Cristina Nita-Rotaru

7610: Distributed Systems

Leader election. Membership. Reliable Multicast. Virtual Synchrony

Required reading for *leader election*…

- An improved algorithm for decentralized extrema-finding in circular elections of processes, E. Chang and R. Roberts, Communications of the ACM, 1979.
- Elections in a distributed computing system, H. Garcia-Molina, IEEE Transactions on Computers, 1982.

1: Leader Election: Ring Algorithm

Leader Election

- Algorithm to select a single process as the coordinator of some task distributed among several processes
- Correctness: When the election algorithm terminates a single process has been selected and every process knows its identity
  - Safety: any process selects as leader the non-faulty process with the best attribute value (usually highest id) or no leader is selected
  - Liveness: any instance of the election algorithm terminates and any non-faulty process has selected a leader

Leader Election - Challenges

- Nodes do not know apriori who is the leader
- Any process can start an election
- Processes communicate through messages, messages can be lost, delayed, network can be partitioned
- Processes can crash, new leader needed
- Previously crashed process recovers may need new election
- Processes can crash during leader election
- All nodes must agree on when election is over and who the new coordinator is

Leader Election Algorithms - Model

- Each process has a unique number
- One process per machine
- Processes know each other's process number
- Processes do not know the status of the other processes, i.e. up or down
- Different network topologies, different algorithms for different topologies
- Goal : In general, the process with the highest ID number will be the new coordinator.

Ring Algorithm

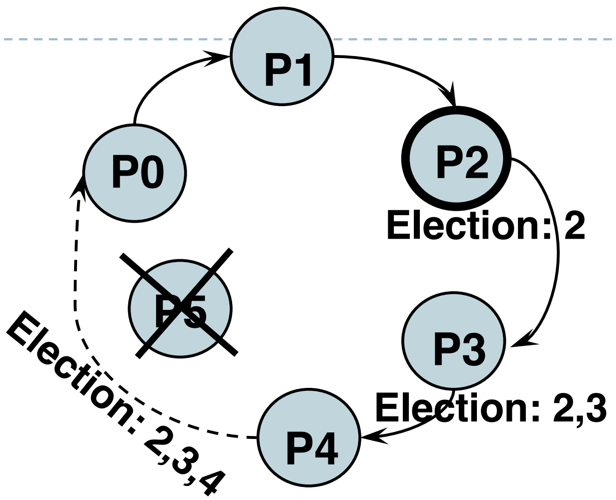
- Assumes that the processes are arranged in a logical ring and each process knows the order of the ring of processes (unidirectional).
  - All messages are sent clockwise around the ring.
- Faulty processes are those that don’t respond in a fixed amount of time.
- Even if two ELECTIONS started at once, everyone will pick same leader since the node with highest identifier is picked.
- Messages go around the ring till they return to the initiator.

Ring Algorithm

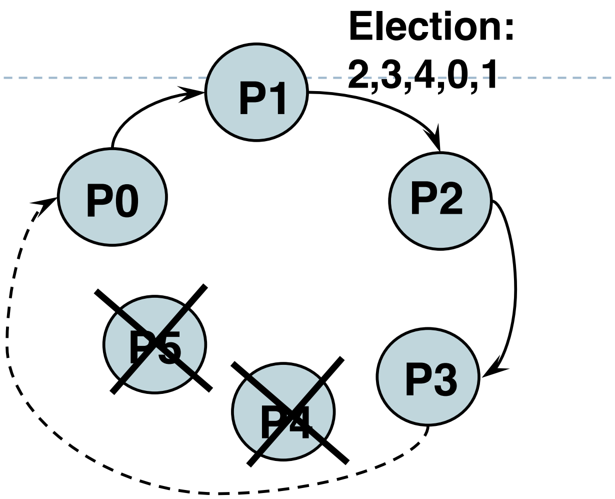
- When a process notices that current leader failed:
  - Sends an ELECTION message to start the algorithm to its successor; It contains its own id.
    - □ (if successor is down, sender skips until a running process is located).
- When a process receives an ELECTION message:
  - process adds its own id to the list and sends to successor.
- When ELECTION message gets back to the initiator (process recognizes the message that contains its own id):
  - Sends a LEADER message that announces the new leader and contains: id of new leader (list member with highest number); List of the members of the new ring. Message circulates around the ring.
- When the LEADER message gets back to initiator:
  - Election is over if id of new leader is in ring id-list.
  - Else the algorithm is repeated (handles election failure).

Example: Ring Election

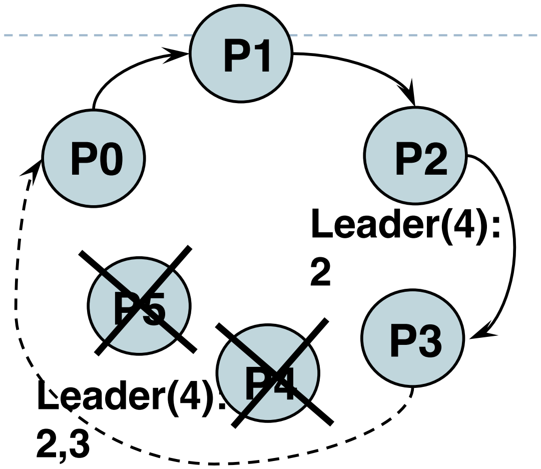
Example from 425, Prof. Klara Nahrstedt



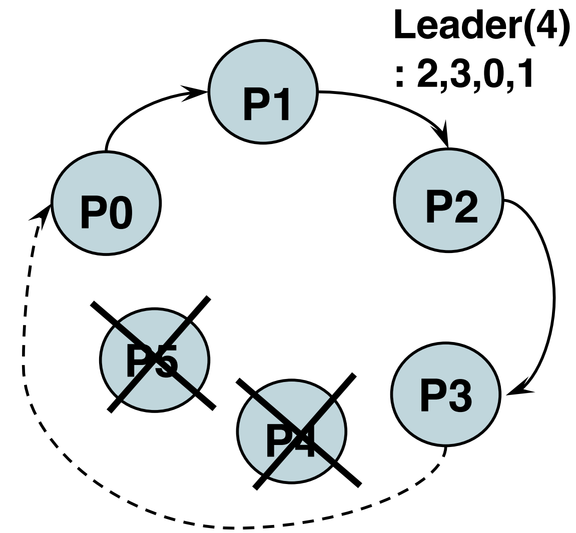
1. P2 initiates election



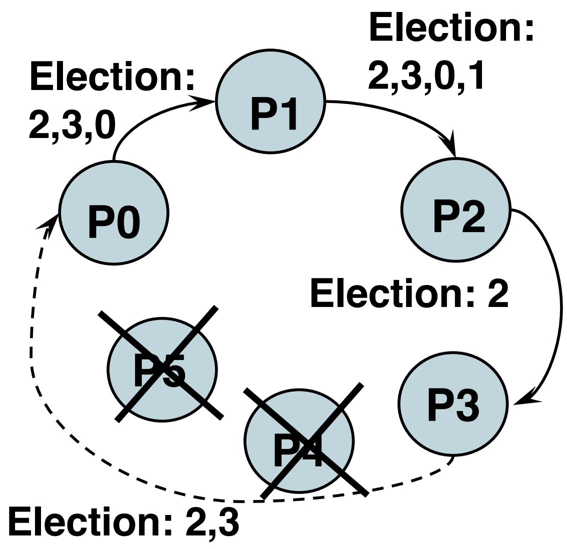
2. P2 receives "election", P4 dies



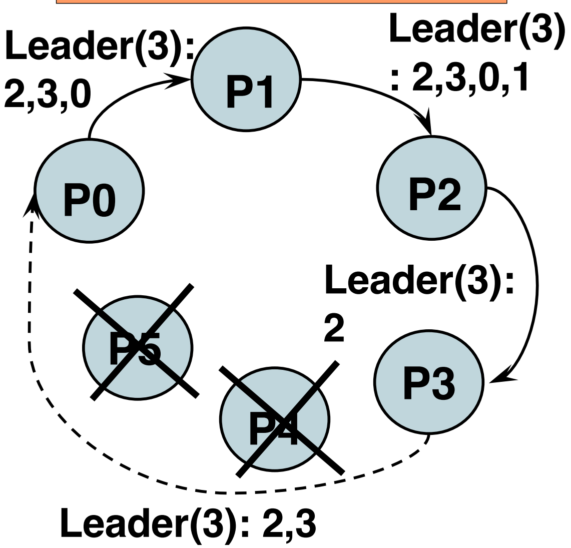
3. P2 selects 4 and announces the result



4. P2 receives "Coord", but P4 is not included



5. P2 re-initiates election



6. P3 is finally elected

Ring Algorithm Analysis

- Worst case 2(N-1) messages are passed (when does this happen?)
  - One round for the ELECTION message
  - One round for the LEADER
  - Assumes that only a single process starts an election.
- Multiple elections cause an increase in messages

1: Leader Election: Bully Algorithm

Bully Algorithm In a Nutshell

- Model
  - Synchronous model
  - Processes know each other’s ids
  - A process can detect that another process failed based on message transmission time and processing time
- Process p starts election
  - When it detected that the coordinator has failed
  - When it recovered from a crash
- High-numbered processes “bully” low-numbered processes out of the election, until only one process remains.

Bully Algorithm

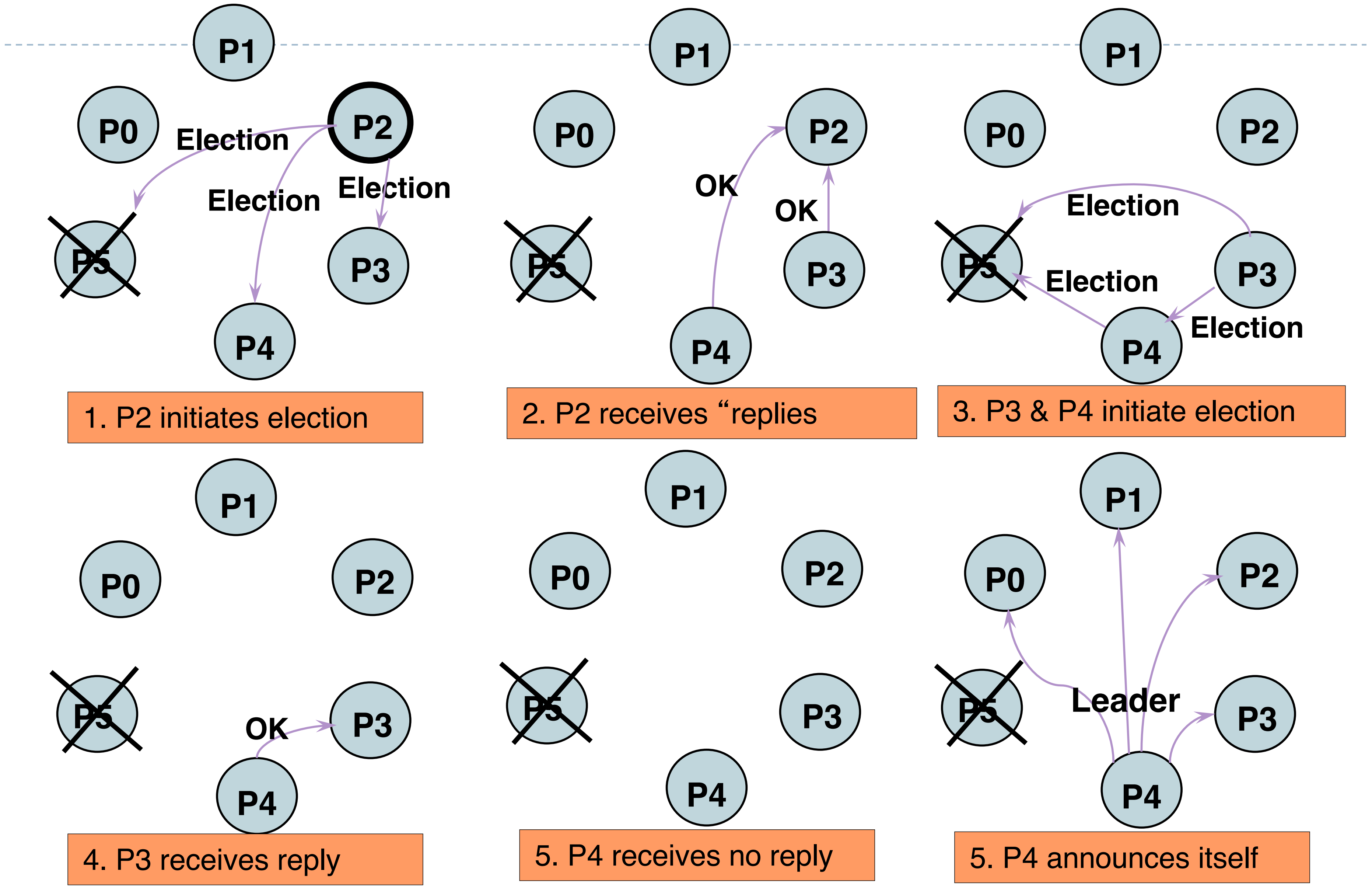
- A process starts the algorithm by sending ELECTION message to only the nodes that have a higher id than itself
  - If no answer OK is received, then it announces itself as the new leader to the lower processes, with a LEADER message
  - If any OK is received, then there is a process with a higher id, wait for the LEADER message; if none received start a new election algorithm
- If a process received an ELECTION message, sends an OK and then starts a new election
- If a process detects that the leader has failed and it has the highest id, then sends a LEADER message to all processes with lower identifiers

Bully Algorithm Message Cost

- Best case: The node with second highest identifier detects failure and elects itself
  - Total messages = N-2
    - One message for each of the other processes indicating the process with the second highest identifier is the new coordinator
- Worst case: The node with lowest identifier detects failure
  - Total messages = O(N2)
    - requires N-1 processes to initiate the election algorithm each sending messages to processes with higher identifiers

Example Bully Election

Example from 425, Prof. Klara Nahrstedt



1. P2 initiates election

2. P2 receives “replies

3. P3 & P4 initiate election

4. P3 receives reply

5. P4 receives no reply

5. P4 announces itself

2: Membership

Membership Service

- Needed for distributed protocols that require knowledge of alive processes
  - Static: list is known before, track processes that crash
  - Dynamic: processes can join, leave and crash
- Need to detect failures (Remember ! we know we can not do it accurately)
- Need to agree on the current list of processes

A Membership Protocol

- Leader: one of the processes (oldest) will act as leader
- Each process: maintains a list with alive processes (list has to be the same)
- All processes: track each other (ping or I am alive)
  - If timeout occurs - process that did not answer is considered crashed, he will have to rejoin with another identifier
- Two cases:
  - leader is alive (normal-case)
  - leader fails

Normal-Case

- Leader detects a failure or receives a join, he starts a two-phase protocol to ensure the list of alive members is updated consistently
- Phase 1: leader sends all add and delete events to everybody
  - Every process acknowledges
  - Leader must wait for a majority of acknowledgements
- Phase 2: If leader receives majority, then sends the modifications (may include any failure detected during first phase)

Leader Fails

- If a process detects that the leader failed, second process on the list becomes the leader, three phase protocol
- Phase 1: new leader informs the other processes that leader has failed, asks for pending add/delete operations, collects acknowledgments and current membership information
- Phase 2 and 3 similar with normal case

3: Reliable multicast

Unicast, Broadcast, Multicast

- **Unicast**: Messages are sent from exactly one process to one process
- **Broadcast**: Messages are sent from exactly one process to all processes on the network
- **Multicast**: Messages are sent from exactly one process to several processes (referred as group) on the network

Reliable Communication

- **Unicast**: one sender and one receiver
- **Multicast**: one sender and many receivers
- **Reliable unicast**: guarantees delivery of messages, if the other party fails, there is no service
- **Reliable multicast**: ? What is the meaning of reliable multicast in the context of process failures?

Naïve Approach

- Use a reliable one-to-one send operation.
- A basic multicast primitive guarantees a correct (non-faulty) process will eventually deliver the message, as long as the sender does not crash.
- What if the sender crashes after he sent the message and some processes received the message and some other did not?

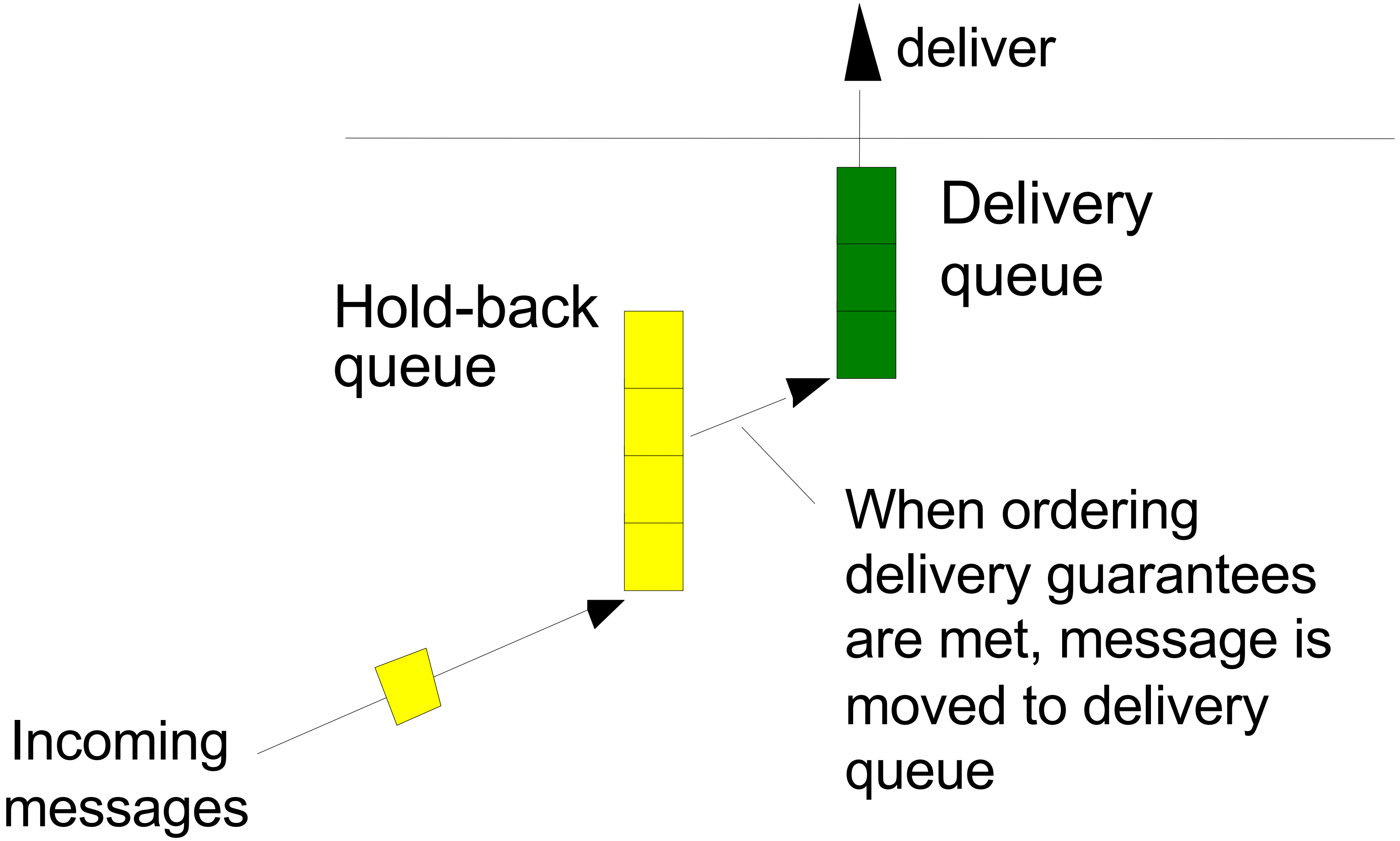
Meaning of Reliability in Multicast

- **Integrity**: A correct process p delivers a message m at most once.
- **Validity**: If a correct process multicasts message m, then it will eventually deliver m.
  - Each process delivers its own messages
- **Agreement**: If a correct process delivers message m, then all the other correct processes in the group will eventually deliver m.

Ordered Multicast

- **FIFO ordering**: If a correct process multicasts m and then multicasts m', then every correct process that delivers m' will have already delivered m.
- **Causal ordering**: If multicast m → multicast m' then any correct process that delivers m' will have already delivered m.
- **Total ordering**: If a correct process delivers message m before m', then any other correct process that delivers m' will have already delivered m.

Message Processing

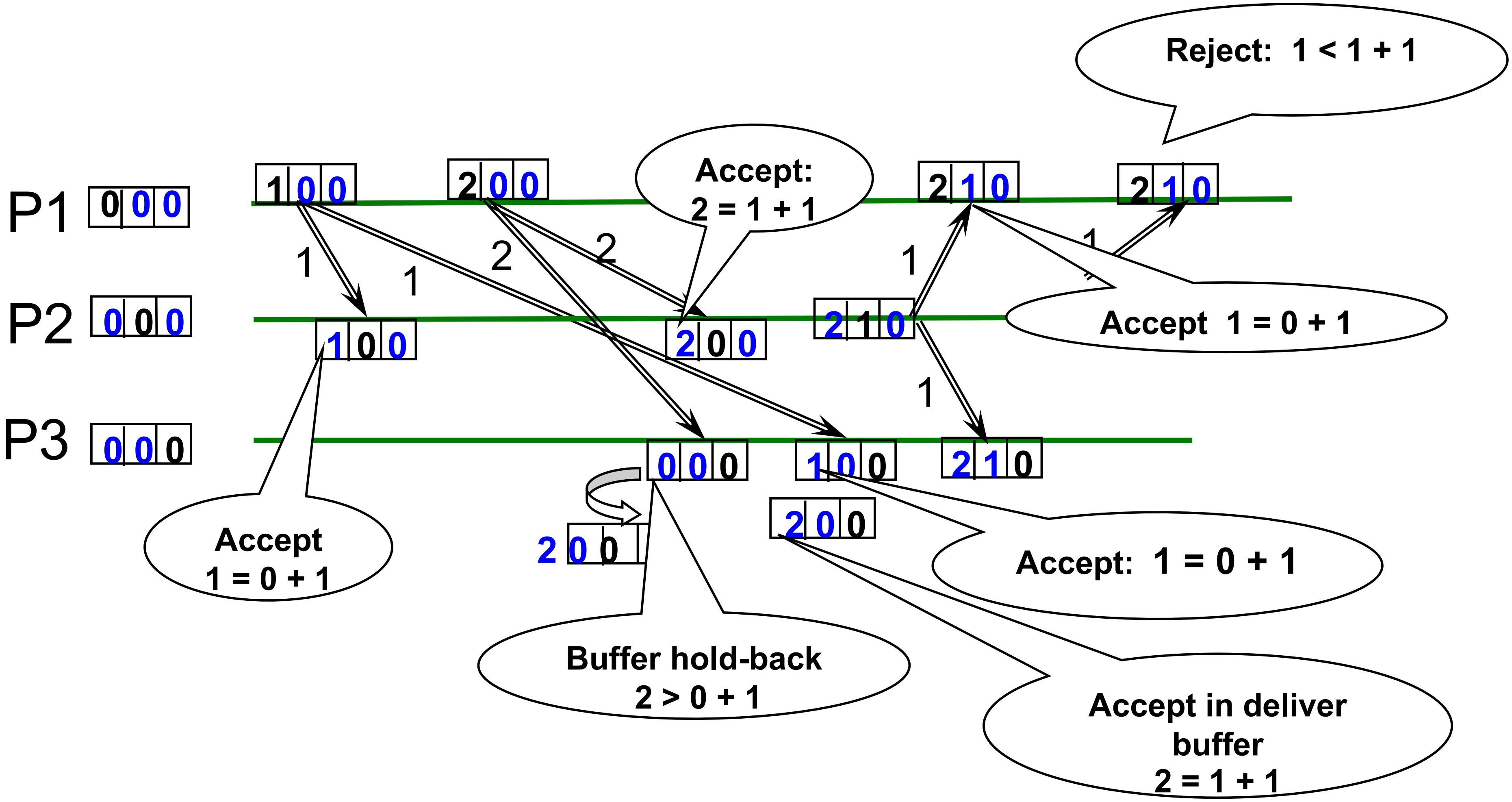

deliver
Delivery queue
Hold-back queue
When ordering delivery guarantees are met, message is moved to delivery queue
Incoming messages

FIFO Reliable Multicast Algorithm

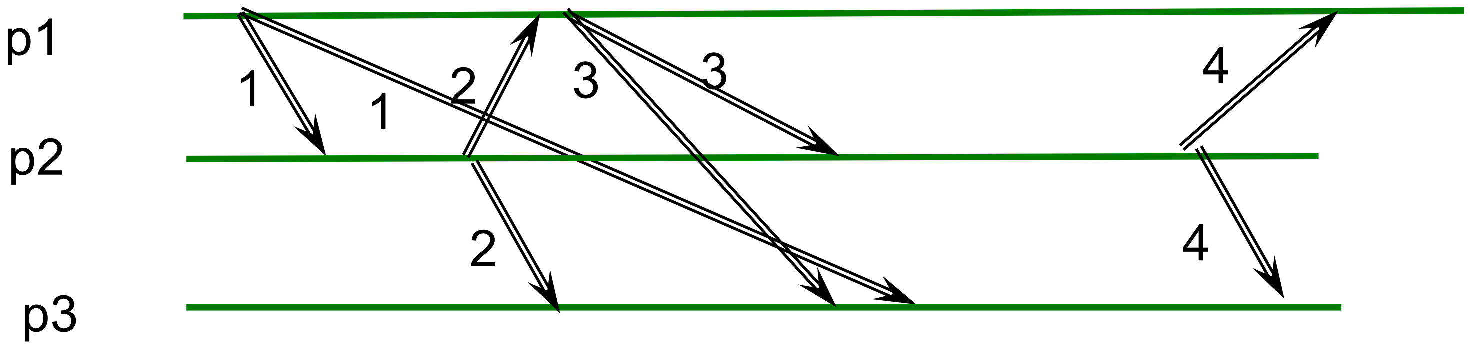
- S^p_G: count of messages p has sent to group G.
- R^q_G: the sequence of the latest message that p has delivered from q to the group G.
- **When sending**: p multicasts message m to group G
  - $S^p_G = S^p_G + 1$
  - S^p_G is included with m
- **When receiving**: p receives message m from q with sequence number S for group G:
  - If $S = R^q_G + 1$, p delivers m and $R^q_G = R^q_G + 1$
  - If $S > R^q_G + 1$, p places the message in the hold-back queue (need to receive other messages first from q)
  - If $S < R^q_G + 1$, p drops the message (old message)

FIFO Example

Example from 425, Prof. Klara Nahrstedt



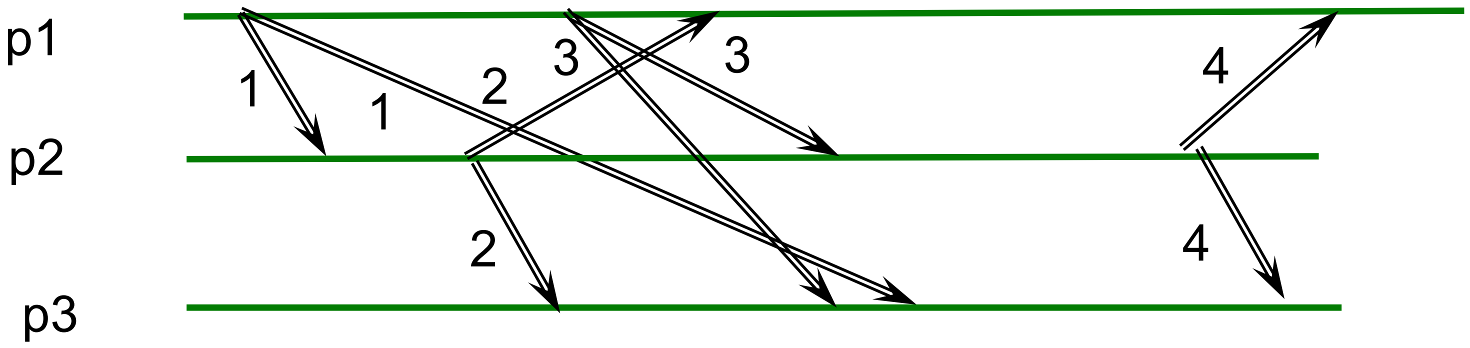
Example of FIFO ordering - 2



P1 will deliver, 1, 2, 3, 4,
P2 will deliver 1, 2, 3, 4
P3 will deliver 2, 1, 3, 4

Messages from different senders can be interleaved, as long as FIFO is enforced for each sender

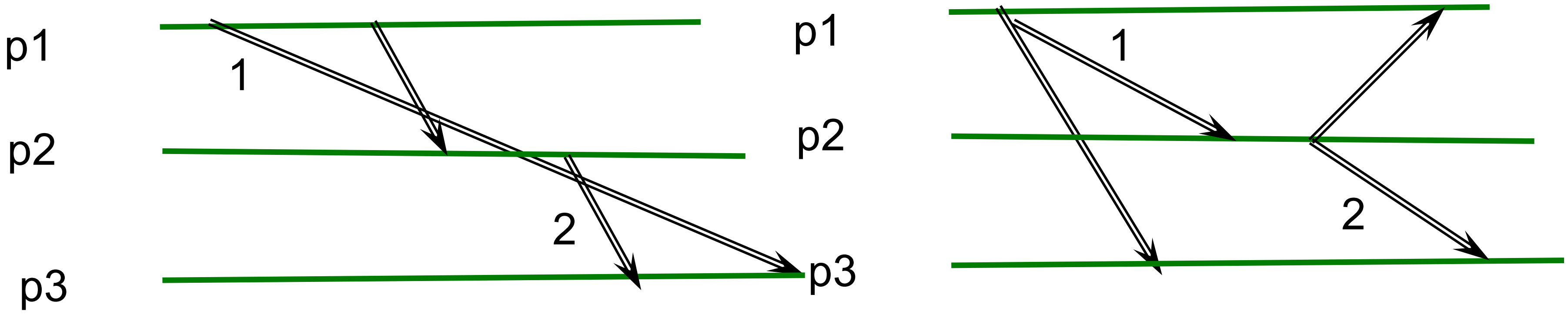
Example of FIFO ordering - 3



P1 will deliver, 1, 3, 2, 4,
P2 will deliver 1, 2, 3, 4
P3 will deliver 2, 1, 3, 4

Causal ordering

Causal ordering: If multicast m → multicast m’ then any correct process that delivers m’ will have already delivered m.



What can you say about messages 1 and 2?
Can p3 deliver 2 is he wants to preserve causal ordering?

Can p3 deliver 2 is he wants to preserve causal ordering?

Causal Multicast Algorithm

- Use vector clocks:

  $V(a) < V(b)$ iff a happens before b

- Each process maintains a vector clock per group
  - $V^G_i[\ j\ ]$ counts the number of group *G* messages from process j to process I delivered to the application
- When process *i* receives a $<m, V^G_j>$ from j, then
  - $V^G_i[k] = \max(V^G_i[k], V^G_j[k])$ if $k \neq i$
  - $V^G_i[k] = V^G_i[k] + 1$ if $k = i$

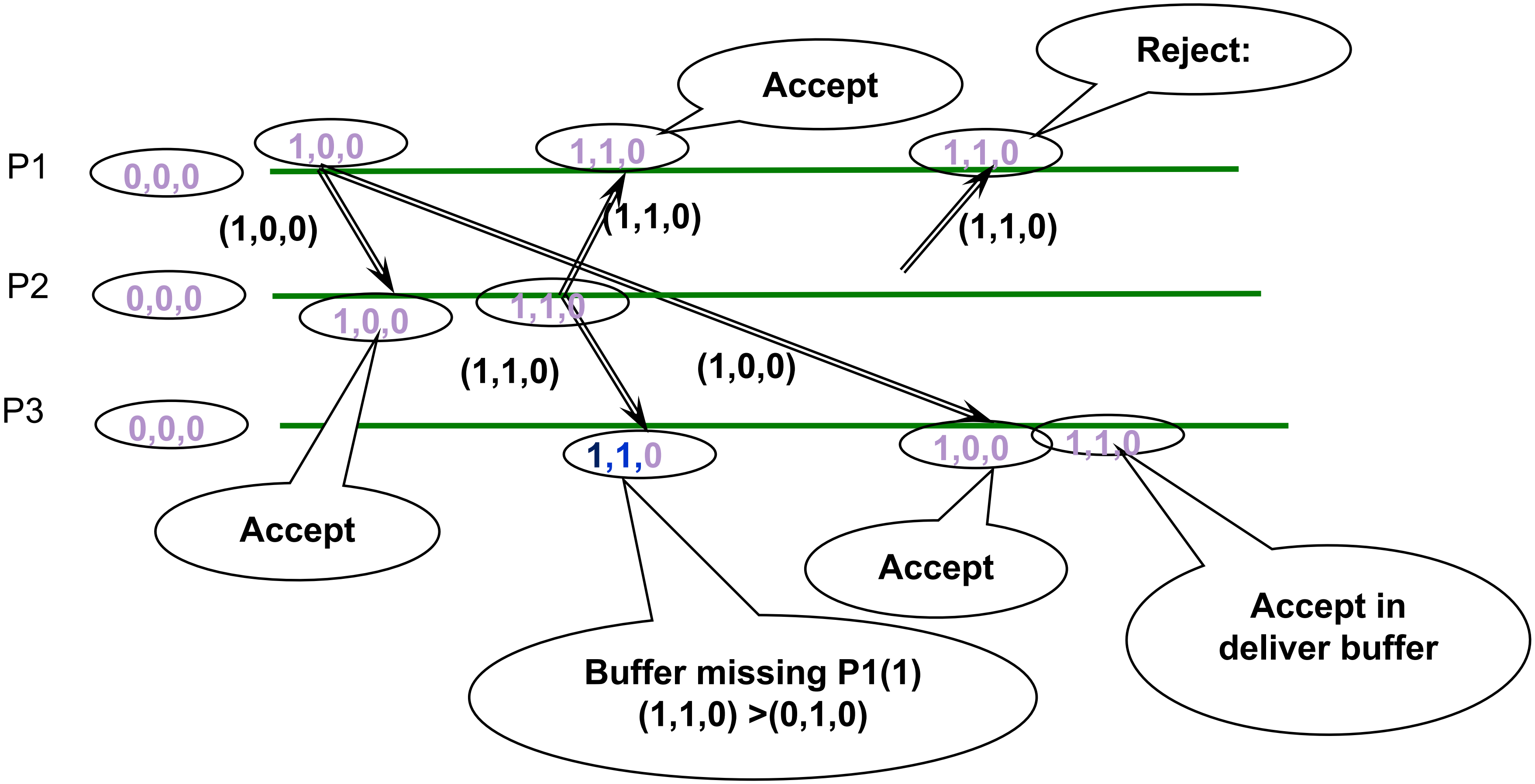
Causal Reliable Multicast

- Initialize V^G_i [j] = 0, j = 1, 2, …N processes
- When sending: process i to group G
  - V^G_i [i] = V^G_i [i] + 1 (increment state of i)
  - Send message with entire vector V^G_i
- When receiving: process i received m from process j for group G
  - Put m, V^G_j in hold-back queue
  - Wait till causality is met V^G_j [j] = V^G_i [j] + 1 and V^G_j [k] ≤ V^G_i [k], any k ≠ j

  then deliver m and V^G_i [j] = V^G_i [j] + 1

Example

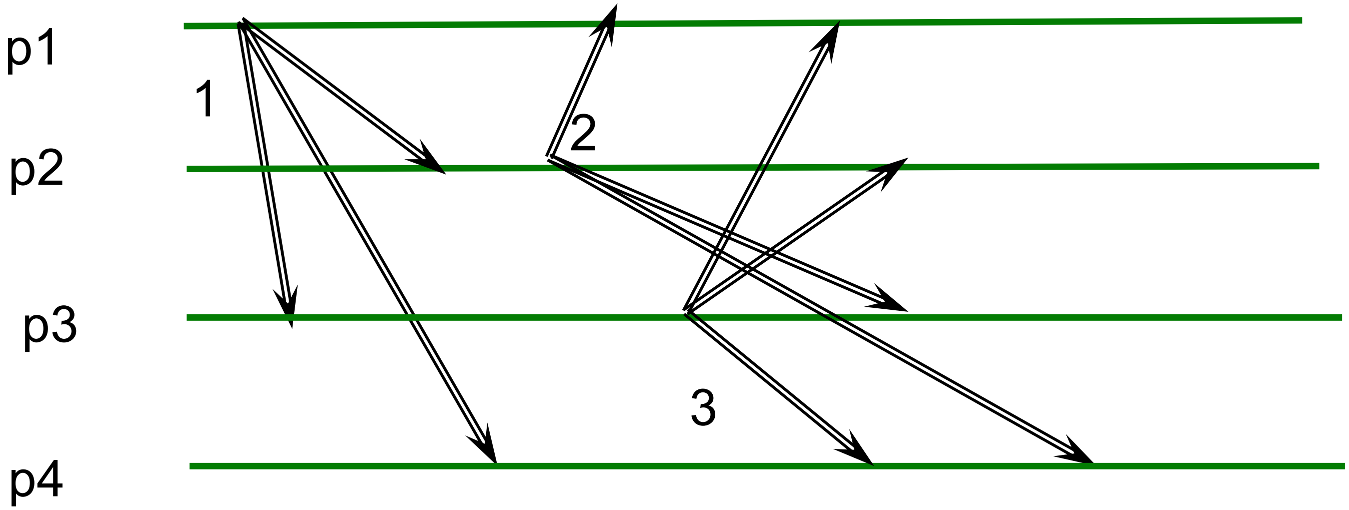
Example from 425, Prof. Klara Nahrstedt



process i received m from process j

Causality is met $V^G_j[j] = V^G_i[j] + 1$ and $V^G_j[k] \leq V^G_i[k]$, any $k \neq j$

Vector Clocks vs Total Order



2 and 3 are not causally related !!!!!

P1 will deliver, 1, 2, 3
P2 will deliver 1, 2, 3
P3 will deliver 1, 3, 2
P4 will deliver 1, 3, 2

Isis

- Toolkit for distributed programming
- Useful for managing replicated data, synchronizing distributed computations, automating recovery, and dynamically reconfiguring a system to accommodate changing workloads
- Developed at Cornell

Isis Total Ordered Multicast

- Uses sequences associated with each message and ID of processes to determine order
- Each process maintains a queue with messages received
- Messages can be ready to deliver or not based on what a process knows about what other processes did (the sequence)

Isis Total Ordered Multicast (cont)

Sender multicasts the message to everyone

Upon receiving a message M each receiver R_i

1. Adds M to the queue
2. Marks the message *undeliverable*
3. Sends ack to the sender with a sequence number seq that is *the latest sequence number received* + 1, suffixed with the R_i's ID.

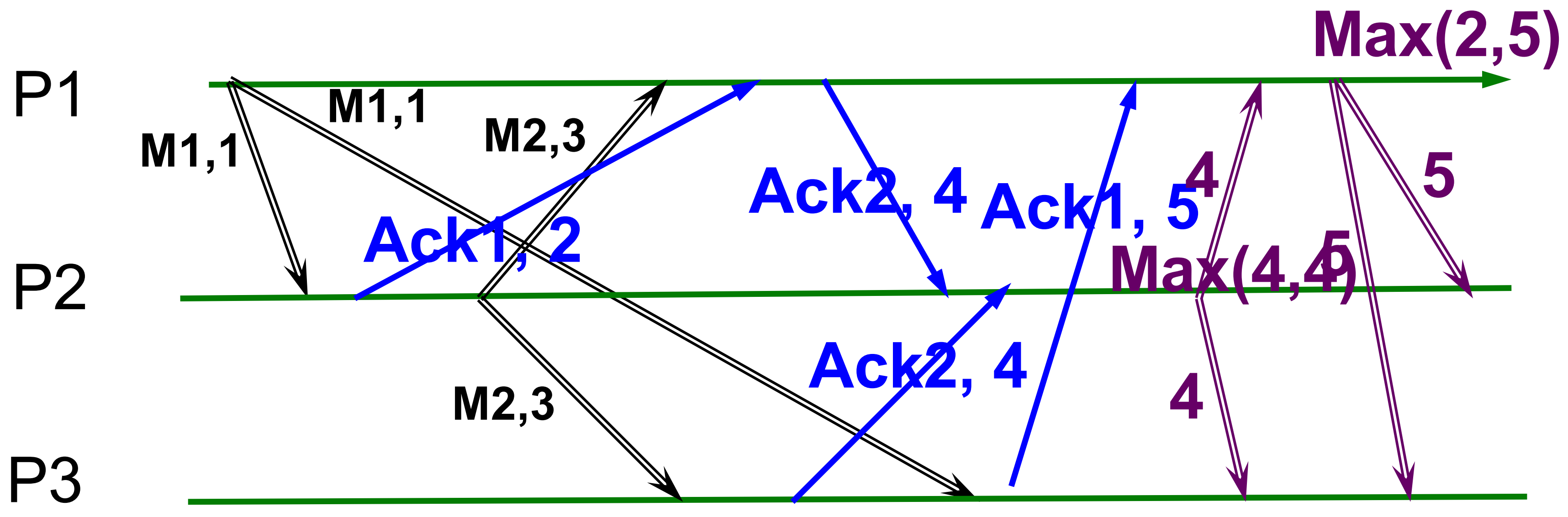
Sender collects all acks from the receivers

1. calculates final_seq = *maximum* (${seq_i}$)
2. multicasts final_seq to all processes

Upon receiving final seq each receiver

1. marks M as *deliverable*,
2. reorders the queue based on seq
3. delivers the set of messages with lower seq and marked as *deliverable*.

Example Total Order



What are the changes if P1 receives Ack1 before M2?
What are the changes if P2 receives m1 after sending mM?

TOTEM: The single-ring protocol

- Uses a circulating token containing among others:
  - A seq field with the sequence number of the last message that was sent
  - An aru field with the sequence number of the last message that has been received by all processors, replaces acks
- Only the processor that holds the token can send a message

Meaning of SEQ and ARU

- Provides total order on message
- Used to detect gaps and request retransmissions through a field in the token
- After a full token rotation process can determine all processes have received all message with lower sequence numbers

Using the aru

- If token.aru = token.seq and have all the messages then the process should set aru higher and seq when sending new messages
- If missed a message with m.seq smaller than then should set token.aru = m.seq
- If is the one that lowered the aru and the token.aru is still the same, should set token.aru = local.aru

Safe Delivery

- Consistent with Total/Agreed order.
- Message is delivered after received by all processors.

TOTEM: The single-ring protocol (II)

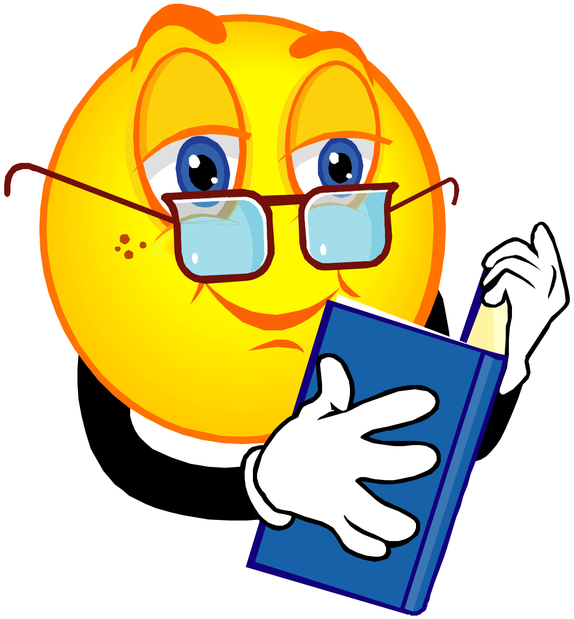
- aru field used to implement safe delivery:
  - Tells processors which messages have been received by every processor in the ring
- Token also provides information about the aggregate message backlog of the processors on the ring
  - Results in a fairer bandwidth allocation among processors

Membership and Reliable Multicast

- Message delivery
- Group membership changes
- They are interleaved
- Does this matter?

Summary

- Leader election algorithms: usually select the process with the highest id, network topology determines complexity in terms of number of messages
- Membership services must take into account leader failures
- Meaning of reliable muticast is more complex than for reliable unicast, different ordering guarantees: FIFO, causal, total order



4: Virtual Synchrony

Required reading for this topic…

- **Exploiting virtual synchrony in distributed systems**, K Birman and T. Joseph, SOSP 1987
- Extended Virtual Synchrony, L. E. Moser , Y. Amir , P. M. Melliar-Smith , D.A. Agarwal, DISC 1994
- Group Communication Specifications: A Comprehensive Study. Gregory V. Chockler, Idit Keidar, and Roman Vitenberg. ACM Computing Surveys, 2001.

Systems …

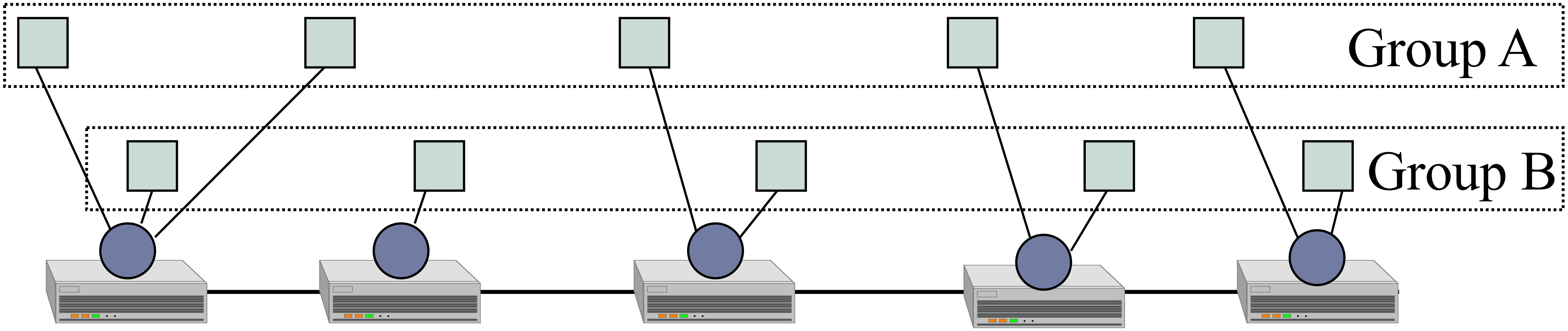
- www.spread.org
- http://www.cs.huji.ac.il/labs/transis/lab-projects/guide/intro.html
- http://www.cs.huji.ac.il/labs/transis/lab-projects/guide/chap3.html
- http://www.cs.cornell.edu/Info/Projects/ISIS/

Process Groups

- One way of building distributed fault-tolerant systems by organizing them in a group and ensuring group membership and group multicast, with different ordering properties.
- Easier to work with when providing in the form of a toolkit.

Implementation

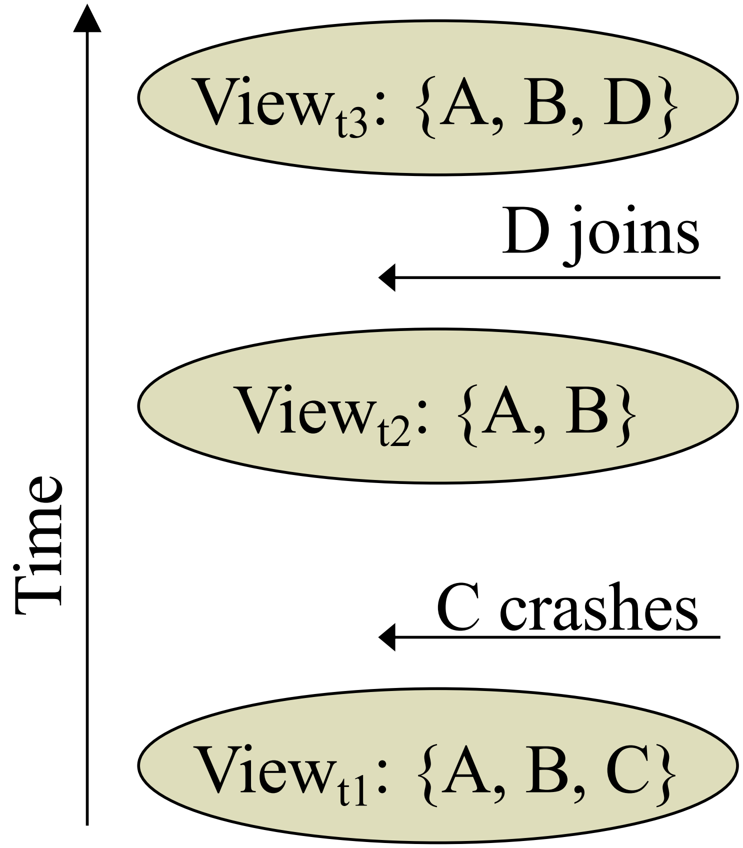
- Either client-server (as in the picture) - servers perform the distributed protocols, clients and groups are lightweight
- Or completely distributed, limited scalability



- Reliable and ordered message delivery (unicast and broadcast)
- Group membership service may support process failures, network partitions and merges

Semantics

- **View**: list of group members at a certain time
- **Semantics**: define how the membership and the messages are interleaved and what is the service provided to the applications
- Useful for implementing other distributed applications such as: state transfer, replicated data, load balancing.
- **Two models: Virtual Synchrony Model (VS) and Extended Virtual Synchrony Model (EVS)**



Time

View_{t3}: {A, B, D}

D joins

View_{t2}: {A, B}

C crashes

View_{t1}: {A, B, C}

Ordered Multicast

- **FIFO ordering**: If a correct process multicasts m and then multicasts m', then every correct process that delivers m' will have already delivered m.
- **Causal ordering**: If multicast m → multicast m' then any correct process that delivers m' will have already delivered m.
- **Total ordering**: If a correct process delivers message m before m', then any other correct process that delivers m' will have already delivered m.

Safe Delivery

- Consistent with Total/Agreed order.
- Message is delivered after received by all processes (processes send ack) .

Why Virtual Synchrony?

- Ideally: events are in the same order for any two processes, messages delivers to all process at the same moment …
- Impossible
- Events need to be synchronized only to the degree application is sensitive to ordering

Virtual Synchrony Model

Processes that move together through the **same views**, deliver the **same set of messages**.

- The model relates to message and view delivery, and relationship between messages and views.
- Views consist of list of members, have unique identifiers.
- Membership changes are totally ordered with respect to all regular messages that pass in the system.
- The order of the regular messages is determined by the delivery service (fifo, causal, agreed).

Virtual Synchrony Model

- 1. Self Inclusion

  If process p installs a view v then p is a member of v.
- 2. Local Monotonicity

  If process p installs a view v after installing a view v' then the identifier id of v is greater than the identifier id' of v'.
- 3. Self Delivery

  If process p sends a message m, then p delivers m unless it crashes.
- 4. Delivery Integrity

  If process p delivers a message m in a view v, then there exists a process q that sent m in v causally before p delivered m.
- 5. No Duplication

  A message is sent only once. A message is delivered only once to the same process.

Virtual Synchrony Model

- 6. Sending View Delivery

  A message is delivered in the view that it was sent in.

- 7. Virtual Synchrony

  Two processes that move together through two consecutive views deliver the same set of messages in the former view.

Virtual Synchrony Model

- 8. Causal Delivery
  If message m causally precedes message m', and both are sent in the same view, then any process q that delivers m' delivers m before m'.
- 9. Agreed Delivery
  - Agreed delivery maintains causal delivery guarantees.
  - If agreed messages m and m' are delivered at process p in this order, and m and m' are delivered by process q, then m is delivered before m' by q.

How to Provide Virtual Synchrony?

- Messages can be lost
- Before moving into new view, exchange message to flush all the messages from previous view
- Application message are blocked during view change
- Joins are not allowed during view change

Virtual Synchrony and Network Partitions

- Virtual Synchrony was introduced in a model that did not consider network partitions, fail stop failure (ISIS)
- Later extended to network partitions (TRANSIS, SPREAD)
- Allows operation to be partitionable in order to support crash recoveries and network partitions:
- If a process group partitions into subsets that cannot communicate with each other, each subset continues observing the (partitionable) Virtual Synchrony model separately.
- Upon re-merging, the merged set will be virtually synchronized from the merging membership change message.

Extended Virtual Synchrony (EVS)

- ~~Major difference is~~
- ~~6. Sending View Delivery~~

  ~~A message is delivered in the view that it was sent in.~~

- 6. Same View Delivery

  A message is delivered in the same view.

- Better performance, message can be delivered faster.
- Delivery view is not necessary the same as sending view

EVS: Main Idea

- While noticing a membership change, the new view is not immediately delivered to the application
  - System switches into a transitional phase trying to
    - recover lost messages from the current view
    - achieve consistency among configuration members that are still connected.
  - New messages from the application are buffered until the transitional phase ends and a new view is reached.
  - The new membership is delivered to the application
  - Previously buffered messages are multicast and processed together with new messages from the applications.

FIFO and EVS

- Assume a membership change takes place and some process p in the current view notices it missed a message
  - p requests the missed message.
  - If p is still connected to other members of the configuration, and some received that message, he will receive the lost message
  - If there is no connected member that received this message - then this does not contradict the virtual synchrony or FIFO
- When we can not deliver messages?
  - If all connected members received the ith message m' from process p, but missed the (i-1)th message m from p, and p is no longer reachable, then m' could not be delivered because it would contradict the FIFO mode guarantees.
- Note that the delivery view will not necessary be the same as the sending view.

Causal and EVS

- Similar to FIFO in recovering lost messages if possible.
- When we can not deliver a message?:
  - If all connected members have received the i^{th} message m from process p that is no longer reachable, but missed the message m' that could be the $(i-1)^{th}$ message from that same process p, or the j^{th} message from another process q that is also no longer reachable, then m could not be delivered because the causality principle is violated.
- Notice that if the network partitions and several detached components of the same configuration are created, then each could deliver a different set of messages, depending on the knowledge of the component members.
  - if p is in another component, then this component will deliver m (unless causality is contradicted by a former lost message m'). This will not contradict the Virtual Synchrony model's guarantees since the following membership change message each component delivers is different.

Agreed and EVS

- Lost messages are recovered if possible
- Messages that may contradict the causality principle are not delivered.
  - For example a message m that causally follows a lost message m' can not be delivered because it will contradict causality.
- (C1) Every process must deliver its own messages; however, although they are buffered, they can not be delivered before they are totally ordered.
- **Messages may be lost, or become undeliverable after a membership change. A lost or an undeliverable message creates a hole in the total order.**

Agreed and EVS (cont.)

- Consider the case that a message m sent by process p is totally ordered after a hole that stands for message m' .
  - If message m is not delivered, C1 is contradicted.
  - If message m is delivered in the current configuration, then total order is not kept throughout the configuration, since in another component message m' may be accessible, and will be delivered before message m.
- Solution: Use a transitional configuration which contains members of the current regular configuration that are still connected
  - It begins when a membership change starts, and lasts until it is completed and a membership change is delivered to the application.
- Messages such as m are delivered in the transitional configuration.

Safe and EVS

- A safe message may only be delivered to the application when all other processes in the configuration have acked that message.
- Consider the case when some process does not ack, or if that ack did not reach all processors just before a membership change started
  - If some of the connected processors received the ack, they can retransmit it, and that message could be delivered as part of the current (unchanged) configuration.
  - If a processor did not ack, or the ack was lost by all connected members of the configuration, then the message cannot be delivered as safe in the current configuration C.
- The solution is to use a transitional configuration. In this configuration, messages receive acks from all members, and therefore can be delivered as safe in this context.

EVS: Regular Configuration

- Regular configuration is the configuration in which regular messages are sent and delivered. FIFO (atomic) and causal communication modes need to use only this configuration type in order to deliver messages.

EVS: Transitional Configuration

- Used to correctly define and implement total order and safe communication modes in a partitionable environment. The transitional configuration consists of members that come directly from the same regular configuration and that will also be members of the same future regular configuration.
  - allows delivery of messages following holes multicast in the prior regular configuration.
- A regular configuration may be followed by several transitional configurations (when several components detach), and preceded by several transitional configurations (when several components merge). A transitional configuration, in contrast, is immediately preceded and followed by a single regular configuration.

Transitional Configuration

- 11. Transitional Set
  - Every process is part of its transitional set for a view v.
  - If two processes p and q install the same view, and q is included in p's transitional set for this view then p's previous view was identical to q's previous view.
  - If two processes p and q install the same view v, and q is included in p's transitional set for v then p and q have the same transitional set for v.
- 12. Transitional Signal
  - Each process delivers exactly one transitional signal per view.
  - If two processes p and q install the same view v and q is included in p's transitional set for v then p and q deliver the same set of agreed messages before and after the transitional signal.

Agreed and Transitional Configuration

- 9. Agreed Delivery
  - Agreed delivery maintains causal delivery guarantees.
  - If agreed messages m and m' are delivered at process p in this order, and m and m' are delivered by process q, then m is delivered before m' by q.
  - If agreed messages m and m' are delivered by process p in view v in this order, and m' is delivered by process q in v before a transitional signal, then q delivers m. If messages m and m' are delivered by process p in view v in this order, and m' is delivered by process q in v after a transitional signal, then q delivers m if r, the sender of m, belongs to q's transitional set.

Safe and Transitional Configuration

- 10. Safe Delivery
  - Safe delivery maintains agreed delivery guarantees.
  - If process p delivers a safe message m in view v before the transitional signal, then every process q of view v delivers m unless it crashes. If process p delivers a safe message m in view v after the transitional signal, then every process q that belongs to p's transitional set delivers m after the transitional signal unless it crashes.

Systems Providing Virtual Synchrony

- Isis: Introduced VS and no longer widely used
  - Developed at Cornell
  - Very successful; has major roles in NYSE, Swiss Exchange, French Air Traffic Control system (two major subsystems of it), US AEGIS Naval warship
  - Also was first to offer a publish-subscribe interface that mapped topics to groups
- Totem and Transis
  - Totem (UCSB) went on to become Eternal and was the basis of the CORBA fault-tolerance standard
  - Transis (Hebrew University) became a specialist in tolerating partitioning failures

Systems Providing Virtual Synchrony

- Horus and Ensemble
  - Developed at Cornell: successors to Isis
  - Both focus on flexible protocol stack linked directly into application address space
    - A stack is a pile of micro-protocols
    - Can assemble an optimized solution fitted to specific needs of the application by plugging together “properties this application requires”, lego-style
    - The system is optimized to reduce overheads of this compositional style of protocol stack
  - Ensemble is relatively popular and supported by a user community. Horus works well but is not widely used.

Systems Providing Virtual Synchrony

- Spread Toolkit
  - Developed at John Hopkins
  - Very simple architecture and system
  - Fairly fast, easy to use, rather popular
  - Supports one large group within which user sees many small “lightweight” subgroups that seem to be free-standing
  - Protocols implemented by Spread servers that relay messages to clients

Summary

- Virtual Synchrony: Processes that move together through the same views, deliver the same set of messages.
- Virtual synchrony blocks application from sending messages
- Both crash failure and network partition supported
- Extended Virtual Synchrony, improved performance, more complexity, uses a transitional configuration