University of Freiburg, Germany
Department of Computer Science

Distributed Systems

Chapter 4 Coordination and Agreement

Christian Schindelhauer

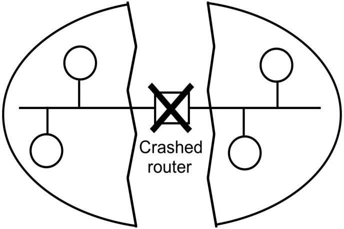
19. May 2014

4.1: Introduction

- Coordination in the absence of master-slave relationship
- Failures and how to deal with it
- Distributed mutual exclusion
- Agreement is a complex problem
- Multicast communication
- Byzantine agreement

Assumptions

- Channels are reliable
- The network remains connected
- Process failures are not a threat to the communication
- Processes only fail by crashing



Failure Detectors

- Failure detector is a service answer queries about the failures of *other* processes
- Most failure detectors are *unreliable failure detectors*
  - Returning either *suspected* or *unsuspected*
  - *suspected*: some indication of process failure
  - *unsuspected*: no evidence for process failure
- *Reliable failure detector*
  - Returning either *failed* or *unsuspected*
  - *failed*: detector has determined that the process has failed
  - *unsuspected*: no evidence for failure

Example of an unreliable failure detector

- Each process p sends a 'p is here' message to every other process every T seconds
- If the message does not arrive within $T + D$ seconds then the process is reported as *Suspected*

4.2: Distributed Mutual Exclusion

- Problem known from operating systems (there: *critical sections*)
- How to achieve mutual exclusion only with messages

Application-Level Protocol

enter()	enter critical section – block if necessary
resourceAccesses()	access shared resources in critical section
exit()	leave critical section – other processes may enter

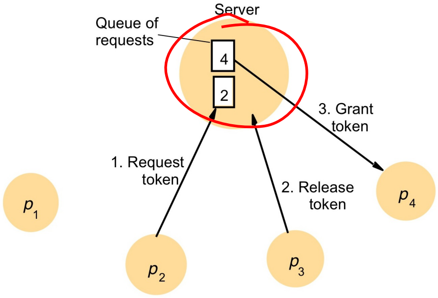
Essential Requirements

ME1: Safety	At most one process may execute the critical section at a time
ME2: Liveness	Requests to enter and exist the critical section eventually succeed
ME3: $\rightarrow$ ordering	requests enter the critical section according to the *happened-before* relationship

Performance of algorithms for mutual exclusion

- *Bandwidth* consumed: proportional to the number of messages sent in each *entry* and *exit* operation
- *Client delay* at each *entry* and *exit* operation
- *Throughput* rate of several processes entering the critical section
- Throughput is measured by the *synchronization delay* between one process exiting the critical section and the next process entering it
- short *synchronization delay* correspond to high *throughput*

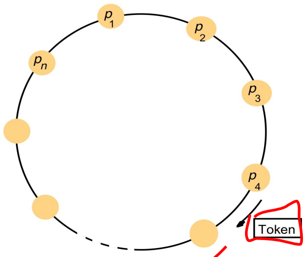
Central Server Algorithm



from *Distributed Systems – Concepts and Design*, Coulouris, Dollimore, Kindberg

- Simplest solution
- Request are handled by queues
- Performance
  - Entering the critical section: two messages (*request*, *grant*)
  - Leaving the critical section: one message (*release*)
- Server is performance bottleneck

Ring Based Algorithm



from *Distributed Systems Concepts and Design*, Coulouris, Dollimore, Kindberg

- Simplest distributed solution
- Arrange processes as ring (not related to physical network)
- A token (permission to enter critical section) is passed around
- Conditions ME1 (safety) and ME2 (liveness) are met
- ME3: $\rightarrow$ ordering is not fulfilled
- Continuous consumption of bandwidth
- Synchronisation delay is between 1 and n messages.

The Algorithm of Ricart and Agrawala

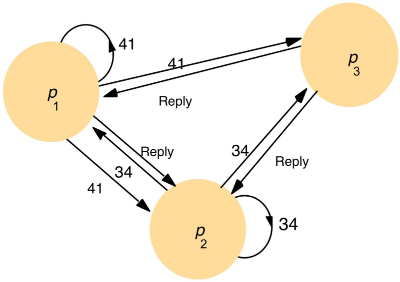
- Mutual exclusion between n peer processes $p_1, p_2, \ldots, p_n$ which
  - have unique numeric identifiers
  - possess communication channels to one another
  - keep Lamport clocks attached to the messages
- Process states
  - `released`: outside the critical section
  - `wanted`: wanting to enter critical section
  - `held`: being in the critical section
- Each process `released` immediately answers a request to enter the critical section
- The process with `held` does not reply to requests until it is finished
- If more than one process requests the entry, the first one collecting the $n-1$ replies is allowed to enter the critical section.
- If the Lamport clocks of the latest messages do not differ, the numeric ID is used to break the tie.

The Algorithm of Ricart and Agrawala

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

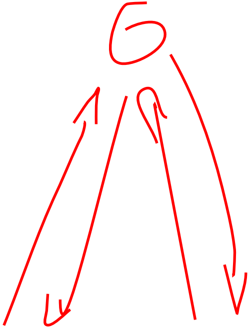
On receipt of a request $<T_i, p_i>$ *at* p_j $(i \neq j)$
    if (*state* = HELD or (*state* = WANTED *and* $(T, p_j) < (T_i, p_i)$))
    then
        queue *request* from p_i without replying;
    else
        reply immediately to p_i;
    end if
To exit the critical section
    state := RELEASED;
    reply to any queued requests;

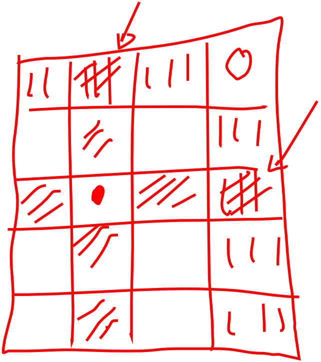


from *Distributed Systems – Concepts and Design*, Coulouris, Dollimore, Kindberg

The Algorithm of Ricart and Agrawala

- Mutual exclusion properties
  - ME1 (safety): processes in state `held` prevent other ones from entering the CS
  - ME2 (liveness): follows from the ordering
  - ME3 (ordering): follows from the use of Lamport clocks
- Cost of gaining access: $2(n-1)$ messages
  - $n-1$ for multicast of request
  - $n-1$ for replies
- Client delay for requesting entry: a round-trip message
- Synchronization delay is one message transmission time





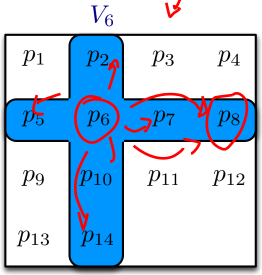
$n = k^2$

$$\boxed{2(k-1)} = 2(\sqrt{n} - 1)$$

$$= O(\sqrt{n})$$

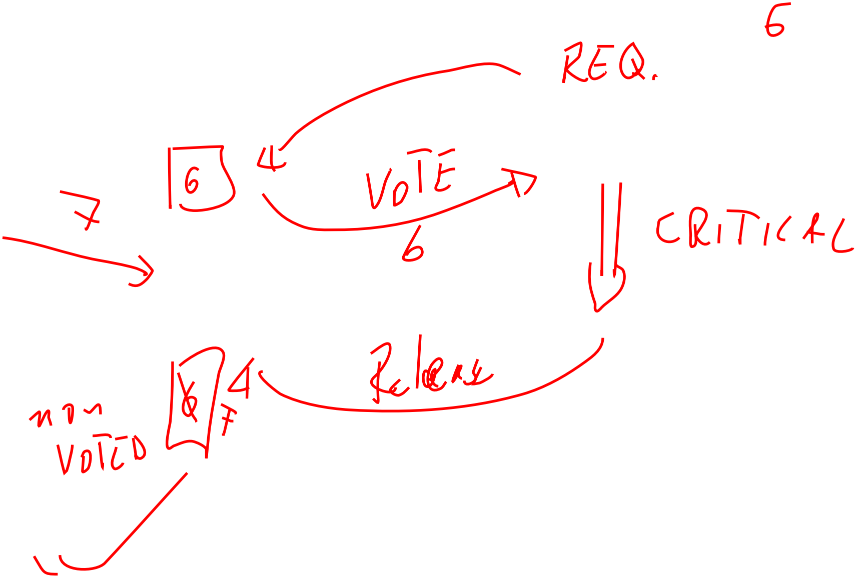
Maekawa's Voting algorithm

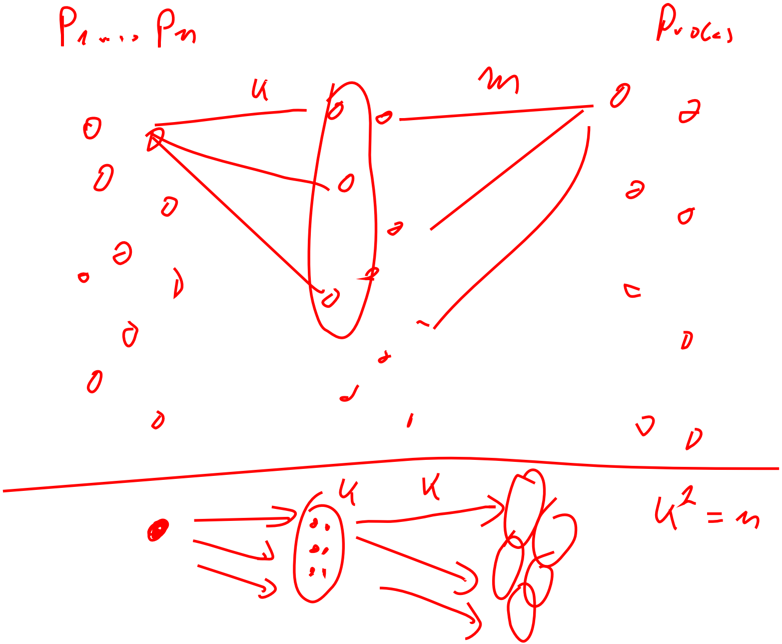
- Reduce the number of messages by asking a subset
- For each process p_i choose a *voting set* V_i such that
  1. $p_i \in V_i$
  2. $V_i \cap V_j \neq \emptyset$ for all i, j
  3. $|V_i| = k$ for all i (fairness)
  4. Each process occurs in at most m voting sets
- Minimal choice of $\max\{m, k\}$ is $k, m \in \Theta(\sqrt{n})$.
- The optimal solution can be approximated by placing all nodes in a square matrix and choosing the row and column as voting set.



from

Distributed Systems – Concepts and Design, Coulouris, Dollimore, Kindberg


6
REQ.
6
4
VOTE
6
7
CRITICAL
Release
4
7
non
VOTED



P1 ..., Pn
Procs
k
m
k
k
k^2 = n

Maekawa's Voting algorithm

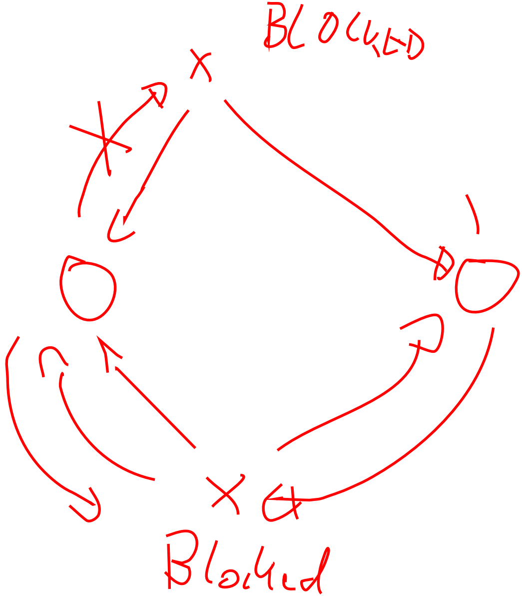
On initialization
  state := RELEASED;
  voted := FALSE;

For p_i to enter the critical section
  state := WANTED;
  Multicast *request* to all processes in V_i;
  Wait until (number of replies received = K);
  state := HELD;

On receipt of a request from p_i at p_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

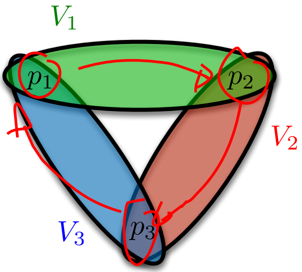
For p_i to exit the critical section
  state := RELEASED;
  Multicast *release* to all processes in V_i;

On receipt of a release from p_i at p_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


BLOCKED
Blocked

Maekawa's Voting algorithm

- Mutual exclusion properties
  - ME1 (safety): follows from the intersections of V_i and V_j
  - ME2 (liveness): not guaranteed.
- Sanders improved this algorithm to achieve ME2 and ME3 (not presented here)
- Cost
  - $2k$ per entry to the critical section
  - k for exit
  - $O(\sqrt{n})$ messages
- Client delay for requesting entry: a round-trip message
- Synchronization delay is a round-trip message



Mutual Exclusion

Fault Tolerance

- What happens when messages are lost
- What happens when process crashes

- All of the above algorithms presented fail
- We will revisit this problem

4.3: Elections

Election Algorithm

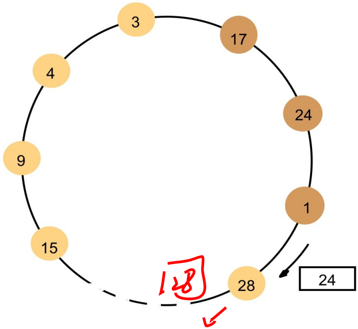
- An algorithm for choosing a unique process from a set of processes $p_1, \ldots, p_n$.
- A process *calls the election* if it initiates a run of an election algorithm
- Several elections could run in parallel where subset of processes are *participants* or *non-participants*.
- We assume processes have numeric IDs and that wlog. the process with the highest will be chosen.

Requirements

E1: Safety	During the run each participant has either $\text{elected}_i = \perp$ or $\text{elected}_i = P$, where P is the non-crashed process with the largest ID
E2: Liveness	All participating processes p_i eventually set $\text{elected}_i \neq \perp$ or crash.

Ring-Based Election: Algorithm of Chang and Roberts

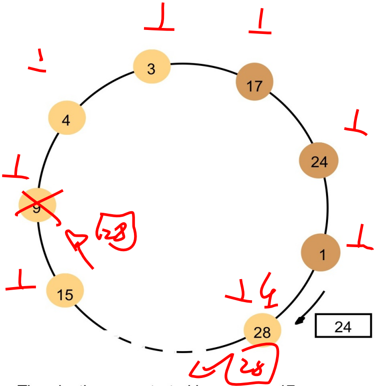
- Each process p_i has a communication channel to the next process in the ring $p_{(i+1) \bmod n}$
- Messages are sent clockwise
- Assumption: no failures occur
- Non-participants are marked
- When a process receives an election message, it compares the identifier
  - If the arrived ID is greater, it forwards it
  - if the arrived ID is smaller and the process participates, it replaces it with its ID
  - if the arrived ID equals the process ID, the process is elected and sends an elected message around (with its ID).



Note: The election was started by process 17.
The highest process identifier encountered so far is 24.
Participant processes are shown darkened

Ring-Based Election: Algorithm of Chang and Roberts

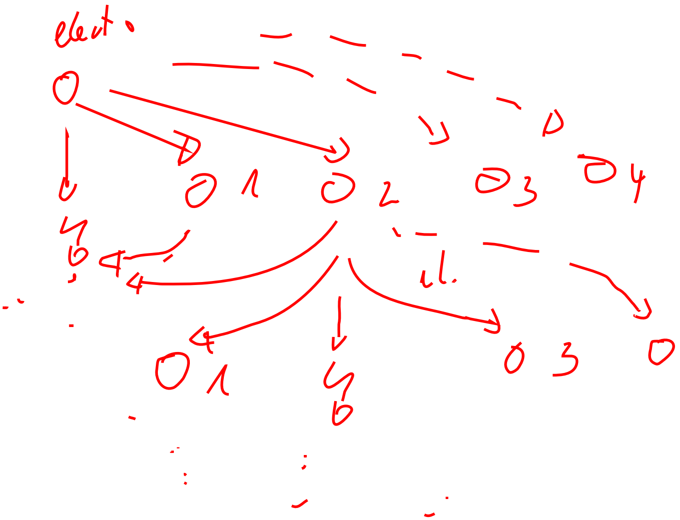
- E1 (Safety): follows directly
- E2 (Liveness): follows in the absence of crashes and communication errors
- Worst-case performance if a single node participates in the process
- Time: $3n - 1$ messages for the election
- Not very practical algorithm fault-prone and high communication overhead
- assumes a-priori knowledge (ring topology)

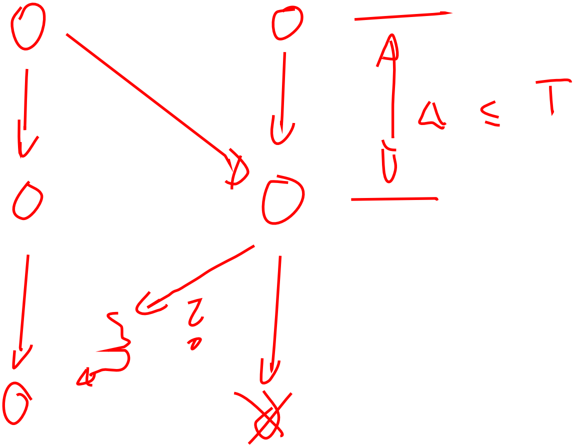


Note: The election was started by process 17.
The highest process identifier encountered so far is 24.
Participant processes are shown darkened

The Bully Algorithm of Garcia & Molina

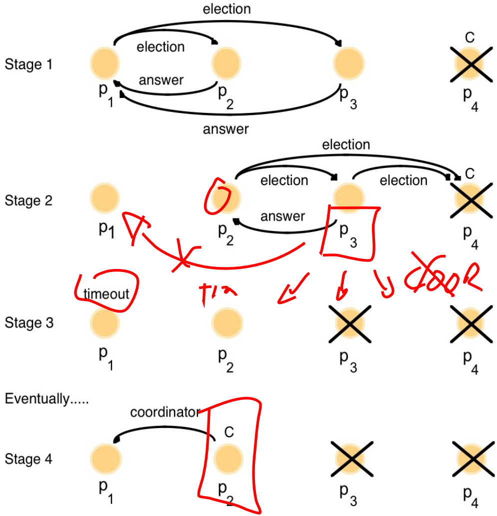
- The distributed system is assumed to be synchronous
  - i.e. after a timeout period T a missing answer is interpreted as crash
  - reliable failure detector
  - fail-stop model
- Message types
  - *election*: Announces an election
  - *answer*: Answers *election* message (contains ID)
  - *coordinator*: Announces the identity of the elected process
- Any process may trigger an *election*
- Every process receiving an *election* messages sends an *answer* and starts a new one (if it has not started one before).
- If a process knows it has the highest ID (based on the answers) it sends the *coordinator* message to all processes
- If answers of lower IDs fail to arrive within time T the sender considers itself a coordinator and sends the *coordinator* message


elect.
0
1
2
3
4
0
4
1
4
6
3
0


Δ ≤ T
?

The Bully Algorithm of Garcia & Molina

- If a process receives an *election* message it sends back an *answer* messages and begins another election — if it has not begun an election
- If a process knows it has the highest ID it sends the *coordinator* message
- New arriving processes with higher ID „bully" existing cordinators



The Bully Algorithm of Garcia & Molina

- E2: liveness condition is guaranteed if messages are transmitted reliably
- E1: safety condition: Not guaranteed if processes are replaced by processes with the same identifier
- different conclusions on which is the coordinator process
- E1 not guaranteed if the timeout value is too small
- In the worst case the algorithm needs $O(n^2)$ messages for n processes

4.4: Multicast communication

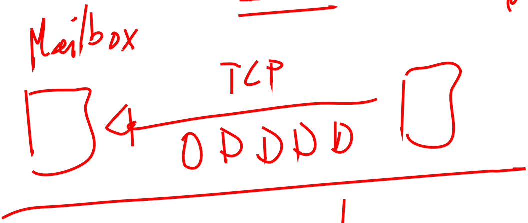
- With a single call of *multicast*(g, m) a process sends a message to all members of the group g
- Using *deliver*(m), received messages are delivered on participating processes
- *Efficiency*
  - Number of messages, transmission time
- *Delivery guarantees*
  - ordering
  - receipt
  - e.g. IP Multicast does not guarantee ordering of success

IPv4

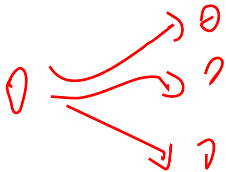
IPv6

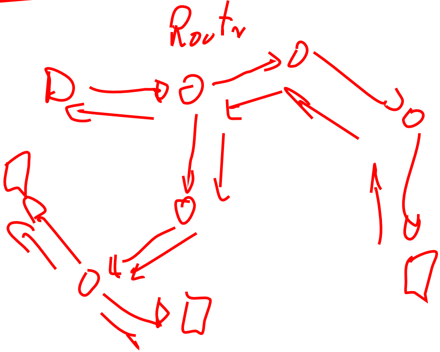
Internet

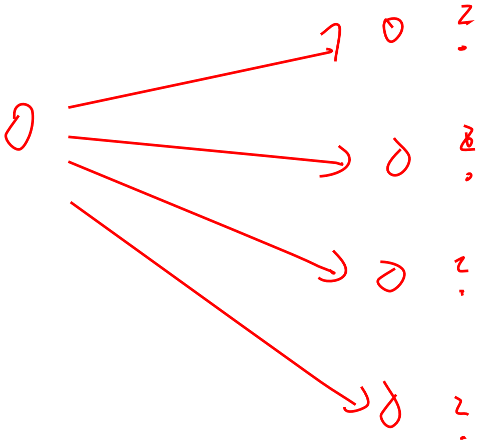
IP-Multicast

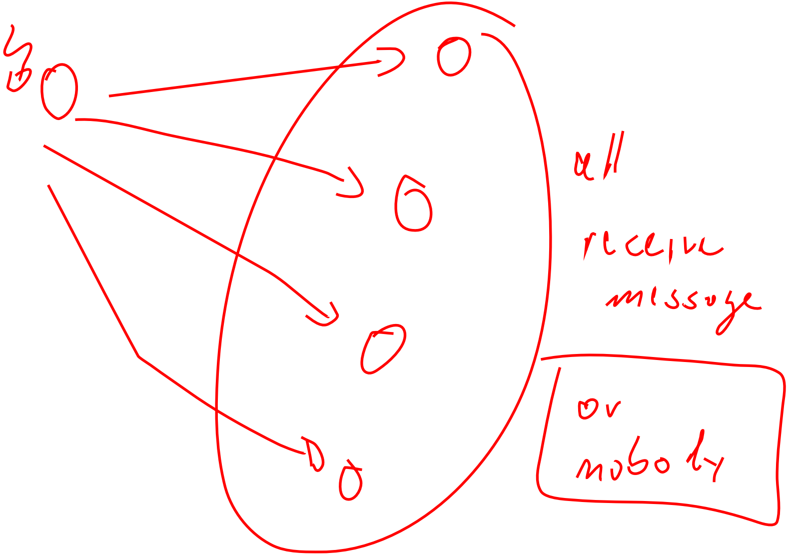


Unicast



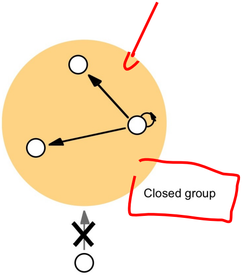


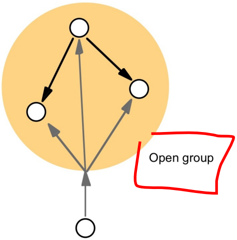


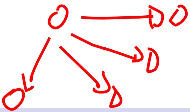

all receive message
or nobody

4.4: Multicast communication

- *System Model*
  - *multicast*(g, m): sends the message m to all members of group g
  - *deliver*(m): delivers a message to the process (message has been received by lower level)
  - *sender*(m): sender of a message m (within the message header)
  - *group*(m): group of a message m (within the message header)
- Allowed senders
  - closed group: senders must be members of a group
  - open group: any process can send a message to the group







Basic Multicast

- *B-multicast*(g, m): for each process $p \in g$, *send*(p, m)
- *B-deliver*(m): if message m is received at p return the message m

Ack Implosion

- if too many processes participate
- if *send* uses acknowledgments, some of them could be dropped
- then the messages could be retransmitted
- further *acks* are lost due to full buffers etc.

Reliable Multicast

- *Safety: Integrity*
  - Every message is delivered at most once
  - Receiver of m is a member of $group(m)$
  - Sender has initiated a $multicast(g, m)$
- *Liveness: Validity*
  - If a correct process multicasts a messages then it eventually delivers m (to itself)
- *Agreement*
  - If a correct process delivers m then all other processes eventually deliver m

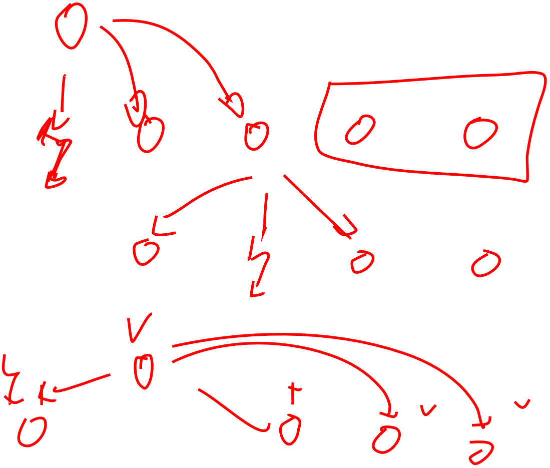
Implementing Reliable Multicast over Basic Multicast

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

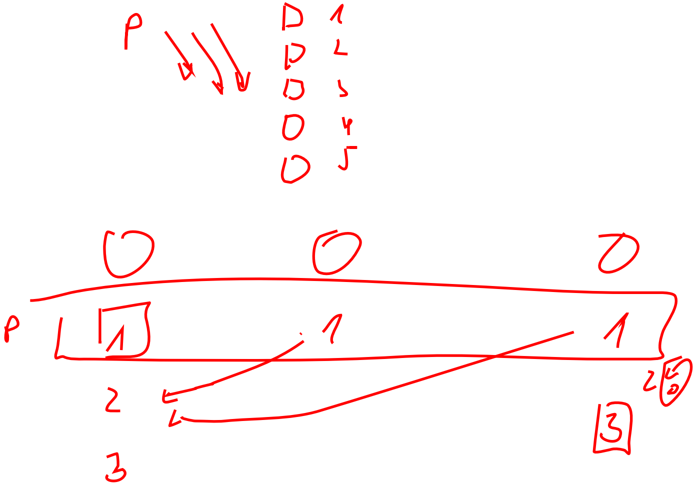
On B-deliver(m) at process q with g = group(m)
    if $(m \notin Received)$
    then
        $Received := Received \cup \{m\};$
        if $(q \neq p)$ *then* $B\text{-}multicast(g, m);$ *end if*
        R-deliver m;
    end if

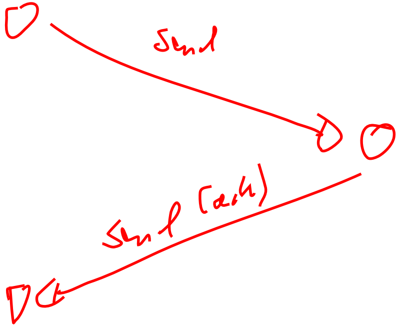
Each message needs to be sent $|g|$ times!



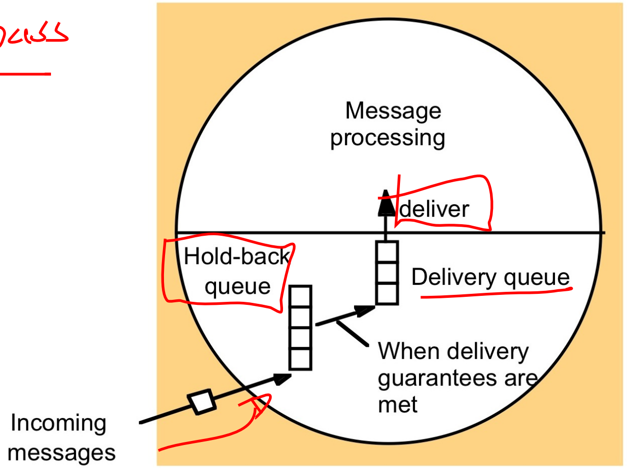
Implementing Reliable Multicast over IP Multicast

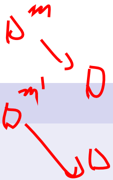
- *R-multicast*(g, m) for sending process p
  - Sender increments a (sending) sequence number S_g^p for group g after each messages
  - Sequence number sent with message
  - Acknowledgements of all received messages with $\langle q, R_g^q \rangle$ are piggybacked with message
  - Negative Acknowledgments: by received sequence number R_g^q causes retransmission of message
- *R-deliver*(g) for receiving process q
  - R_g^q is the sequence number of the latest message it has delivered.
  - it is sent with each acknowledgment and allows the sender (and all receivers) to learn about missing messages
  - Process q *delivers* a message m (with piggybacked S) only if $S = R_g^q + 1$.
  - messages with $S > R_g^q + 1$ are kept in a hold-back queue
  - messages with $S < R_g^q + 1$ are erased
  - After delivery $R_g^q := R_g^q + 1$


P
1
2
3
4
5
P
1
1
1
2
3
2
3


Send
Send (ack)

Hold-Back Queue for Arriving Multicast Messages



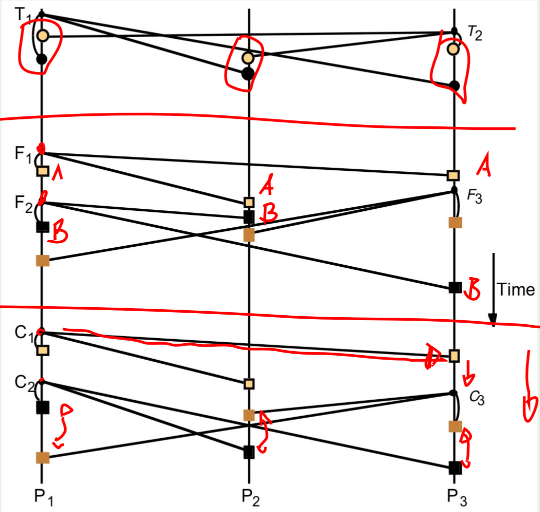


Ordered Multicast

- *FIFO Ordering*
  - If a process casts $\texttt{multicast}(g, m)$ before $\texttt{multicast}(g, m')$
  - then m is delivered before m'
  - in each process of group g
- *Causal Ordering:*
  - If $\texttt{multicast}(g, m) \rightarrow \texttt{multicast}(g, m')$
  - then m is delivered before m'
  - $\rightarrow$ is based only on messages within the group g
- *Total Ordering:*
  - If a process delivers m before m'
  - then m is delivered before m' on any other process of g

Total, FIFO and Causal Ordering

- *Total Ordering*
- *FIFO Ordering*
- *Causal Ordering*



Bulletin Board

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		

- *FIFO Ordering*
- *Causal Ordering*
- *Total Ordering*