Coordination and Agreement

From **Coulouris, Dollimore, and Kindberg**
Updated and revised by **Kulik**

Aims

For a set of processes in a distributed system

- Coordinate of actions of the independent processes to achieve common goals
- Agree on values
- Even if there are failures in the system reach these aims

Synchronous systems

- Computation step of a process has known lower and upper bounds
- Message delivery times are bounded to a known value
- Each process has a clock whose drift rate from real time is bounded by a known value

Asynchronous systems

- No bounds on process execution times, message delivery times clock drift rate

Coordination Problems in DS

Asynchronous distributed systems

- No single process has a view of the current global system state

Failure detection

- How does a process determine whether a peer is dead or alive?

Mutual exclusion

- No two process can access a shared resource in a critical section at the same time
- Simple solution: a single server and a locking mechanism to allow orderly access
- Sought solution: processes, among themselves, agree on how to access a resource or any other shared entity through passing messages

Coordination Problems in DS

Election in master-slave systems

- While booting up or when the master fails

Multicast

- Reliability of multicast (correct delivery, only once, etc.)
- Order preservation

Consensus in the presence of faults (byzantine problems):

- Determine whether acknowledgement was received over an unreliable communication medium
- Determine whether peer process knows about one's own intentions in the presence of a non-confidential communication channel

Assumptions

Processes are connected via reliable channels

Processes do not rely on others to communicate

Processes themselves can crash or act unexpectedly

- A failure detector is a service that is used to realize that another process has failed
- Failure detectors may also be unreliable
  - We often use timeouts for detection, this may give false indications

Failure Detection

Failure Detector

- Service that can decide whether or not a particular process has crashed
- Can collaborate with other processes to detect failure

Unreliable failure detector

- Unsuspected peer process
  - Failure is unlikely, e.g., failure detector has recently received communication from unsuspected peer
  - May be inaccurate
- Suspected peer process
  - Indication that peer process failed, e.g., no message received in quite some time
  - May be inaccurate, e.g., peer process has not failed, but the communication link is down or peer process is much slower than expected

Reliable failure detector

- Unsuspected: potentially inaccurate as above
- Failed: accurate determination that peer process has failed

Failure Detector

Implementation of an unreliable failure detector

- Periodically, every *T* seconds each process *p* sends a “p is here” message to every other process
- If a local failure detector at *q* does not receive “p is here” from *p* within *T*+*D* (*D* = estimated maximum transmission delay), then *p* is suspected
- If message is subsequently received, *p* is declared OK

Problem: calibration of transmission delay

- For small *D*, intermittent network performance downgrades will lead to suspected nodes
- For large *D* crashes will remain unobserved (crashed nodes will be fixed before timeout expires)

Solution

- Variable *D* that reflect the observed network latencies

Implementation of reliable failure detectors

- Only possible in a synchronous network

Distributed Mutual Exclusion

Task

- Distributed processes need to coordinate activities, e.g., for resource sharing
- Processes access shared resources, variables, etc. in a *critical section*
- Processes have to enter to this code part properly

Assumptions

- There is only one critical section
- The system is asynchronous
- Message delivery is reliable and messages are delivered intact
- For this case: processes do not fail

Mutual Exclusion Problems

Prominent problem in multitasking operating systems

- Access to shared memory
- Access to shared resources
- Access to shared data
- Various algorithms to ensure mutual exclusion

Mutual exclusion in distributed systems

- No shared memory
- Usually, no centralized instance like operating system kernel that would coordinate access
- Based on a synchronous or asynchronous, usually failure-prone network infrastructure

Examples

- Consistent access to shared files (e.g., Network File Systems)
- Coordination of access to an access point in an IEEE 802.11 WaveLAN

Application Level Protocol

***enter*()**

- Enter critical section
- Block if necessary

***resourceAccess*()**

- Access shared resources in critical section

***exit*()**

- Leave critical section
- Other processes may now enter

Conditions for Mutual Exclusion

ME1: (Safety)

- At most one process may execute in the critical section at a time

ME2: (Liveness)

- Requests to enter and exit the critical section eventually succeed (i.e., no deadlocks or starvation)
- Impossible for one process to enter critical section more than once while other processes are awaiting entry

Use of happened-before relation

- It is not possible to order the access requests using time perfectly as there is no global clock

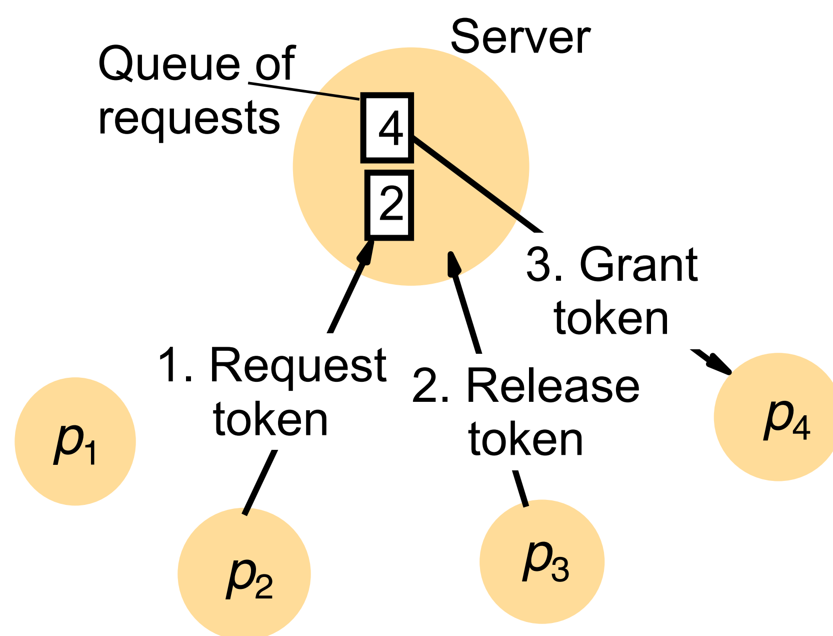
ME3: ($\rightarrow$ Ordering)

- If one request r_1 to a resource happened before another request r_2 then r_1 should enter the critical section before r_2

Central Server Algorithm

Central Server-based Algorithm

- Central server receives access requests
  - If no process in critical section, request will be granted
  - If process in critical section, request will be queued
- Process leaves critical section
- Grant access to next process in queue, or wait for new requests if queue is empty



Safety and Liveness is guaranteed

Ordering (ME3) is not guaranteed

- Network delays may reorder requests

Performance and availability of server are the bottlenecks

Evaluation of Mutual Exclusion Algorithms

Bandwidth consumption

- Number of messages sent in each entry and exit operation

Client Delays

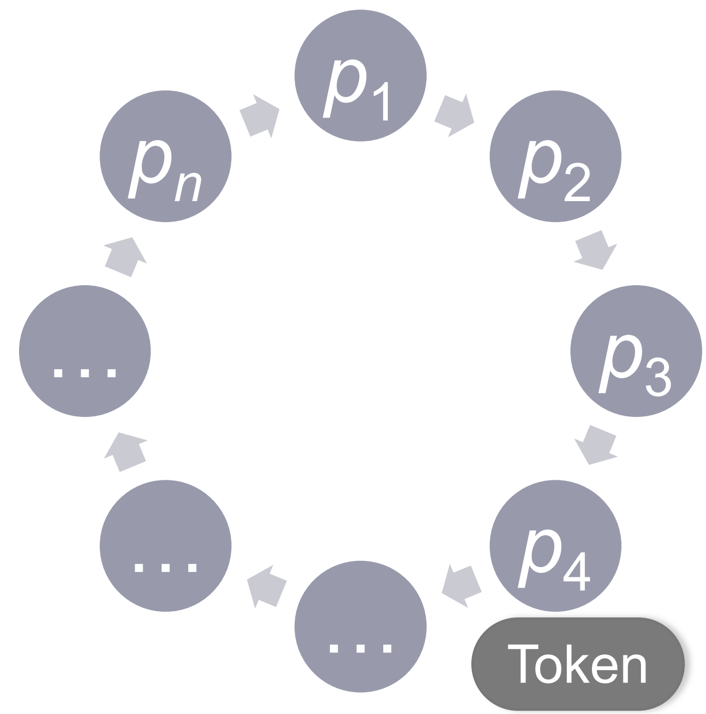
- At each entry and exit

Throughput

- Number of critical region accesses that the system allows
- Measure effect using the *synchronization delay* of one process exiting the critical section and the next process entering it
- Throughput is greater when the synchronization delay is shorter

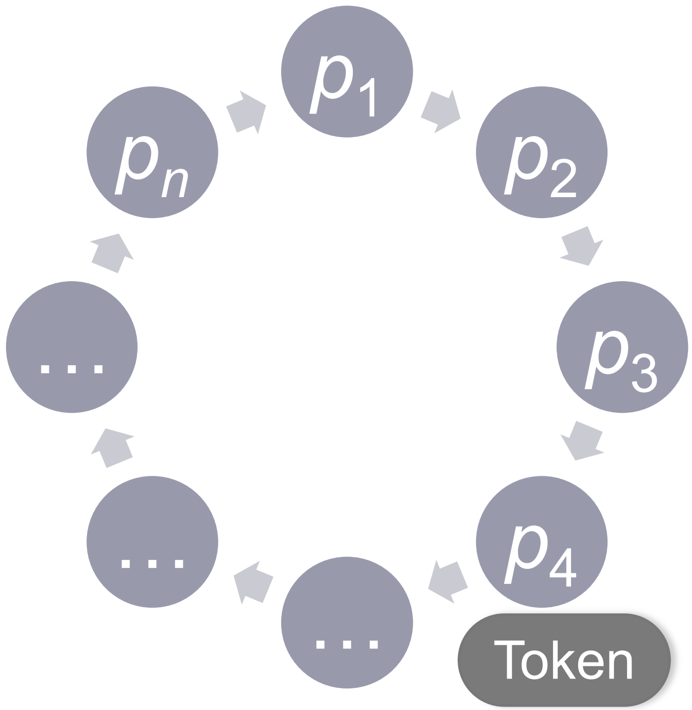
Ring-based Algorithm

- Logical, not necessarily physical link: every process p_i has connection to process $p_{(i+1) \bmod N}$
- Token passes in one direction through the ring
- Token arrival
  - Only process in possession of token may access critical region
  - If no request upon arrival of token, or when exiting critical region, pass token on to neighbor
- Algorithm satisfies ME1, ME2, but not ME3



Ring-based Algorithm II

- Algorithm satisfies ME1, ME2, but not ME3 (happened before relation)
- Performance
  - Constant bandwidth consumption
  - Entry delay between 0 and *N* message transmission times
  - Synchronization delay between 1 and *N* message transmission times
  - The algorithm wastes bandwidth if no process requests the token



Using Multicast and Logical Clocks

- Each process requests to enter a critical section via multicast
- When others reply to the multicast, then you can decide on the request
- All the conditions ME1 – ME3 are met
- Processes use Lamport clocks and their IDs
  - Timestamps are used to resolve simultaneous requests
  - If two timestamps are the same, then process IDs are used

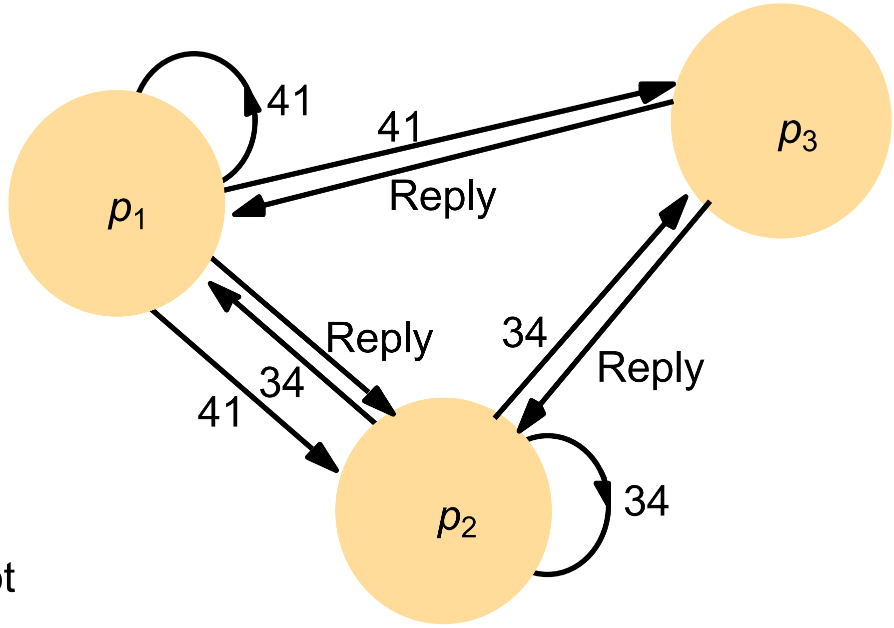
Ricart and Agrawala's Algorithm

```
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;

On receipt of a request <Ti, pi> at pj (i ≠ j)
    if (state = HELD or (state = WANTED and (T, pj) < (Ti, pi)))
    then
        queue request from pi without replying;
    else
        reply immediately to pi;
    end if

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

Multicast Synchronization

Example

- p_3 does not want to enter the critical section
- p_1 and p_2 request entry concurrently
- Timestamp of p_1's request is 41, and that of p_2 is 34
- p_3 replies immediately to p_1's and p_2's request
- p_2 receives p_1's request: its own request has the lower timestamp and thus does not reply to p_1
- p_1 finds that p_2's request has a lower timestamp than that its own request and replies immediately
- Once p_2 receives a reply from from p_1, receiving this second reply, it can enter the critical section
- When p_2 exits the critical section, it will reply to p_1's request



Multicast-based Mutual Exclusion Contd

Performance

- Expensive access requires 2(*N*-1) messages per request
- Synchronization delay: just one message transmission time (previous algorithms: up to *N*)
- Improved versions exist for reducing bandwidth costs, e.g., repeated entry of same process without executing protocol

ME1 – ME3:

- Algorithm satisfies ME1
  - Two processes p_i and p_j can only access critical section at the same time in case they would have replied to each other
  - BUT pairs $<T_i, p_i>$ are totally ordered (contradiction!)
- It can be shown that ME2 and ME3 are satisfied as well

Maekawa's Voting Algorithm

Idea

- Not all processes have to agree to decide on who enters to the critical section
- Subsets can be used: split set of processes up into overlapping subsets ("voting sets")
- Processes vote for other processes to decide
- A candidate has to collect enough votes to enter to the critical section
- Sufficient condition: there is consensus within every subset

Important observation

- Subsets used by any two process should overlap: the intersection processes in sets ensure the safety condition is met
- At most one process can enter the critical section, by casting their votes for only one candidate

Maekawa's Voting Algorithm

Model

- For N processes $p_1, ..., p_N$, voting sets $V_1, ..., V_N$ are chosen such that:
  - $\forall i\ \forall j : i \neq j, 1 \leq i, j \leq N : V_i \cap V_j \neq \{\}$ (any two voting sets overlap)
  - $\forall i : 1 \leq i \leq N : p_i \in V_i$
  - $\forall i : 1 \leq i \leq N : |V_i| = K$ (fairness: all voting sets have equal size)
  - Each process p_j is contained in K of the voting sets V_i

Optimization goal

- Minimize K while achieving mutual exclusion
- Achieved when $K \sim \sqrt{N}$

Optimal voting sets

- Nontrivial to calculate, approximation: derive V_i so that $|V_i| \sim 2\sqrt{N}$
- Place processes in a $\sqrt{N}$ by $\sqrt{N}$ matrix
- Let V_i the union of the row and column containing p

Maekawa's Algorithm

```
On initialization
  state := RELEASED;
  voted := FALSE;
For pi to enter the critical section
  state := WANTED;
  Multicast request to all processes in
Vi;
  Wait until (number of replies received
= K);
  state := HELD;
On receipt of a request from pi at pj
  if (state = HELD or voted = TRUE)
  then
    queue request from pi without
replying;
  else
    send reply to pi;
    voted := TRUE;
  end if
```

```
For pi to exit the critical section
  state := RELEASED;
  Multicast release to all processes
in Vi;
On receipt of a release from pi at pj
  if (queue of requests is non-empty)
  then
    remove head of queue – say pk;
    send reply to pk;
    voted := TRUE;
  else
    voted := FALSE;
  end if
```

Discussion of Maekawa's Voting Algorithm

Maekawa's Voting Algorithm

Satisfies ME1

- If two processes could enter a critical section, then the processes in the non-empty intersection of their voting sets would have both granted access
- Impossible, since all processes make at most one vote after receiving request

Deadlocks are possible

- Given three processes with $V_1 = \{p_1, p_2\}$, $V_2 = \{p_2, p_3\}$, $V_3 = \{p_3, p_1\}$
- Construct a cyclic wait graph
  - p_1 replies to p_2, but queues request from p_3
  - p_2 replies to p_3, but queues request from p_1
  - p_3 replies to p_1, but queues request from p_2

Voting Contd

Algorithm is deadlock prone

- p_1, p_2, p_3 with sets $\{p_1,p_2\}$, $\{p_2,p_3\}$, $\{p_3,p_1\}$ can deadlock if all three call for a critical section access and then they vote for themselves

Extensions for satisfying ME2 and ME3

- Modification to ensure absence of deadlocks
- Use of logical clocks: processes queue requests in happened-before order

Performance

- Bandwidth utilization: $2\sqrt{N}$ per entry, $\sqrt{N}$ per exit, total $3\sqrt{N}$ is better than Ricart and Agrawala for $N>4$
- Client delay: same as for Ricart and Agrawala
- Synchronization delay: round-trip time instead of single-message transmission time in Ricart and Agrawala

Failures

All of the covered algorithms cannot tolerate a crash failure

- Ring-algorithms cannot tolerate a single crash failure

Maekawa’s algorithm

- Tolerates crash failures if process is in a voting set not required, rest of the system not affected

Central-Server

- Tolerates crash failures of a node that has neither requested access nor is currently in the critical section

Ricart and Agrawala algorithm

- Tolerate crash failures if we assume that a failed process grants all requests immediately
- Requires reliable failure detector

Elections

Election algorithms

- Choosing a unique process to play a particular role
- For example: the server-based algorithm for mutual exclusion requires the election of a server process

Fundamentals

- Each process can call an election (all processes can call simultaneously)
- Elected process should be unique
- The process with the largest value to an identifier needs to be found and elected
  - Example: elect process with lowest computational load, i.e., use $< 1/load, i >$, as its identifier, where *load* > 0 and the process index *i* is used to order identifiers with the same load
- Use of special value such as “not yet defined” (⊥) in cases where we do not hear from a process

Elections

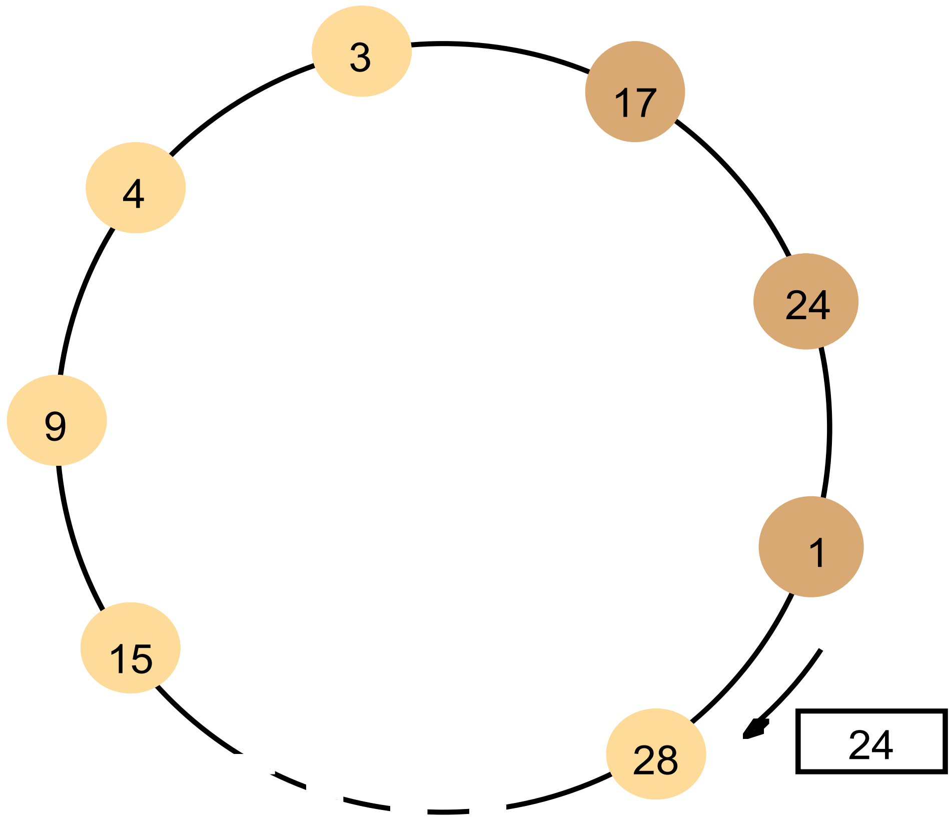
Requirements for a run of an election algorithm

- E1: a process p_i has $elected_i = \perp$(undefined) or $elected_i = P$ for some non-crashed process P that will be chosen at the end of the run with the largest identifier (*safety*)
- E2: all processes p_i will eventually set $elected_i \neq \perp$(*liveness*) or crash

Performance

- Network bandwidth utilization (proportional to total number of messages sent)
- Turnaround: number of serialized message transmission times between initiation and termination of a single run

Ring-based Election Algorithm



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

Assumptions

- All processes have a unique integer ID
- All N nodes communicate on a uni-directional ring structure
- Each process p_i has a communication channel to the next process in the ring, $p_{(i+1) \bmod N}$

Algorithm

- Initially all processes marked as non-participants
- Any process can start an election
- Election start: a process marks itself as a participant and places its ID in a message and sends it clockwise to the next

Ring-based Election II

- If a message is received, a process compares the received ID with its own
  - If the ID is greater then forward it to the next process
  - If the ID is smaller then
    - If own status is “non-participant”, then substitute own ID in election message and forward on ring
    - otherwise, do not forward message (as “participant”)
  - If received ID is identical to own ID
    - This process ID must be greatest and it becomes elected
    - Marks own status as “non-participant”
    - Sends out “elected” message
- If a message is forwarded, upon any forwarding, mark own state as “participant”
- When receiving “elected” message
  - Mark own status as “non-participant”
  - Sets its variable $elected_i$ to the ID in the message (unless it is the new coordinator)
  - Forward *elected* message

Ring-based Election Contd

E1: Safety

- Met as all identifiers are considered
- Even if elections start simultaneously the smaller identifier cannot pass the larger identifier

E2: Liveness

- Met due to reliable communication

Worst-case behavior:

- $3N - 1$ messages in total and sequentially
  - At most $2N - 1$ messages for electing the left-hand neighbor
  - Another N *elected* messages

Failures

- Not tolerated

The Bully-Algorithm

Synchronous networks

- Nodes can crash
- Crashes will be detected reliably

Assumptions

- Each node knows identifiers of *all* other nodes
- Every node can communicate with every other node

Message types

- Election: announce an election
- Answer: reply to an election message
- Coordinator: announce identity of elected process

The Bully-Algorithm II

Initiation of algorithm

- Reliable failure detection
- Failure: no answer to request within $T = 2T_{trans} + T_{process}$

Coordinator

- Process decides whether to become coordinator by comparing own ID with all other IDs (highest wins)
- Announce its role by sending coordinator message to all other nodes with lower ID
- Process with lower ID can bid to become coordinator by sending election message to all processes with higher ID
  - If no response within T, considers itself elected coordinator, sends coordinator message to all processes with lower ID
  - Otherwise: wait for another T' time units for a coordinator message to arrive from new coordinator
    - If no response, then begin another election process

The Bully-Algorithm III

Receipt of an election message

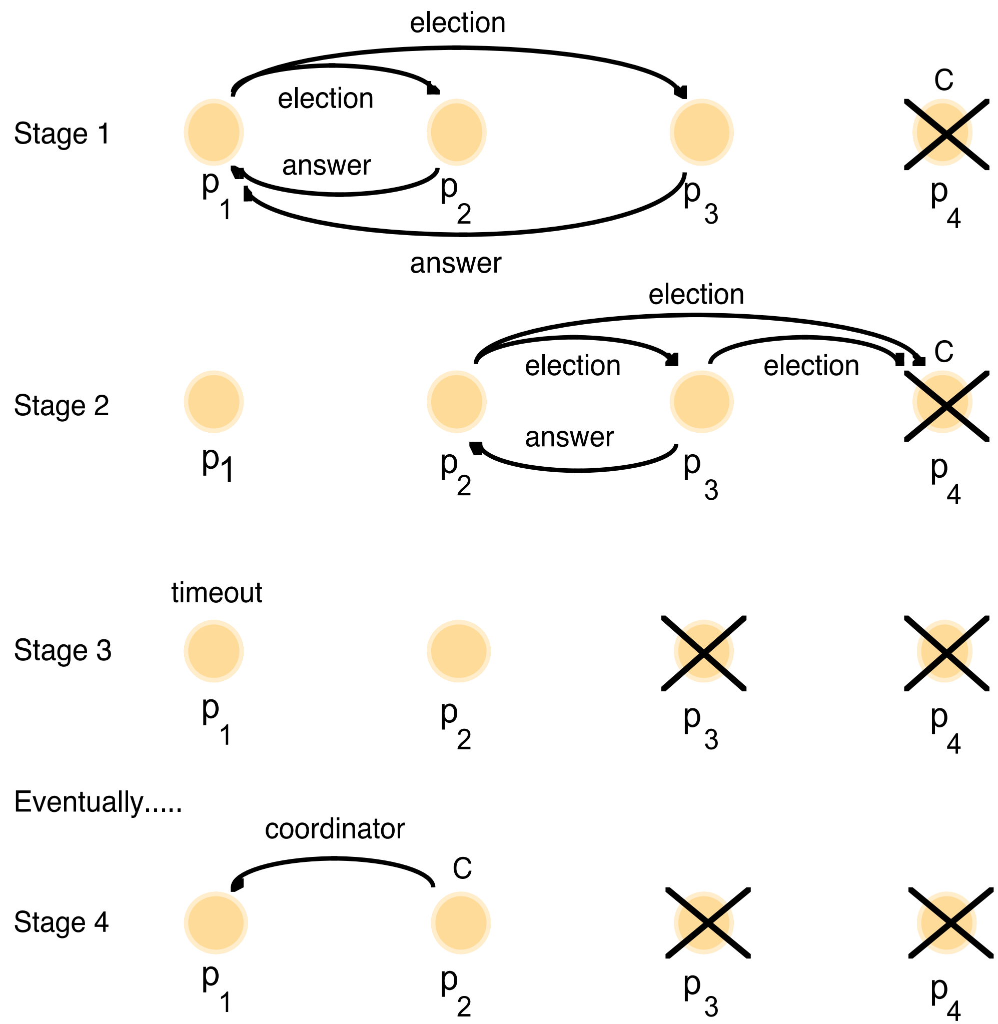
- Process sets variable *election*$_i$ to the ID of the coordinator received in the election message
- Process sends back an answer message and begins another election – unless an election was already initiated

New process replacing crashed process

- If it has the highest ID, it will immediately send coordinator message and “bully” current coordinator to resign

The Bully Algorithm

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



The Bully Algorithm: Evaluation

Properties

- E1 satisfied if no process is replaced and the timeout T estimate is accurate
- E2 satisfied for a synchronous network and reliable transmission
- E1 not satisfied if crashed process replaced at the same time while another process has announced that it is the new coordinator

Performance

- Best case: process with the second highest identifier detects coordinators failure and elects itself coordinator and sends N-2 coordinator messages
- requires $O(N^2)$ messages in worst case when lowest ID detects failure $\Rightarrow$ N-1 processes with higher IDs start election

Multicast Communication

Group communication

- Sending and delivery of messages to a subset of (broadcast is to all) processes
- Receiving of message: queuing of arriving message in network interface buffer
- Delivery of message: passing message from network interface buffer to target application
- Membership in recipient group is transparent to sender: a single send operation without having to send individual messages to all group members
  - Example: IP address for which first 4 bits are XXXX in IPv4

Issues

- Addressing
- Delivery guarantees that messages are received by a group
- Coordination: delivery ordering amongst group members

Applications of multicast

- Computer Supported Collaborative Work (CSCW)
- Communication with replicated servers (fault-tolerance)
- Event notification in networks

Multicast Communication: System Model

Message `m`:

- Contains ID of sender (`sender(m)`) and of destination group (`group(m)`)

`multicast(g, m)`

- Multicast message `m` to group `g`

`delivery(m)`

- Delivery of a message at recipient

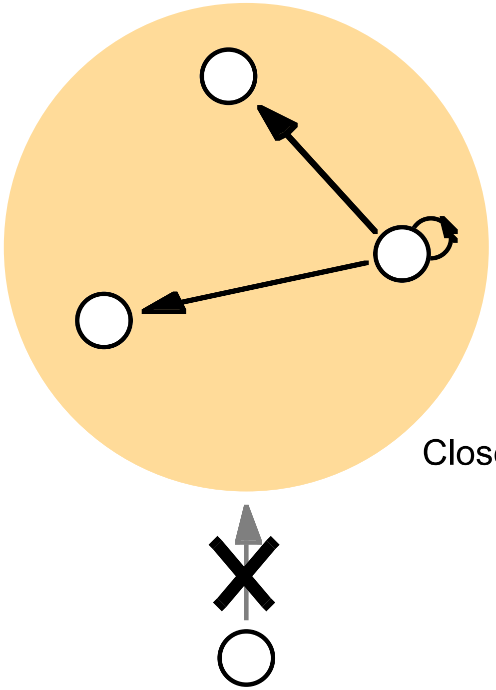
Open and Closed Multicast Groups

Closed

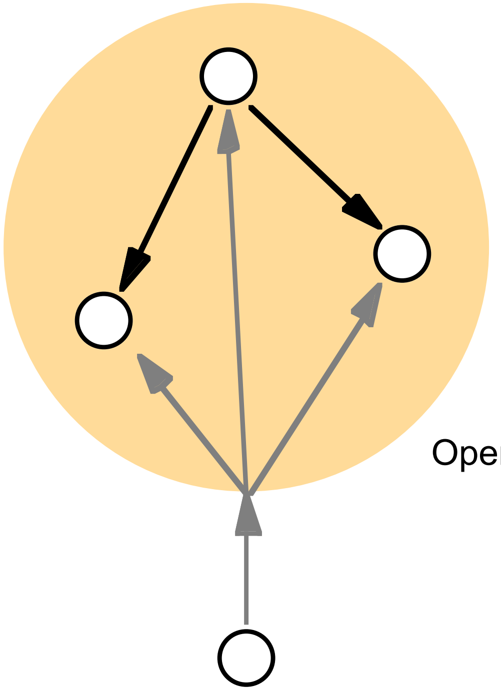
- Multicast only within group possible

Open

- Processes outside the group may send to it



Closed group



Open group

Basic Multicast

Property

- Guaranteed delivery, unless multicasting process crashes (different to IP multicast)
- Primitive *B-multicast* and its basic delivery primitive *B-deliver*

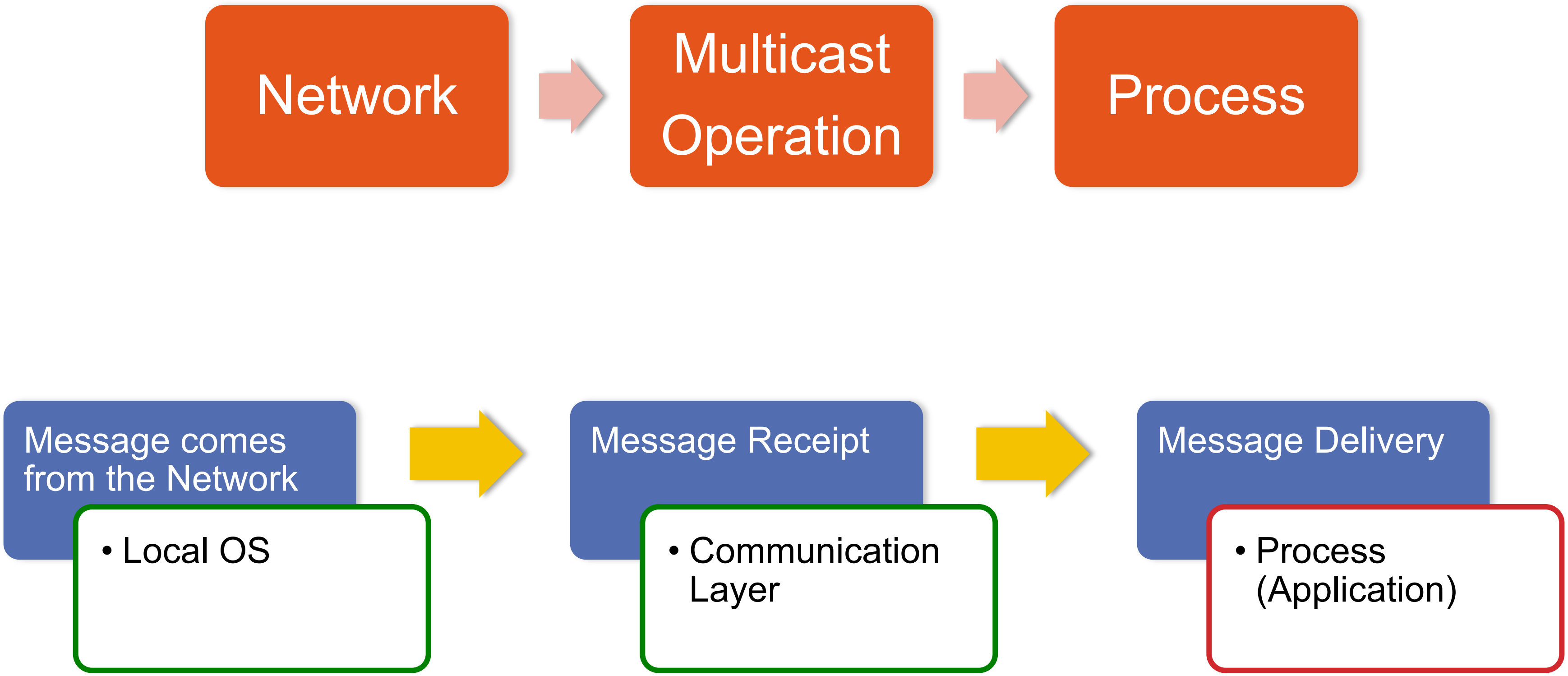
Primitives and implementation

- Use a reliable one-to-one `send()` operation
  - `B-multicast(g, m)`: for each process `p` ∈ `g`, `send(p, m)`
  - `B-deliver(m)` at `p`: when `receive(m)` at `p`, for all `p`

`ack`-Implosion:

- Problem in using concurrent `send(p, m)` operations
  - All recipients acknowledge receipt at about same time
  - Buffer overflow leads to dropping of `ack` messages
  - Retransmits, even more `ack` messages

Multicast: Layered Architecture



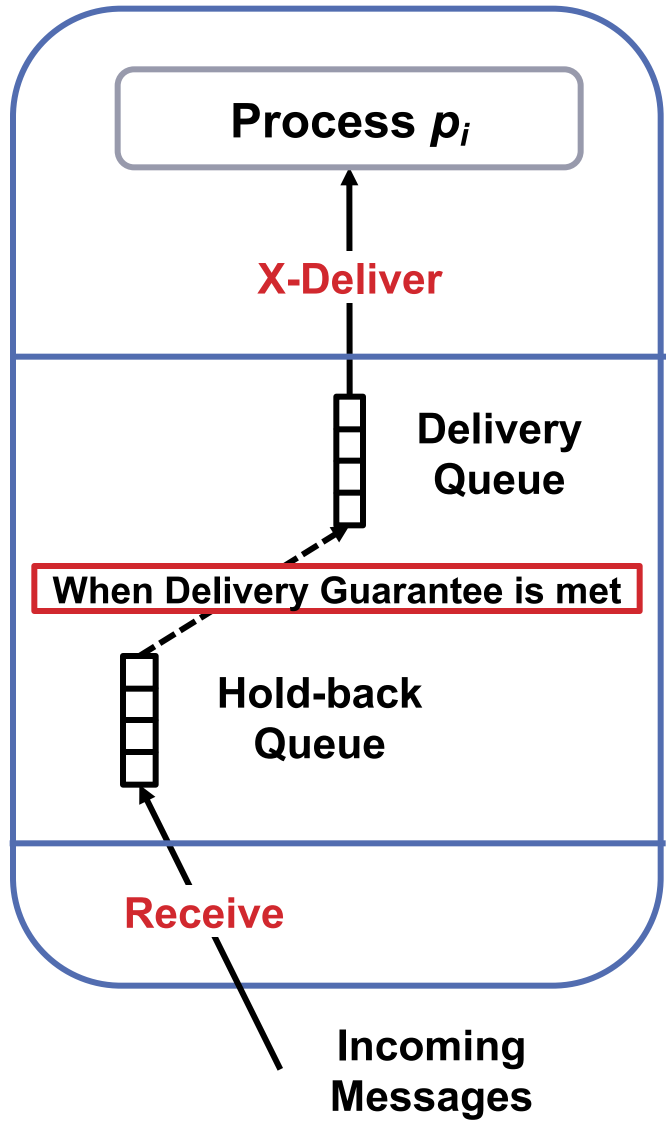
X-Multicast

X-multicast(g, m)

X-deliver(m)

X is

- B: basic
- R: reliable
- FO: FIFO order
- CO: causal order
- TO: total order



Reliable Multicast

Primitives

- `R-multicast(m, g)`
- `R-deliver(m)`

Desired properties

- Integrity
  - A correct process `p` delivers a message at most once
  - Safety: a delivered message is identical to the message sent in the multicast send operation
- Validity
  - If a correct process multicasts message `m`, then it will eventually deliver `m` (liveness)
- Agreement
  - If a correct process delivers a message `m`, then all other correct processes in the target group of message `m` will also deliver message m

Note

- Validity and agreement ensure overall liveness: if one process delivers a message `m`, then `m` will eventually be delivered to all group members

Reliable Multicast

Properties

- Validity: a correct process will eventually B-deliver to itself
- Integrity: based on underlying communication medium
- Agreement: B-multicast to all other processes after B-deliver
- Inefficient: each message is sent |g| times to each process

On initialization

```
Received := {};
```

For process p to R-multicast message m to group g

```
B-multicast(g,m); (p∈g is included as destination)
```

On B-deliver(m) at process q with g = group(m)

```
if (m ∉ Received):   Integrity
    Received := Received ∪ {m};
    if (q ≠ p):
        B-multicast(g,m);   Agreement
    R-deliver(m)   Integrity, Validity
```

Idea: q sends m to other working nodes before it delivers m to process

Reliable Multicast over IP Multicast

R-IP-multicast

- Observation: multicast successful in most cases
- No separate acknowledgement messages
  - Piggyback acknowledgements on sent messages
  - Use negative acknowledgement (absence of a message) to indicate non-delivery

Basic idea

- Assumption: closed multicast groups
- S_g^p: sequence number for group g that process p belongs to (number for the next message to be sent)
- R_g^q: sequence number of latest message that a process has delivered from process q and that was sent to group g (last message delivered from q)

Hold-back Queue for Arriving Multicast Messages

Basic algorithm

- p R-multicasts message to group g and piggybacks onto message
    - S_g^p
    - Acknowledgements <q, R_g^q> for all q
  - IP-multicast message with its piggybacked values to g
  - Increment S_g^p by one
- R-deliver message from p:
    - Only if received sequence number $S = R_g^p+1$
    - Then increment R_g^p by 1
    - Retain any message that cannot yet be delivered in *hold-back-queue*

Reliable Multicast over IP Multicast

Basic idea

- R-deliver message from p
  - If $S \leq R_g^p$, then message is already delivered, discard
  - If $S > R_g^p + 1$ or $R > R_g^p$ for any enclosed acknowledgement <q, R>
    - Then receiver has missed one or more messages
    - Request retransmit through negative acknowledgement

Integrity

- Follows from detection of duplicates and properties of IP multicast (e.g., checksum to detect message corruption)

Validity

- Message loss can only be detected when a successor message is eventually transmitted
- Requires processes to multicast messages indefinitely

Agreement

- Requires unbounded history for multicast messages so that retransmit is always possible

Note

- There exist practical variants that ensure validity and agreement

Ordered Multicast

Example

- Order ignored in multicasting until now
- Bulletin-boards, chat rooms, instant messaging

Assumption

- Every process belongs to at most one group

FIFO ordering

- If a correct process issues a multicast(g, m) and then multicast(g, m’), then every correct process that delivers m’ will deliver m before m’

Causal ordering

- If multicast(g, m) → multicast(g, m’), where → is induced by message passing only, then every correct process that delivers m’ will deliver m before m’

Ordered Multicast II

Total ordering

- If a correct process delivers m before it delivers m’, then any other correct process that delivers m’ will deliver m before m’

Notes

- Causal ordering implies FIFO ordering
- FIFO ordering and causal ordering are partial (pre-)orders
- Total order allows arbitrary ordering of deliver events relative to multicast events, as long as this order is identical in all correct processes
- Atomic multicast: reliable, totally ordered multicast

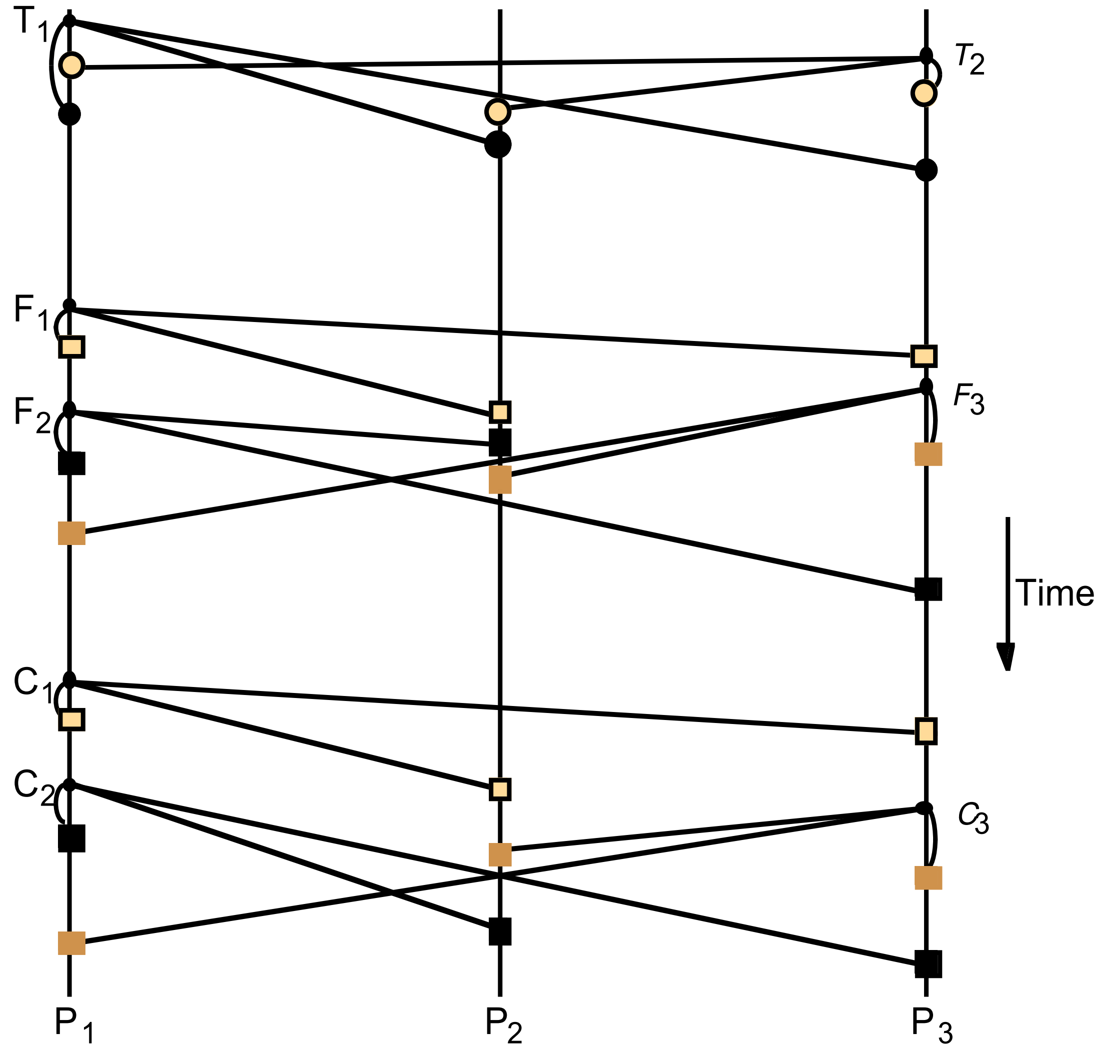
Total, FIFO and Causal Ordering

The consistent ordering of totally ordered messages T_1 and T_2

The FIFO-related messages F_1 and F_2

The causally related messages C_1 and C_3

The otherwise arbitrary delivery ordering of messages



Display from a Bulletin Board Program

- FIFO: Every message from same sender should be ordered
- Causal: Responses should appear after the posting
- Total: All the left-hand numbers should be the same on all users

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		

How Do We Achieve Ordering?

FIFO

- Use sequence numbers and delay delivery until this number is reached

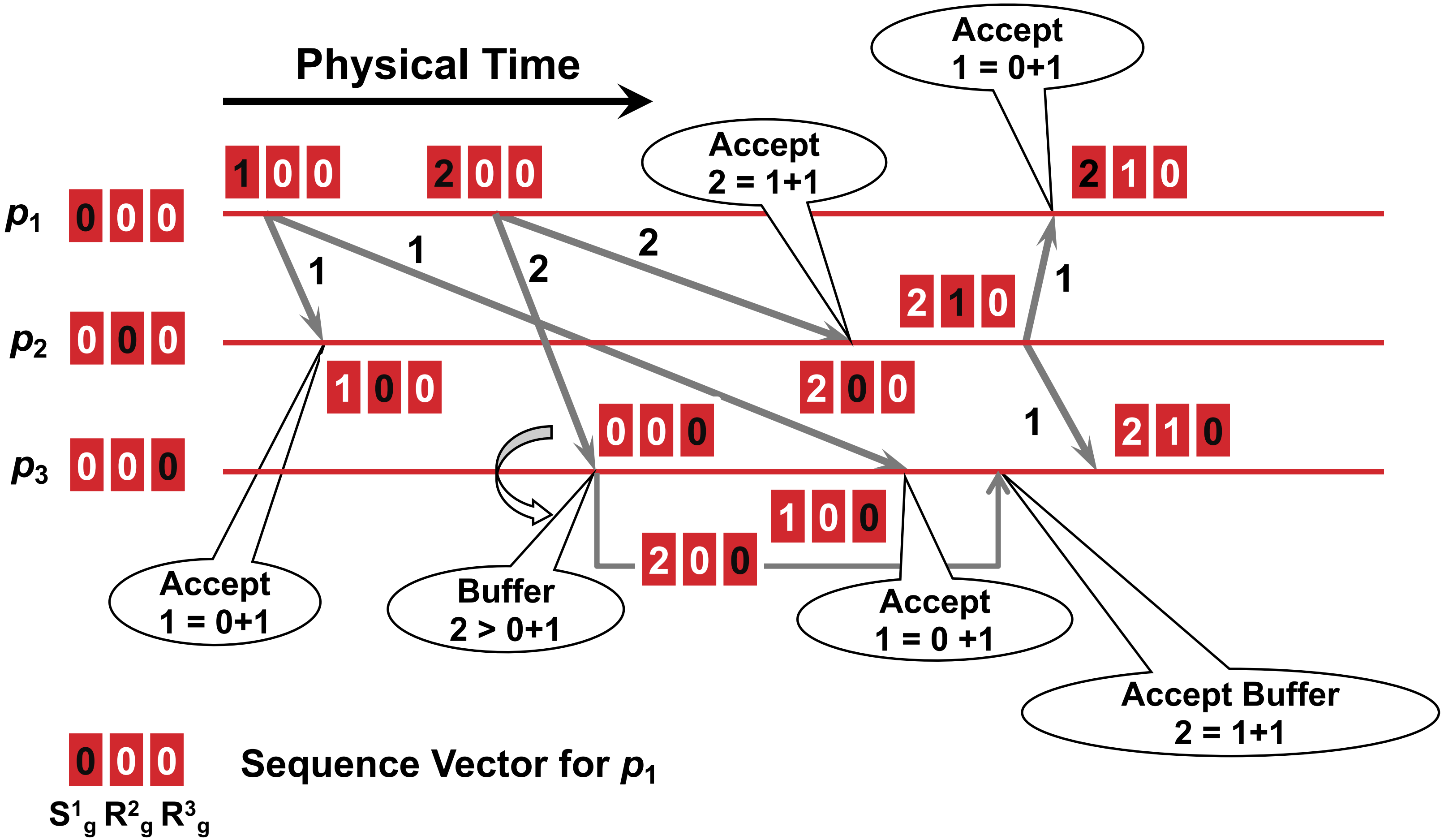
For total ordering we need non-process specific sequence numbers

Ordered Multicast

Implementing FIFO ordering

- S_g^p: sequence number for group g that process p belongs to g
- R_g^p: sequence number of latest message that a process has delivered g from process p and that was sent to group g
- *Assumption: non-overlapping groups*
- FO-multicast(m, g)
  - Increment S_g^p by 1
  - B-multicast(m, g, $<S_g^p>$)
- Receipt of a message from q with sequence number S
  - If $S = R_g^p+1$, then this is the next message,
    - Therefore FO-deliver(m)
    - $R_g^p := S$
- If $S > R_g^p+1$, then
  - Place message on hold-back queue until intervening messages have been delivered and $S = R_g^p+1$

Example: FIFO Multicast



Source: Klara Nahrstedt

Ordered Multicast

Implementing total ordering

- Idea: assign totally ordered identifiers to multicast messages so that every process makes the same delivery decision based on these identifiers
- Delivery similar to FIFO delivery, only that group-specific sequence numbers rather than process-specific sequence numbers are used
- Assumption: non-overlapping groups
- Two main methods for the assignment of identifiers
  - Sequencer
  - Collective agreement on the assignment of message identifiers

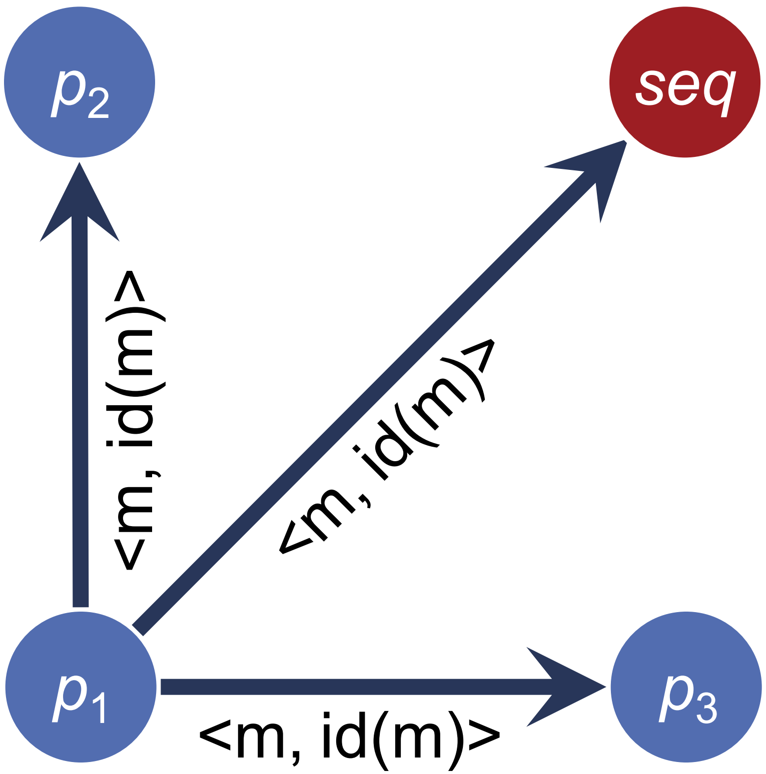
Ordered Multicast

Implementing total ordering

- Sequencer
- Process wishing to TO-multicast attaches a unique identifier ID(m) to the message
- Message is sent to sequencer as well as all members of g
- Sequencer maintains group-specific sequence number s_g which it uses to assign increasing and consecutive sequence numbers to the messages it B-delivers
- Announces the order in which members of g have to deliver these messages using a B-multicast order message

Total Ordering Using a Sequencer

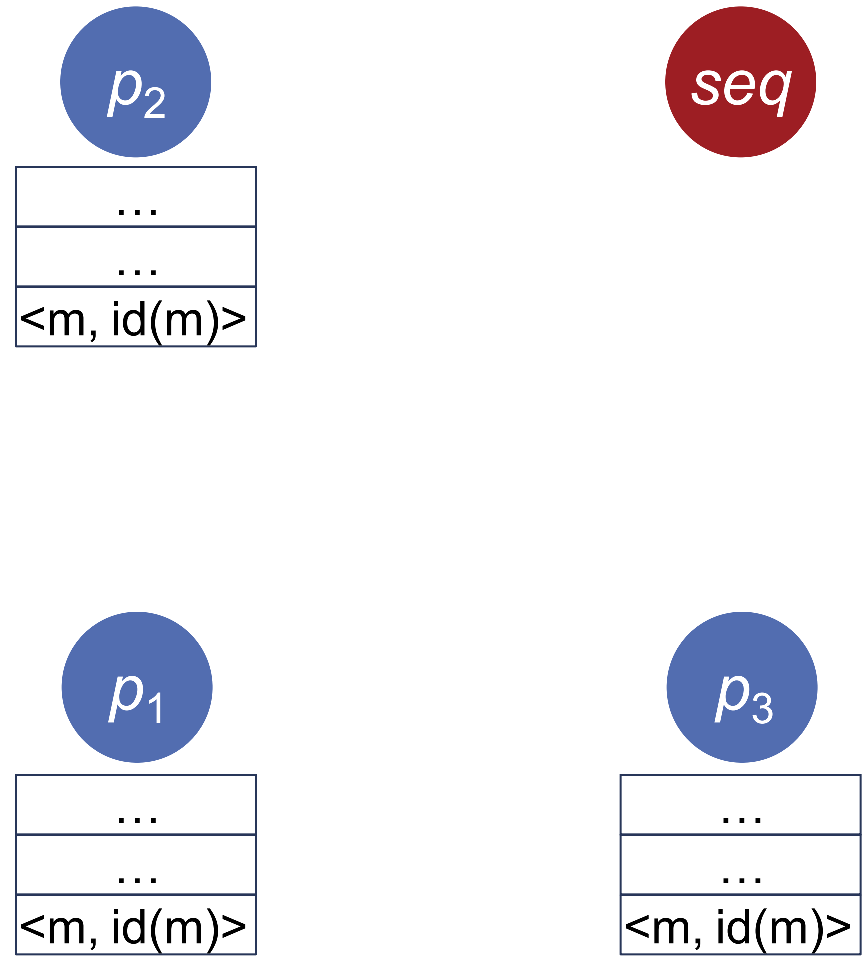
- To TO-multicast a message m to a group g, p_1 attaches a unique identifier id(m) to it
- The messages for g are sent to the sequencer for g and to the members of g
- The sequencer may be a member of g



Total Ordering Using a Sequencer

On B-deliver(<m, id(m)>)

- A process (but NOT the sequencer) places the message <m, id(m)> in its hold-back queue



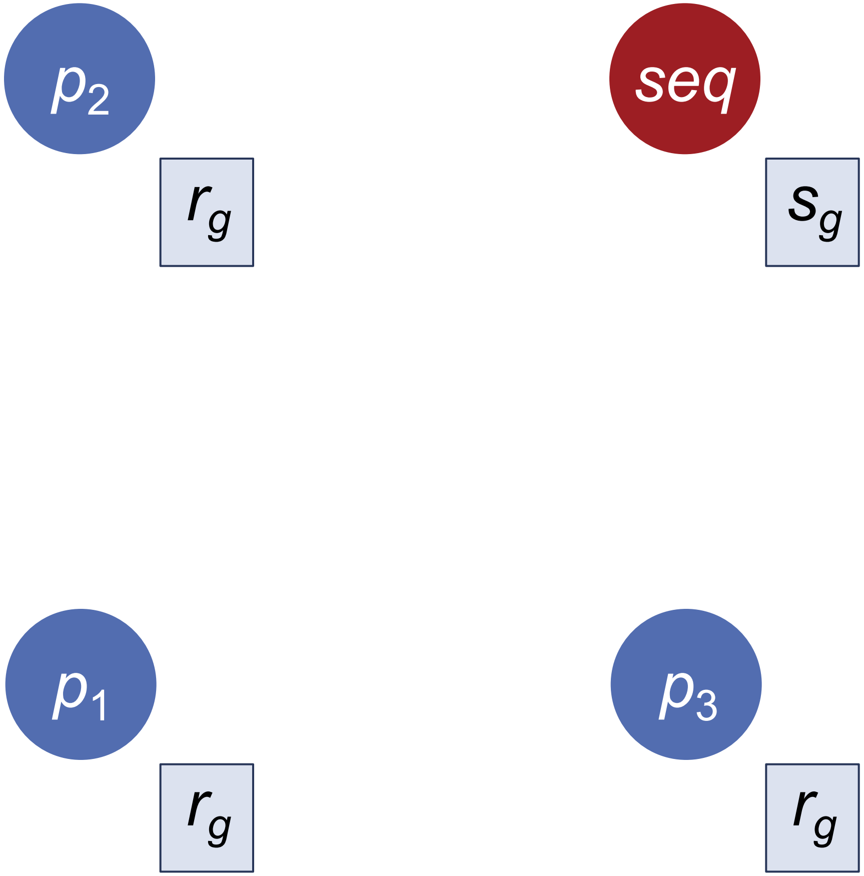
Total Ordering Using a Sequencer

Sequencer

- Maintains a group-specific sequence number s_g
- Assigns increasing and consecutive sequence numbers to the messages that it B-delivers

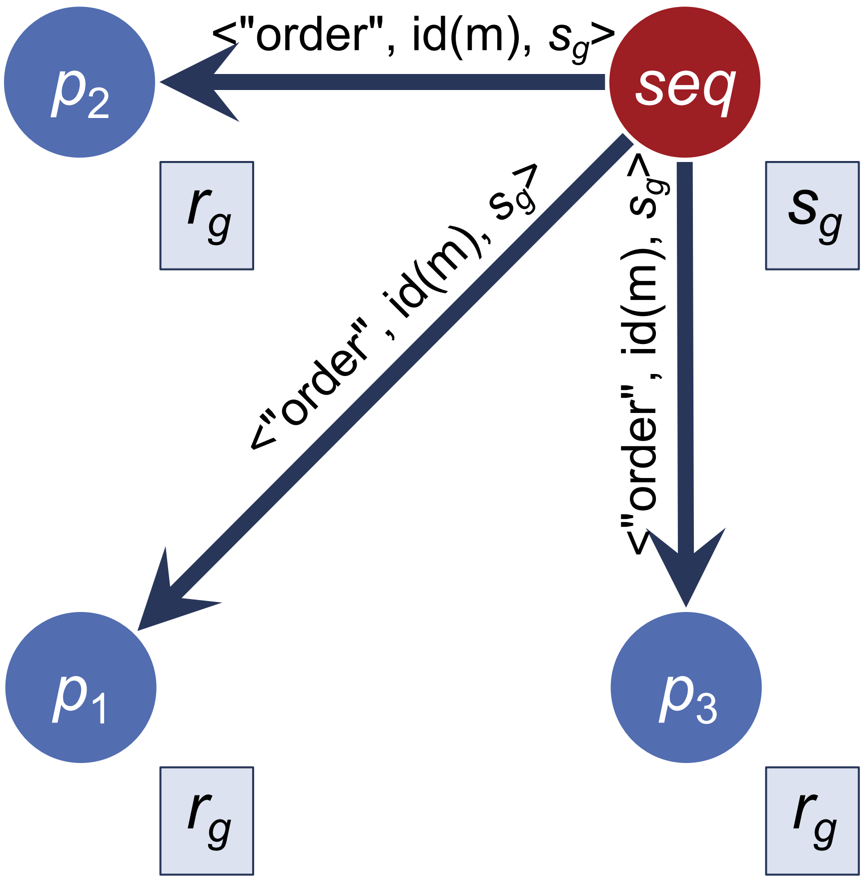
Processes

- Have their local group-specific sequence number r_g



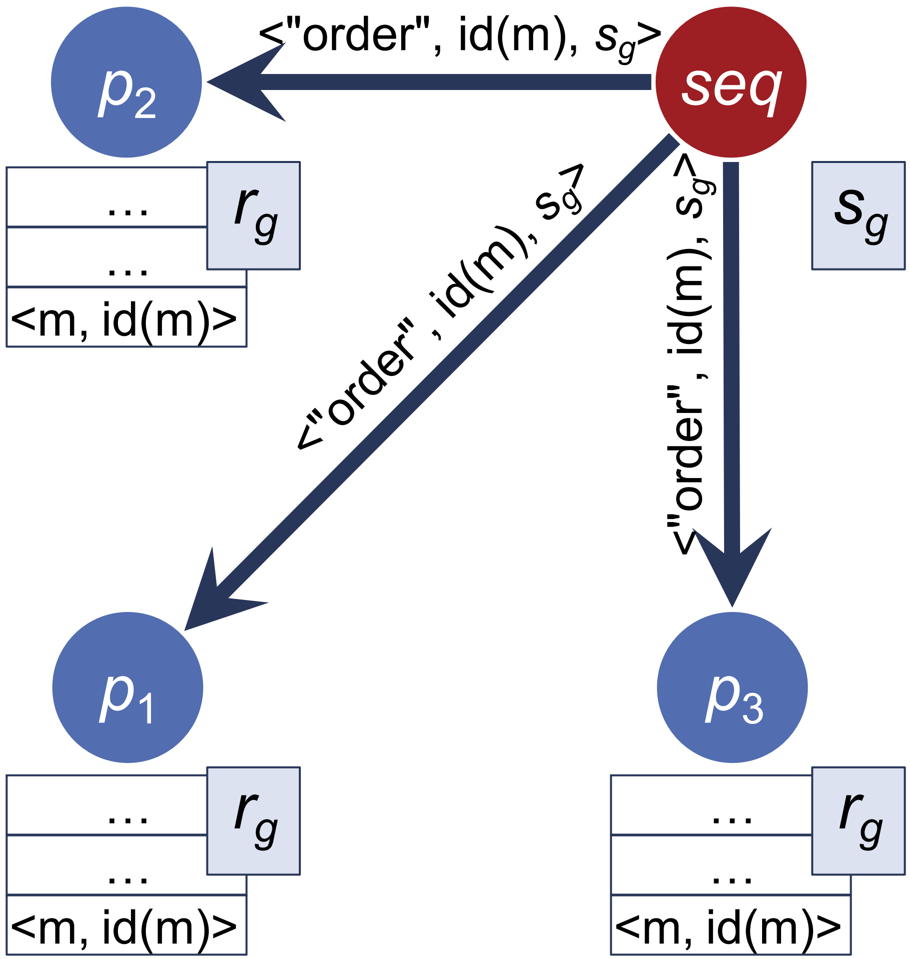
Total Ordering Using a Sequencer

- On B-deliver(<m, id(m)>) the sequencer announces the sequence numbers
- B-multicasting “order” messages to g.



Total Ordering Using a Sequencer

- A message will remain in a hold-back queue until it can be TO-delivered
- To-delivery: if the corresponding sequence number $s_g = r_g$



Total Ordering Using a Sequencer

Algorithm for group member p

On initialization: $r_g := 0$;

To TO-multicast message m to group g
B-multicast$(g \cup \{sequencer(g)\}, \langle m, i\rangle)$;

On B-deliver$(\langle m, i\rangle)$ *with* $g = group(m)$
Place $\langle m, i\rangle$ in hold-back queue;

On B-deliver$(m_{order} = \langle\text{"order"}, i, S\rangle)$ *with* $g = group(m_{order})$
wait until $\langle m, i\rangle$ in hold-back queue and $S = r_g$; **Ensures total ordering**
TO-deliver m; // (after deleting it from the hold-back queue)
$r_g = S + 1$;

Algorithm for sequencer of g

On initialization: $s_g := 0$;

On B-deliver$(\langle m, i\rangle)$ *with* $g = group(m)$
B-multicast$(g, \langle\text{"order"}, i, s_g\rangle)$;
$s_g := s_g + 1$;

Total Ordering Using Distributed Agreement

Sequencer-based approach

- Sequencer may become a bottleneck
- Sequencer is a critical point of failure
- There are algorithms to address this problem

Approach without a sequencer

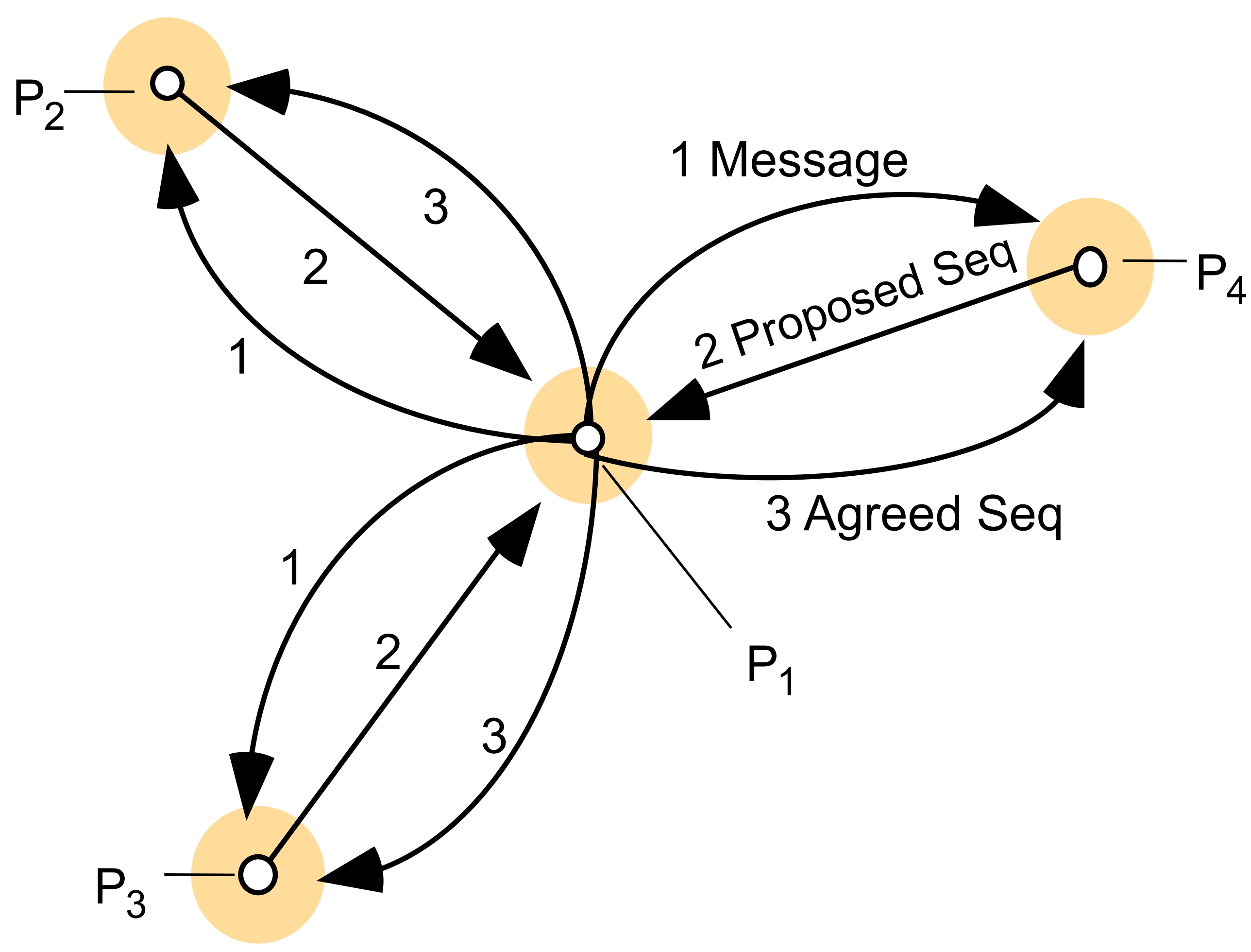
- Processes collectively agree on the assignment of sequence numbers to messages in a distributed fashion
- Receiving processes bounce proposed sequence numbers to sender
- Sender returns agreed sequence numbers
- Each process *q* in group *g* maintains
  - A_g^q: the largest agreed sequence number it has observed so far for group g
  - P_g^q: own largest proposed sequence number

Total Ordering Using Distributed Agreement

Collective agreement on message identifiers

- Each process *q* in group *g* keeps A_g^q, its largest observed agreed sequence number, and P_g^q, its own largest proposed number
- *p* B-multicasts <*m*, *i*> to *g*, where *i* is unique identifier for *m*
- Each recipient *q* replies to *g* with proposed agreed sequence number
  - $P_g^q := \max(A_g^q, P_g^q) + 1$
  - Each process *q* provisionally assigns own proposed sequence number to message and queues message in hold back queue, ordered according to proposed sequence number
- *p* chooses largest proposed number as sequence number *a*
- *p* B-multicasts <*i*, *a*> to *g*
- Each process *q* in group
  - Sets $A_g^q := \max(A_g^q, a)$
  - Reorders received message in hold-back queue if received sequence number differs from proposed number
  - Only when message at head of hold-back queue is assigned an agreed sequence number, it will be queued in delivery queue

The ISIS algorithm for total ordering



Implementing Causal Ordering

Causal ordering

- Takes into account of the happened-before relationship only established by multicast messages

Idea

- Each process *p* maintain its own vector timestamp for the group *g*
- |g| entries count the number of multicast messages from each process that happened-before the next message to be multicast
- $V^g_i[j]$ counts the number of group *g* messages from p_j to p_i

Review Vector Timestamps

- If process p_i receives a message $<m, V^g_j>$ from process p_j, then
  - $V^g_i[k] = \max(V^g_i[\mathrm{k}], V^g_j[k])$ if $k \neq i$
  - $V^g_i[k] = V^g_i[k] + 1$ if $k = i$
- Remember $V(a) < V(b)$ iff event *a* happens before event *b*

Causal Ordering Using Vector Timestamps

Algorithm for group member p_i $(i = 1, 2\ldots, N)$

On initialization

$V_i^g[j] := 0$ $(j = 1, 2\ldots, N)$;

The number of messages in group g from process p_j that have been seen at process p_i so far

To CO-multicast message m to group g

$V_i^g[i] := V_i^g[i] + 1;$

B-multicast$(g, <V_i^g, m>)$;

p_i has delivered any message that p_j had delivered

On B-deliver$(<V_j^g, m>)$ *from* p_j, *with* $g = group(m)$

place $<V_j^g, m>$ in hold-back queue;

wait until $\boxed{V_j^g[j] = V_i^g[j] + 1}$ and $\boxed{V_j^g[k] \le V_i^g[k]\ (k \ne j);}$

CO-deliver m; // after removing it from the hold-back queue

$V_i^g[j] := V_i^g[j] + 1$;

p_i has delivered any earlier message sent by p_j

Example: $V^g{}_2$ = [3,6,2] from p_2 is received by p_3 with $V^g{}_3$ = [2,5,2], i.e., p_3 needs to deliver a message from p_1 first

Overlapping Groups

We assumed that the multicast groups do not overlap

- Obvious solution: multicast to all but this is not practical
- More details in the textbook …

Consensus

Agreement on a value or action to be taken

- Mutual exclusion: processes agree on the process that enters critical section
- Election: processes agree on elected process
- Totally ordered multicast: processes agree on the order of message delivery
- Banking: processes agree on whether or not to perform a transaction such as debit or credit

Byzantine process failures

- Processes fail, but may still respond with arbitrary, erratic behavior
- Examples: software bugs or malicious attacks

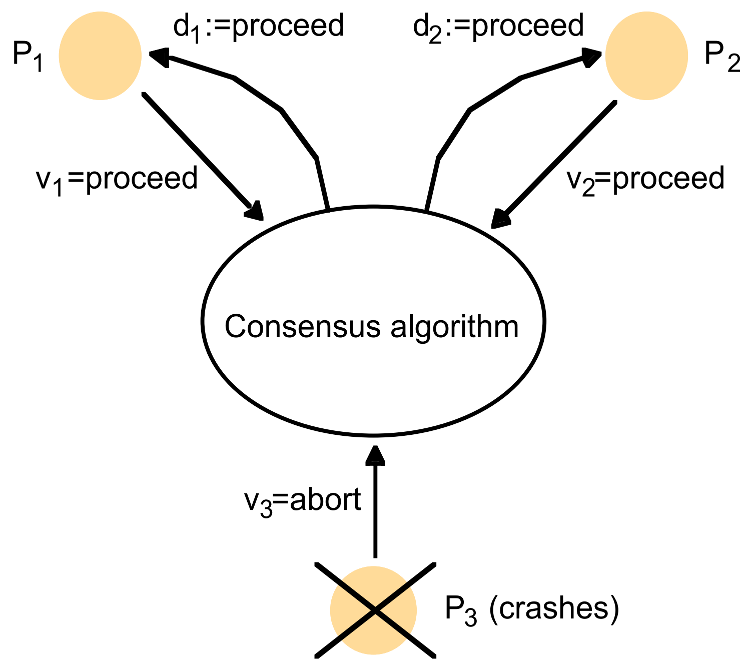
Digital signatures

- During an agreement algorithm a faulty process cannot make a false claim about the values that a correct process has sent to it

Consensus for three processes

Agreement in the value of a decision variable amongst all correct processes

- p_i is in state undecided and proposes a single value v_i drawn from a set D
- Processes communicate with each other to exchange values
- p_i sets decision variable d_i and enters the decided state after which the value of d_i remains unchanged



Consensus Problem: Properties

Termination

- Eventually each process sets its decision variable

Agreement

- The decision value is the same on all processes
- For all correct p_i and p_j holds $d_i = d_j$

Integrity

- If the correct processes all proposed the same value, then any correct process in the decided state has chosen that value

Consensus: Majority

Majority algorithm

- Solve consensus in a *failure-free* environment
- Each process *reliably multicasts* proposed values
- After receiving response, solves consensus function
  - `majority(`v_1`, ...,` v_N`)`
  - Returns most often proposed value or undefined (⊥) if there is no majority

Properties

- Termination guaranteed by reliability of multicast
- Agreement, integrity: definition of majority, and integrity of reliable multicast (all processes apply the same function to the same data)

Other functions

- Minimum or maximum instead of majority

Byzantine Generals

Problem

- Three or more generals are to agree on an attack or retreat
- The commander issues order and the lieutenants have to decide to attack or retreat
- One of the generals may be treacherous

Treacherous commander

- Proposes attack to one general and retreat to the other

Treacherous lieutenants

- Tell one of their peers that commander ordered to attack, and others that commander ordered to retreat

Difference to consensus problem

- One process supplies a value that others have to agree on (other processes cannot propose a value)

Integrity

- *If* the commander is correct, then all correct processes decide on the value proposed by the commander (but the commander may not be correct)

Interactive Consistency

Goal

- All correct processes agree on a *vector of values*, each component corresponding to one process' agreed value
- Example: agreement about each process' local state

Requirements

- Termination: eventually each correct process sets it decision variable
- Agreement: the decision vector of all correct processes is the same
- Integrity: if p_i is correct, then all correct processes decide on v_i as the i-th component of their vector

Relationship of Consensus to Other Problems

Decision variables

- $C_i(v_1, ..., v_N)$ returns the decision value of p_i as a solution to the consensus problem, where $v_1, ..., v_N$ are the proposed values of the processes
- $BG_i(j, v)$ returns the decision value of p_i where p_j is the commander proposing value v
- $IC_i(v_1, ..., v_N)[j]$ returns the j-th value in the decision vector of p_i where $v_1, \ldots, v_N$ are the proposed values of the processes

Relationship of Consensus to Other Problems

BG → IC

- Run BG N times, once with each p_i acting as commander
  $IC_i\,(v_1, ..., v_N)[j] = BG_i(j, v_j)$

IC → C

- Run IC to produce a vector of values at each process
- Apply an appropriate function on the vector's values to derive a single value $C_i(v_1, ..., v_N) = \text{majority}(IC_i(v_1, ..., v_N)[1], ..., IC_i(v_1, ..., v_N)[N])$

C → BG

- Commander p_j sends its proposed value v to itself and of the remaining processes
- All processes run C with the values $v_1, ..., v_N$ that they receive
- $BG_i(j, v) = C_i(v_1, ..., v_N)$

Consensus from RTO Multicast

Idea

- Implementing consensus through a reliable and totally ordered multicast operation RTO-multicast

Algorithm

- Assumption: all processes form a group
- p_i performs RTO-multicast (v_i, g)
- p_i sets $d_i = m_i$, where m_i is the first value that p_i RTO-delivers

Properties

- Termination guaranteed by reliable multicast
- Agreement and validity because reliability and total ordering of multicast delivery

RTO multicast from consensus

- Chandra and Toueg [1996]

Consensus in Synchronous system: Multicast

Assumption

- No more than f of the N processes crash

Algorithm proceeds in in f+1 rounds

- Processes B-multicast values between them
- At the end of f+1 rounds, all surviving processes agree

Algorithm for process $p_i \in g$; algorithm proceeds in $f + 1$ rounds

On initialization
$Values_i^1 := \{v_i\}$; $Values_i^0 = \{\}$;

In round r $(1 \le r \le f + 1)$
B-multicast$(g,\ Values_i^r - Values_i^{r-1})$; // Send only values that have not been sent
$Values_i^{r+1} := Values_i^r$;
while (in round r)
{
On B-deliver(V_j) *from some* p_j
$Values_i^{r+1} := Values_i^{r+1} \cup V_j$;
}

After $(f + 1)$ *rounds*
Assign $d_i = minimum(Values_i^{f+1})$;

Consensus in Synchronous system: Multicast

Dolev-Strong algorithm

- $Values_i^r$: set of proposed values known to process p_i before round r
- Every process multicasts the set of values it has not sent in previous rounds
- Every process takes delivery of values from other processes
- At the end of f+1 rounds: each process choses minimum value

Algorithm for process $p_i \in g$; algorithm proceeds in $f+1$ rounds

On initialization
$Values_i^1 := \{v_i\}$; $Values_i^0 = \{\}$;

In round r $(1 \le r \le f+1)$
B-multicast(g, $Values_i^r - Values_i^{r-1}$); // Send only values that have not been sent
$Values_i^{r+1} := Values_i^r$;
while (in round r)
{
On B-deliver(V_j) *from some* p_j
$Values_i^{r+1} := Values_i^{r+1} \cup V_j$;
}

After $(f+1)$ *rounds*
Assign $d_i = minimum(Values_i^{f+1})$;

Dolev-Strong Algorithm

Correctness

- Is every process going to arrive at the same set of values?
- If yes: integrity and agreement will follow, since processes consistently apply the minimum function to this set

Proof sketch

- Assume two processes differ in their final set of values (with f crashes)
- Thus: some correct process p_i has a value v that another correct process p_j ($i \neq j$) does not have
- This means that some other process p_m, which sent v to p_i crashed before v could be delivered to p_j
- In turn: any process sending v in the previous round must have crashed, i.e., at least one crash per round
- We have f+1 rounds, at most f crashes … hence contradiction!

General property in synchronous systems

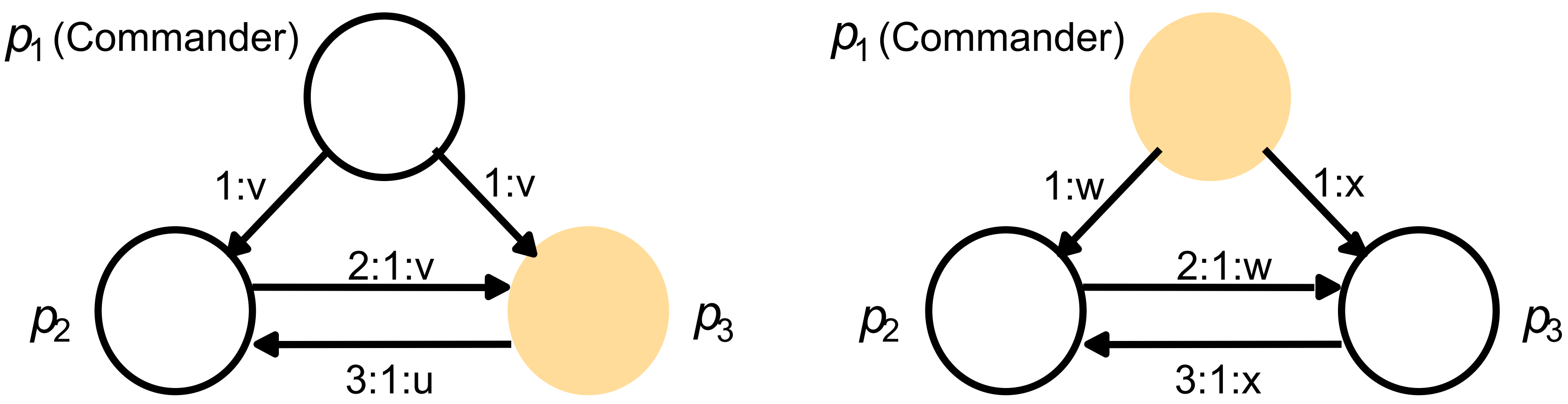
- Any algorithm to reach consensus, tolerating up to f crashes or byzantine failures, requires at least f+1 rounds

Impossibility of three Byzantine Generals

Impossibility for N = 3 and f =1

- '3:1:u' is the message '3 says (1 says u)'
- Both scenarios show two rounds of messages
- Left: p_2 knows is that it has received two different values (p_3 is faulty)
- Right: same situation, but commander is faulty
- Assume a solutions exists: p_2 decides on w, p_3 on x. Contradiction!

Impossibility for N ≤ 3f



Faulty processes are shown coloured

Solution For $N \geq 4$ Byzantine Generals

Correct generals reach agreement in two rounds

- First, commander sends value to each lieutenant
- Second, each lieutenant sends value it received to all peers
- Lieutenant receives
  - Value from commander
  - *N*-2 values from peers

Commander is faulty

- All lieutenants correct: each will gather exactly the set of values the commander sent out

1 lieutenant is faulty

- each of its peers receives *N*-2 copies of the value the commander sent out, plus the faulty lieutenant value

Solution For 4 Byzantine Generals

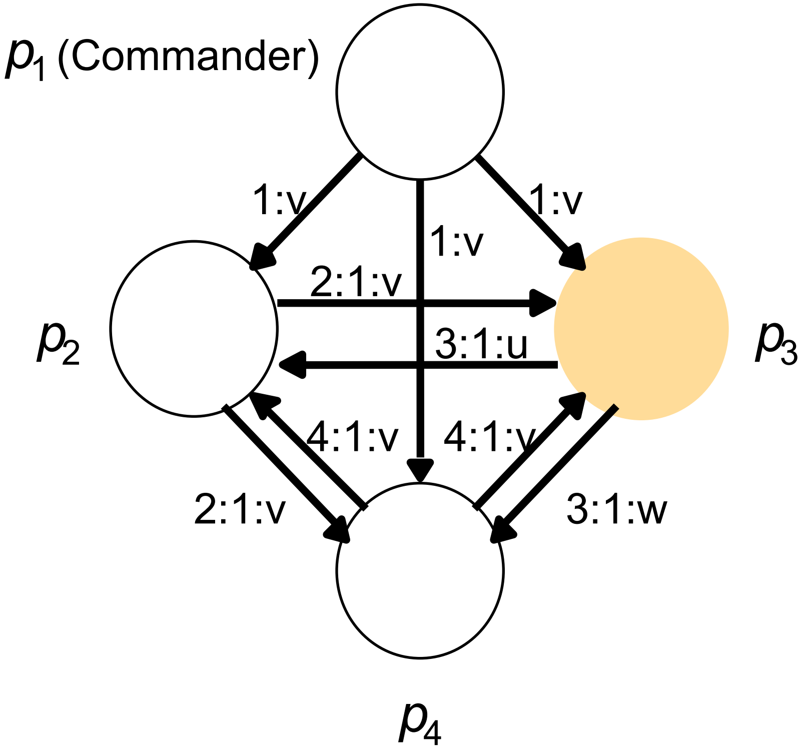
Reaching agreement

- Simple majority function
- Commander is correct
  - Since $N \geq 4$, N-2 $\geq$ 2, majority function will ignore value of faulty lieutenant
  - Majority function produces value of commander
- Commander is incorrect
  - There is no majority and the majority function will produce $\perp$
  - Note: BG requires agreement only if commander correct

Solution For 4 Byzantine Generals

p_2: majority({v,u,v}) = v

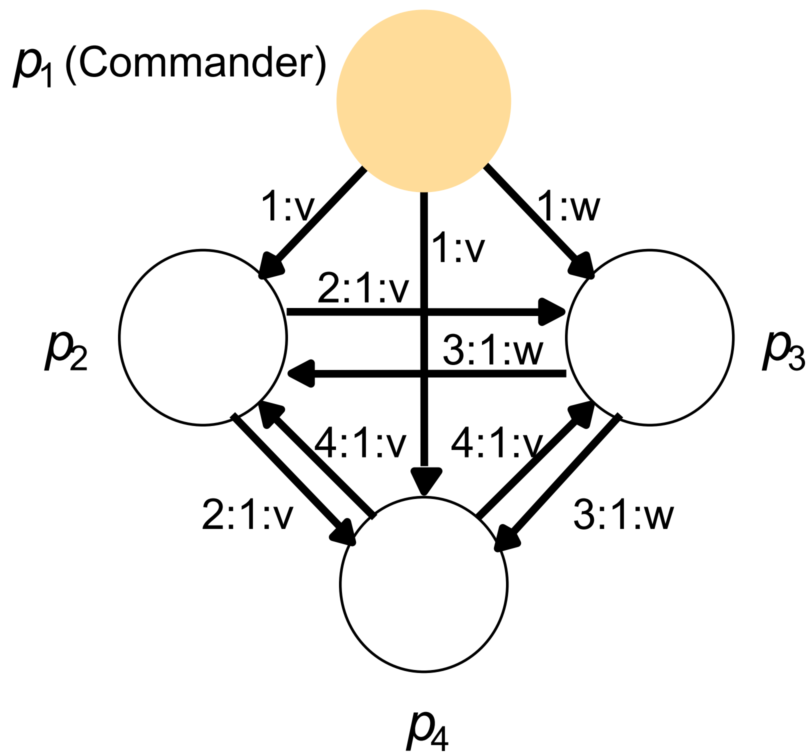
p_4: majority({v,v,w}) = v



Faulty processes are shown coloured

Solution For 4 Byzantine Generals

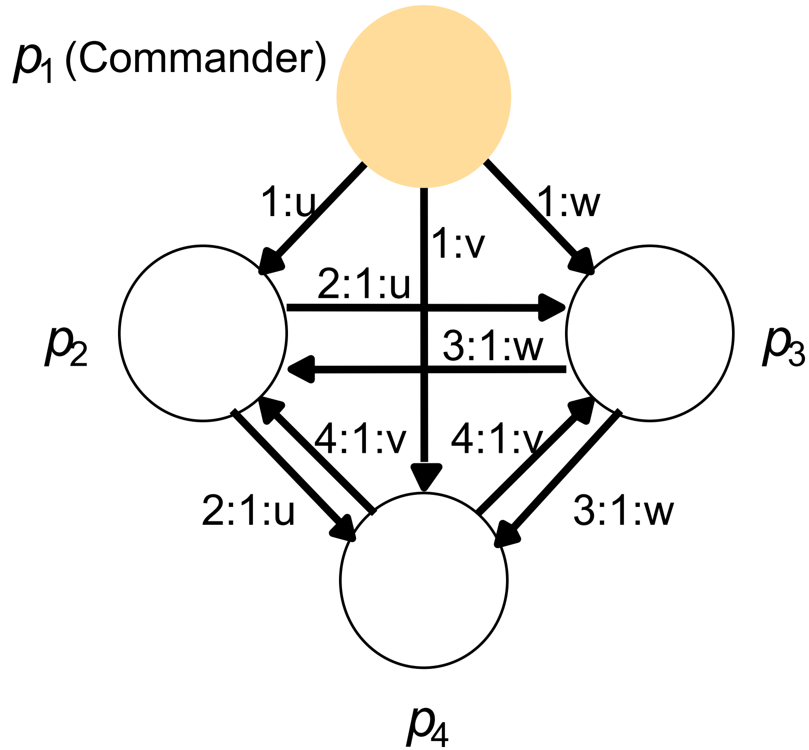
p_2: majority({v,w,v}) = v
p_3: majority({v,v,w}) = v
p_4: majority({w,v,v}) = v



Faulty processes are shown coloured

Solution For 4 Byzantine Generals

p_2, p_3, p_4:
majority({u,v,w}) = ⊥



Faulty processes are shown coloured

Impossibility in Asynchronous Systems

Previous algorithms: synchronous systems

- Message exchanges in rounds
- Timeouts

No algorithm can guarantee consensus

- Even with a single process crash failure (Fischer, Lynch and Paterson, 1985)
- Proof idea: show that there is always some continuation of the process's execution that avoids consensus being reached
- But: in practice consensus can often be reached, but a small probability remains that consensus cannot be reached

Asynchronous Systems …

Reaching consensus by weakening system assumptions

- *Masking faults*
  - Design system so that failures appear like intermittent slowdown in processing of messages
    - Store system state on persistent storage before crash
    - Restart system in that state after recovery
- *Modified failure detectors*
  - Deem process that has not responded as failed
  - Treat this process as fail-safe, i.e., discard any subsequent messages from this process
  - Problems
    - Long timeouts necessary
    - False negatives possible that reduce effectiveness of system
  - Turn an asynchronous system into a synchronous system

Asynchronous Systems …

Consensus using randomization

- Adversary: interferes with the processes' attempts to reach consensus
  - Manipulates the network to delay messages so that they arrive at just the wrong time,
  - Slows down or speeds up the processes just so that they are in the 'wrong' state when they receive a message
- Use element of chance
  - Adversary cannot exercise its counter-strategy
  - Consensus might still not be reached in some cases
  - But: processes can reach consensus in a finite expected time