Fault Tolerance – Reliable Group Communication

timofei.istomin@unitn.it

Thanks for the slides to:

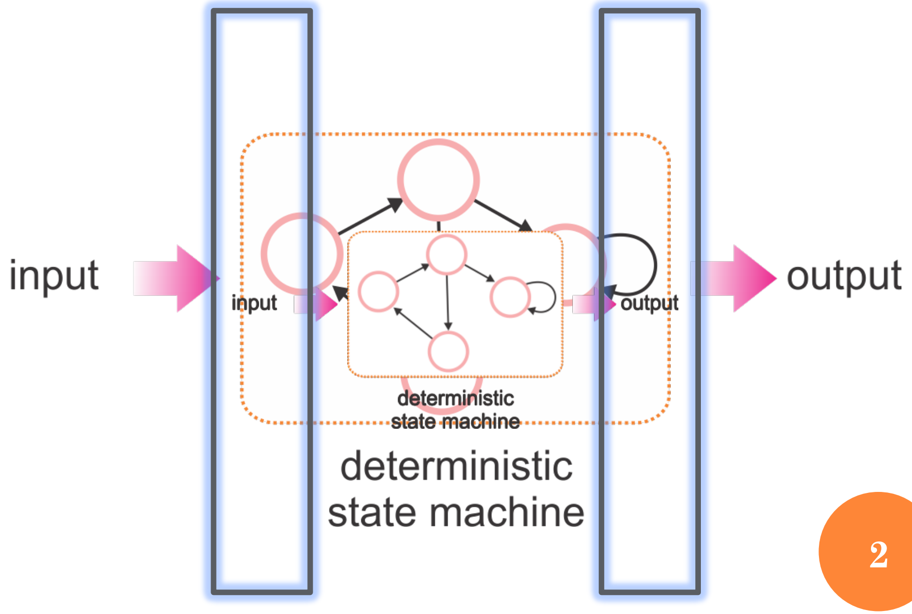
Giuliano Mega

Davide Frey

Luca Mottola

Reliable group communication

- It is of critical importance if we want to exploit process resilience by replication
- Classical example: the *state machine* approach;
  - make your process a *deterministic state machine*, so that…
  - … given *n* copies of your process, if you feed them the same inputs:
    1. they will emit the same outputs
    2. their internal state will be the same



Reliable group communication

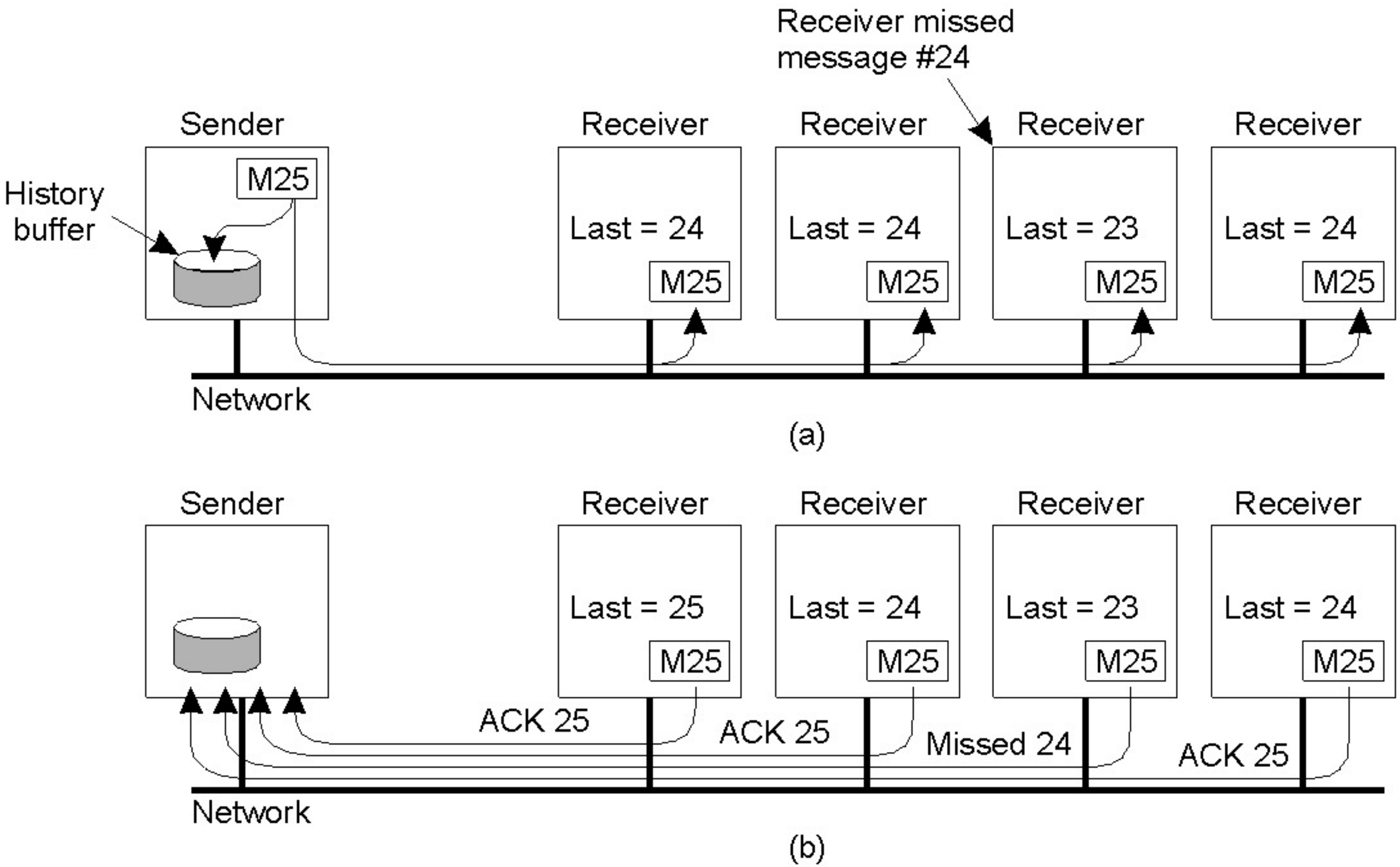
- Reliable multicast is surprisingly tricky
  - Yet, we need it to deliver messages to all members in a process group
- Usually, all we have is reliable point-to-point service or unreliable multicast service
  - Achieving reliable multicast through multiple reliable point-to-point channels may be not efficient
- Plus…
  - … what happens if the sender crashes while sending?
  - What if the group changes while a message is being sent? Who should get the message, and who should not?

Reliable group communication

- We look at two cases for reliable multicast
- **Case 1:** groups are fixed (i.e., processes do not crash), communication failures only
  - All group members should receive the multicast
  - Unreliable multicast primitive is available
- **Case 2:** faulty processes, but reliable channels

- We start from **Case 1**

BASIC RELIABLE MULTICAST

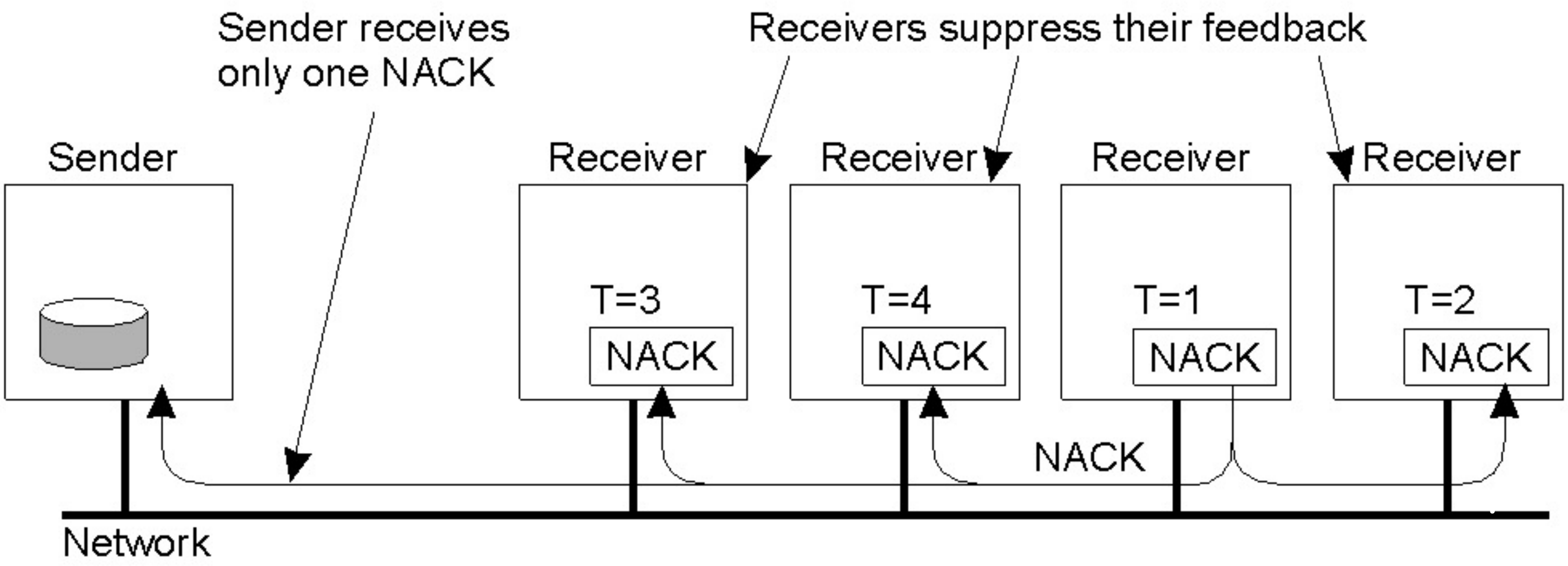


Basic Reliable Multicast

- What if we make the receivers ACK all messages?
- Key problem: too many ACKs:
  - "feedback implosion"
- Mitigation strategies:
  - piggyback ACK messages on traffic (only works if traffic is constant enough), and
  - retransmit using point-to-point primitive
- Scalability problems still persist
- Negative ACKs: only send feedback if message not received
  - sender might have to cache messages forever
  - still may result in too many ACKs

Scalable Reliable Multicast

- A first solution is non-hierarchical feedback control, implemented in SRM
  - Negative acknowledgements are multicast
  - Randomly staggered
- Used in practice in several applications

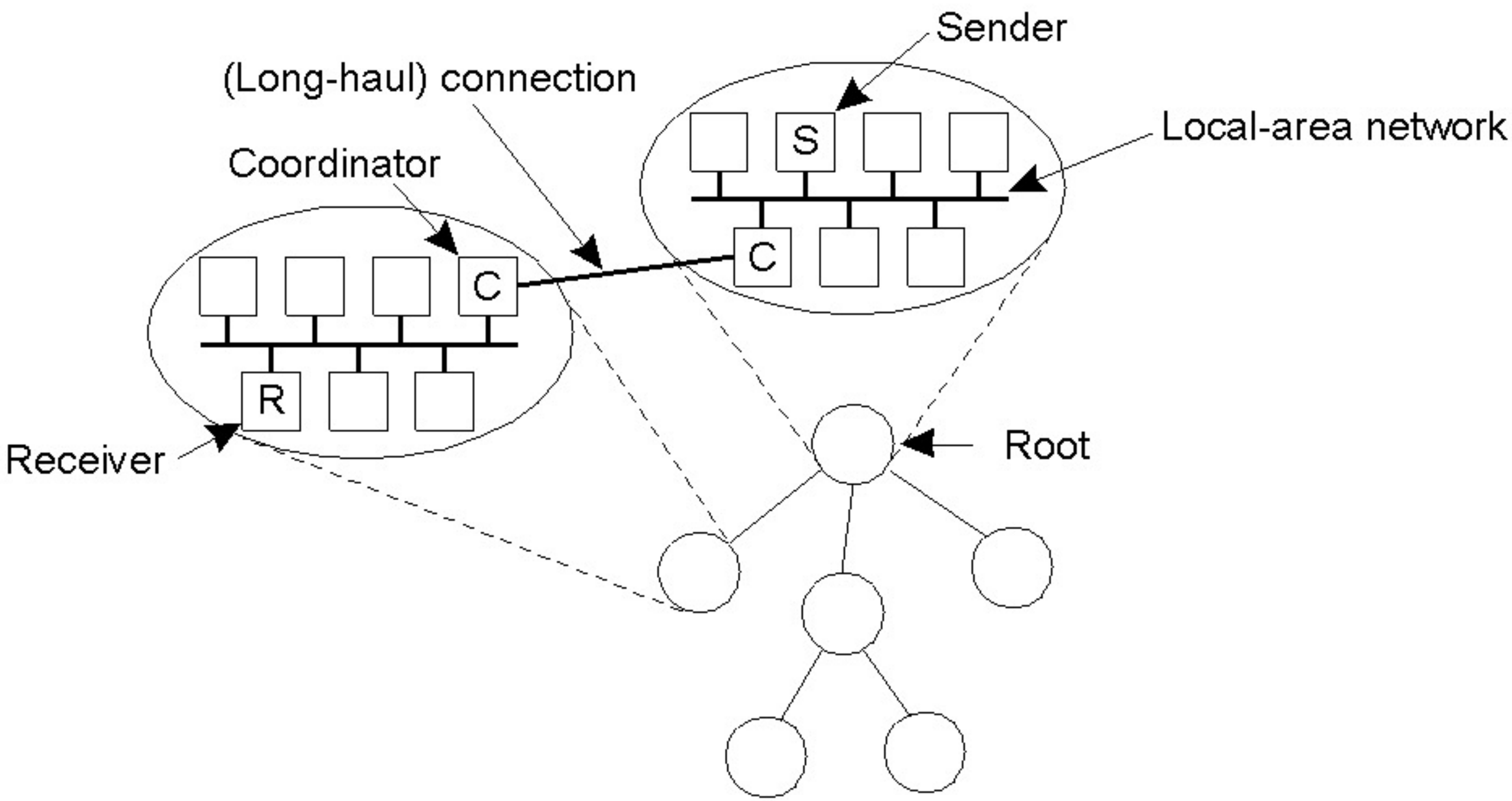


SCALABLE RELIABLE MULTICAST

- Staggering “the right amount” is non-trivial
  - depends on network delays, right value might be group-dependent, especially over WANs
- Processes that don’t lose messages still receive NACKs from those who do
  - Dynamically create separate groups for processes that tend to miss messages together, difficult to achieve in large-scale settings
- Ultimately, flat groups will always have scalability limitations

Hierarchical Feedback Control

- With Hierarchical Feedback control, receivers are organized in groups headed by a coordinator



Hierarchical Feedback Control

- The coordinator can adopt any strategy within its group
- In addition it can request retransmissions to its parent coordinator
  - A coordinator can remove a message from its buffer if it has received an ACK from
    - All the receivers in its group
    - All of its child coordinators
- The problem is that the hierarchy has to be constructed and maintained
  - Difficult to implement on physical networks, usually done via application-level overlays

Reliable Multicast

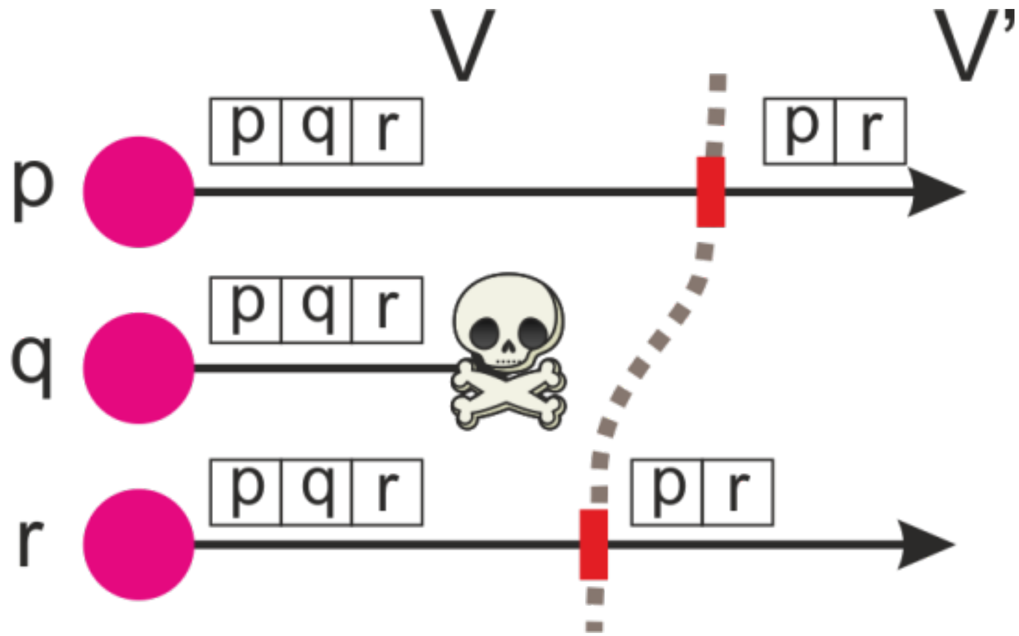
- Recall – two cases:
  - **Case 1:** fixed groups, unreliable channels;
  - **Case 2:** dynamic groups (processes can join, leave or crash), reliable channels

- Now we look at **Case 2**

- The main problem is **dynamic membership**:
  - group membership may change while multicasts are being issued;
  - to simplify the application programmer's life the middleware can provide **virtual synchrony (VS)**

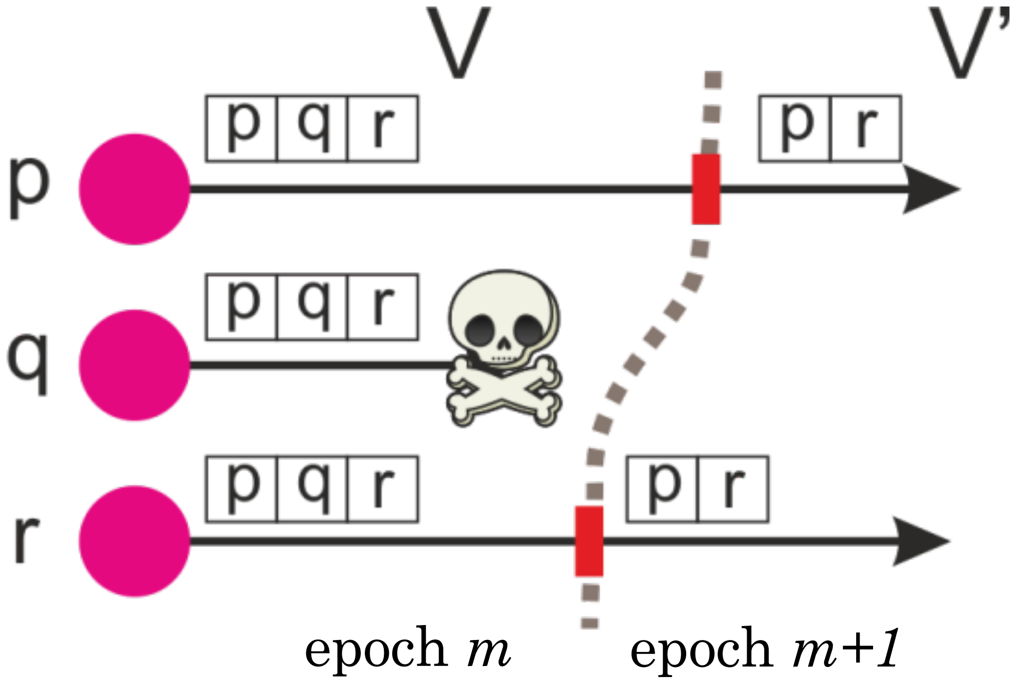
VIEWS AND VIEW SYNCHRONY

- In VS, each process maintains a "view" of the group
  - i.e., local knowledge of who is part of the group, **and therefore to whom messages should be delivered to**
- Views "get replaced" as the group changes: processes *install new views* when a new member joins the group or someone leaves the group



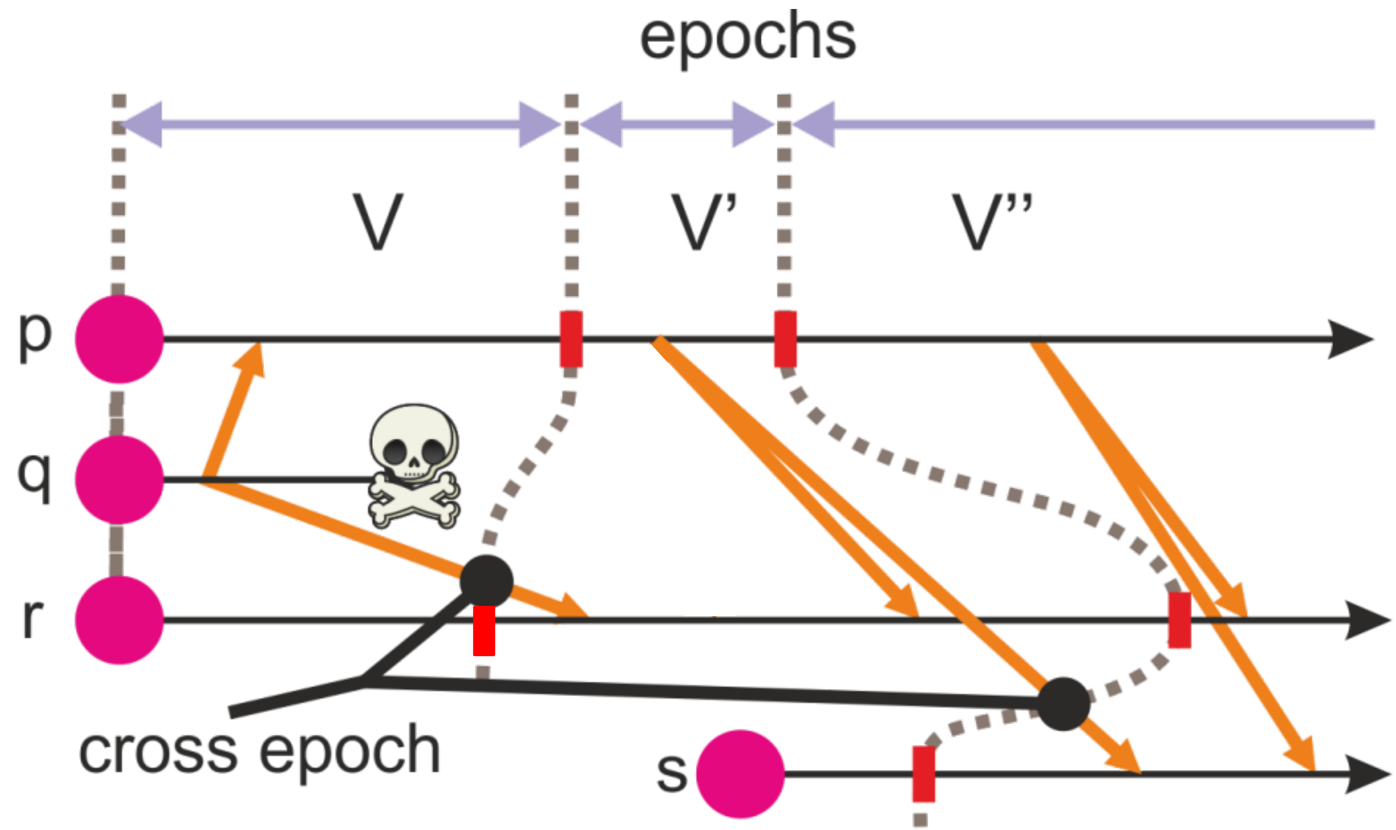
Views and View Synchrony

- The system guarantees (under some assumptions) **that all correct processes will see the same sequence of group views**
- Therefore, each view defines a global **epoch**



VIEWS AND VIEW SYNCHRONY

- Another important property guaranteed by virtual synchrony: **multicasts do not cross epoch boundaries**

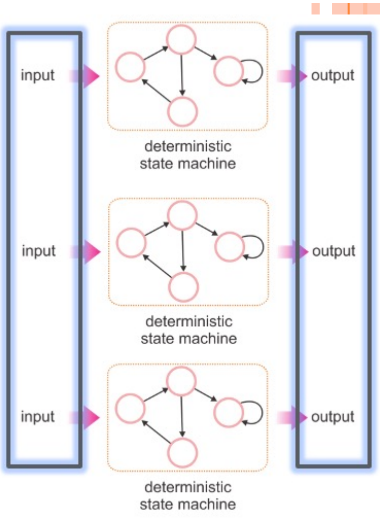


V={p,q,r}

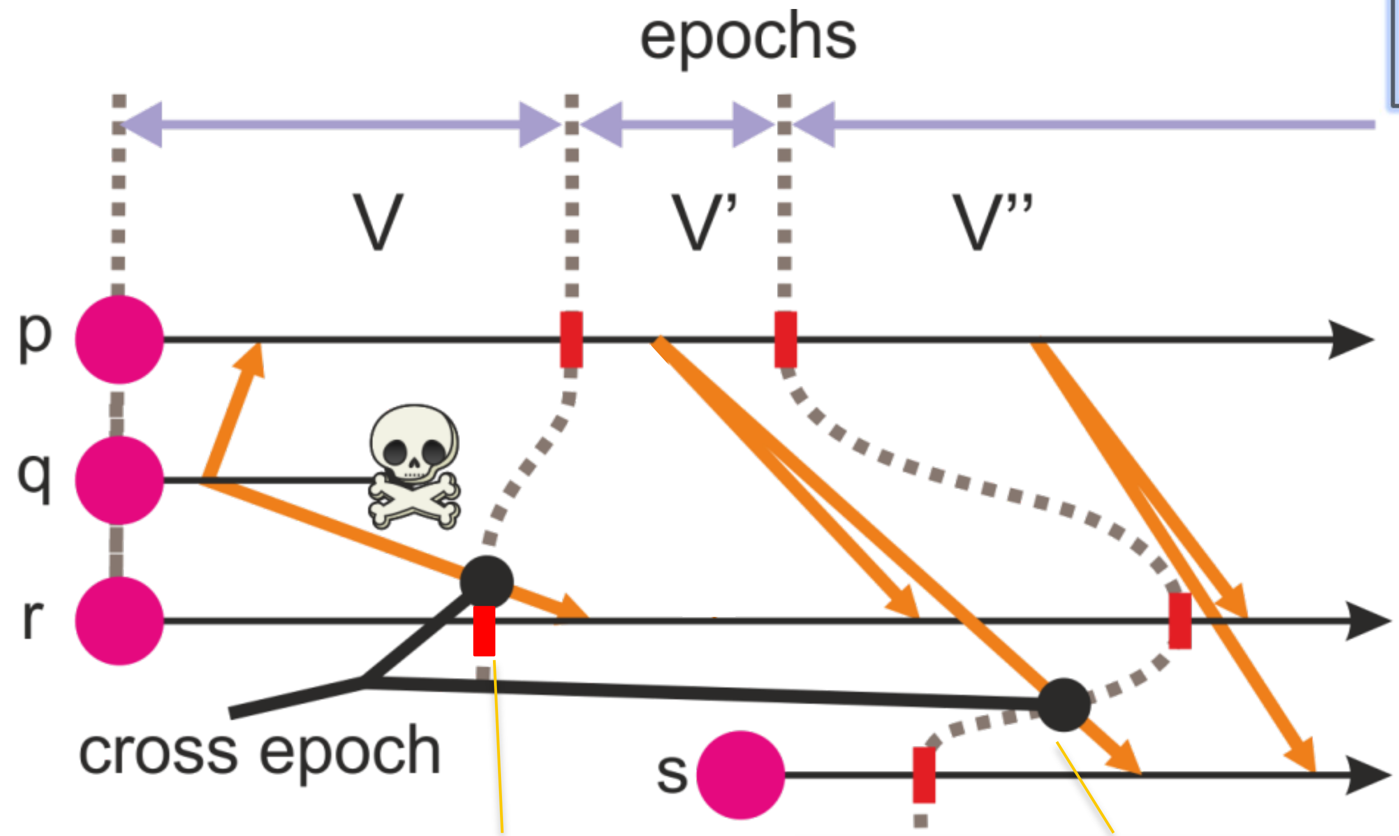
V'={p,r}

V''={p,r,s}

VIEWS AND VIEW SYNCHRONY



- Why crossing epoch boundaries is a problem?



V={p,q,r}

V'={p,r}

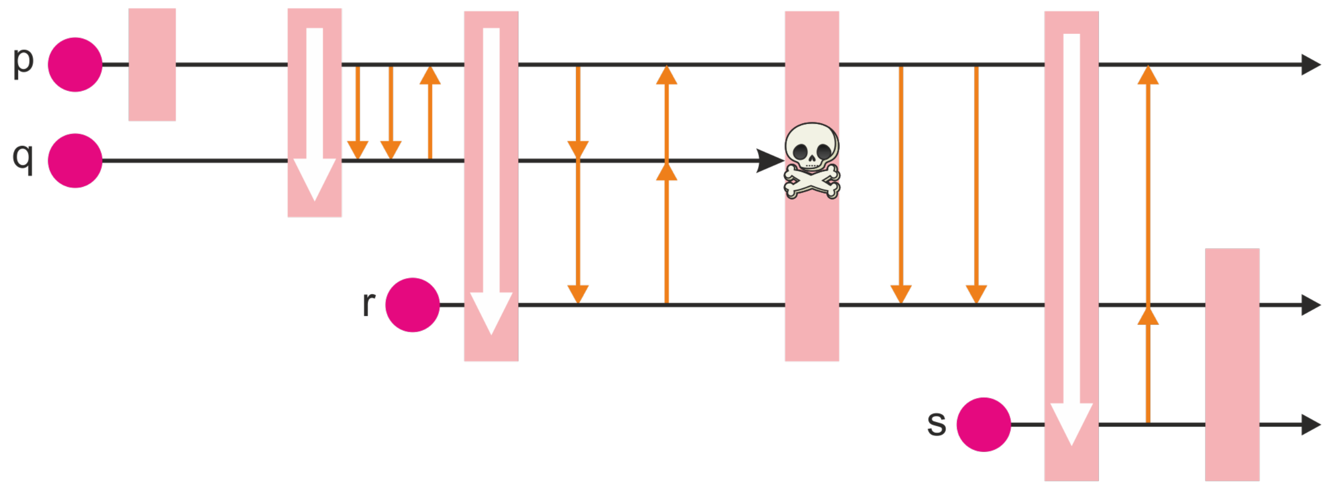
V''={p,r,s}

R does not have the same information as P does when it gets informed about Q's crash

P and R are not sure what information S have received since it has joined

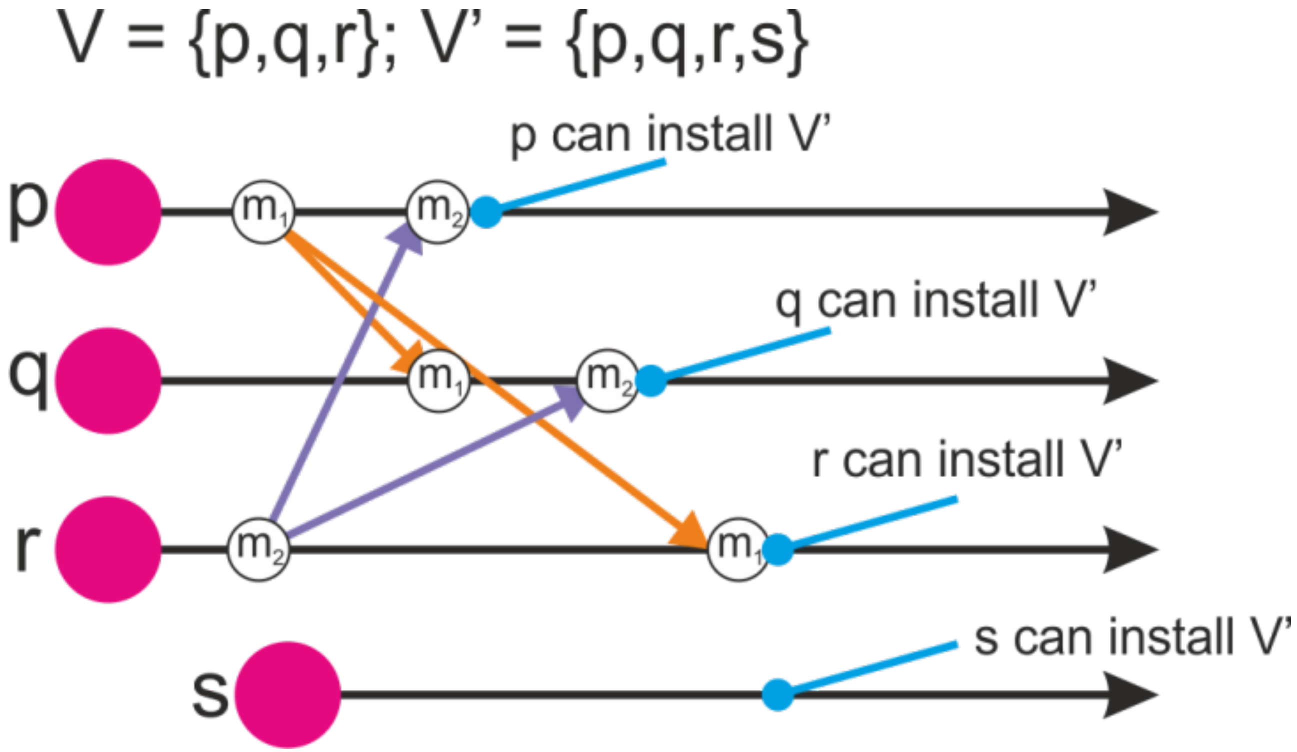
Close Synchrony

- Virtual synchrony *emulates* close synchrony where the events (multicasts, group changes) are *instantaneous*
- Easy to reason about, since crashes or joins never overlap with message delivery
- Multicasts also reliable: always get to everyone in the group



Views and View Synchrony

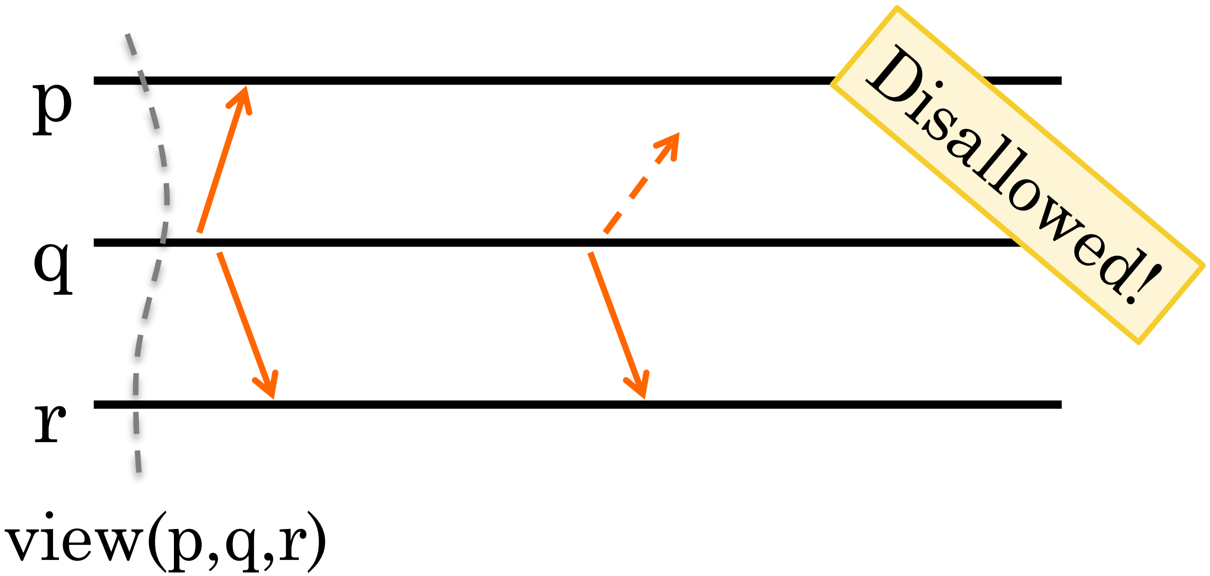
- "Multicasts do not cross epoch boundaries"
  - If there are multicasts for messages m_1 and m_2, and a view change occurs …
  - … to satisfy view synchrony, these multicasts **must complete** before the new view is installed

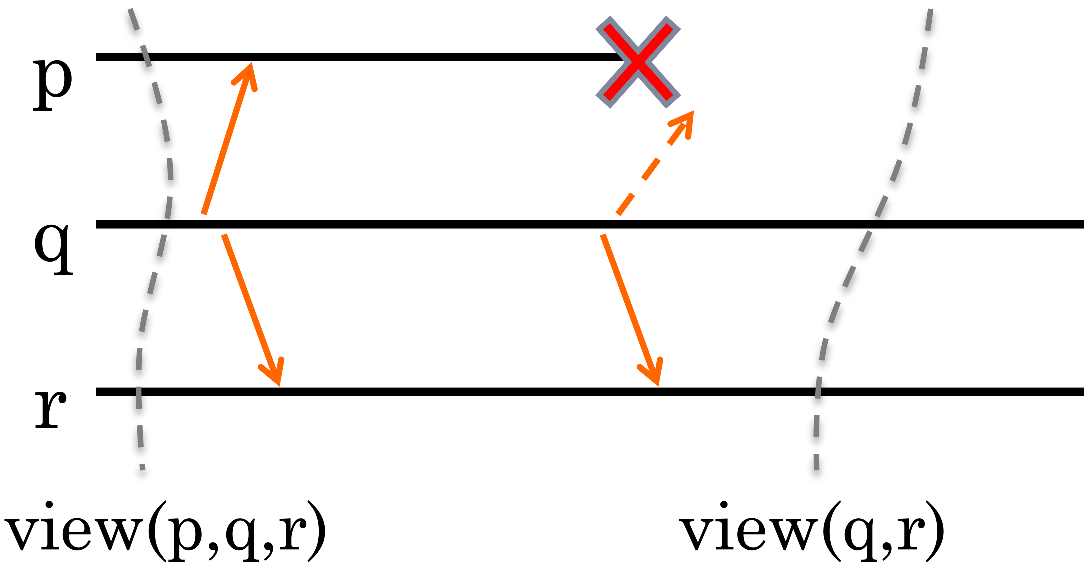


Multicasts in Virtual Synchrony

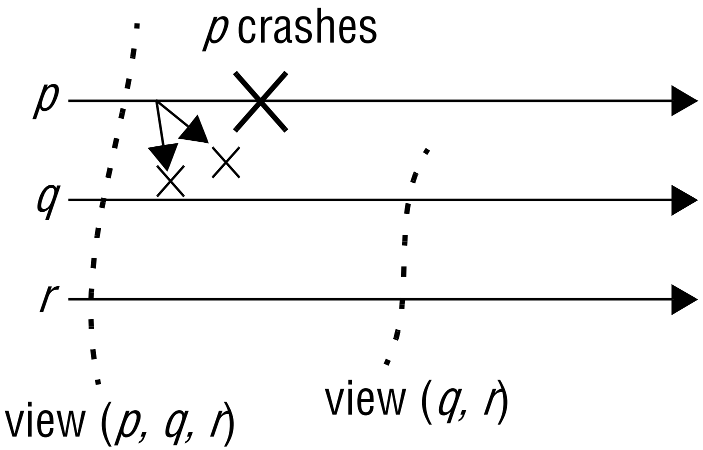
- What does it mean for multicast to "complete"? Who has to get it or not get it?
- Virtual synchrony multicasts are **reliable**
- A message multicast in an epoch E defined by view V should either:
  - be **delivered** within E by <u>all</u> **operational** participants
  - be **delivered** by <u>none</u> of the **operational** participants
  - One of these conditions is met: multicast is complete
- To exclude trivial protocols (just discard everything), the *"or none"* case <u>is only allowed if the sender crashes</u>
  - All "evidence" erased: as if it had never been sent
  - We only care about operational participants…

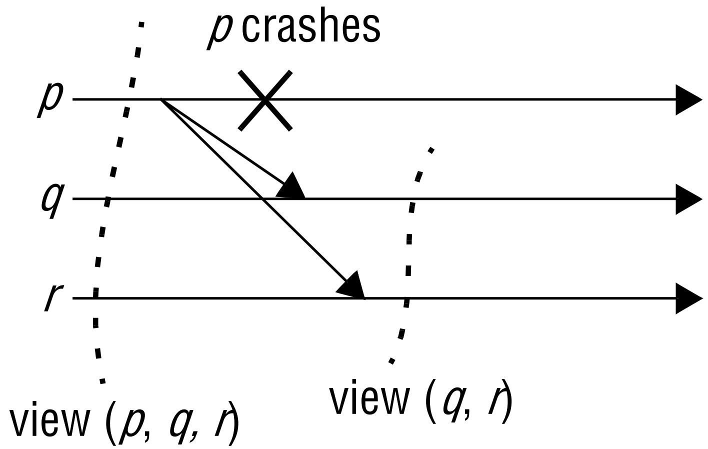
Examples

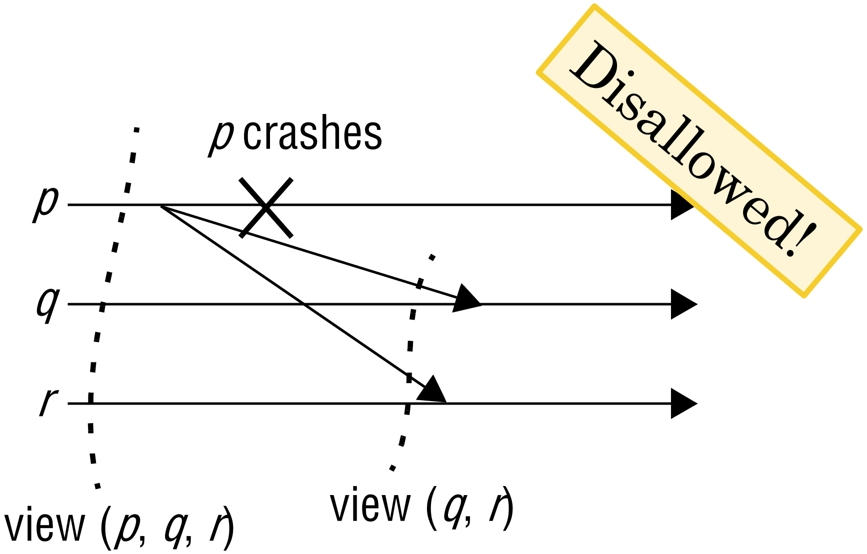


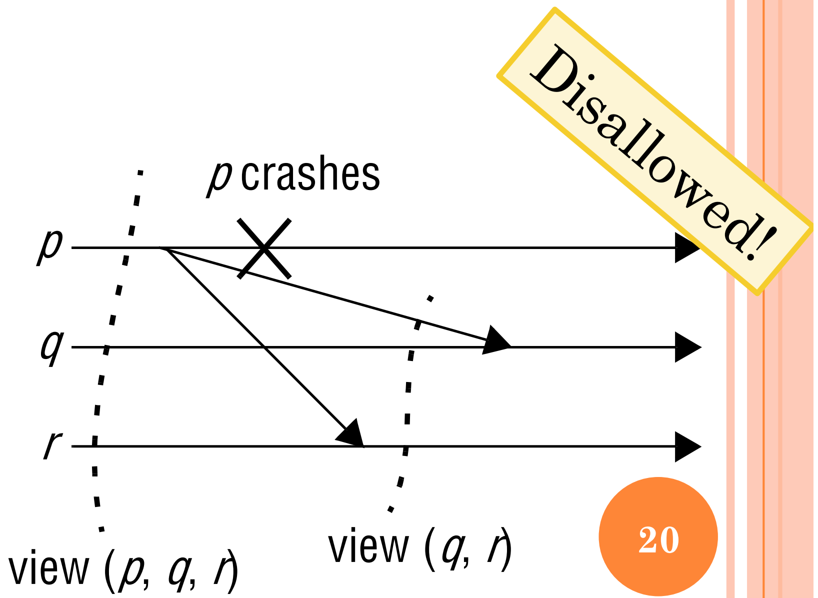


EXAMPLES


p crashes
p
q
r
view (p, q, r)
view (q, r)


p crashes
p
q
r
view (p, q, r)
view (q, r)


Disallowed!
p crashes
p
q
r
view (p, q, r)
view (q, r)


Disallowed!
p crashes
p
q
r
view (p, q, r)
view (q, r)

The Flavors of Virtual Synchrony

- In virtual synchrony, messages can't cross epochs …
- … but what can happen inside an epoch?
- Three base flavors:
  - unordered;
  - FIFO ordered;
  - causally ordered;
- where each of those can be:
  - totally ordered;
  - not totally ordered;
- for a total of six flavors

Detour: slides about message ordering…

Naming the Flavors

- The six versions of virtually synchronous reliable multicast

Multicast	**Basic Message Ordering**	**Total-ordered Delivery?**
Reliable multicast	None	No
FIFO multicast	FIFO-ordered delivery	No
Causal multicast	Causal-ordered delivery	No
Atomic multicast	None	Yes
FIFO atomic multicast	FIFO-ordered delivery	Yes
Causal atomic multicast	Causal-ordered delivery	Yes

Exercise 1

Is this execution virtually synchronous? What about FIFO, causally, or total ordered? Assume all start sharing the same view V0.

P1
1. delivers M1
2. installs view V1
3. delivers M2
4. multicasts M3
5. delivers M3
6. installs view V2

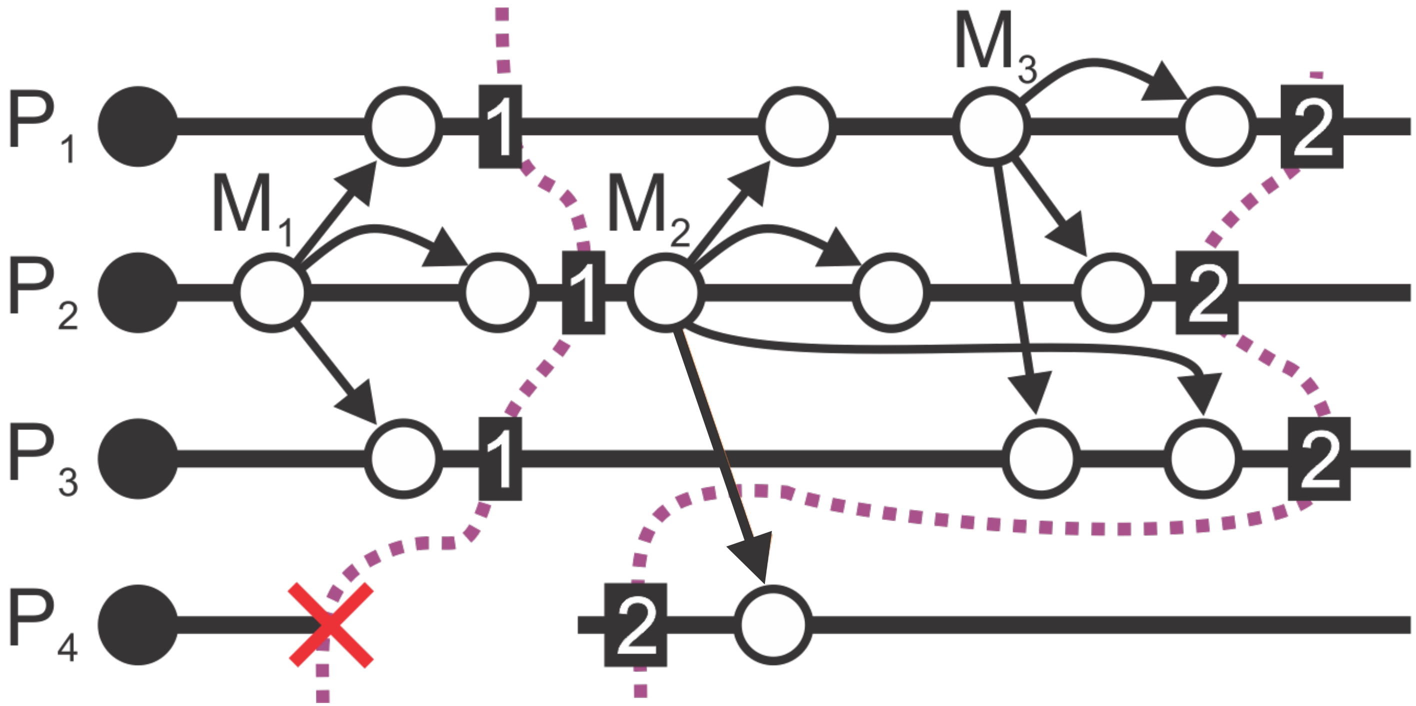
P2
1. multicasts M1
2. delivers M1
3. installs view V1
4. multicasts M2
5. delivers M2
6. delivers M3
7. installs view V2

P3
1. delivers M1
2. installs view V1
3. delivers M3
4. delivers M2
5. installs view V2

P4
1. installs view V2
2. delivers M2

Group views
V0={P1,P2,P3,P4}
V1={P1,P2,P3}
V2={P1,P2,P3,P4}

Exercise 1



- **not** virtually synchronous
- FIFO
- **not** causal
- **not** total

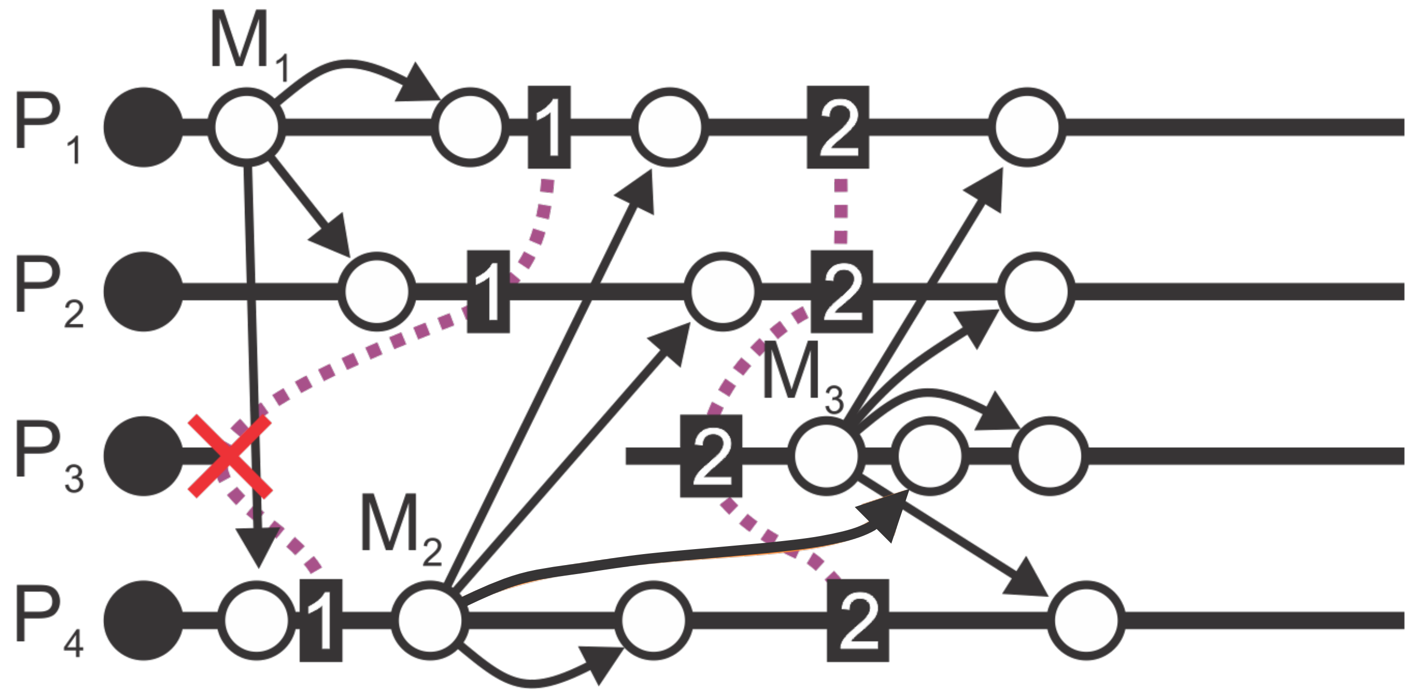
EXERCISE 2

Is this execution virtually synchronous? What about FIFO, causally, or total ordered? Assume all start sharing the same view V0.

P1	P2	P3	P4
1. multicasts M1	1. delivers M1	1. installs view V2	1. delivers M1
2. delivers M1	2. installs view V1	2. multicasts M3	2. installs view V1
3. installs view V1	3. delivers M2	3. delivers M2	3. multicasts M2
4. delivers M2	4. installs view V2	4. delivers M3	4. delivers M2
5. installs view V2	5. delivers M3		5. installs view V2
6. delivers M3			6. delivers M3

Group views
V0={P1,P2,P3,P4}
V1={P1,P2,P4}
V2={P1,P2,P3,P4}

Exercise 2



- **not** virtually synchronous
- FIFO
- causal
- total

Implementing Virtual Synchrony

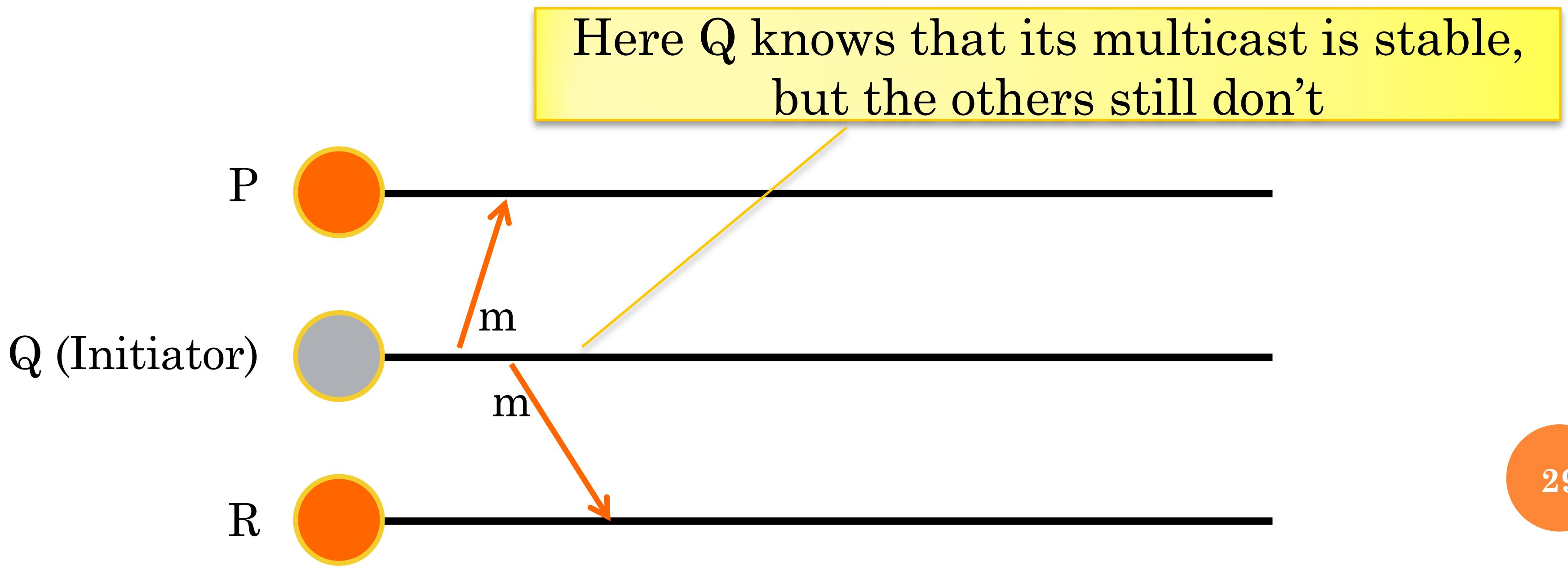
- We have seen the execution model…
- … but how can you enforce it in a real system?
  - ISIS toolkit, now called **VSync** [Birman et. al 1991]
  - JGroups (part of JBoss)
- **Assumptions:**
  - Reliable FIFO channels
  - Processes may silently crash

Multicast stability

- Multicast is implemented as a sequence of unicasts to all group members
- Channels are reliable, therefore all the unicasts eventually reach their destination
  - … unless the destination crashes
- The only case in which a multicast can fail is when the initiator crashes in the middle of it
- To overcome this case, every process *p*
  - Delivers immediately the message *m* it received, but …
  - .. keeps a copy of *m* until it is **sure** that every *correct* process in the *current view* has it …
  - … once this happens, *m* is said to be ***stable***, and the copy can be dropped
  - sends the copy to processes that might need it

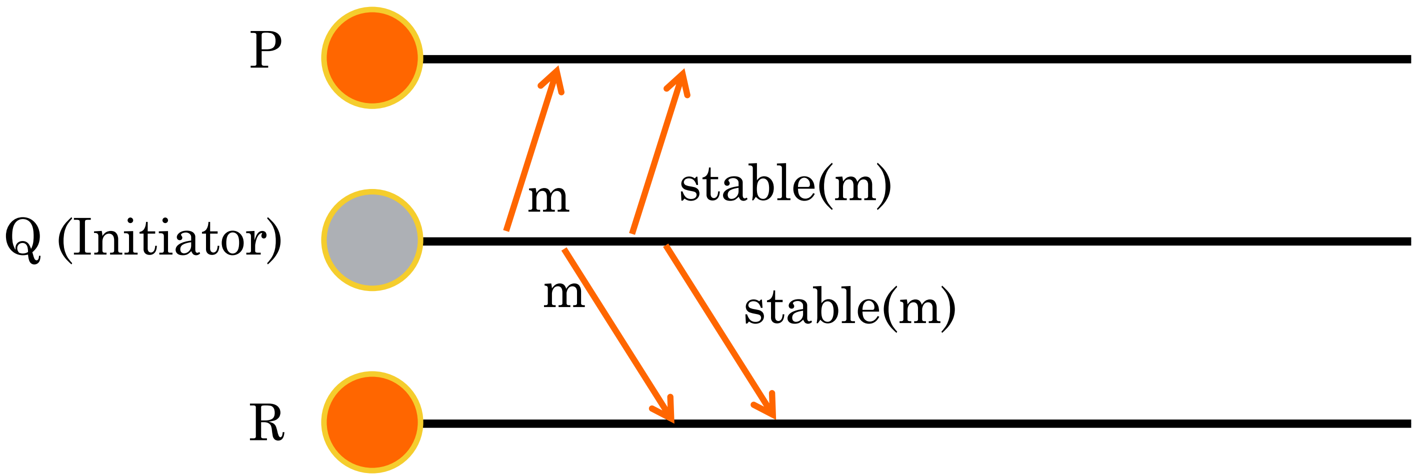
DETERMINING MESSAGE STABILITY

- Based on assumptions, the initiator learns that its multicast *m* is **stable** after it has sent the individual (unicast) messages to all the group members



ANNOUNCING MESSAGE STABILITY

- To inform the other nodes (and free their memory by deleting their copy of *m*), the initiator announces to the group that *m* is now stable
  - Piggybacking this information on the following multicasts or sending a dedicated one, if needed



HANDLING GROUP CHANGES

- When a process leaves, joins, or detects a crash, it sends a view change message to the group
- Upon receiving a view change with view V_1:
  - until the new view is installed:
    - pause sending *new* multicasts
    - keep delivering incoming messages sent in the old view, but
    - defer messages sent in the new view
  - do an "all-to-all" echo: send *all* **unstable** messages to all processes in the **new** view V_1, followed by a **FLUSH** message
  - wait until the **FLUSH** message arrives from **every** other process in V_1
  - install the view V_1

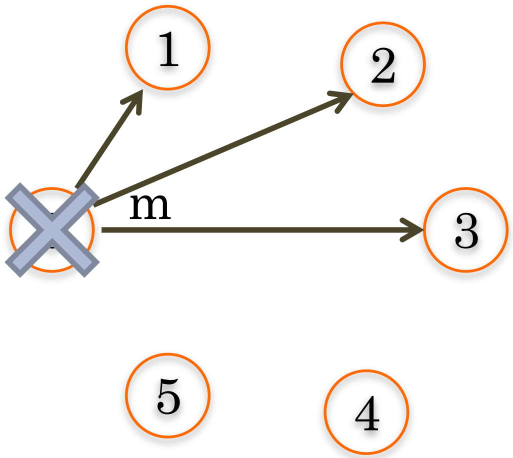
Unstable messages are ignored by a joining node; only the **FLUSH** messages matter to it

HANDLING GROUP CHANGES

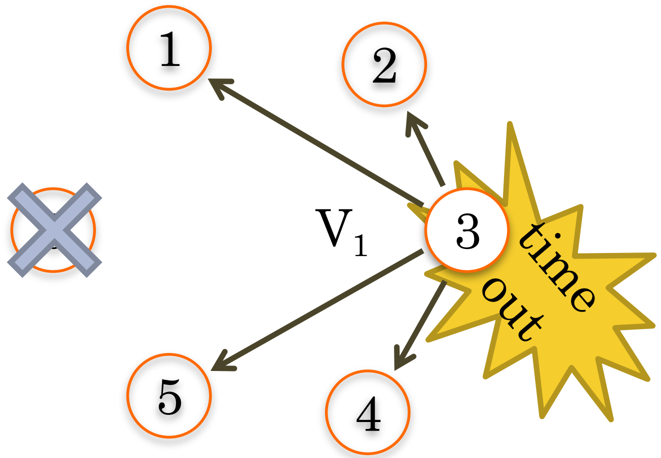
- Having received **FLUSH** from all processes in V_1, we know that we have received all the unstable messages sent in V_0 from all currently operational processes (because of FIFO), therefore
  - we deliver them in the old view V_0
  - we install the new view V_1
- If someone else crashes before sending **FLUSH**: repeat the all-to-all phase with the next view V_2

Example Execution

initiator crashes, processes 1, 2, 3 deliver *m* in V_0



crash detected in V_0



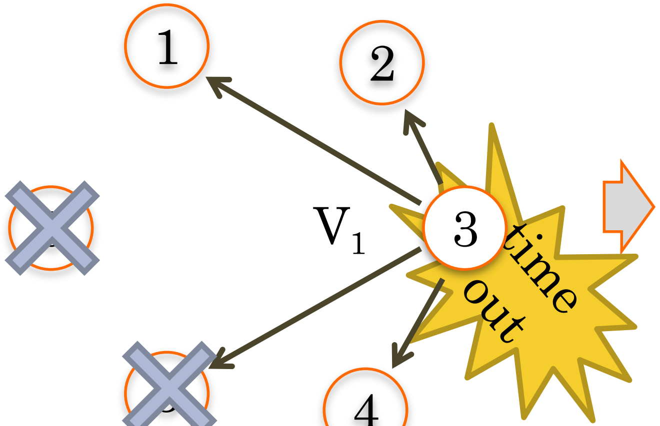
- all receive V_1
- all send to all: *m*, FLUSH(V_0)

- 4 and 5 deliver *m* in V_0
- all wait till FLUSH(V_0) comes from all in V_1
- all install V_1

V_0={0,1,2,3,4,5} V_1={1,2,3,4,5}

Example Execution

crash detected in V_0

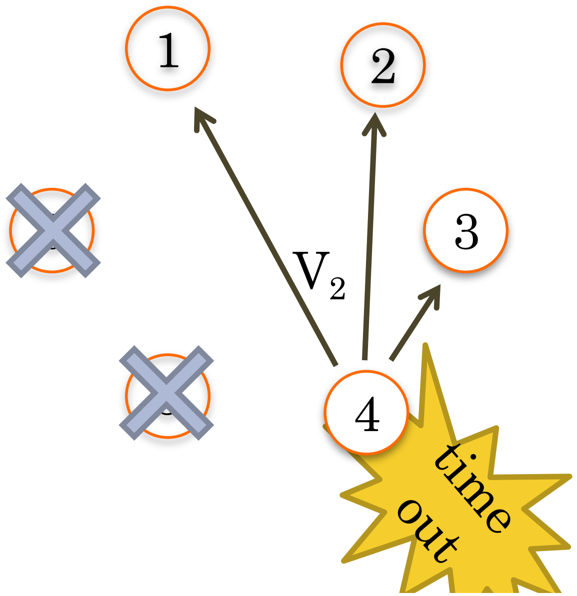


- all receive V_{1},
- all send to all: m, FLUSH(V_0)

- 4 delivers m
- All wait till FLUSH(V_0) comes from **all** in V_1… bad luck!

What if another node crashes during all-to-all phase?

Another crash detected in V_0



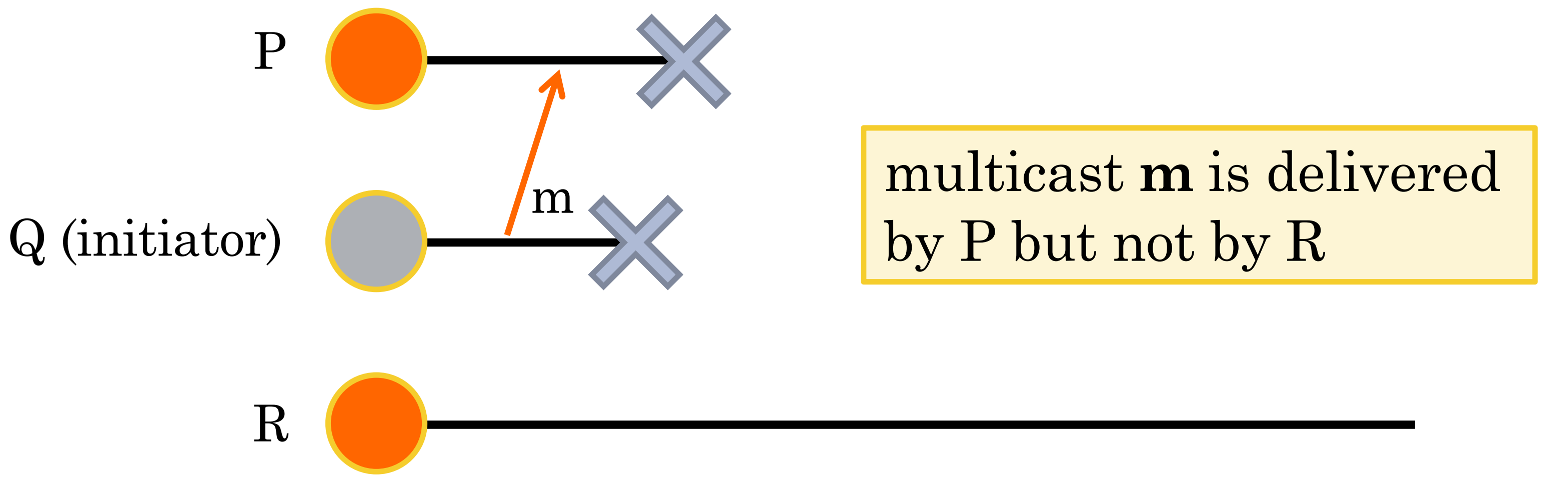
all send to all: V_2, FLUSH(V_1)

- all receive V_2
- wait till FLUSH(V_1) comes from all in V_2
- all install V_1
- all install V_2

V_1={1,2,3,4,5} V_2={1,2,3,4}

A NOTE ON MULTICAST STABILITY

- Note that the described algorithm delivers arriving messages right away, without ensuring that they are stable, therefore, the following is possible

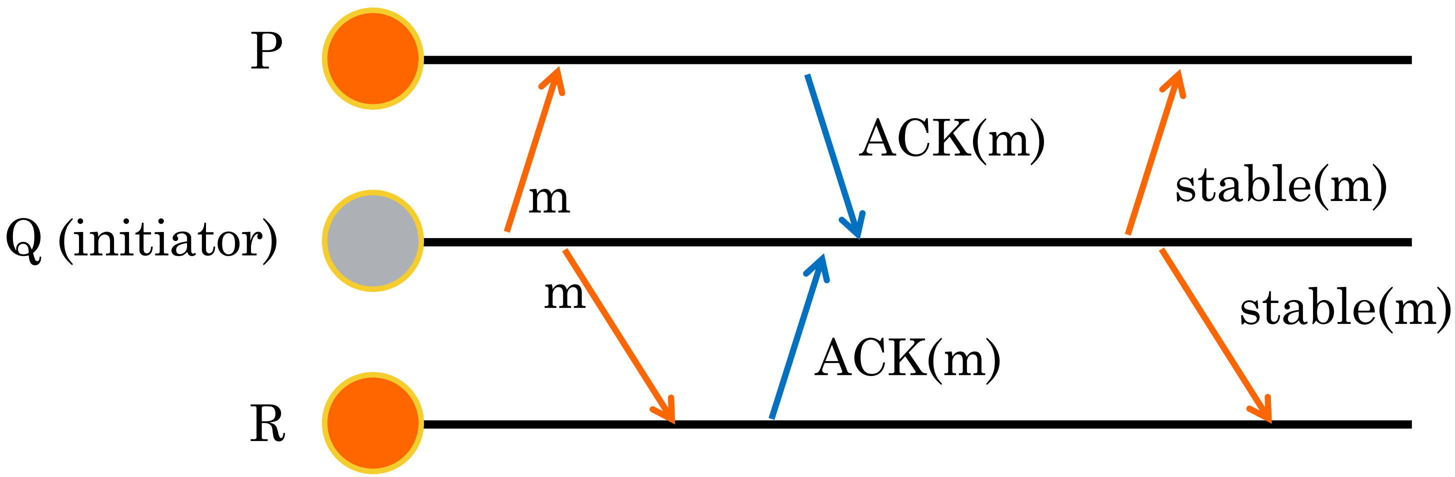


- The VS properties still hold (see previous slides)

- However, if this behaviour is undesirable, it can be solved by adding a second phase to each multicast…

A TWO-PHASE VARIANT

- The initiator collects acknowledgements about each multicast and confirms the stability by sending a second-phase message
- The receivers deliver the message only after they have learnt that the message is stable



This will slow down the protocol significantly!

LITERATURE

Reliable group communication:

- G. Coulouris, J. Dollimore, T. Kindberg. Distributed Systems: Concepts and Design (5th edition). Addison-Wesley, 2012
- A. S. Tanenbaum, M. van Steen. Distributed Systems: Principles and Paradigms (2nd edition)
- http://jgroups.org/manual4/index.html