Group Communication

Outline

- Reliability of group communication
  - IP multicast
  - Atomic multicast
- Ordering of group communication
  - Absolute ordering
  - Total ordering
  - Causal ordering
  - FIFO ordering

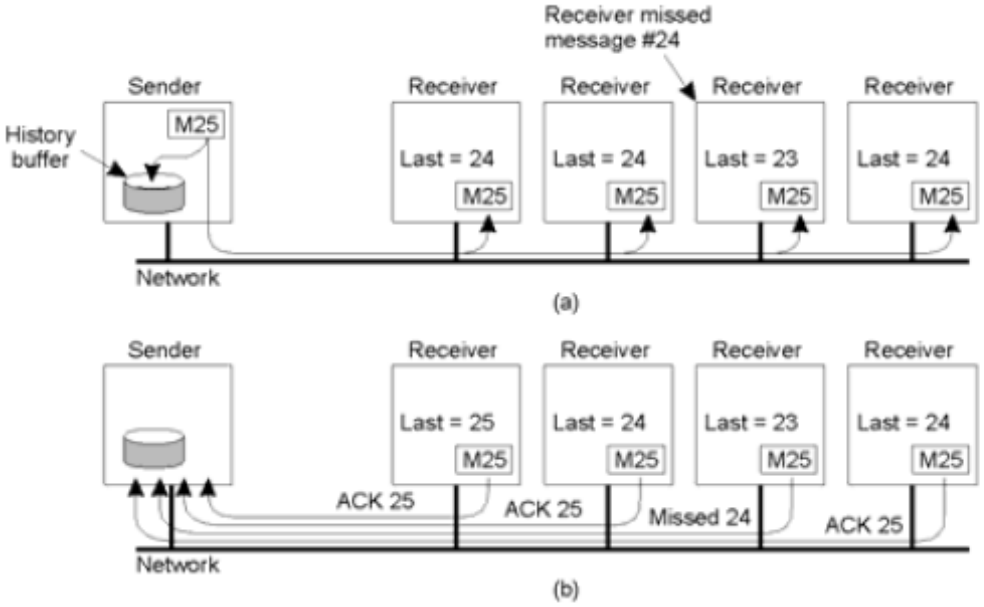
Group Communication

- Communication types
  - One-to-many: broadcast
  - Many-to-one: synchronization, collective communication
  - Many-to-many: gather and scatter
- Applications
  - Fault tolerance based on replicated services (type: one-to-many)
  - Finding the discovery servers in spontaneous networking (type: a one-to-many request, a many-to-one response)
  - Better performance through replicated data (type: one-to-many, many-to-one)
  - Propagation of event notification (type: one-to-many)

Failure

Type of failure	Description
Crash failure	A server halts, but is working correctly until it halts
Omission failure *Receive omission* *Send omission*	A server fails to respond to incoming requests A server fails to receive incoming messages A server fails to send messages
Timing failure	A server's response lies outside the specified time interval
Response failure *Value failure* *State transition failure*	The server's response is incorrect The value of the response is wrong The server deviates from the correct flow of control
Arbitrary failure	A server may produce arbitrary responses at arbitrary times

Basic Reliable Multicast



A simple solution to reliable multicasting when all receivers are known and are assumed not to fail

a) Message transmission

b) Reporting feedback

Process Groups

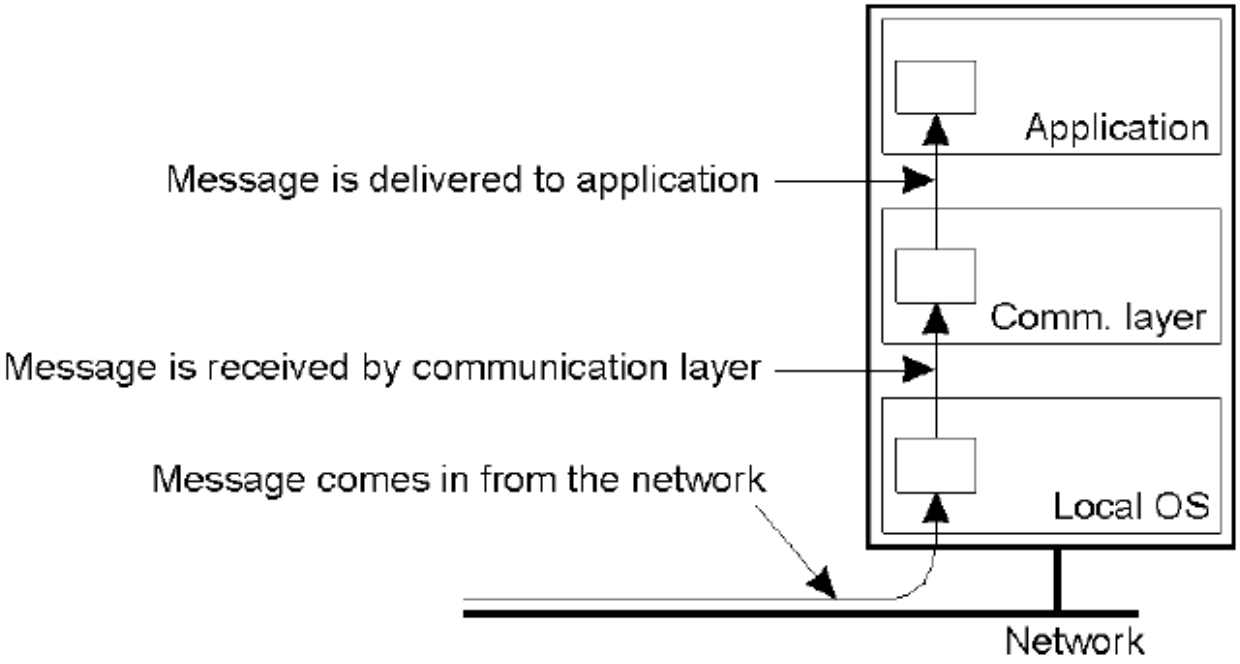
- Process Groups: a paradigm for building fault-tolerant distributed systems based on two primitives –
  - Group Multicast Communication
  - Group Membership

- Group view refers to the set of processes in the group
- Processes join and leave a group --- view change
- Membership changes and multicast messages are interleaved

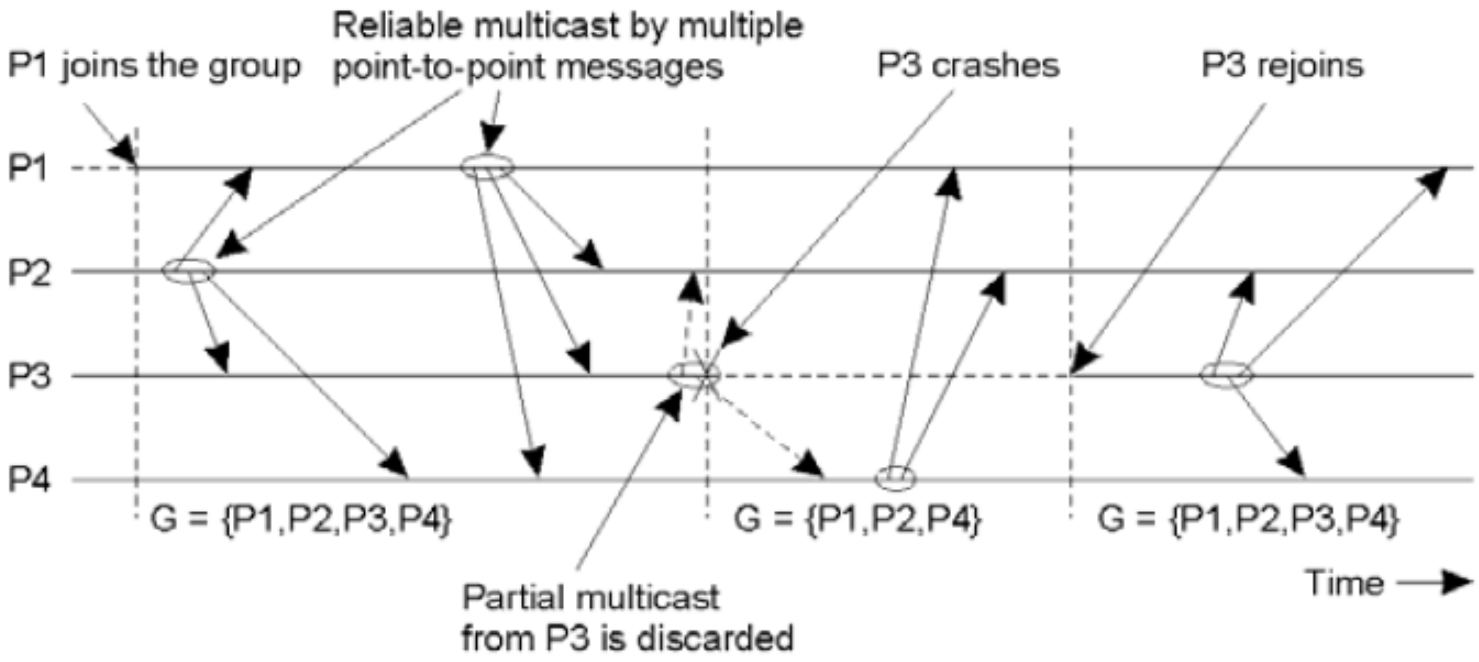
Virtual Synchrony

- **Virtual synchrony:** a strong execution model for process groups in which the same set of multicast messages are delivered between view changes events by all non-faulty members. Also membership (view) changes seen by all processes in the same order.
- What happens if there is a process failure in the middle of multicasting a message?

Virtual Synchrony



- The logical organization of a distributed system to distinguish between message receipt and message delivery
- Distinguish between message reception (by host) and delivery (to application)



The principle of virtual synchronous multicast.

Atomic Multicast

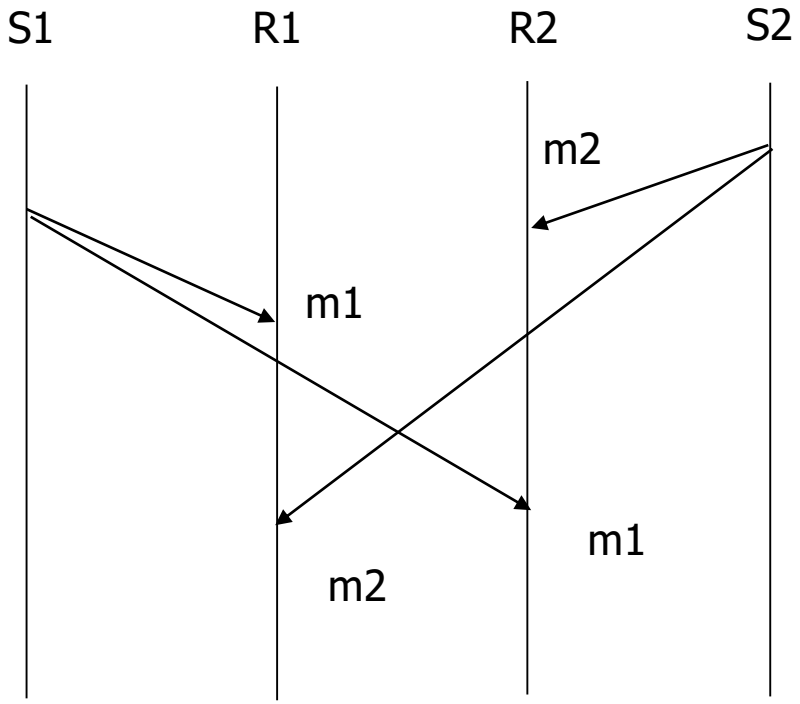
- Send-to-all semantics and all-reliable
- Simple emulation:
  - A repetition of one-to-one communication with acknowledgment
- What if a receiver fails
  - Time-out retransmission
- What if a sender fails before all receivers receive a message
  - All receivers forward the message to the same group and thereafter deliver it to themselves.
  - A receiver discard the 2nd or the following messages.

Reliable Unordered Delivery

Process P1	Process P2	Process P3
sends m1	receives m1	receives m2
sends m2	receives m2	receives m1

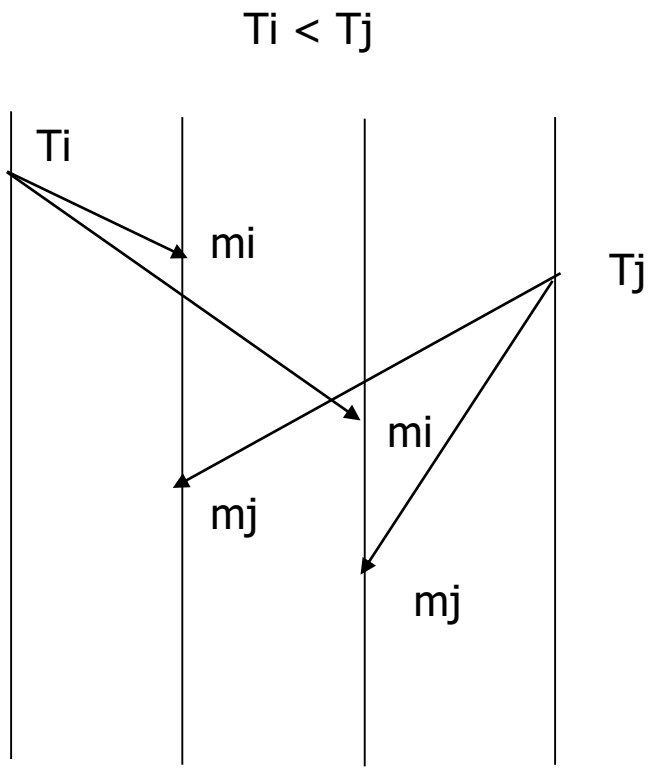
Three communicating processes in the same group. The ordering of events per process is shown along the vertical axis.

Message Ordering



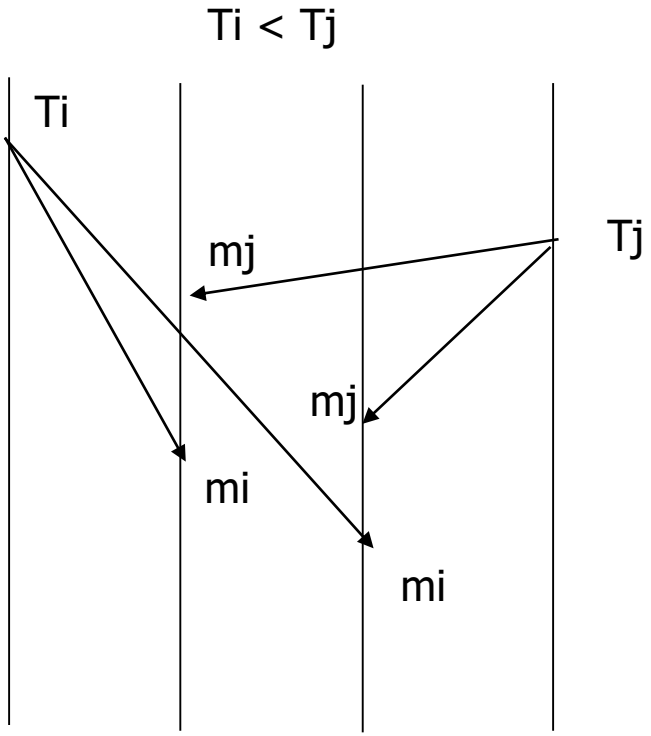
- R1 and R2 receive m1 and m2 in a different order!
- Some message ordering required
  - Absolute ordering
  - Consistent/total ordering
  - Causal ordering
  - FIFO ordering

Absolute Ordering



- Rule:
  - Mi must be delivered before mj if Ti < Tj
- Implementation:
  - A clock synchronized among machines
  - A sliding time window used to commit message delivery whose timestamp is in this window.
- Example:
  - Distributed simulation
- Drawback
  - Too strict constraint
  - No absolute synchronized clock
  - No guarantee to catch all tardy messages

Consistent/Total Ordering



- Rule:
  - Messages received in the same order (regardless of their timestamp).
- Implementation:
  - A message sent to a sequencer, assigned a sequence number, and finally multicast to receivers
  - A message retrieved in incremental order at a receiver
- Example:
  - Replicated database updates
- Drawback:
  - A centralized algorithm

Total Ordering Using A Sequencer

1. Algorithm for group member p

On initialization: $r_g := 0$;

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

Message sent to all group members and a sequencer

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

Receive an incoming message in a temporary queue

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

Receive a sequence number message from the sequencer, Reorder the incoming message with this sequence #, and Deliver it if my local counter reaches this number.

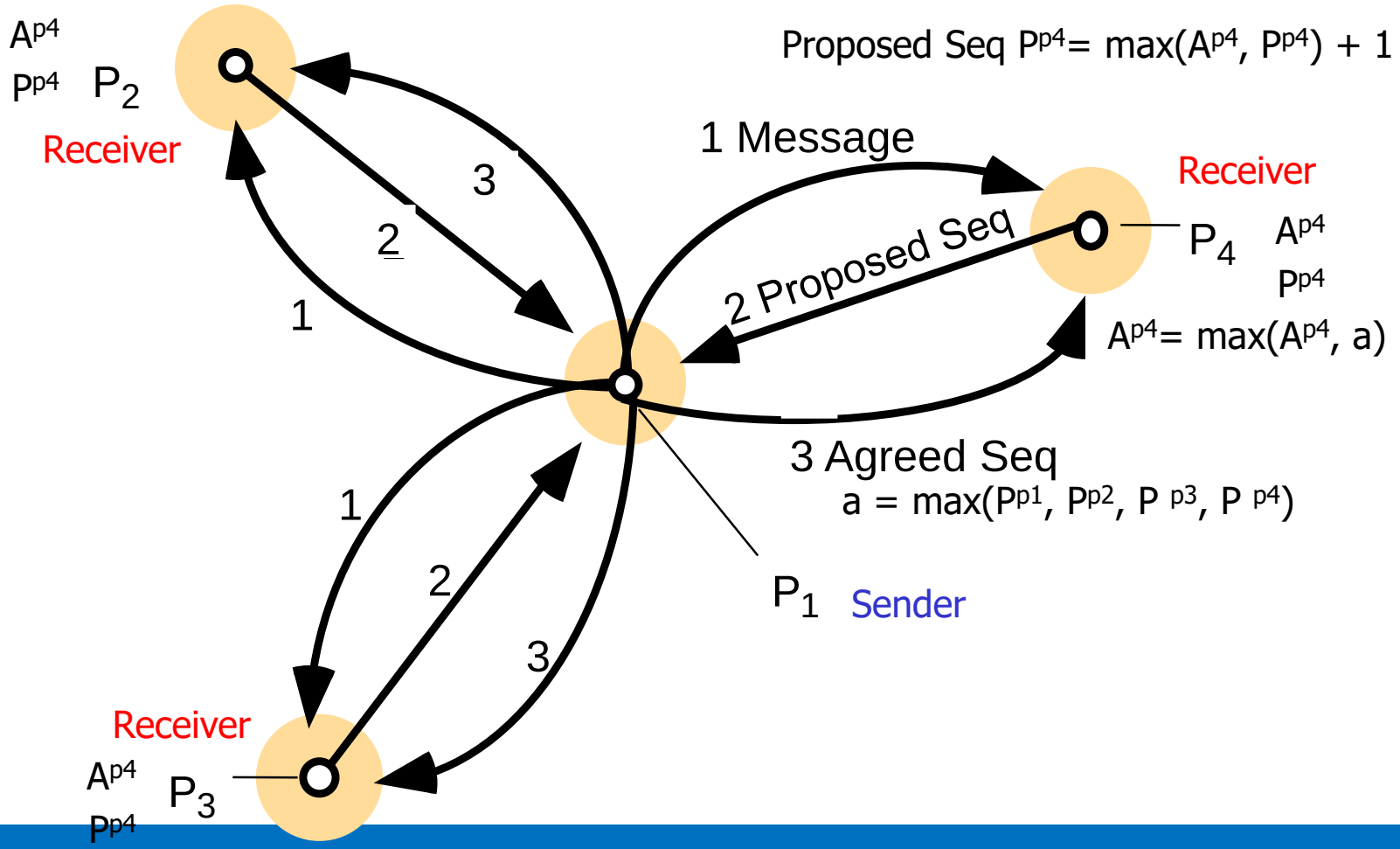
2. Algorithm for sequencer of g

On initialization: $s_g := 0$;

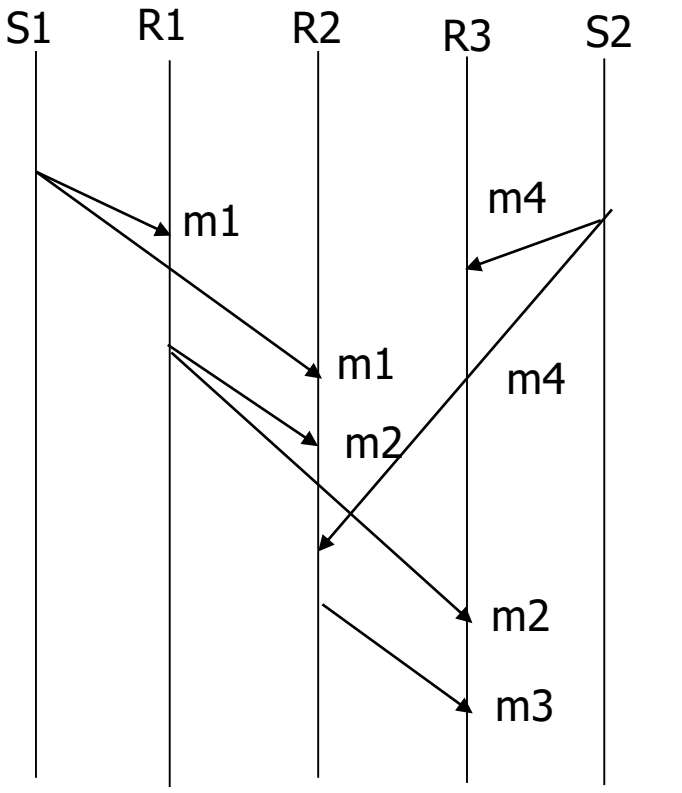
On B-deliver$(<m, i>)$ *with* $g = group(m)$
  B-multicast$(g, <$"order", $i, s_g>)$;
  $s_g := s_g + 1$;

Receive a message, associate it with a sequence number. multicast it to all group members, and increments the sequence number

The ISIS System for Total Ordering



Causal Ordering



From R2's view point m1 →m2

- Rule:
  - Happened-before relation
    - If e^k_i, $e^l_i \in h$ and $k < l$, then $e^k_i \rightarrow e^l_i$,
    - If e_i = send(m) and e_j = receive(m), then $e_i \rightarrow e_j$,
    - If $e \rightarrow e'$ and $e' \rightarrow e''$, then $e \rightarrow e''$
- Implementation:
  - Use of a vector stamps
- Example:
  - Distributed file system
- Drawback:
  - Vector as an overhead
  - Broadcast assumed

Causal Ordering Using A Vector Stamps

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

On initialization

$V_i^g[j] := 0\ (j = 1, 2\ldots, N);$ Each process maintains a vector.

To CO-multicast message m to group g

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

$B\text{-}multicast(g, <V_i^g, m>);$ Increment my vector element, and Send a messge with a vector

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 $V_j^g[j] = V_i^g[j] + 1$ and $V_j^g[k] \le V_i^g[k]\ (k \neq j)$;

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

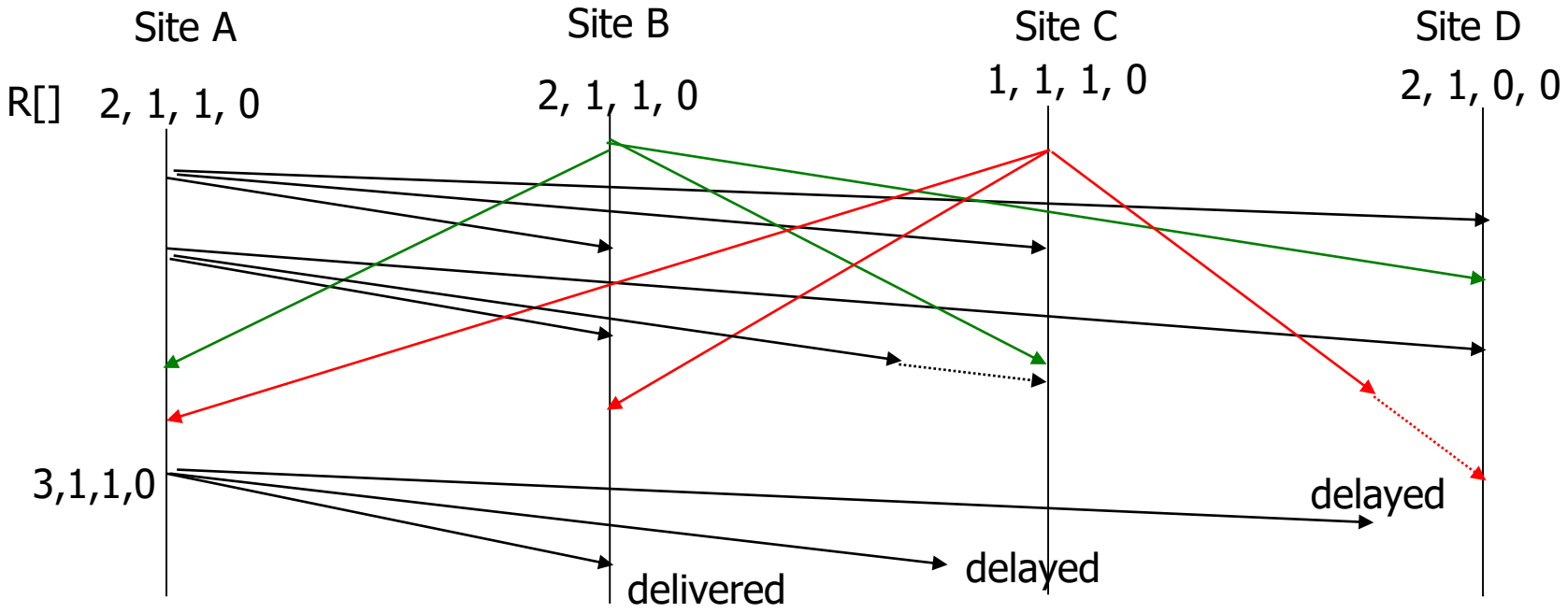
$V_i^g[j] := V_i^g[j] + 1$;

Make sure J's element of J's vector reaches J's element of my vector. (All J's previous message has been delivered.)

Make sure all elements of J's vector <= all elements of my vector. (I've delivered all messages that J has delivered.)

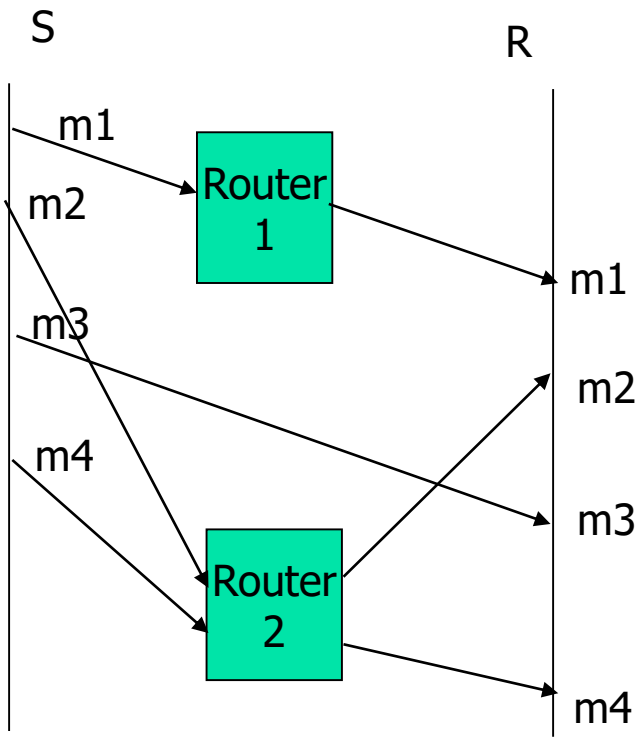
Increment the sender's vector element

Vector Stamps



- S[i] = R[i] + 1 where i is the source id
- S[j] ≤ R[j] where i≠j

FIFO Ordering



- Rule:
  - Messages received in the same order as they were sent.
- Implementation:
  - Messages assigned a sequence number
- Example:
  - TCP
  - This is the weakest ordering.

FIFO Ordering

Process P1	Process P2	Process P3	Process P4
sends m1	receives m1	receives m3	sends m3
sends m2	receives m3	receives m1	sends m4
	receives m2	receives m2	
	receives m4	receives m4	

Four processes in the same group with two different senders, and a possible delivery order of messages under FIFO-ordered multicasting

Total Ordered (Atomic) Delivery

Process P1	Process P2	Process P3	Process P4
sends m1	receives m1	receives m1	sends m3
sends m2	receives m3	receives m3	sends m4
	receives m2	receives m2	
	receives m4	receives m4	

- All processes see same delivery order
- Often used in replication

ISIS

ISIS System --- first group communication system built upon virtual synchronous execution models

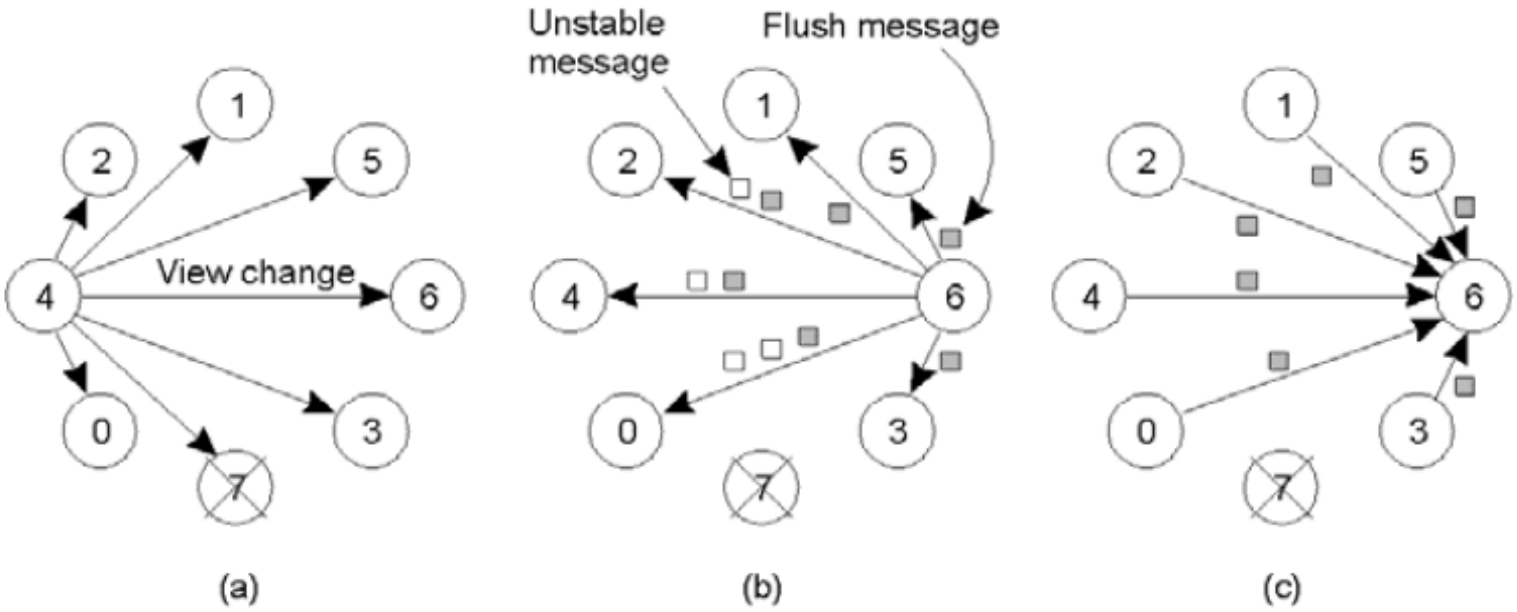
- Group multicast communication and group membership services built on top of TCP

- Multicast semantics supported:
- abcast: view-synchronous totally-ordered group multicast
- cbcast: view-synchronous causally-ordered group multicast
- cabast: view-synchronous causally and totally-ordered group multicast

Implementing VS with ISIS

- Main issue: ensure that all messages sent to view G are delivered to all non-faulty processes in G before next group view G is installed.
- If the sender fails before its message m is delivered to all members, processes that haven't received the message must get it from other processes.
- How? Every process in G keeps m until it knows that all members in G have received it, i.e., m is **stable**.
- Upon receiving a view change message, a process forwards all of its unstable messages and then send a flush message for the new view.
- After a process receives a flush message for the new view from each other process, it can safely install the new view.

Implementing Virtual Synchrony



a) Process 4 notices that process 7 has crashed, sends a view change

b) Process 6 sends out all its unstable messages, followed by a flush message

c) Process 6 installs the new view when it has received a flush message from everyone else

What if ?

- Another process fails during a view change?

• Protocol fails – no flush message received from failed process

- How to fix?

• Announce new view changes while old ones are pending
• After messages are stable in most recently announced view change, install new view

JGroups

The end