Group Communication

A **group** is a collection of users sharing some common interest.Group-based activities are steadily increasing.

There are many types of groups:

- **Open** group (anyone can join, **customers of Walmart**)
- **Closed** groups (membership is closed, **class of 2000**)
- **Peer-to-peer** group (all have equal status, **graduate students of CS department, members in a videoconferencing / netmeeting**)
- **Hierarchical** groups (one or more members are distinguished from the rest. **President and the employees of a company, distance learning**).

Major issues

Various forms of multicast to communicate with the members.

Examples are

- Atomic multicast
- Ordered multicast

Dynamic groups

- How to correctly communicate when the membership constantly changes?
- Keeping track of membership changes

Atomic multicast

A multicast is ***atomic***, when the message is delivered to **every correct** member, or to **no member** at all.

In general, processes may crash, yet the atomicity of the multicast is to be guaranteed.

How can we implement atomic multicast?

Basic vs. reliable multicast

Basic multicast does *not* consider failures.

Reliable multicast tolerates (certain kinds of) failures.

Three criteria for **basic multicast**:

Liveness.	Each process must receive every message
Integrity.	No spurious message received
No duplicate.	Accepts exactly one copy of a message

Reliable atomic multicast

Sender's program

```
i:=0;
do  i ≠ n →
   send message to member [i];
   i:= i+1
od
```

Receiver's program

```
if   m is new →
           accept it;
           multicast m;
 [] m is duplicate → discard m
fi
```

Tolerates process crashes. Why does it work?

Multicast support in networks

Multicast is an extremely important operation in networking, and there are numerous protocols for multicast.

Sometimes, certain features available in the infrastructure of a network simplify the implementation of multicast. Examples are

- Multicast on an ethernet LAN
- IP multicast for wide area networks

IP Multicast

IP *multicast* is a bandwidth-conserving technology where the **router** reduces traffic by **replicating a single stream of information and forwarding them** to multiple clients. Very popular for for streaming content distribution.

Sender sends a single copy to a **special multicast IP address** (Class D) that represents a group, where other members register.

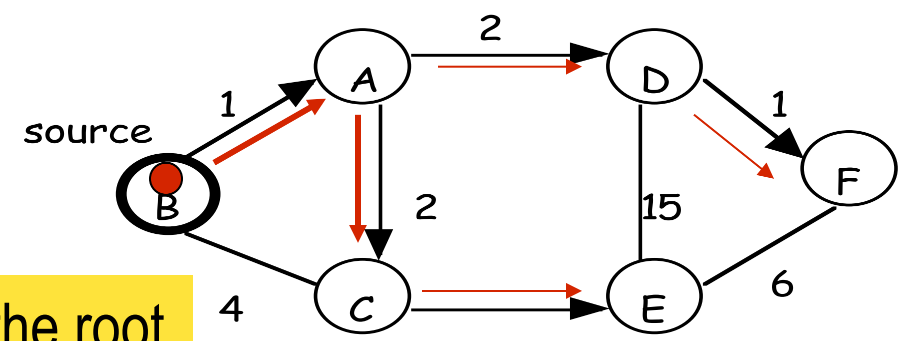
IP Multicast

Internet radio. BBC has encouraged UK-based Internet service providers to adopt multicast-addressable services in their networks by providing BBC Radio at higher quality than is available via their unicast-addressed services.

Instant movie or TV. Netflix uses a version of IP multicast for the instant streaming of their movies at homes through the Blu-ray Disc players

Distance learning. Involves live transmission of the course material to a large number of students at geographically distributed locations.

Distribution trees

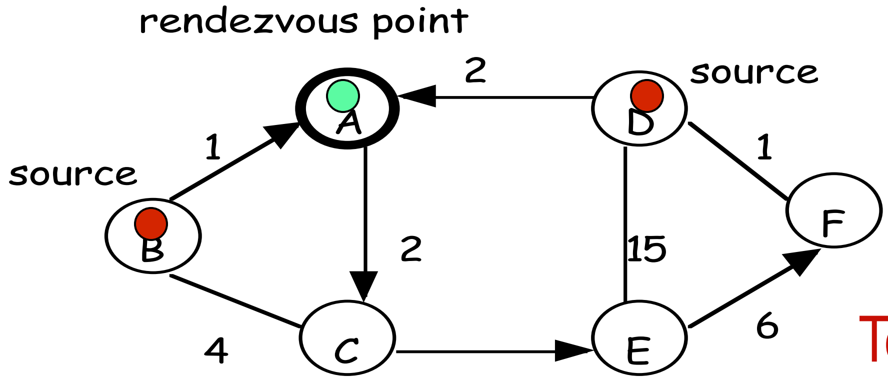


Source is the root of a spanning tree

(a) Source tree

Routers maintain & update distribution trees whenever members join / leave a group

Uses shortest path trees.
One tree for each source



(b) Shared tree

All multicasts are Routed via a Rendezvous point. The shared tree is also called core-based tree.

Too much load on routers.
Application layer multicast overcomes this.

Distribution trees

The routers have to know about group composition. As groups memberships change, routers have to be updated.

Too much load on routers.

Application layer multicast overcomes this. The responsibility of multicast is left to the applications layer. Each multicast is implemented as a series of unicasts.

Reverse Path Forwarding

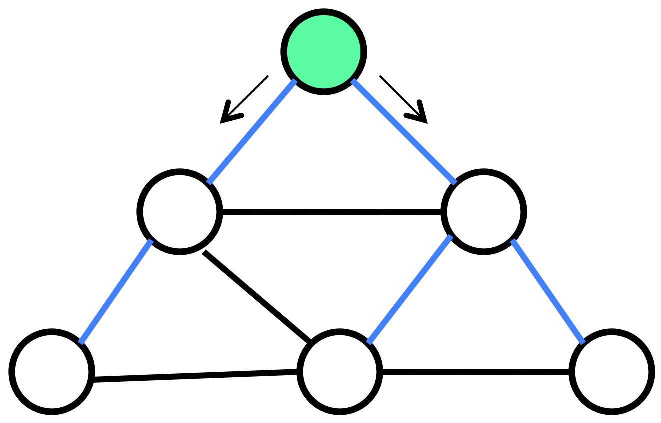
RPF is a multicast algorithm that uses both source and destination addresses. Each recipient of a packet from a source node

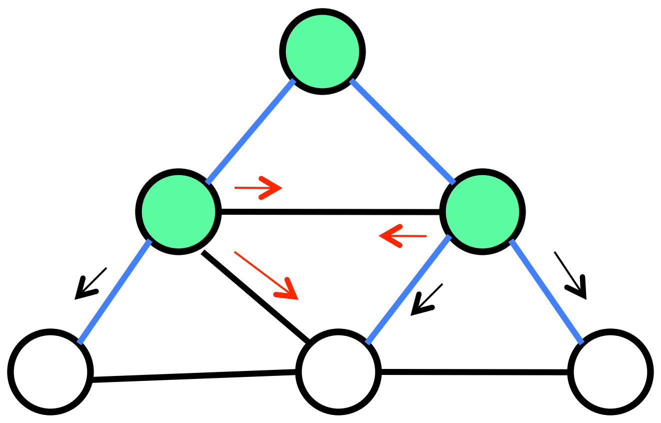
- looks up source address in its own routing table
- compares route entry with receiving interface
- if wrong interface (other than the shortest path port), drop

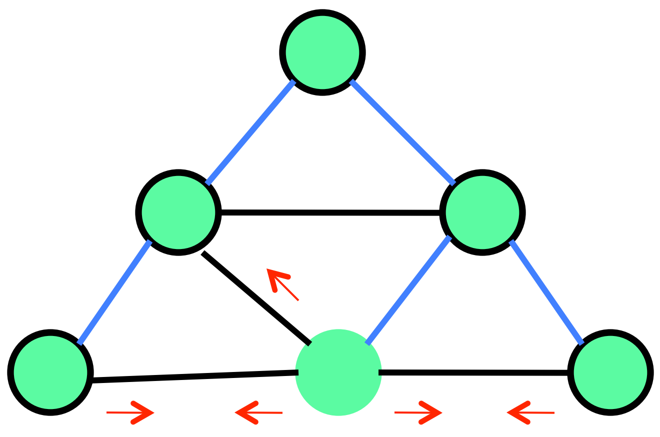
(**Note**: the packet must be received on the interface that the router would use to forward the return packet. This prevents possible looping as in flooding, IP spoofing etc.)

- for each outgoing interface with group members downstream, forward the packet.

Reverse Path Forwarding







The blue link denotes the *first hop* for the return path to the source. The messages represented by the red arrows will be dropped

Ordered multicasts:
Basic versions only

Total order multicast. Every member must receive **all updates** in the same order. Example: consistent update of replicated data on servers

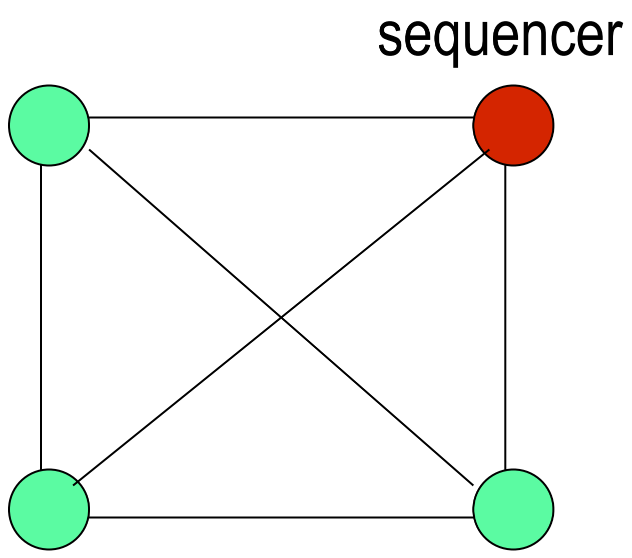
Causal order multicast. If a, b are two updates and **a happened before b**, then every member must accept a before accepting b. Example: *implementation of a bulletin board*.

Local order (a.k.a. *Single source FIFO*). Example: video distribution, distance learning using “push technology.”

Implementing total order multicast

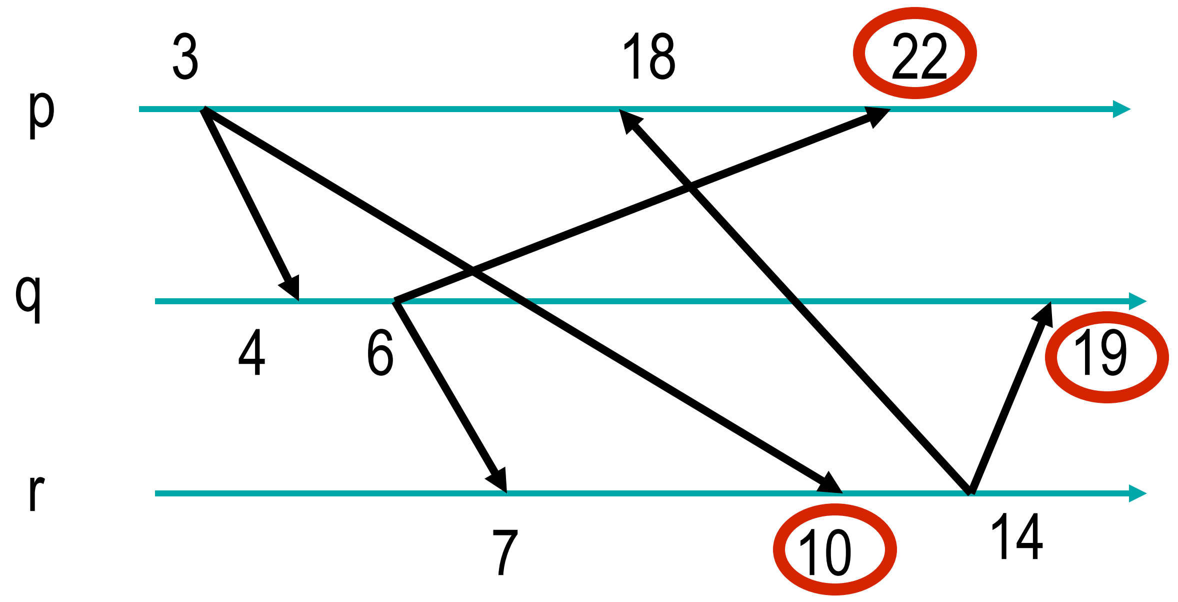
First method. Basic multicast using a sequencer

```
{The sequencer S}
define seq: integer (initially 0}
do  receive m →
    multicast (m, seq);
    seq := seq+1;
    deliver m
od
```



Implementing total order multicast

Second method. Basic multicast without a sequencer.
Uses the idea of 2PC (two-phase commit)



Implementing total order multicast

Step 1. Sender **i** sends (**m, ts**) to all

Step 2. Receiver **j** saves it in a *holdback queue*, and sends an ack (**a, ts**)

Step 3. Receive all acks, and pick the largest **ts**. Then send (**m, ts, commit**) to all.

Step 4. Receiver removes it from the holdback queue and delivers **m** in the ascending order of timestamps.

Why does it work?

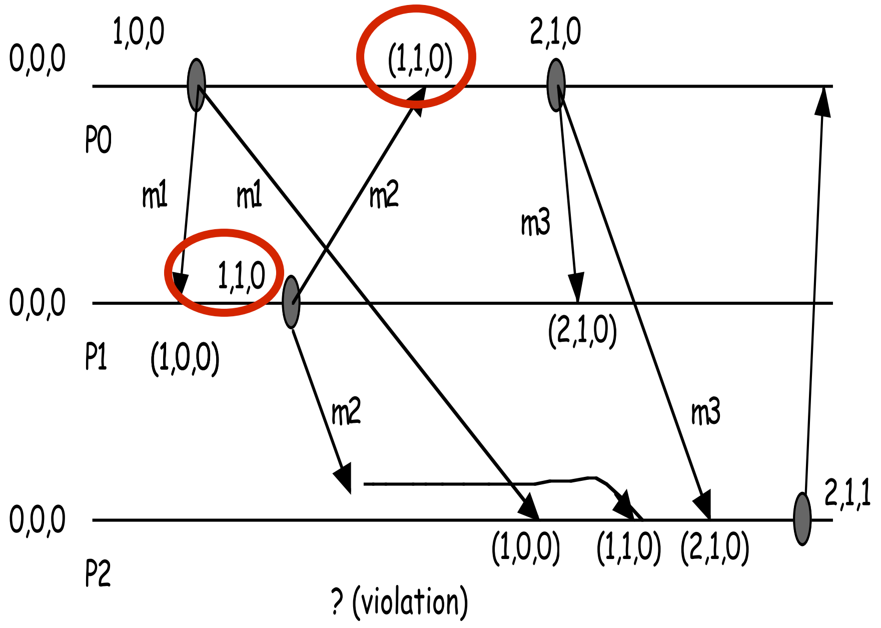
Implementing causal order multicast

Basic multicast only. Uses vector clocks. Recipient **i will** deliver a message from **j** iff

1. $VC_j(j) = LC_j(i) + 1$
{LC = local vector clock}

2. $\forall k: k \neq j :: VC_k(j) \leq LC_k(i)$

VC = incoming vector clock
LC = Local vector clock



Note the slight difference in the implementation of the vector clocks

Reliable multicast

Tolerates process crashes. The additional requirements are:

Only **correct processes** are required to receive the messages from **all correct processes** in the group. Multicasts by faulty processes will either be received by every correct process, or by none at all.

A theorem on reliable multicast

Theorem.

In an *asynchronous* distributed system, total order reliable multicasts cannot be implemented when even a single process undergoes a crash failure.

(Hint) The implementation will violate the FLP impossibility result. Complete the arguments!

Scalable Reliable Multicast

IP multicast or application layer multicast provides *unreliable datagram* service. Reliability requires the detection of the message omission followed by retransmission. This can be done using **ack**. However, for large groups (as in distance learning applications or software distribution) **scalability is a major problem.**

The reduction of acknowledgements and retransmissions is the main contribution in ***Scalable Reliable Multicasts*** (SRM) (Floyd et. al).

Scalable Reliable Multicast

If omission failures are rare, then then instead of using **ACK**, receivers will only report the *non-receipt* of messages using **NACK**.

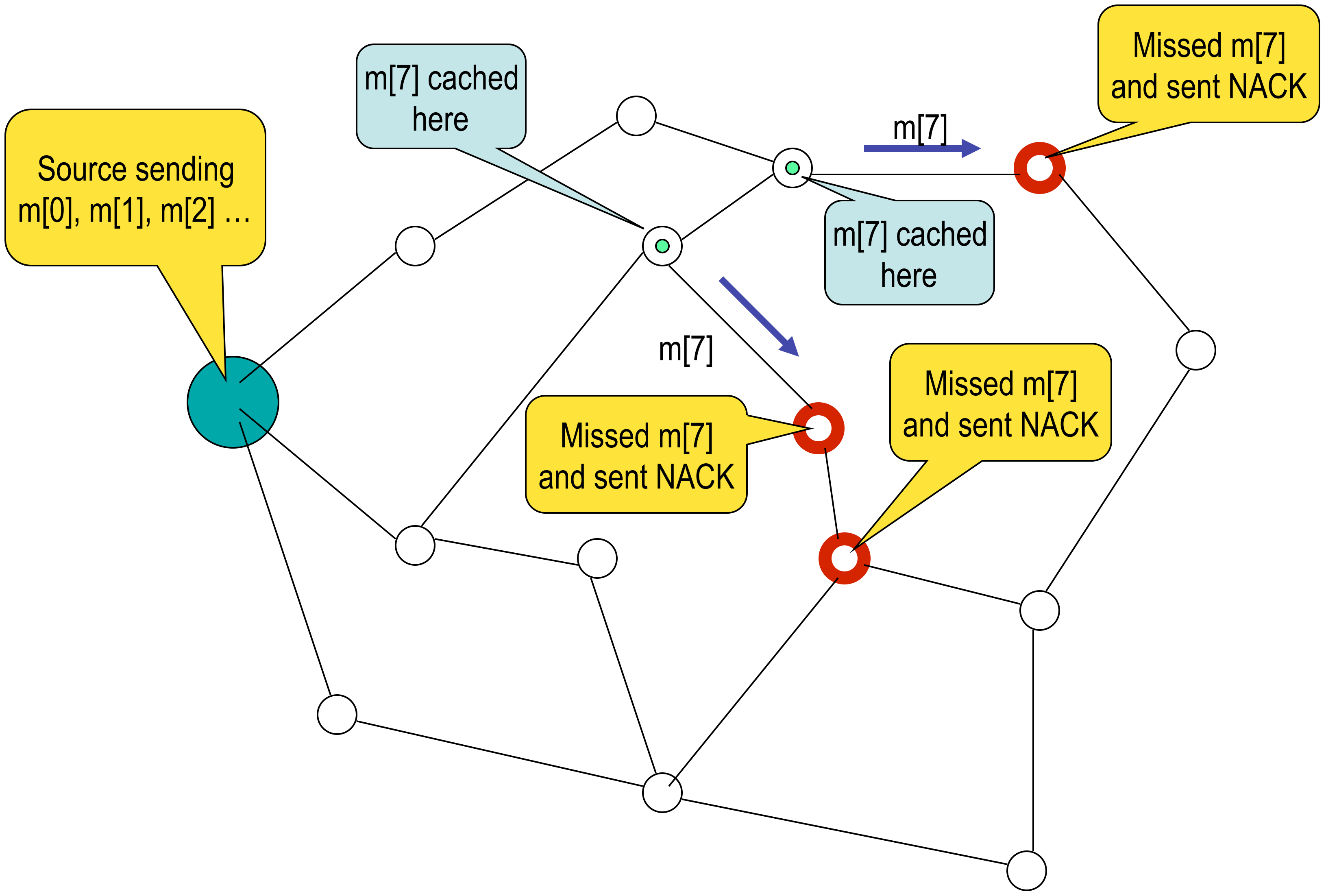
If several members of a group fail to receive a message, then each such member waits for a random period of time before sending its NACK. This helps to suppress redundant NACKs. Sender multicasts the missing copy only once.

Use of cached copies in the network and selective point-to-point retransmission further reduces the traffic.

Scalable Reliable Multicast

Receiving processes have to detect the non receipt of messages from the source. Each member periodically broadcasts a *sessions message* containing the largest sequence number received by it. Other members figure out which messages they did not receive.

Scalable Reliable Multicast


Source sending m[0], m[1], m[2] …
m[7] cached here
m[7]
Missed m[7] and sent NACK
m[7] cached here
m[7]
Missed m[7] and sent NACK
Missed m[7] and sent NACK

Dealing with open groups

The ***view*** of a process is its current knowledge of the membership.
It is important that all processes have identical views.
Inconsistent views can lead to problems. Example:

Four members (0,1,2,3) will send out 144 emails.

Assume that 3 left the group but only 2 knows about it. So,
0 will send 144/4 = 36 emails (first quarter 1-36)
1 will send 144/4 = 36 emails (second quarter 37-71)
2 will send 144/3 = 48 emails (last one-third 97-144)
3 has left. The mails 72-96 will not be delivered!

Dealing with open groups

Views can change unpredictably, and no member may have exact information about who joined and who left at any given time.

These views and their changes should propagate in the same order to all members.

Dealing with open groups

Example. Current view (of all processes) $v_0(g) = \{0, 1, 2, 3\}$.
Let 1, 2 leave and 4 join the group concurrently. This view change can be serialized in many ways:

- {0,1,2,3}, {0,1,3} {0,3,4}, OR
- {0,1,2,3}, {0,2,3}, {0,3}, {0,3,4}, OR
- {0,1,2,3}, {0,3}, {0,3,4}

To make sure that every member observe these changes in the same order, changes in the view should be sent via total order multicast.

View propagation

{Process 0}:

- $v_0(g)$;
- send m1, ... ;
- $v_1(g)$;
- send m2, send m3;
- $v_2(g)$;

{Process 1}:

- $v_0(g)$;
- send m4, send m5;
- $v_1(g)$;
- send m6;
- $v_2(g)$ ...;

$v_0(g) = \{0,1,2,3\}$,

$v_1(g) = \{0,1,3\}$,

$v_2(g) = \{0,3,4\}$

View delivery guidelines

Rule 1. If a process **j** joins and continues its membership in a group **g** that already contains a process **i**, then *eventually* **j** *appears in all views* delivered by process **i.**

Rule 2. If a process ***j*** *permanently leaves* a group **g** that contains a process **i**, then *eventually* ***j*** *is excluded from all views* delivered by process **i**.

View-synchronous communication

Rule. *With respect to each message, all correct processes have the same view.*

m sent in view V ⇒ m received in view V

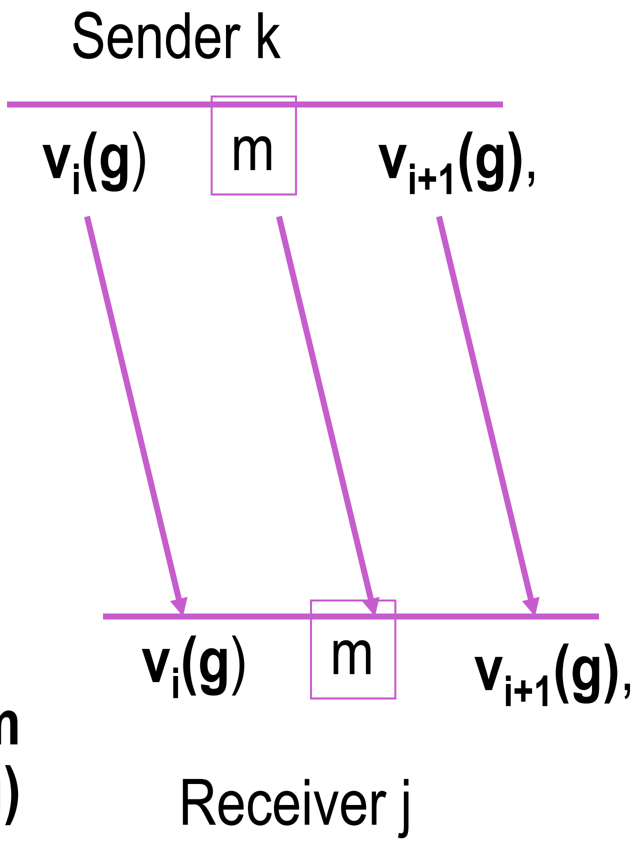
This is also known as **virtual synchrony**

View-synchronous communication

Agreement. If a correct process **k** delivers a message **m** in **v_i(g**) before delivering the next view **v_{i+1}(g)**, then every correct process **j** $\in$ **v_i(g)** $\cap$ **v_{i+1}(g)** must deliver **m** before delivering **v_{i+1}(g).**

Integrity. If a process **j** delivers a view **v_i(g**), then **v_i(g**) must include **j**.

Validity. If a process **k** delivers a message **m** in view **v_i(g)** and another process **j** $\in$ **v_i(g)** does not deliver that message **m**, then the next view **v_{i+1}(g)** delivered by **k** must exclude **j**.



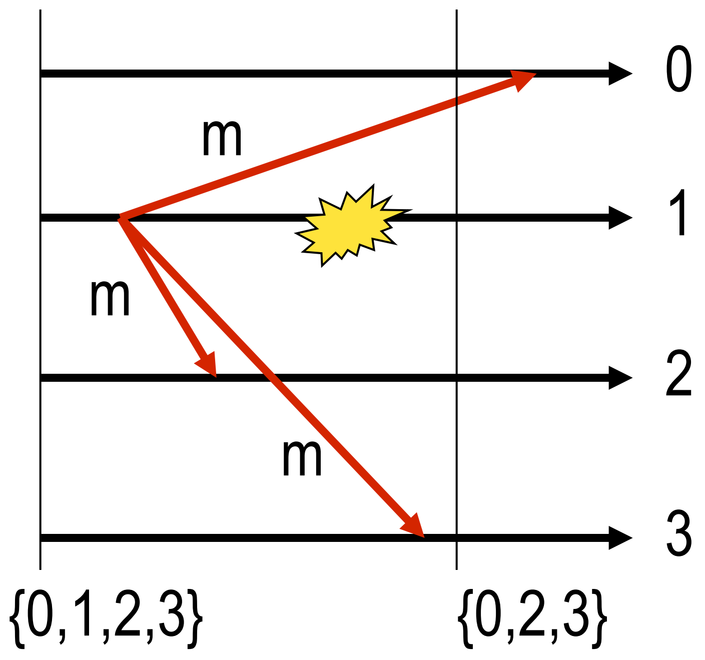
Example

Let process 1 deliver m and then crash.

Possibility 1. No one delivers **m**, but each delivers the new view {0,2,3}.

Possibility 2. Processes 0, 2, 3 deliver **m** and then deliver the new view {0,2,3}

Possibility 3. Processes 2, 3 deliver **m** and then deliver the new view {0,2,3} but process 0 first delivers the view {0,2,3} and then delivers **m.**



Possibility 3

Are these acceptable?

Overview of Transis

What is Transis?

A group communication system developed by Danny Dolev and his group at the Hebrew University of Jerusalem. (see http://www.cs.huji.ac.il/labs/transis/). Important objectives of Transis are to develop a framework for Computer Supported Cooperative Work (CSCW) applications, such as multi-media and desktop conferencing, as well as a scalable Video-on-demand service.

Overview of Transis (2)

What is Transis?

- Deals with **open group**
- Supports **scalable reliable multicast**
- Tackles **network partition.** Allows the partitions to continue working and later restore consistency upon merger

Overview of Transis (3)

1. IP multicast (and ethernet LAN) used to support high bandwidth multicast.

2. **ACK and NACK are piggybacked** with the next message and message loss is detected transparently, leading to ***selective retransmission.*** Example:

(**Notation:** Let A, B, C, D … be the different processes in a System. b_k is the ack of message B_k sent by process B. a_2B_1 denotes the ack of A_2 piggybacked on B_1)

A process that receives A_1, A_2, a_2B_1, b_3C_1 … suspects that it did not receive message B_2 and B_3, and sends a NACK to request their retransmission.

It will postpone sending c_1 until it received B_2 and B_3

Overview of Transis (4)

FIFO (single source)

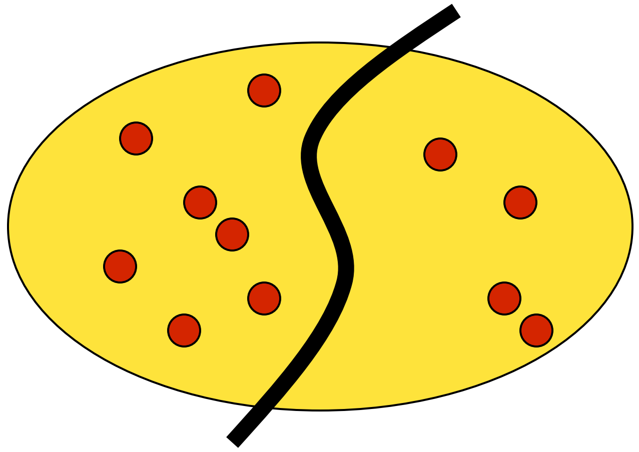
Causal mode (maintains causal order)

Agreed mode (maintains total order that does not conflict with the causal order)

Safe mode (Delivers a message only after all processes in the current configuration have acknowledged its reception) If safe message M is delivered in a configuration that includes process P, then P will deliver M unless it crashes.

Overview of Transis (5)

Dealing with partition



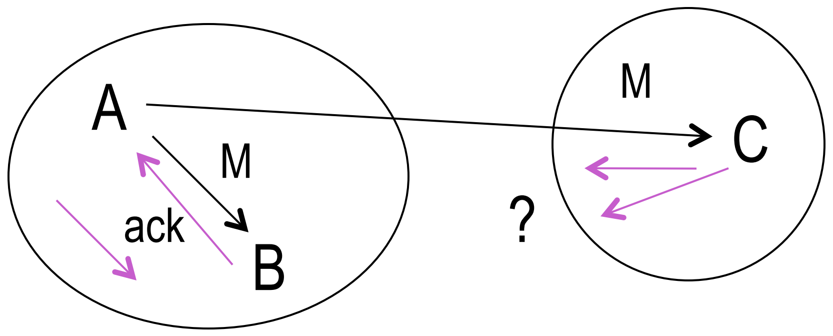
Each partition assumes that the machines in the other partition have failed, and maintains virtual synchrony within its own partition only.

After repair, consistency is restored in the entire system.

Dealing with partitions

Assume that when A was sending a **safe message** M, the configuration changed to **{A, B, C} → {A, B}, {C}.**

Case 1. All but C sent ack to A, B. Now, to deliver M, A, B must receive the new view {A,B} first.

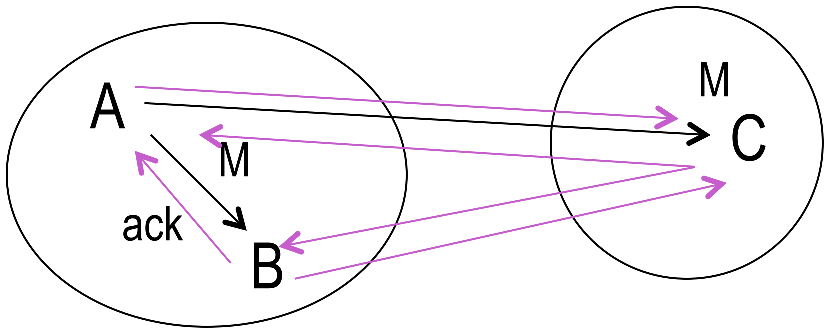


Extended virtual synchrony

Dealing with partitions

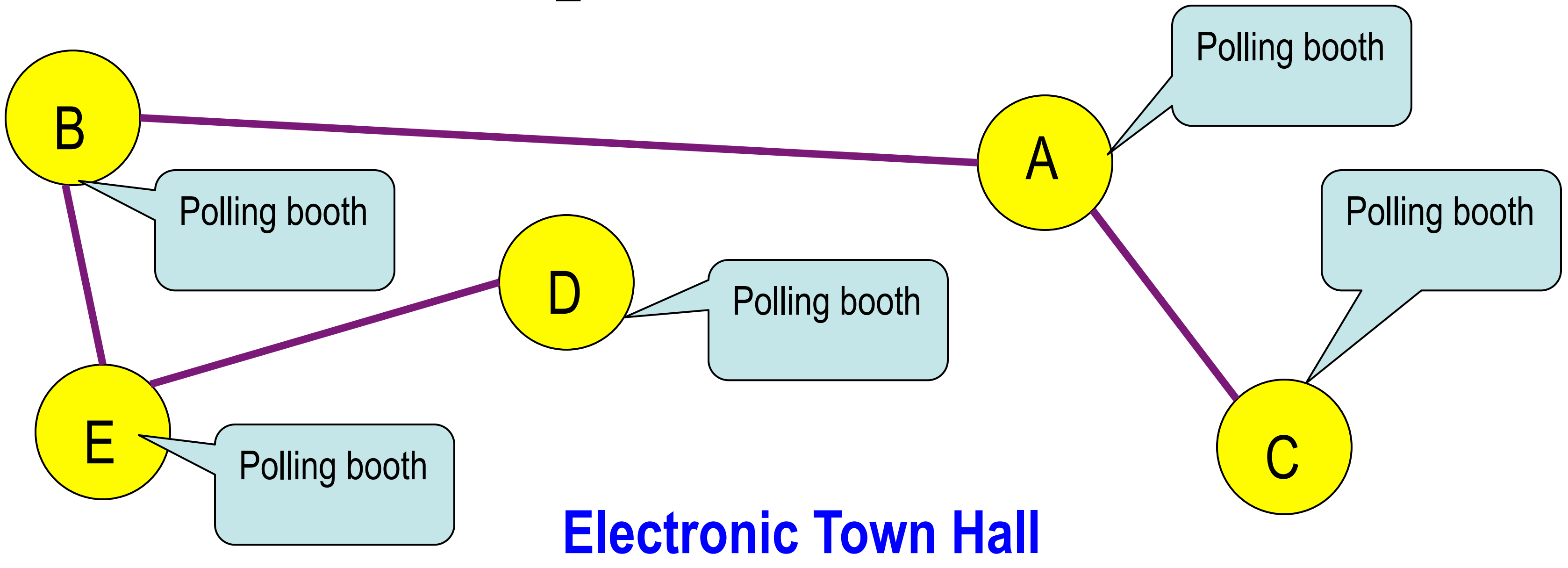
Assume A was sending a **safe message** M, and the configuration changed to **{A, B, C}** → **{A, B}, {C}.**

Case 2. If C ack'ed M and also received acks from A and B *before the partition*, then C will deliver M before it receives the new view {C}.



Otherwise, C will ignore message M as spurious without contradicting any guarantee.

Continuing operation in spite of a partition



Votes are counted manually at each booth. The count of **each vote** is multicast to every other booth, so that the latest count is displayed at each booth. Apparently, if a single wire breaks, the counting will stop.

However, it is silly. Counting could easily continue at the individual booths, and the results could be merged later. This is the Transis approach: supporting operations within partitions whenever possible.