen as the non-crashed process at the end of the run with the largest id
  - E2 (liveness)
    - all process p_i participate and eventually choose the elected one or crash
- Algorithms
  - ring-based & Bully

Election: Ring-based Algorithm

- Algorithm
  - initially, every process is marked as a *non-participant*
  - any process begins election by marking itself as a *participant* and sending an *election* message to its neighbor
  - when an *election* message is received, compare id
    - if my id is higher, claims myself as a participant and pass the message
  - election is done when id in an *election* message is the same as the claimed participant; then it sends an *elected* message
- Analysis
  - (3N-1) messages in worst case and 2N in best case
  - does not tolerate failures

Election: Ring-based Algorithm (cont.)

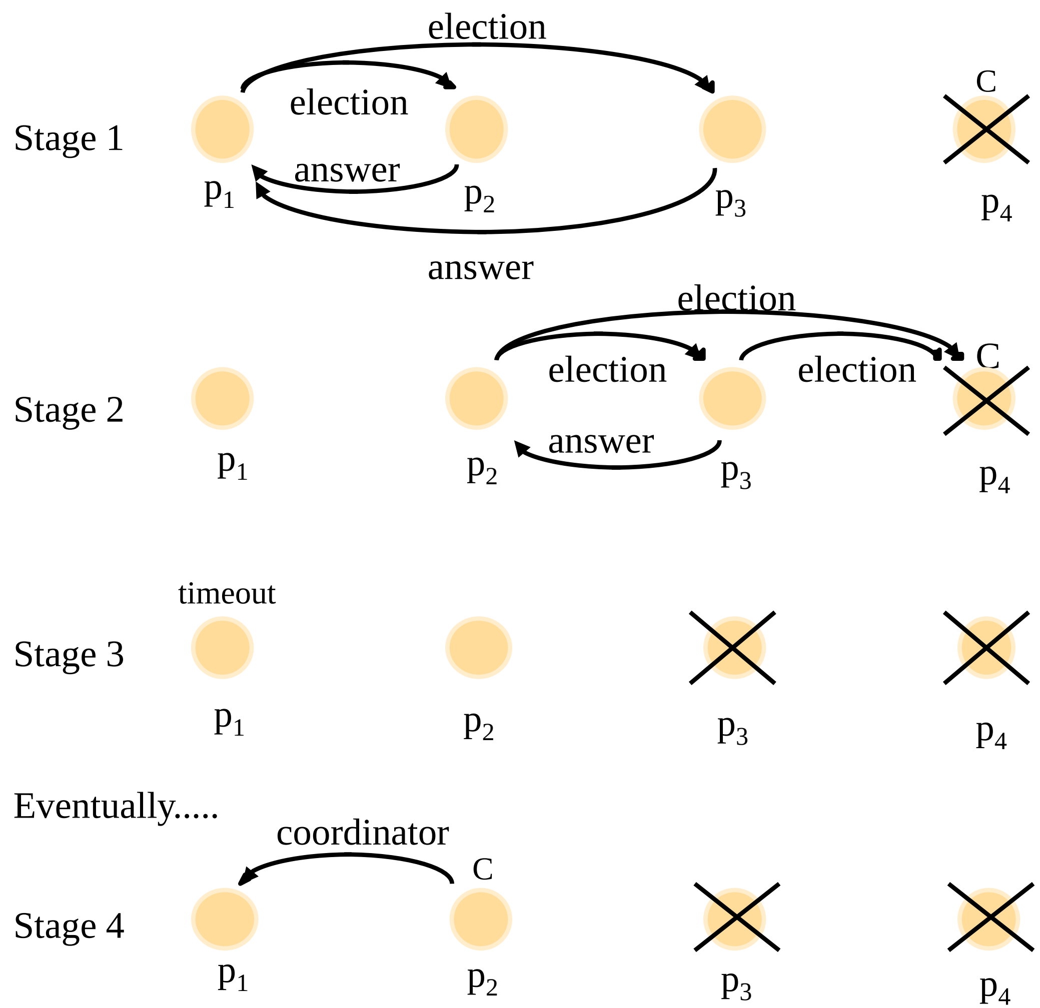


Election: Bully Algorithm

- Assumptions
  - each process has a unique id
  - processes know the id and address of every other process
  - communication is assumed reliable but a process can fail during election
  - election begins when detecting the coordinator has failed
- Algorithm
  - to begin election, a process sends an election message to all processes with higher id's and awaits answer messages
  - if there is no answer message within the timeout, the process becomes a coordinator and sends a coordinator message to the processes with lower id's
  - if the process receives an answer message, waits for a coordinator message
  - if the process receives an election message, it returns an answer and starts an election
  - if the process receives a coordinator message, it treats the sender as a coordinator
  - if there is no coordinator message within the timeout, it begins another election
  - if the failed process with a highest id is restarted, it overrides the current coordinator

Election: Bully Algorithm (cont.)

The election of coordinator p_2, after the failure of p_4 and then p_3



Election: Bully Algorithm

- Analysis
  - E1 (safety) may not be met when
    - the crashed process is replaced by a process with the same id or
    - the assumed timeout values turns out to be inaccurate
  - Performance
    - (N-2) message in the best case
    - $O(N^2)$ messages in the worst case

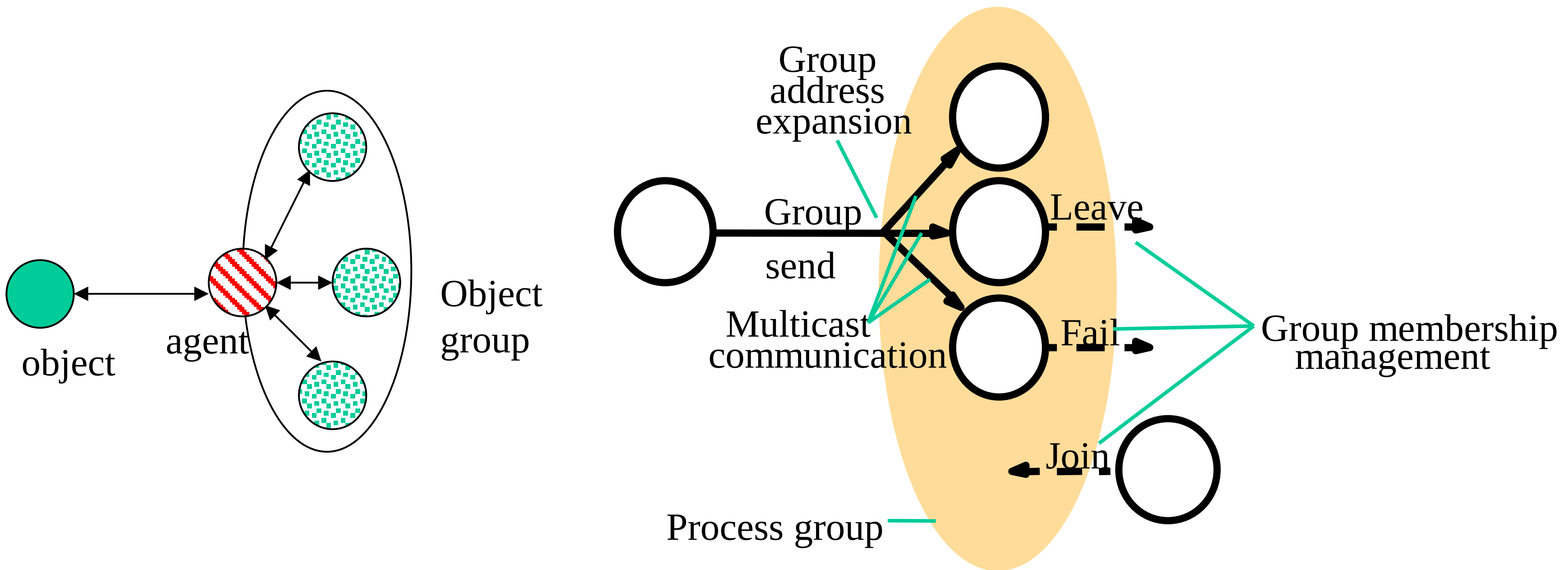
Group System Model

- Definition
  - a set of one or more objects, joined through a common interface, acting as a single unit for purposes of naming and function
- Group types
  - replicate
    - replicated members for highly availability and/or reliability
      - primary/stand-by
      - modular redundant
  - partition
    - members executing a common job in a divided manner
      - e.g. highly parallel array processing
  - aggregate
    - non-replicated members sharing the provision of the service defined by group’s interface
      - e.g. group conferencing

Group Communication Model

- Group communication =

  membership service + multicast with ordering support



Object group interaction model

Membership Service

- Membership service
  - interface for group membership changes
  - failure detection
  - membership change notification
  - group address expansion
- View delivery
  - view: a list of the currently active and connected members in a group
  - basic requirements for view delivery (view notification)
    - order
      - If a process p delivers view $v(g)$ and the $v'(g)$, then no other process $q \neq p$ delivers $v'(g)$ before $\underline{v}(g)$
    - integrity
      - If process p delivers view $v(g)$ then $p \in v(g)$
    - non-triviality
      - If process q joins a group and is or becomes indefinitely reachable from process $p \neq q$, then eventually q is always in the views that p delivers

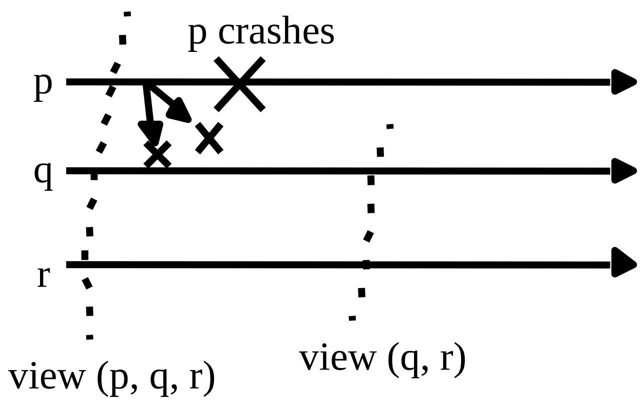
Membership Service (cont.)

- View-synchronous group communication
  - Guarantees provided by view-synchronous group communication
    - agreement
      - correct processes deliver the same set of messages in any given view
    - integrity
      - if a process *p* delivers message *m*, then it will not deliver *m* again
    - validity
      - correct processes always deliver the messages that they send

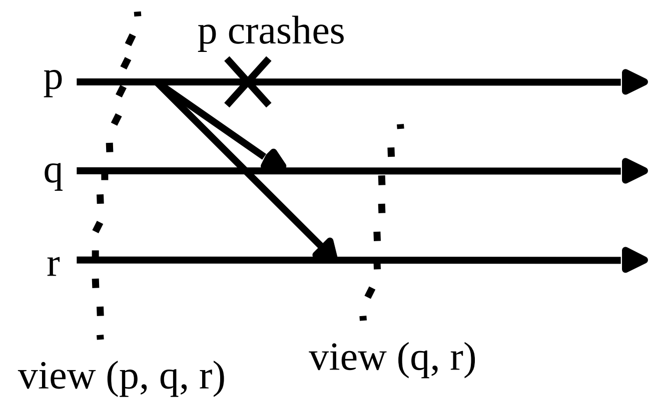
Membership Service (cont.)

- View-synchronous group communication (cont.)

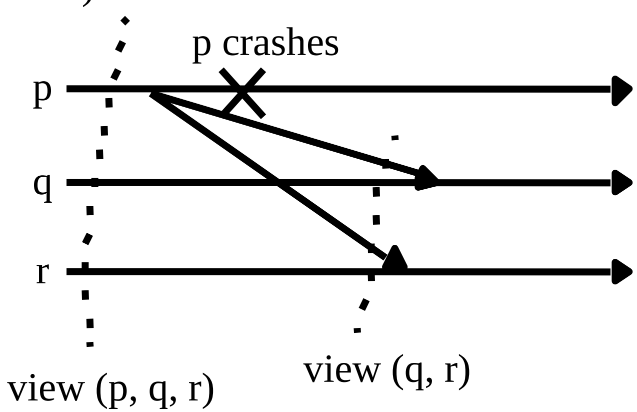
a (allowed).



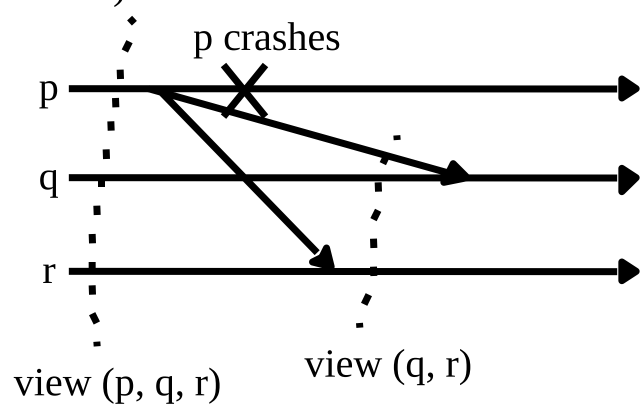
b (allowed).



c (disallowed).



d (disallowed).

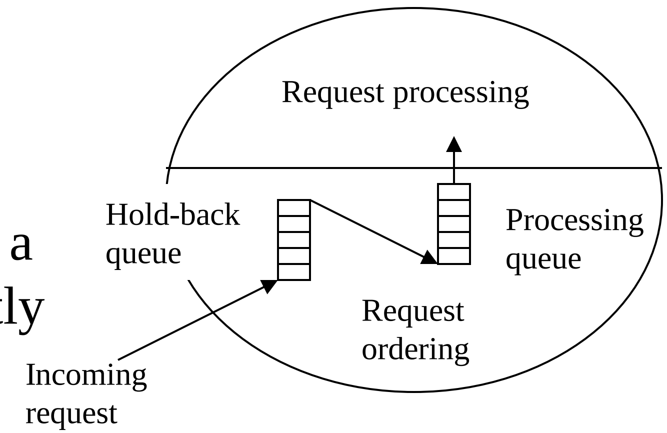


Request Ordering

- Why ordering is concerned?
  - concurrent execution of update requests at replicas may result in inconsistency among replicated data
  - ⇨ serial equivalence of update requests is required
    - expense of ordering should also be considered
- Ordering requirements
  - total ordering
    - requests are processed in the same order at all replicas
  - causal ordering
    - causally related requests are only ordered at all replicas
  - sync ordering
    - requests are ordered in sync before or after a certain request at all replicas

Request Ordering (cont.)

- Request handling at replicas
  - every request is *held-back* until ordering constraints can be met
  - request is defined to be *stable* at a replica once no request from a client and bearing a lower unique identifier can be subsequently delivered to replica; that is, all prior requests have been processed



- Properties for request ordering
  - safety
    - no message will be delivered out of order from hold-back queue to processing queue
    - once a message has been in the processing queue, no prior request should not be in there
  - liveness
    - no message should wait indefinitely in hold-back queue

Request Ordering (cont.)

- Total ordering implementation
  - requires a mechanism to uniquely sequence each request, which enables sequential ordering among messages
  - unique id generation
    - sequencer approach
      - request id is generated by a designated process, sequencer
      - every request is sent to the sequencer which assigns a unique id being incremented monotonically and forwards the request to replicas
      - sequencer may become performance bottleneck and point of failure
    - data update protocol approach
      - token holder sends a request with a temporary id to all replicas
      - each replica (site i) replies with a new id of max(temp id, id) + 1 + i/N; token holder selects largest id among proposed id from all replicas and uses it as the agreed id
      - token holder notifies all replicas of the final id; replica readjusts the message's position at hold-back queue

Request Ordering (cont.)

- Causal ordering implementation
  - requires a mechanism to enable causally related requests to be ordered
  - vector timestamp approach
    - all replicas p_i initializes VT_i (vector time) to zeros
    - when p_i generates a new event, it increments $VT_i[I]$ by 1; it attaches the value $vt = VT_i$ on outgoing messages
    - when p_j handles a request with timestamp *vt*, it updates it vector clock such as Vt_j = merge (Vt_j, vt)