DISTRIBUTED SYSTEMS

04

Reliable Broadcast

Konrad Iwanicki

Copyright notice

Acknowledgments

This lecture is (partly) based on:

1. C. Cachin, R. Guerraoui, and L. Rodrigues: *Introduction to Reliable and Secure Distributed Programming*, Second Edition, Springer-Verlag, (February 12, 2011), 386 pages, ISBN 978-3642152597, Chapter 3.

Introduction

Three fundamental abstractions:

- Interpreters,
- Storage,
- Communication channels.

Introduction

Three fundamental abstractions:

- Interpreters,
- Storage,
- Communication channels.

Introduction

- Traditional network protocols typically provide point-to-point communication.
- In distributed systems, often more than two parties need to communicate.
- More specifically, we will consider the problem of broadcast communication:
  - There is a population of processes.
  - Broadcasting a message to this population should ensure that all members of the population receive the message.

Introduction

Our assumptions for today:

- The population of processes is relatively small (think of 3-4 up to a dozen processes).
- Only a member of the population may broadcast messages to the other members (and itself).
- All processes are pairwise connected with communication links.
- We consider various models of failures, communication, and timing.

Introduction

Our goal is to provide so-called ***reliable broadcast***.

- What does it mean reliable in reference to broadcast?
- How to implement reliable broadcast in different models?

Considered Failure Model

Crash-stop failures

Best-effort Broadcast

Simple idea: make the sender (broadcaster) coordinate the entire broadcast as much as it can.

This is referred to as ***best-effort broadcast***.

Best-effort Broadcast

Module:
Name: BestEffortBroadcast, **instance** *beb*.

Events:
Request: ⟨*beb*, *Broadcast* | *m*⟩: Broadcasts a message, *m*, to all processes.
Indication: ⟨*beb*, *Deliver* | *p*, *m*⟩: Delivers a message, *m*, broadcast by process *p*.

Properties:
BEB1: *validity*: If a correct process broadcasts a message, *m*, then every correct process eventually delivers *m*.
BEB2: *no duplication*: No message is delivered more than once.
BEB3: *no creation*: If a process delivers a message, *m*, with sender *s*, then *m* must have been previously broadcast by process *s*.

Best-effort Broadcast

Algorithm: Basic Broadcast

Implements:
BestEffortBroadcast, **instance** *beb*.

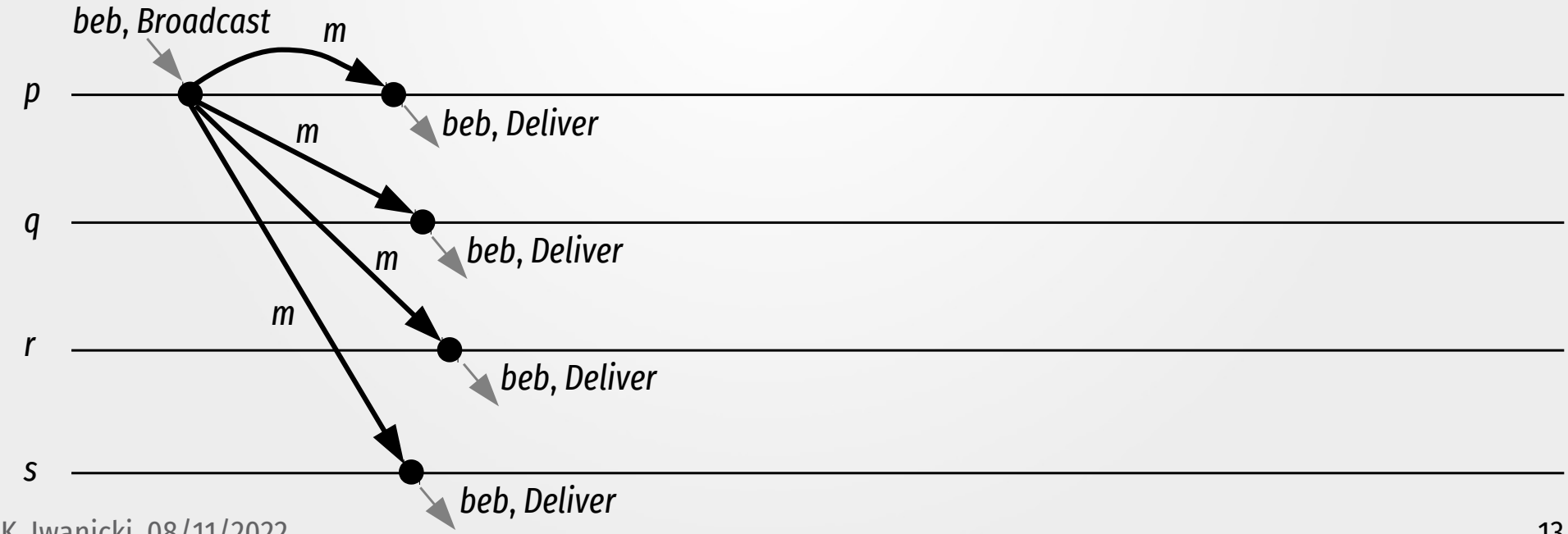
Uses:
PerfectLinks, **instance** *pl*.

upon event ⟨*beb*, *Broadcast* | *m*⟩ **do**
  forall $q \in \Pi$ **do**
    trigger ⟨*pl*, *Send* | *q*, *m*⟩;

upon event ⟨*pl*, *Deliver* | *p*, *m*⟩ **do**
  trigger ⟨*beb*, *Deliver* | *p*, *m*⟩;

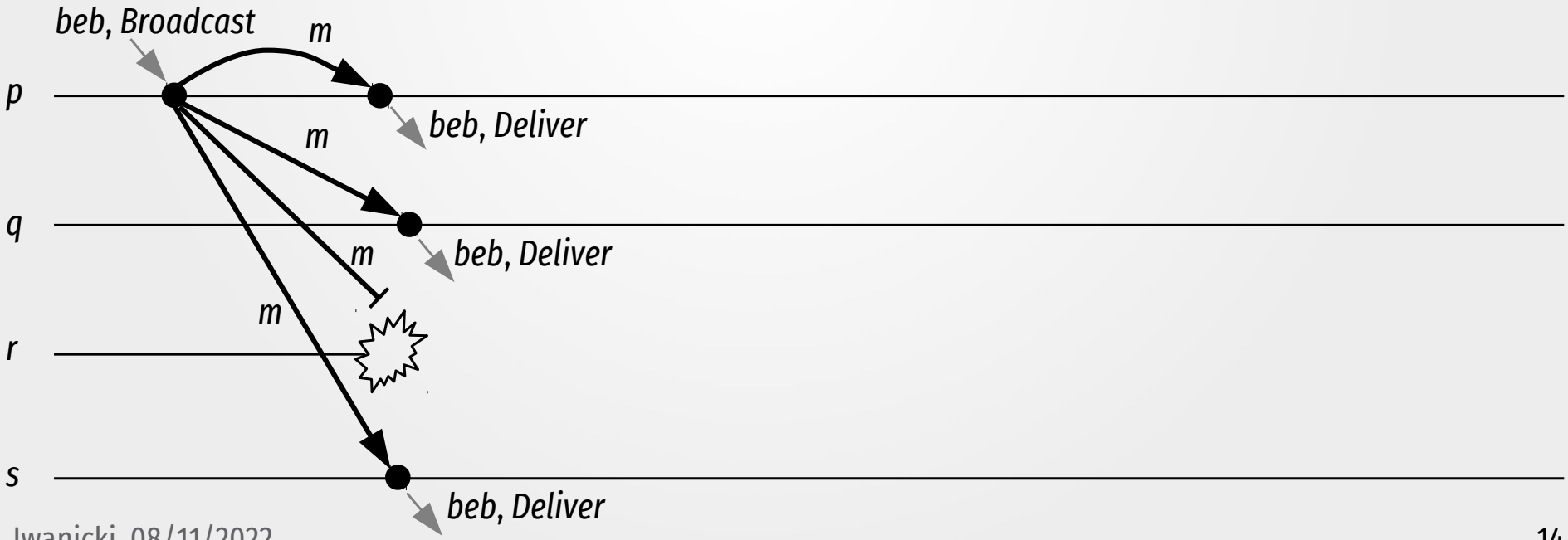
Best-effort Broadcast

- The algorithm is simple.
- It works because of the properties of perfect links.



Best-effort Broadcast

- The algorithm is simple.
- It works because of the properties of perfect links.



Best-effort Broadcast

Correctness of the algorithm – Cheatsheet:

Module:
Name: PerfectLinks, **instance** *pl*.

Events:
Request: ⟨*pl*, *Send* | *q*, *m*⟩: Requests to send message *m* to process *q*.
Indication: ⟨*pl*, *Deliver* | *p, m*⟩: Delivers message *m* from process *p*.

Properties:
PL1: *reliable delivery*: If a correct process, *p*, sends a message, *m*, to a correct process, *q*, (and never crashes afterward), then *q* eventually delivers *m*.
PL2: *no duplication*: No message is delivered by a process more than once.
PL3: *no creation*: If some process, *q*, delivers a message, *m*, with sender *p*, then *m* must have been previously sent to *q* by process *p*.

Best-effort Broadcast

Correctness of the algorithm:

- No creation (follows from no creation of perfect links):
  - Since perfect links do not spoof message, the algorithm does not do this either.
- No duplication (follows from no duplication of perfect links plus message uniqueness):
  - Messages broadcast by different processes are unique.
  - Therefore, since perfect links do not duplicate messages, no message sent by the algorithm is delivered more than once.
- Validity:
  - The sender sends a message to every process over a perfect link.
  - The reliable delivery property of perfect links ensures that each such message is eventually delivered.

Best-effort Broadcast

Performance of the algorithm:

- The work of the algorithm is N.
- The span of the algorithm is 1.

Best-effort Broadcast

Module:
Name: BestEffortBroadcast, **instance** *beb*.

Events:
Request: ⟨*beb*, *Broadcast* | *m*⟩: Broadcasts a message, *m*, to all processes.
Indication: ⟨*beb*, *Deliver* | *p*, *m*⟩: Delivers a message, *m*, broadcast by process *p*.

Properties:
BEB1: *validity*: If a correct process broadcasts a message, *m*, then every correct process eventually delivers *m*.
BEB2: *no duplication*: No message is delivered more than once.
BEB3: *no creation*: If a process delivers a message, *m*, with sender *s*, then *m* must have been previously broadcast by process *s*.

Question: Why can best-effort broadcast be insufficient?

Best-effort Broadcast

Module:
Name: BestEffortBroadcast, **instance** *beb*.

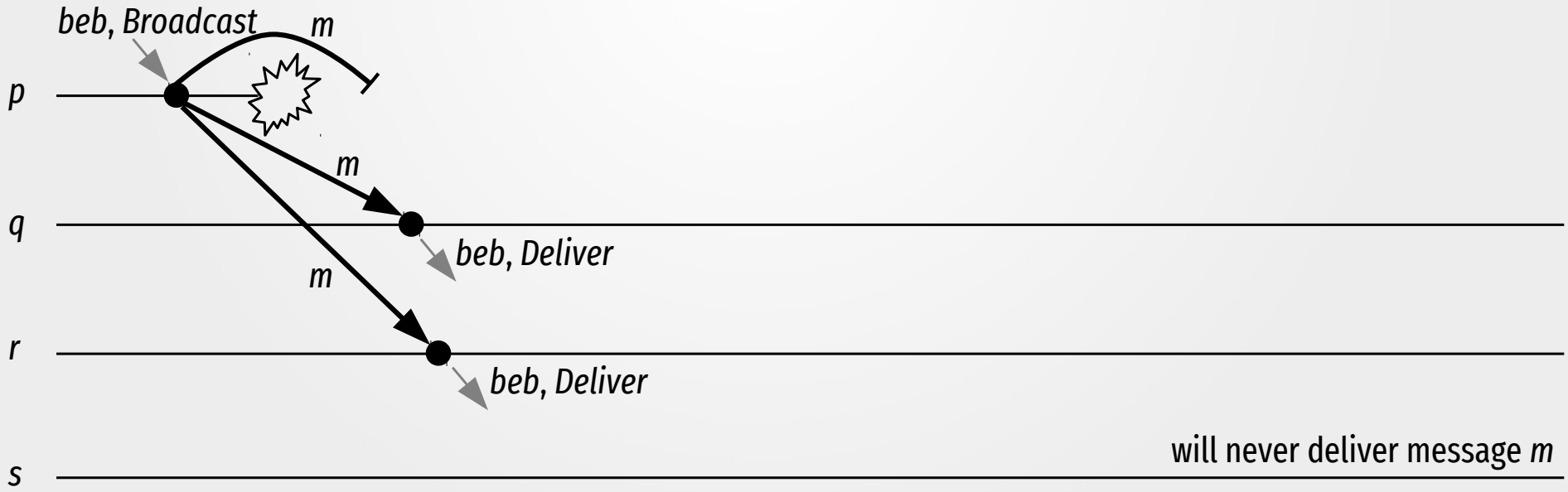
Events:
Request: ⟨*beb*, *Broadcast* | *m*⟩: Broadcasts a message, *m*, to all processes.
Indication: ⟨*beb*, *Deliver* | *p*, *m*⟩: Delivers a message, *m*, broadcast by process *p*.

Properties:
BEB1: *validity*: If a correct process broadcasts a message, *m*, then every correct process eventually delivers *m*.
BEB2: *no duplication*: No message is delivered more than once.
BEB3: *no creation*: If a process delivers a message, *m*, with sender *s*, then *m* must have been previously broadcast by process *s*.

Question: Why can best-effort broadcast be insufficient?

Best-effort Broadcast

If we drop the assumption of the sender being correct, the basic broadcast algorithm no longer ensures that a broadcast message is delivered by every correct process.



Regular Reliable Broadcast

Module:

Name: ReliableBroadcast, **instance** *rb*.

Events:

Request: ⟨*rb*, *Broadcast* | *m*⟩: Broadcasts a message, *m*, to all processes.

Indication: ⟨*rb*, *Deliver* | *p*, *m*⟩: Delivers a message, *m*, broadcast by process *p*.

Properties:

RB1: *validity*: If a correct process, *p*, broadcasts a message, m, then process *p* eventually delivers *m*.

RB2: *no duplication*: No message is delivered more than once.

RB3: *no creation*: If a process delivers a message, *m*, with sender *s*, then *m* must have been previously broadcast by process *s*.

RB4: *agreement*: If a message, *m*, is delivered by some correct process, then *m* is eventually delivered by every correct process.

Regular Reliable Broadcast

Module:
Name: ReliableBroadcast, **instance** *rb*.

Events:
Request: ⟨*rb*, *Broadcast* | *m*⟩: Broadcasts a message, *m*, to all processes.
Indication: ⟨*rb*, *Deliver* | *p*, *m*⟩: Delivers a message, *m*, broadcast by process *p*.

Properties:
RB1: *validity*: If a correct process, *p*, broadcasts a message, m, then process *p* eventually delivers *m*.
RB2: *no duplication*: No message is delivered more than once.
RB3: *no creation*: If a process delivers a message, *m*, with sender *s*, then *m* must have been previously broadcast by process *s*.
RB4: *agreement*: If a message, *m*, is delivered by some correct process, then *m* is eventually delivered by every correct process.

Question: Why this way and not by just removing the “correct” word from the assumption in the validity property?

Regular Reliable Broadcast

Algorithm: Lazy Reliable Broadcast

Implements:
ReliableBroadcast, **instance** *rb*.

Uses:
BestEfforBroadcast, **instance** *beb*.
PerfectFailureDetector, **instance** $\mathcal{P}$.

upon event ⟨*rb*, *Init*⟩ **do**
correct := Π;
from := $[\emptyset]^N$;

upon event ⟨*rb*, *Broadcast* | *m*⟩ **do**
trigger ⟨*beb*, *Broadcast* | [Data, *self*, *m*]⟩;

upon event ⟨*beb*, *Deliver* | *p*, [Data, *s*, *m*]⟩ **do**
if $m \notin from[s]$ **then**
trigger ⟨*rb*, *Deliver* | *s*, *m*⟩;
$from[s] := from[s] \cup \{ m \}$;
if $s \notin correct$ **then**
trigger ⟨*beb*, *Broadcast* | [Data, *s*, *m*]⟩;

upon event ⟨$\mathcal{P}$, *Crash* | *p*⟩ **do**
$correct := correct \setminus \{ p \}$;
forall $m \in from[p]$ **do**
trigger ⟨*beb*, *Broadcast* | [Data, *p*, *m*]⟩;

Regular Reliable Broadcast

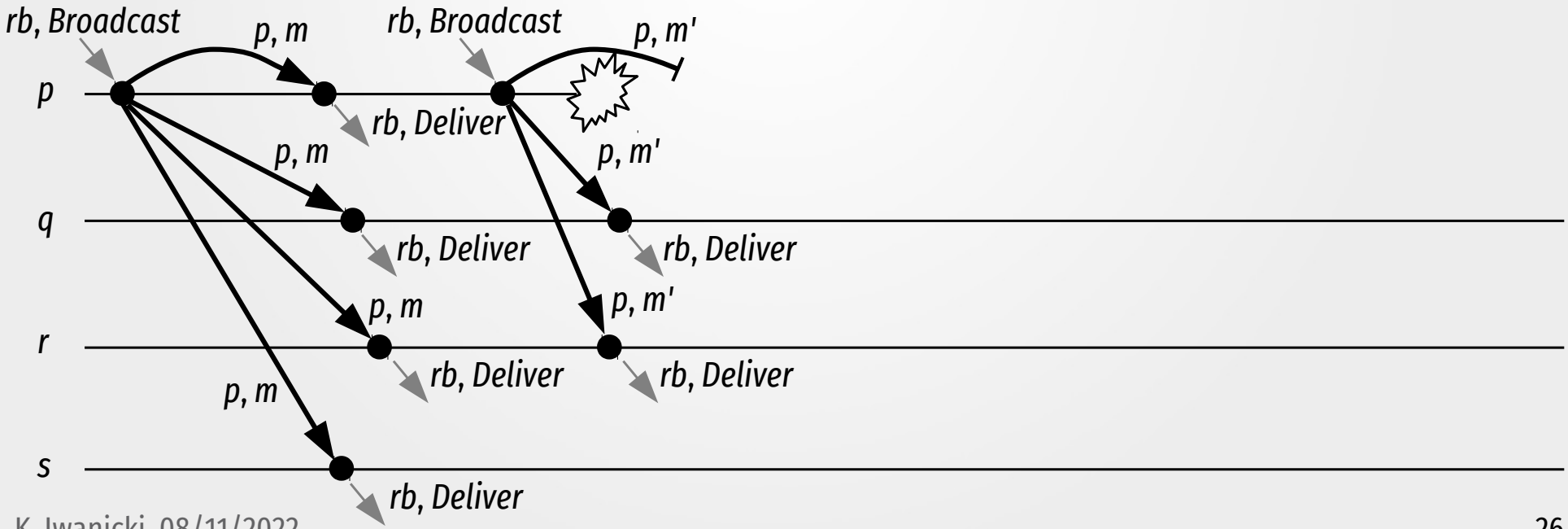
- In the absence of failures, the algorithm works as the previous one.

Regular Reliable Broadcast

- When a process crashes, the other correct processes take over to ensure that each of them delivers all messages that the failing process has managed to broadcast.

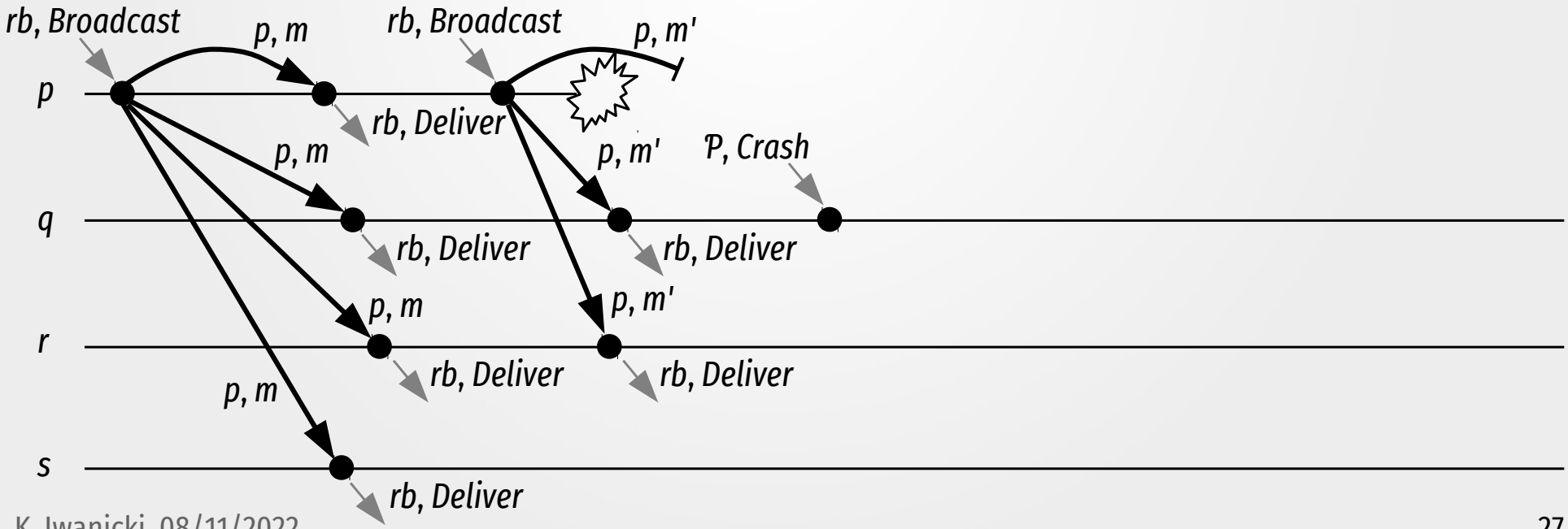
Regular Reliable Broadcast

- When a process crashes, the other correct processes take over to ensure that each of them delivers all messages that the failing process has managed to broadcast.



Regular Reliable Broadcast

- When a process crashes, the other correct processes take over to ensure that each of them delivers all messages that the failing process has managed to broadcast.



Regular Reliable Broadcast

Algorithm: Lazy Reliable Broadcast

Implements:
ReliableBroadcast, **instance** *rb*.

Uses:
BestEfforBroadcast, **instance** *beb*.
PerfectFailureDetector, **instance** $\mathcal{P}$.

upon event ⟨ *rb*, *Init* ⟩ **do**
correct := Π;
from := $[\emptyset]^N$;

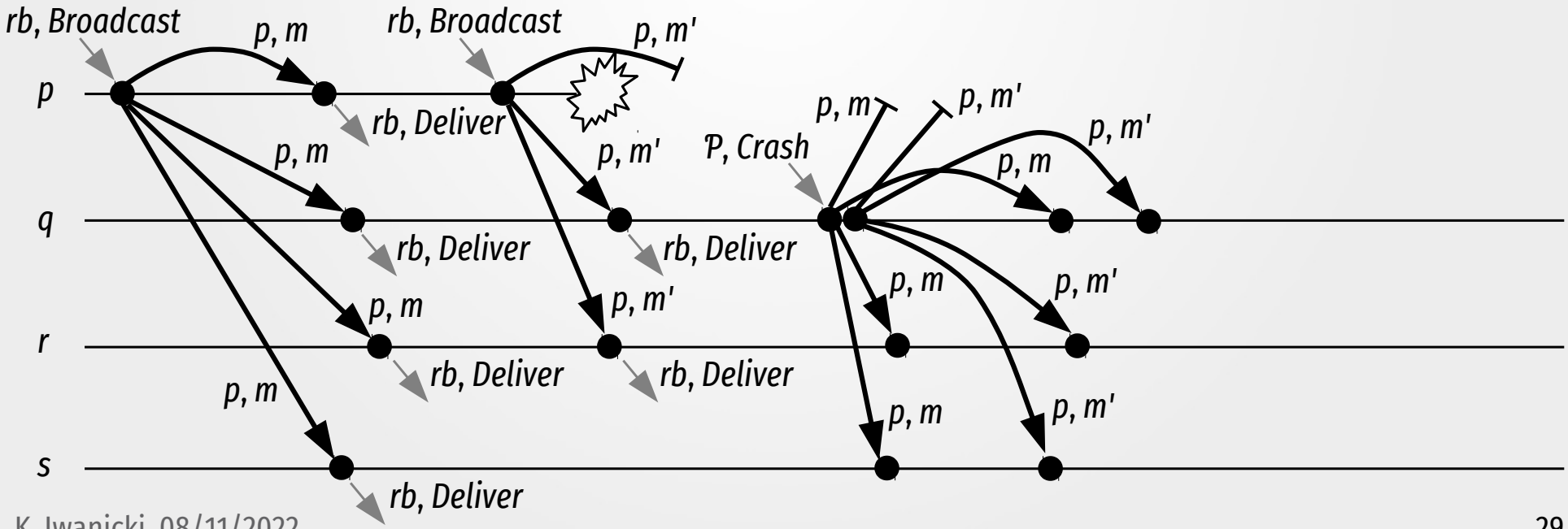
upon event ⟨ *rb*, *Broadcast* | *m* ⟩ **do**
trigger ⟨ *beb*, *Broadcast* | [Data, *self*, *m*] ⟩;

upon event ⟨ *beb*, *Deliver* | *p*, [Data, *s*, *m*] ⟩ **do**
if $m \notin from[s]$ **then**
trigger ⟨ *rb*, *Deliver* | *s*, *m* ⟩;
$from[s] := from[s] \cup \{ m \}$;
if $s \notin correct$ **then**
trigger ⟨ *beb*, *Broadcast* | [Data, *s*, *m*] ⟩;

upon event ⟨ $\mathcal{P}$, *Crash* | *p* ⟩ **do**
correct := *correct* \ { *p* };
forall $m \in from[p]$ **do**
trigger ⟨ *beb*, *Broadcast* | [Data, *p*, *m*] ⟩;

Regular Reliable Broadcast

- When a process crashes, the other correct processes take over to ensure that each of them delivers all messages that the failing process has managed to broadcast.



Regular Reliable Broadcast

Algorithm: Lazy Reliable Broadcast

Implements:
ReliableBroadcast, **instance** *rb*.

Uses:
BestEfforBroadcast, **instance** *beb*.
PerfectFailureDetector, **instance** $\mathcal{P}$.

upon event ⟨*rb*, *Init*⟩ **do**
correct := Π;
from := $[\emptyset]^N$;

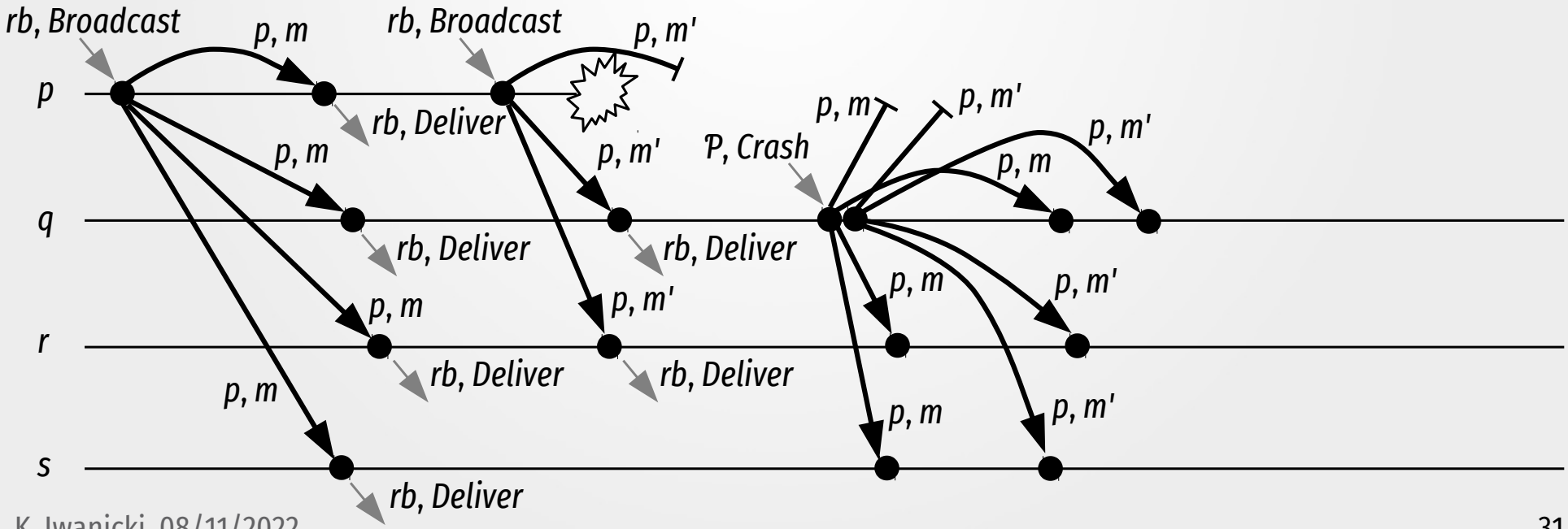
upon event ⟨*rb*, *Broadcast* | *m*⟩ **do**
trigger ⟨*beb*, *Broadcast* | [Data, *self*, *m*]⟩;

upon event ⟨*beb*, *Deliver* | *p*, [Data, *s*, *m*]⟩ **do**
if $m \notin from[s]$ **then**
trigger ⟨*rb*, *Deliver* | *s*, *m*⟩;
$from[s] := from[s] \cup \{ m \}$;
if $s \notin correct$ **then**
trigger ⟨*beb*, *Broadcast* | [Data, *s*, *m*]⟩;

upon event ⟨$\mathcal{P}$, *Crash* | *p*⟩ **do**
correct := *correct* \ { *p* };
forall $m \in from[p]$ **do**
trigger ⟨*beb*, *Broadcast* | [Data, *p*, *m*]⟩;

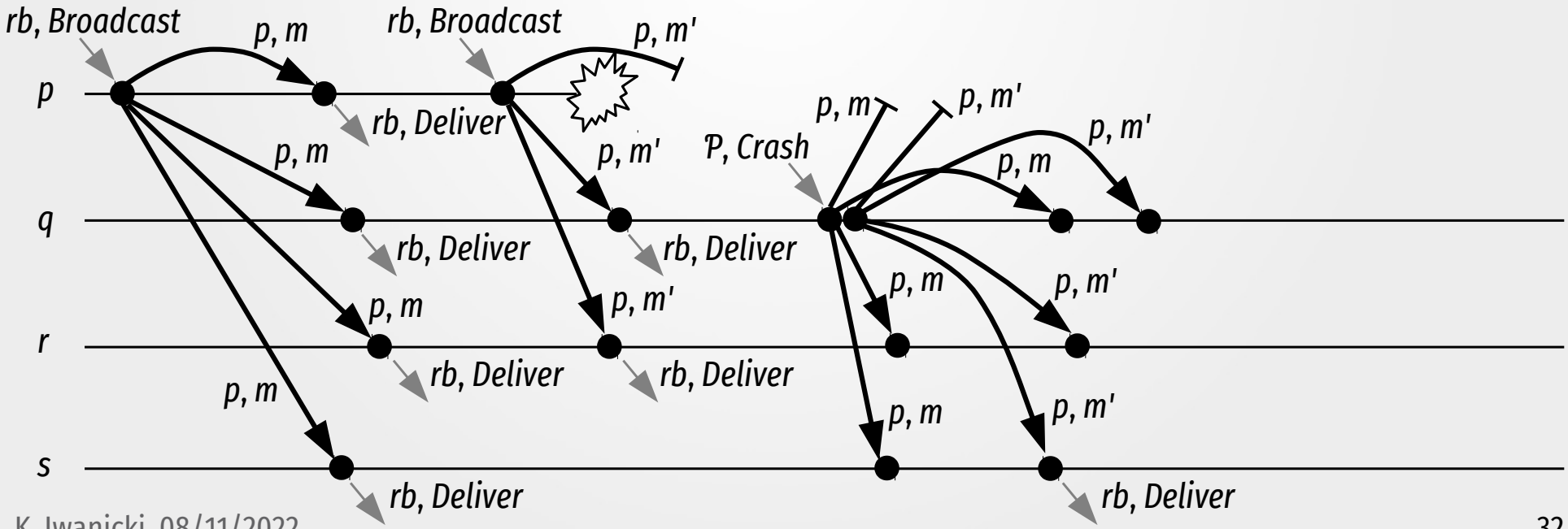
Regular Reliable Broadcast

- When a process crashes, the other correct processes take over to ensure that each of them delivers all messages that the failing process has managed to broadcast.



Regular Reliable Broadcast

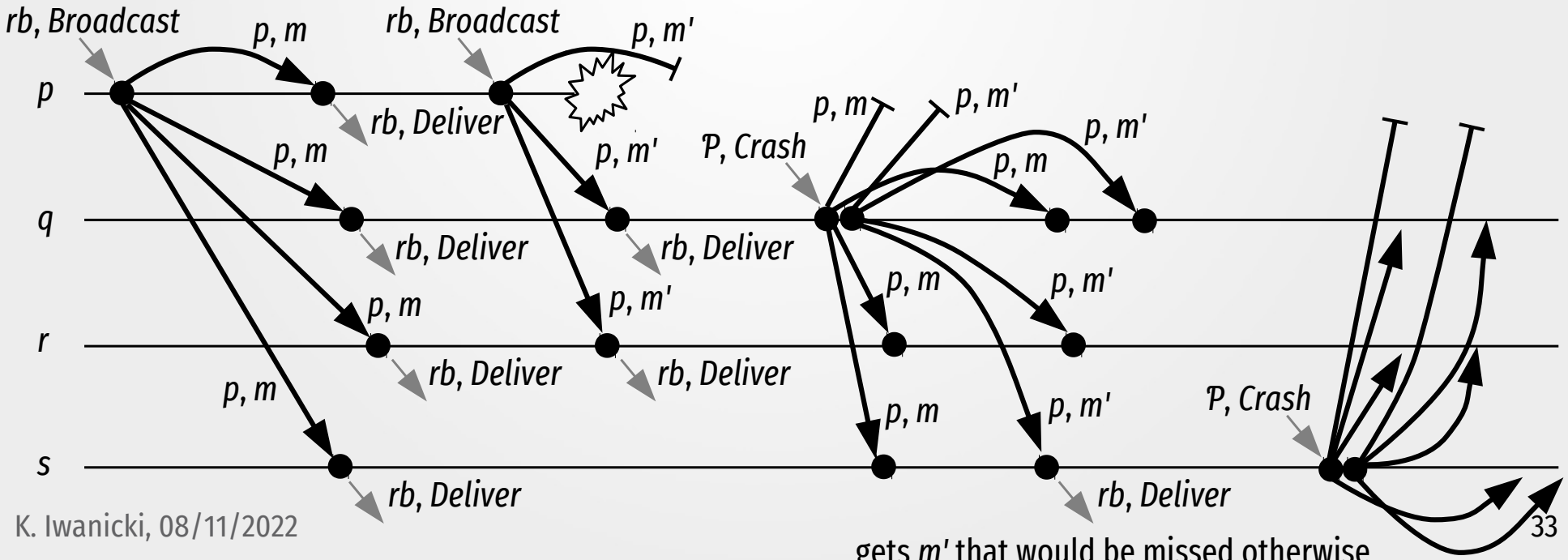
- When a process crashes, the other correct processes take over to ensure that each of them delivers all messages that the failing process has managed to broadcast.



gets *m'* that would be missed otherwise

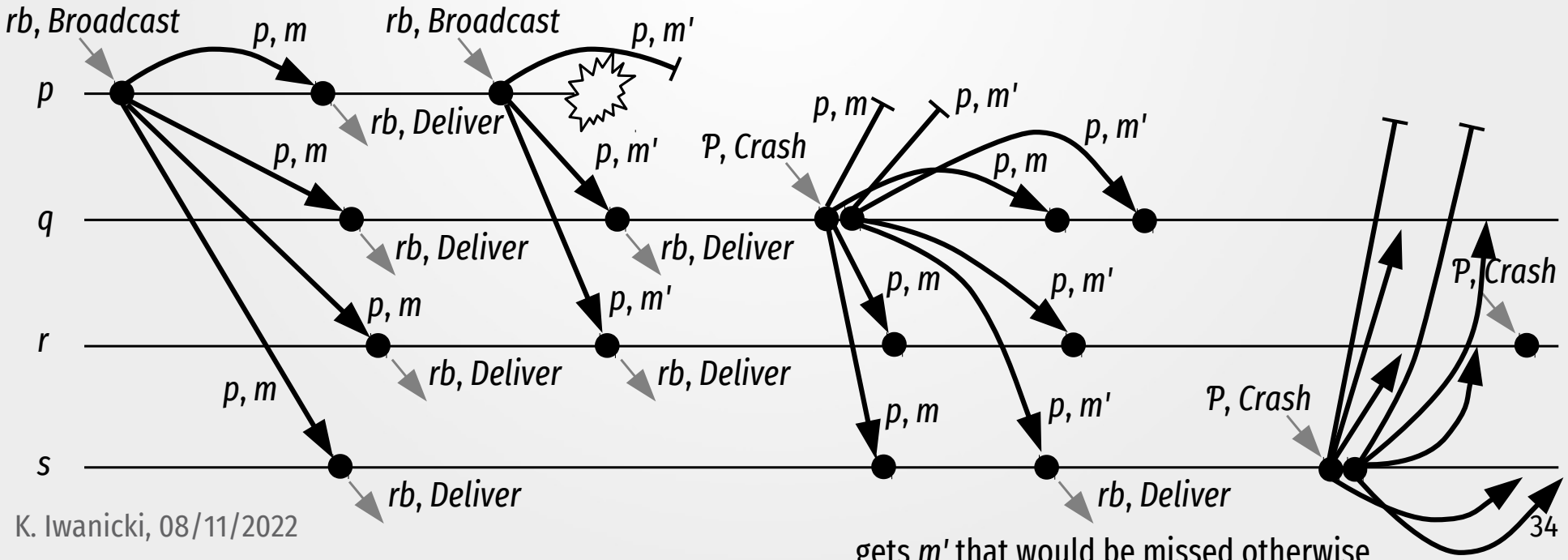
Regular Reliable Broadcast

- When a process crashes, the other correct processes take over to ensure that each of them delivers all messages that the failing process has managed to broadcast.



Regular Reliable Broadcast

- When a process crashes, the other correct processes take over to ensure that each of them delivers all messages that the failing process has managed to broadcast.



Regular Reliable Broadcast

Algorithm: Lazy Reliable Broadcast

Implements:
ReliableBroadcast, **instance** *rb*.

Uses:
BestEfforBroadcast, **instance** *beb*.
PerfectFailureDetector, **instance** $\mathcal{P}$.

upon event ⟨ *rb*, *Init* ⟩ **do**
correct := Π;
from := $[\emptyset]^N$;

upon event ⟨ *rb*, *Broadcast* | *m* ⟩ **do**
trigger ⟨ *beb*, *Broadcast* | [Data, *self*, *m*] ⟩;

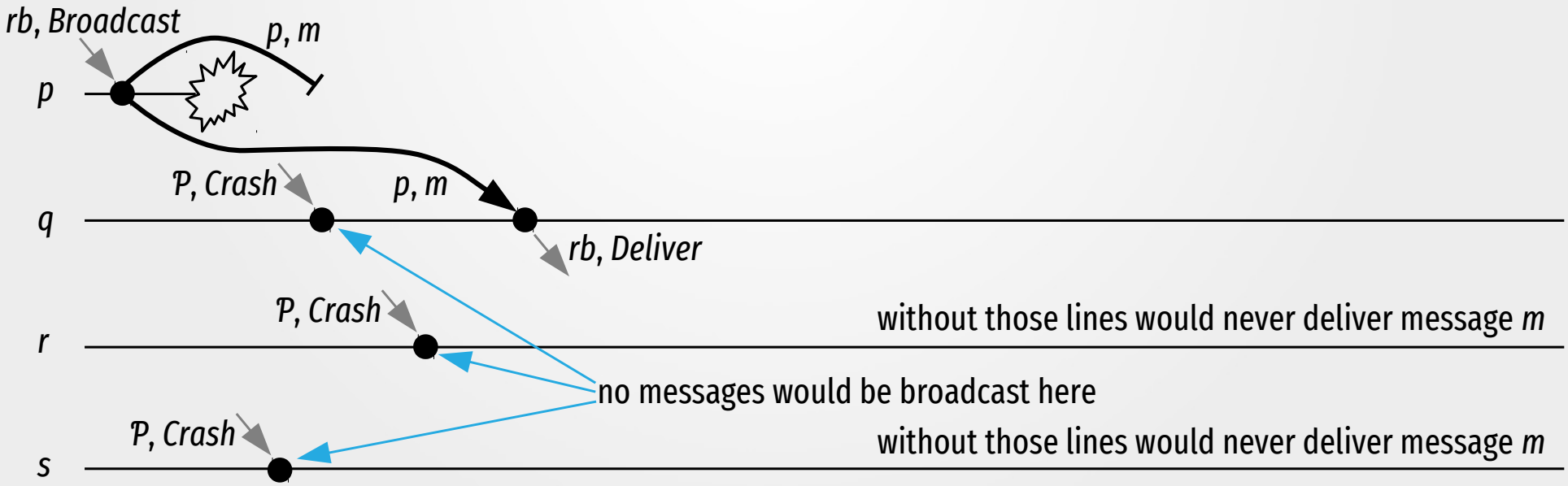
upon event ⟨ *beb*, *Deliver* | *p*, [Data, *s*, *m*] ⟩ **do**
if $m \notin from[s]$ **then**
trigger ⟨ *rb*, *Deliver* | *s*, *m* ⟩;
from[*s*] := *from*[*s*] ∪ { *m* };
if $s \notin correct$ **then**
trigger ⟨ *beb*, *Broadcast* | [Data, *s*, *m*] ⟩;

upon event ⟨ $\mathcal{P}$, *Crash* | *p* ⟩ **do**
correct := *correct* \ { *p* };
forall $m \in from[p]$ **do**
trigger ⟨ *beb*, *Broadcast* | [Data, *p*, *m*] ⟩;

Question: Why are the highlighted lines necessary (if at all)?

Regular Reliable Broadcast

- Without them, we could not guarantee that all correct processes deliver the same set of messages.



Regular Reliable Broadcast

Correctness of the algorithm – Cheatsheet:

Module:
Name: PerfectFailureDetector, **instance** $\mathcal{P}$.

Events:
Indication: $\langle \mathcal{P}, \mathit{Crash} \mid p \rangle$: Detects that process p has crashed.

Properties:
PFD1: *strong completeness*: Eventually, every process that crashes is permanently detected by every correct process.
PFD2: *strong accuracy*: If a process, p, is detected by any process, then p must have crashed.

Regular Reliable Broadcast

Correctness of the algorithm:

- No creation – follows directly from the no creation property of BEB.
- Validity – same here.
- No duplication:
  - Messages broadcast by different processes are unique.
  - Each process maintains variable *from* that filters out received duplicates originating at the same broadcaster.
- Agreement:
  - A perfect failure detector detects every process that has crashed (strong completeness).
  - Each message broadcast by a crashed process and delivered by some correct process is rebroadcast by the correct process (upon the crash or delivery) using BEB.
  - From the validity property of BEB, the message is thus eventually BEB-delivered by every correct process.

Regular Reliable Broadcast

Performance of the algorithm:

- The work of the algorithm is:
  - $O(N)$ in the absence of failures
  - up to $O(M \cdot N^2)$ under a failure of a process, where M is the number of messages that the crashed process has managed to broadcast.
- The span of the algorithm is:
  - 1 in the absence of failures
  - up to $O(M)$ under a failure of a process.
- In particular, if f processes fail, each having broadcast M messages, the work and span are respectively $O(f \cdot M \cdot N^2)$ and $O(f \cdot M)$. Note that f is $O(N)$.

Regular Reliable Broadcast

Question: Can we get rid of the failure detector?

Regular Reliable Broadcast

Algorithm: Eager Reliable Broadcast

Implements:
ReliableBroadcast, **instance** *rb*.

Uses:
BestEfforBroadcast, **instance** *beb*.

upon event ⟨*rb*, *Init*⟩ **do**
delivered := ∅;

upon event ⟨*rb*, *Broadcast* | *m*⟩ **do**
trigger ⟨*beb*, *Broadcast* | [Data, *self*, *m*]⟩;

upon event ⟨*beb*, *Deliver* | *p*, [Data, *s*, *m*]⟩ **do**
if *m* ∉ *delivered* **then**
delivered := *delivered* ∪ { *m* };
trigger ⟨*rb*, *Deliver* | *s*, *m*⟩;
trigger ⟨*beb*, *Broadcast* | [Data, *s*, *m*]⟩;

Regular Reliable Broadcast

Module:
Name: ReliableBroadcast, **instance** *rb*.

Events:
Request: ⟨*rb*, *Broadcast* | *m*⟩: Broadcasts a message, *m*, to all processes.
Indication: ⟨*rb*, *Deliver* | *p*, *m*⟩: Delivers a message, *m*, broadcast by process *p*.

Properties:
RB1: *validity*: If a correct process, *p*, broadcasts a message, m, then process *p* eventually delivers *m*.
RB2: *no duplication*: No message is delivered more than once.
RB3: *no creation*: If a process delivers a message, *m*, with sender *s*, then *m* must have been previously broadcast by process *s*.
RB4: *agreement*: If a message, *m*, is delivered by some correct process, then *m* is eventually delivered by every correct process.

Question: Can we push reliability even further?

Regular Reliable Broadcast

Module:
Name: ReliableBroadcast, **instance** *rb*.

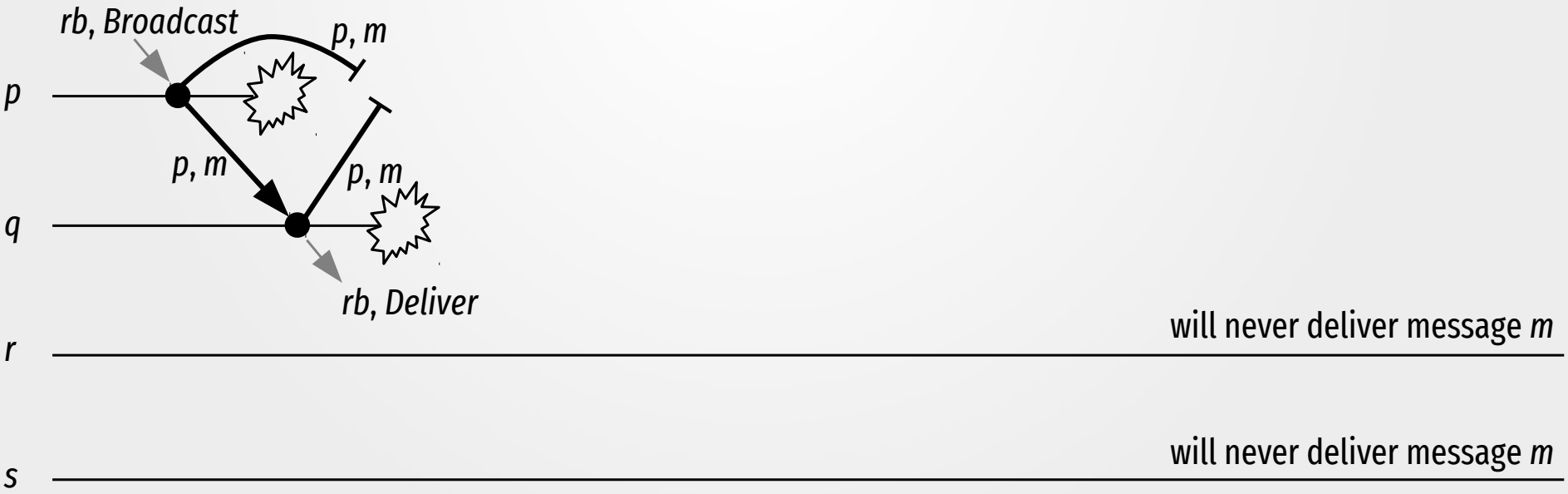
Events:
Request: ⟨*rb*, *Broadcast* | *m*⟩: Broadcasts a message, *m*, to all processes.
Indication: ⟨*rb*, *Deliver* | *p*, *m*⟩: Delivers a message, *m*, broadcast by process *p*.

Properties:
RB1: *validity*: If a correct process, *p*, broadcasts a message, m, then process *p* eventually delivers *m*.
RB2: *no duplication*: No message is delivered more than once.
RB3: *no creation*: If a process delivers a message, *m*, with sender *s*, then *m* must have been previously broadcast by process *s*.
RB4: *agreement*: If a message, *m*, is delivered by some **correct** process, then *m* is eventually delivered by every correct process.

Question: Can we push reliability even further?

Regular Reliable Broadcast

- When processes deliver a message and crash, there may be no process left to ensure that the message is delivered to all correct processes.



Uniform Reliable Broadcast

Module:
Name: UniformReliableBroadcast, **instance** *urb*.

Events:
Request: ⟨*urb*, *Broadcast* | *m*⟩: Broadcasts a message, *m*, to all processes.
Indication: ⟨*urb*, *Deliver* | *p*, *m*⟩: Delivers a message, *m*, broadcast by process *p*.

Properties:
URB1: *validity*: If a correct process, *p*, broadcasts a message, *m*, then process p eventually delivers *m*.
URB2: *no duplication*: No message is delivered more than once.
URB3: *no creation*: If a process delivers a message, *m*, with sender *s*, then *m* must have been previously broadcast by process *s*.
URB4: *uniform agreement*: If a message, *m*, is delivered by some process (correct or faulty), then *m* is eventually delivered by every correct process.

Uniform Reliable Broadcast

- Principal idea: Ensure that enough processes have received a message before delivering it locally.

Uniform Reliable Broadcast

Algorithm: All-Ack Uniform Reliable Broadcast

Implements:
UniformReliableBroadcast, **instance** *urb*.

Uses:
BestEfforBroadcast, **instance** *beb*.
PerfectFailureDetector, **instance** $\mathcal{P}$.

upon event ⟨*urb*, *Init*⟩ **do**
delivered := ∅;
pending := ∅;
correct := Π;
forall *m* **do**
ack[*m*] := ∅;

upon event ⟨*urb*, *Broadcast* | *m*⟩ **do**
pending := *pending* ∪ { (*self*, *m*) };
trigger ⟨*beb*, *Broadcast* | [Data, *self*, *m*]⟩;

upon event ⟨$\mathcal{P}$, *Crash* | *p*⟩ **do**
correct := *correct* \ { *p* };

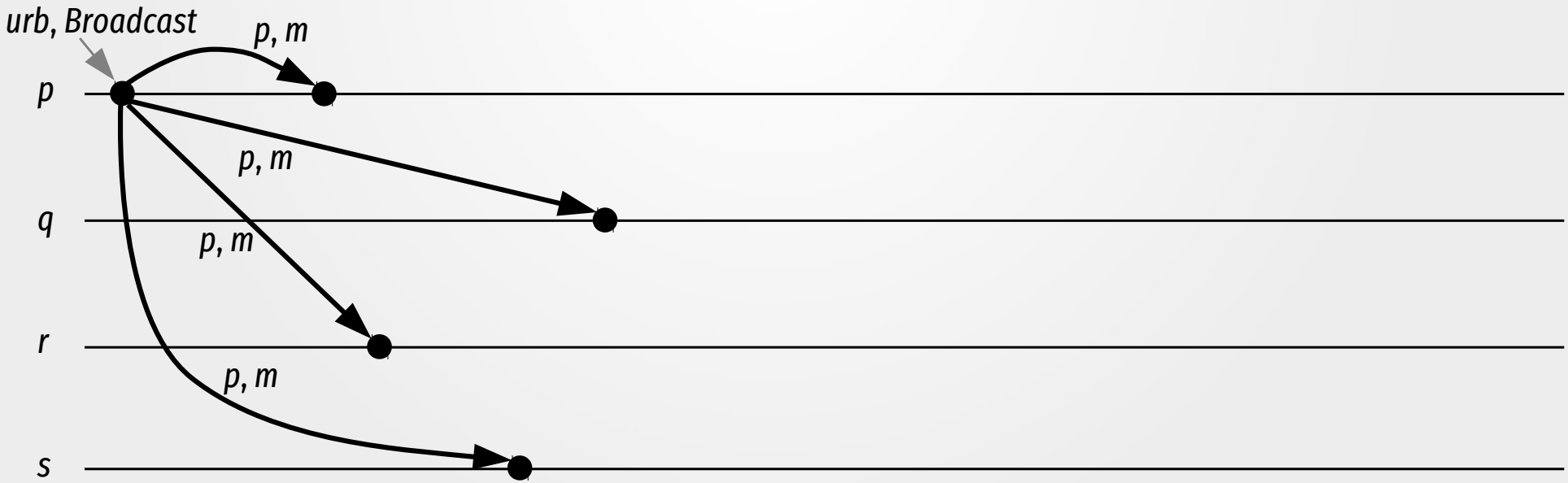
upon event ⟨*beb*, *Deliver* | *p*, [Data, *s*, *m*]⟩ **do**
ack[*m*] := *ack*[*m*] ∪ { *p* };
if (*s*, *m*) ∉ *pending* **then**
pending := *pending* ∪ { (*s*, *m*) };
trigger ⟨*beb*, *Broadcast* | [Data, *s*, *m*]⟩;

fn *candeliver*(m) **is**
return *correct* ⊆ *ack*[*m*];

upon exists (*s*, *m*) ∈ *pending* **such that**
candeliver(m) ∧ *m* ∉ *delivered* **do**
delivered := *delivered* ∪ { *m* };
trigger ⟨*urb*, *Deliver* | *s*, *m*⟩;

Uniform Reliable Broadcast

- Principal idea: Ensure that all correct processes have received a message before delivering it locally.



Uniform Reliable Broadcast

Algorithm: All-Ack Uniform Reliable Broadcast

Implements:
UniformReliableBroadcast, **instance** *urb*.

Uses:
BestEfforBroadcast, **instance** *beb*.
PerfectFailureDetector, **instance** $\mathcal{P}$.

upon event ⟨*urb*, *Init*⟩ **do**
delivered := ∅;
pending := ∅;
correct := Π;
forall *m* **do**
ack[*m*] := ∅;

upon event ⟨*urb*, *Broadcast* | *m*⟩ **do**
pending := *pending* ∪ { (*self*, *m*) };
trigger ⟨*beb*, *Broadcast* | [Data, *self*, *m*]⟩;

upon event ⟨$\mathcal{P}$, *Crash* | *p*⟩ **do**
correct := *correct* \ { *p* };

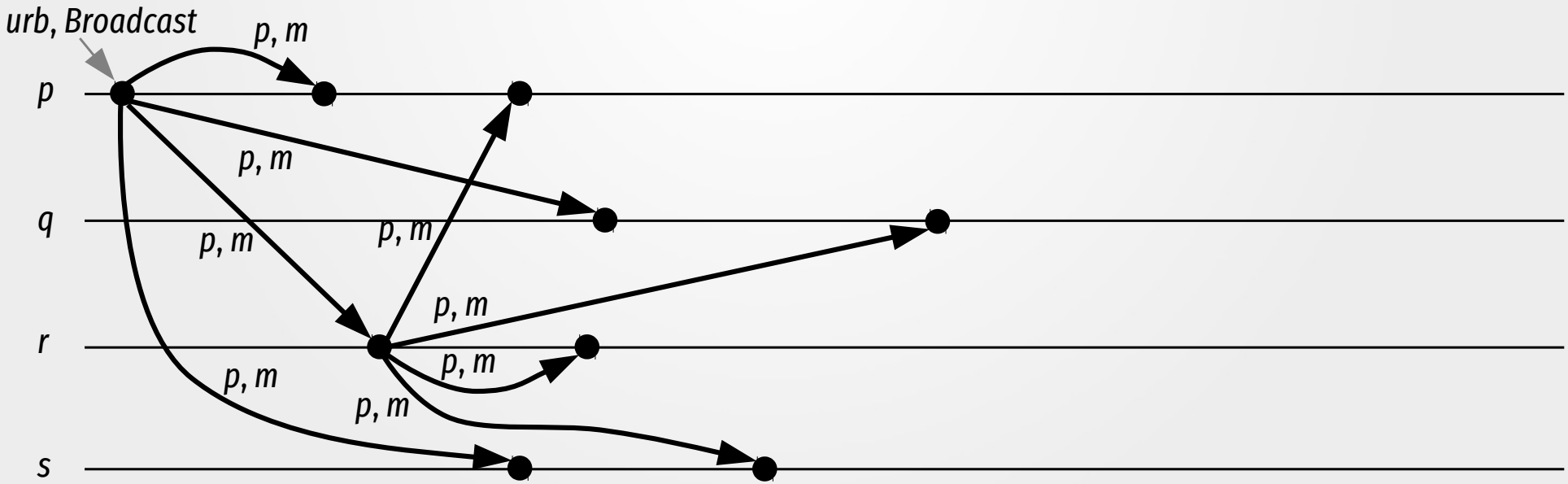
upon event ⟨*beb*, *Deliver* | *p*, [Data, *s*, *m*]⟩ **do**
ack[*m*] := *ack*[*m*] ∪ { *p* };
if (*s*, *m*) ∉ *pending* **then**
pending := *pending* ∪ { (*s*, *m*) };
trigger ⟨*beb*, *Broadcast* | [Data, *s*, *m*]⟩;

fn *candeliver*(m) **is**
return *correct* ⊆ *ack*[*m*];

upon exists (*s*, *m*) ∈ *pending* **such that**
candeliver(m) ∧ *m* ∉ *delivered* **do**
delivered := *delivered* ∪ { *m* };
trigger ⟨*urb*, *Deliver* | *s*, *m*⟩;

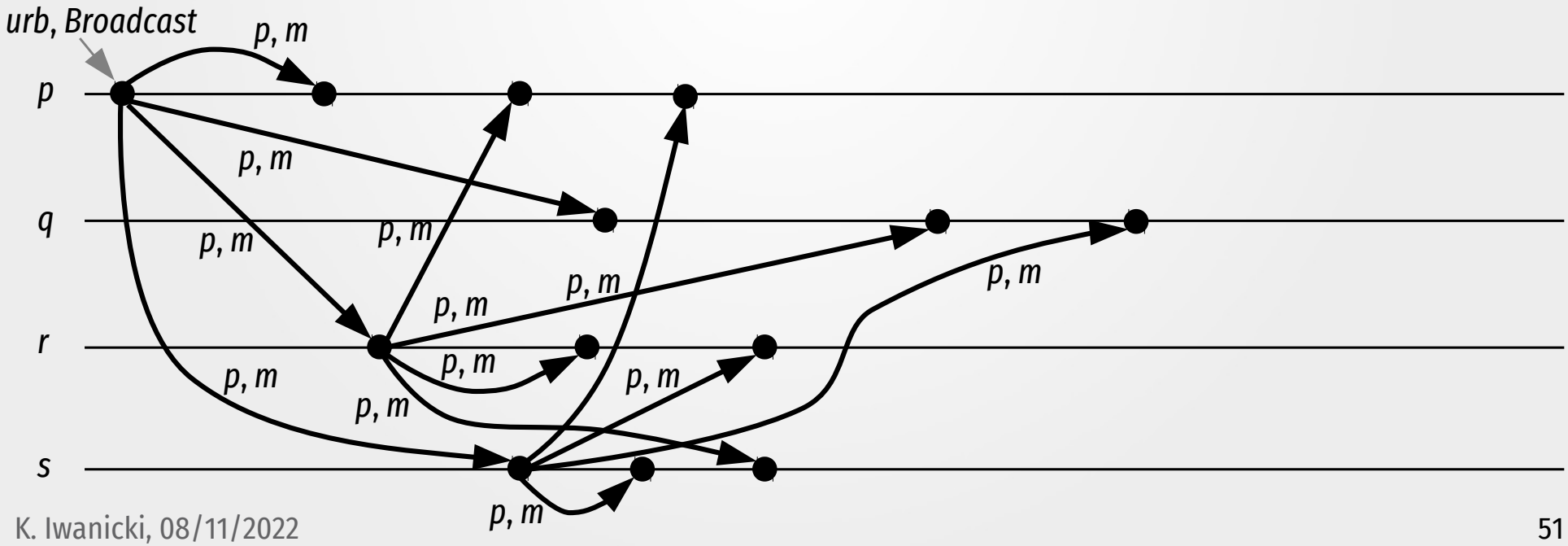
Uniform Reliable Broadcast

- Principal idea: Ensure that all correct processes have received a message before delivering it locally.



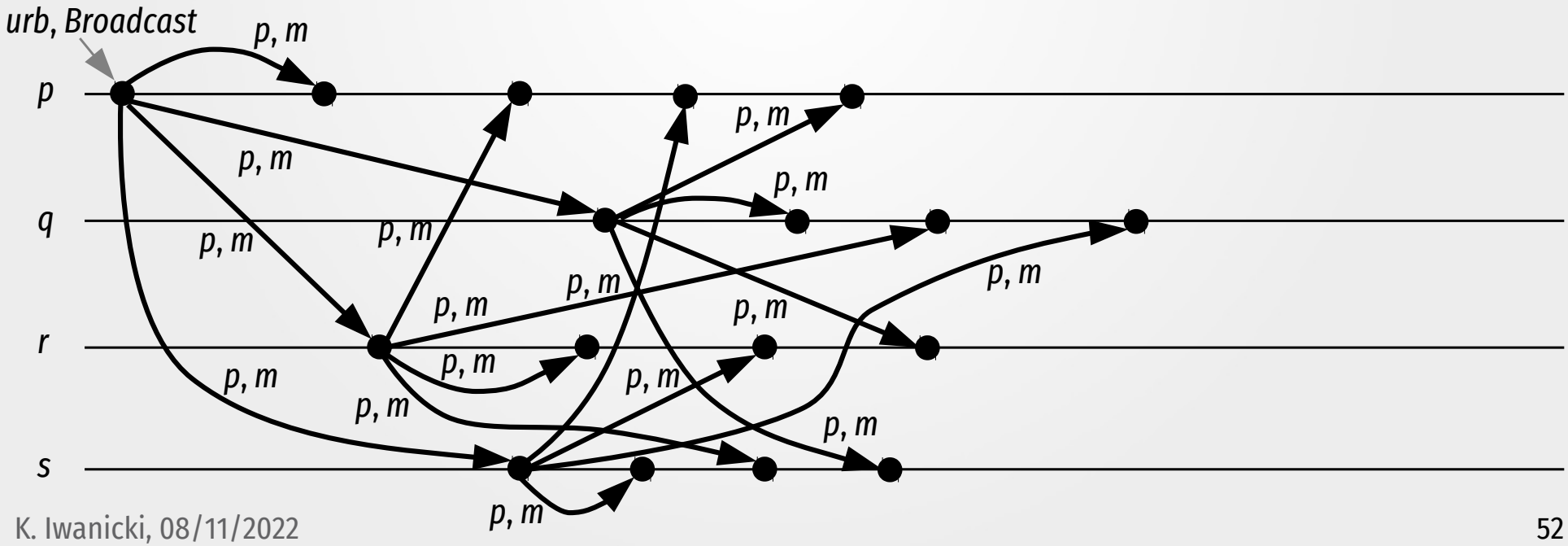
Uniform Reliable Broadcast

- Principal idea: Ensure that all correct processes have received a message before delivering it locally.



Uniform Reliable Broadcast

- Principal idea: Ensure that all correct processes have received a message before delivering it locally.



Uniform Reliable Broadcast

Algorithm: All-Ack Uniform Reliable Broadcast

Implements:
UniformReliableBroadcast, **instance** *urb*.

Uses:
BestEfforBroadcast, **instance** *beb*.
PerfectFailureDetector, **instance** $\mathcal{P}$.

upon event ⟨*urb*, *Init*⟩ **do**
delivered := ∅;
pending := ∅;
correct := Π;
forall *m* **do**
ack[*m*] := ∅;

upon event ⟨*urb*, *Broadcast* | *m*⟩ **do**
pending := *pending* ∪ { (*self*, *m*) };
trigger ⟨*beb*, *Broadcast* | [Data, *self*, *m*]⟩;

upon event ⟨$\mathcal{P}$, *Crash* | *p*⟩ **do**
correct := *correct* \ { *p* };

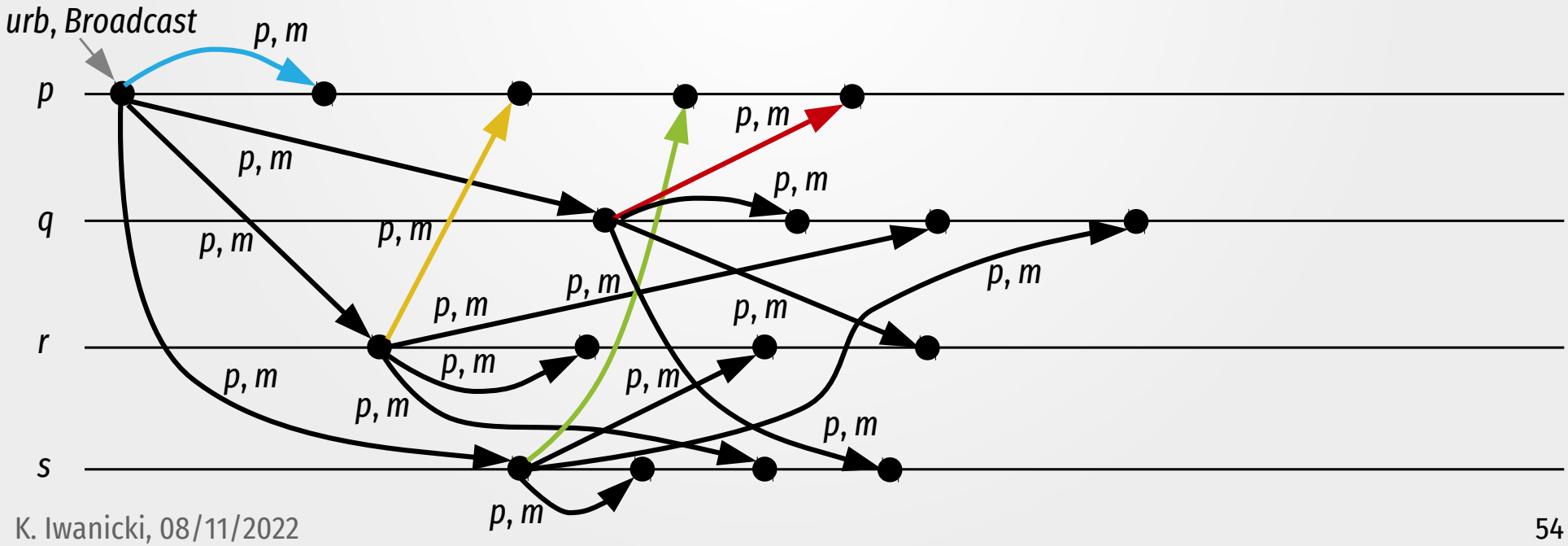
upon event ⟨*beb*, *Deliver* | *p*, [Data, *s*, *m*]⟩ **do**
ack[*m*] := *ack*[*m*] ∪ { *p* };
if (*s*, *m*) ∉ *pending* **then**
pending := *pending* ∪ { (*s*, *m*) };
trigger ⟨*beb*, *Broadcast* | [Data, *s*, *m*]⟩;

fn *candeliver*(m) **is**
return *correct* ⊆ *ack*[*m*];

upon exists (*s*, *m*) ∈ *pending* **such that**
candeliver(m) ∧ *m* ∉ *delivered* **do**
delivered := *delivered* ∪ { *m* };
trigger ⟨*urb*, *Deliver* | *s*, *m*⟩;

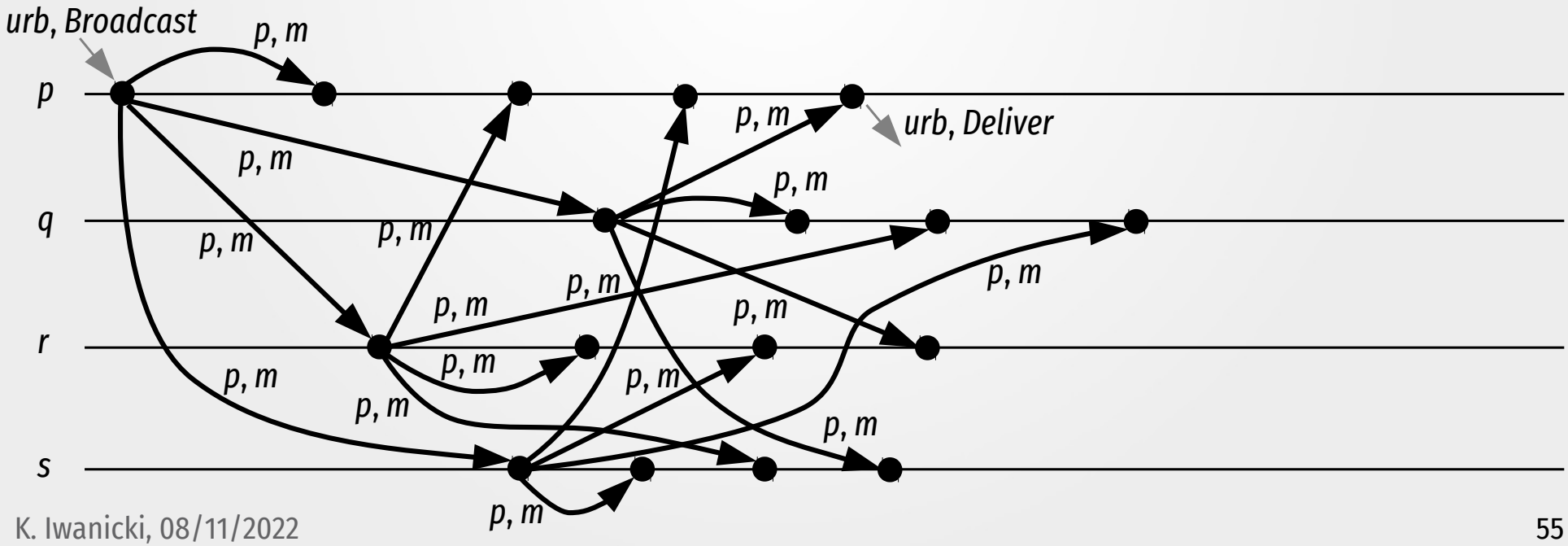
Uniform Reliable Broadcast

- Principal idea: Ensure that all correct processes have received a message before delivering it locally.



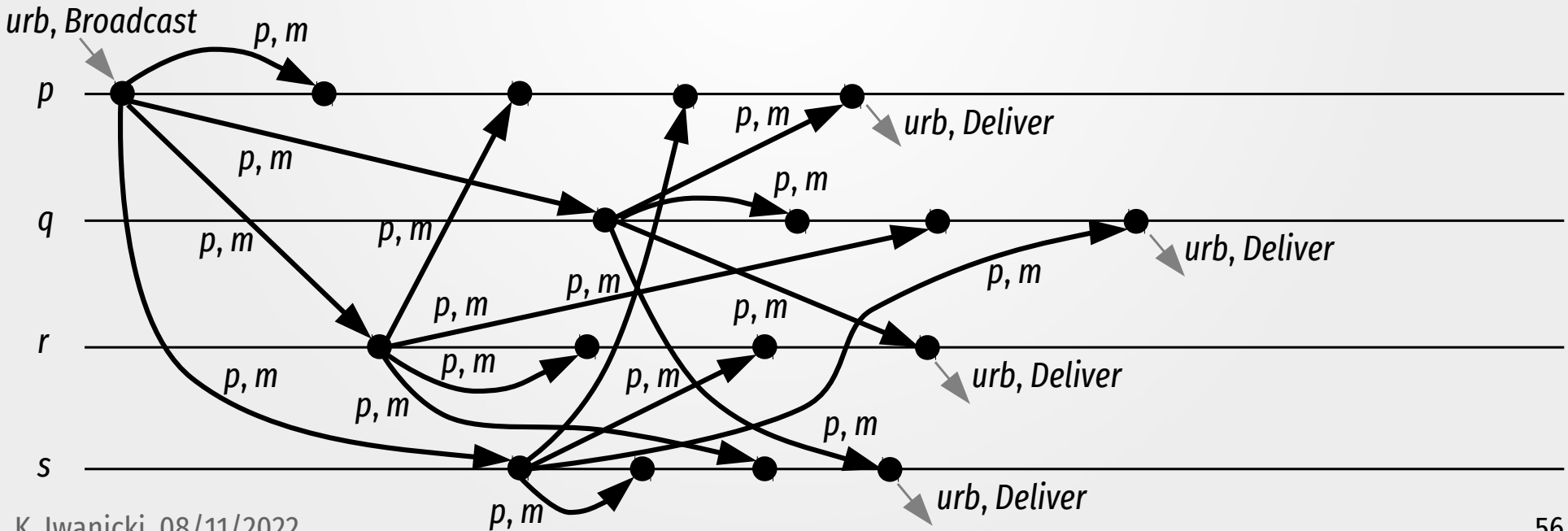
Uniform Reliable Broadcast

- Principal idea: Ensure that all correct processes have received a message before delivering it locally.



Uniform Reliable Broadcast

- Principal idea: Ensure that all correct processes have received a message before delivering it locally.



Uniform Reliable Broadcast

Correctness of the algorithm:

- No creation – follows directly from the no creation property of BEB.
- Validity:
  - If a correct process, p, URB-broadcasts m, then every correct process BEB-delivers m (validity of BEB).
  - Upon the first delivery, every correct process BEB-rebroadcasts m (the algorithm), which is BEB-delivered by p (again, validity of BEB).
  - In contrast, every crashed process is eventually detected (completeness of $\mathcal{P}$).
  - Therefore, relation $correct \subseteq ack[m]$ eventually holds at p and p thus URB-delivers m.
- No duplication:
  - Messages broadcast by different processes are unique.
  - Each process maintains variable *delivered* that filters out URB-delivered duplicates originating at the same broadcaster.

Uniform Reliable Broadcast

Correctness of the algorithm (cont.):

- Uniform agreement – by contradiction:
  - Suppose that some correct process, p, does never URB-deliver a message, m, that has been URB-delivered by another process, q.
  - From the accuracy property of $\mathcal{P}$, q must have BEB-delivered m from every correct process (including p).
  - From the no creation property of BEB, every correct process (including p) must have BEB-(re-)broadcast m.
  - However, from the validity property of BEB, p must have also BEB-delivered all these (re-)broadcasts from the correct processes and accumulated the process identifiers in its $ack[m]$ variable.
  - The strong completeness property of $\mathcal{P}$, in turn, guarantees p to detect all crashed processes, and hence to ensure eventually the following relation at p: $correct \subseteq ack[m]$.
  - Being correct, process p will thus URB-deliver m – contradiction!

Uniform Reliable Broadcast

Performance of the algorithm:

- The work of the algorithm is N^2.
- The span of the algorithm is 2.

Uniform Reliable Broadcast

Question: Can we get rid of the failure detector?

Uniform Reliable Broadcast

Algorithm: Majority-Ack Uniform Reliable Broadcast

Implements:
UniformReliableBroadcast, **instance** *urb*.

Uses:
BestEfforBroadcast, **instance** *beb*.
~~PerfectFailureDetector, **instance** *P*.~~

upon event ⟨*urb*, *Init*⟩ **do**
delivered := ∅;
pending := ∅;
~~*correct* := Π;~~
forall *m* **do**
ack[*m*] := ∅;

upon event ⟨*urb*, *Broadcast* | *m*⟩ **do**
pending := *pending* ∪ { (*self*, *m*) };
trigger ⟨*beb*, *Broadcast* | [Data, *self*, *m*]⟩;

~~**upon event** ⟨*P*, *Crash* | *p*⟩ **do**~~
~~*correct* := *correct* \ { *p* };~~

upon event ⟨*beb*, *Deliver* | *p*, [Data, *s*, *m*]⟩ **do**
ack[*m*] := *ack*[*m*] ∪ { *p* };
if (*s*, *m*) ∉ *pending* **then**
pending := *pending* ∪ { (*s*, *m*) };
trigger ⟨*beb*, *Broadcast* | [Data, *s*, *m*]⟩;

~~**fn** *candeliver*(m) **is**~~
~~**return** *correct* ⊆ *ack*[*m*];~~

fn *candeliver*(m) **is**
return #(*ack*[*m*]) > *N* / 2;

upon exists (*s*, *m*) ∈ *pending* **such that**
candeliver(m) ∧ *m* ∉ *delivered* **do**
delivered := *delivered* ∪ { *m* };
trigger ⟨*urb*, *Deliver* | *s*, *m*⟩;

Uniform Reliable Broadcast

Algorithm: Majority-Ack Uniform Reliable Broadcast

Implements:
UniformReliableBroadcast, **instance** *urb*.

Uses:
BestEfforBroadcast, **instance** *beb*.
~~PerfectFailureDetector, **instance** *P*.~~

upon event ⟨*urb*, *Init*⟩ **do**
delivered := ∅;
pending := ∅;
~~*correct* := Π;~~
forall *m* **do**
ack[*m*] := ∅;

upon event ⟨*urb*, *Broadcast* | *m*⟩ **do**
pending := *pending* ∪ { (*self*, *m*) };
trigger ⟨*beb*, *Broadcast* | [Data, *self*, *m*]⟩;

~~**upon event** ⟨*P*, *Crash* | *p*⟩ **do**~~
~~*correct* := *correct* \ { *p* };~~

upon event ⟨*beb*, *Deliver* | *p*, [Data, *s*, *m*]⟩ **do**
ack[*m*] := *ack*[*m*] ∪ { *p* };
if (*s*, *m*) ∉ *pending* **then**
pending := *pending* ∪ { (*s*, *m*) };
trigger ⟨*beb*, *Broadcast* | [Data, *s*, *m*]⟩;

~~**fn** *candeliver*(m) **is**~~
~~**return** *correct* ⊆ *ack*[*m*];~~

fn *candeliver*(m) **is**
return #(*ack*[*m*]) > *N* / 2;

upon exists (*s*, *m*) ∈ *pending* **such that**
candeliver(m) ∧ *m* ∉ *delivered* **do**
delivered := *delivered* ∪ { *m* };
trigger ⟨*urb*, *Deliver* | *s*, *m*⟩;

Question: What do we lose here?

Regular Reliable Broadcast

Module:
Name: ReliableBroadcast, **instance** *rb*.

Events:
Request: ⟨*rb*, *Broadcast* | *m*⟩: Broadcasts a message, *m*, to all processes.
Indication: ⟨*rb*, *Deliver* | *p*, *m*⟩: Delivers a message, *m*, broadcast by process *p*.

Properties:
RB1: *validity*: If a correct process, *p*, broadcasts a message, m, then process *p* eventually delivers *m*.
RB2: *no duplication*: No message is delivered more than once.
RB3: *no creation*: If a process delivers a message, *m*, with sender *s*, then *m* must have been previously broadcast by process *s*.
RB4: *agreement*: If a message, *m*, is delivered by some correct process, then *m* is eventually delivered by every correct process.

Question: What about message ordering?

FIFO Reliable Broadcast

Module:
Name: FIFOReliableBroadcast, **instance** *frb*.

Events:
Request: ⟨*frb*, *Broadcast* | *m*⟩: Broadcasts a message, *m*, to all processes.
Indication: ⟨*frb*, *Deliver* | *p*, *m*⟩: Delivers a message, *m*, broadcast by process *p*.

Properties:
FRB1: *validity*: If a correct process, *p*, broadcasts a message, *m*, then process *p* eventually delivers *m*.
FRB2: *no duplication*: No message is delivered more than once.
FRB3: *no creation*: If a process delivers a message, *m*, with sender *s*, then *m* must have been previously broadcast by process *s*.
FRB4: *agreement*: If a message, *m*, is delivered by some correct process, then *m* is eventually delivered by every correct process.
FRB5: *FIFO delivery*: If some process broadcasts message m_1 before it broadcasts message m_2, then no correct process delivers m_2 unless it has already delivered m_1.

FIFO Reliable Broadcast

Algorithm: Broadcast with Sequence Number

Implements:
FIFOReliableBroadcast, **instance** *frb*.

Uses:
ReliableBroadcast, **instance** *rb*.

upon event ⟨*frb*, *Init*⟩ **do**
lsn := 0;
pending := ∅;
next := $[1]^N$;

upon event ⟨*frb*, *Broadcast* | *m*⟩ **do**
lsn := *lsn* + 1;
trigger ⟨*rb*, *Broadcast* | [Data, *self*, *m*, *lsn*]⟩;

upon event ⟨*rb*, *Deliver* | *p*, [Data, *s*, *m*, *sn*]⟩ **do**
pending := *pending* ∪ { (*s*, *m*, *sn*) };
while exists (*s*, *m'*, *sn'*) ∈ *pending* **such that** *sn'* = *next*[*s*] **do**
next[*s*] := *next*[*s*] + 1;
pending := *pending* \ { (*s*, *m'*, *sn'*) };
trigger ⟨*frb*, *Deliver* | *s*, *m'*⟩;

FIFO Reliable Broadcast

Correctness of the algorithm:

- Validity, no creation, no duplication, agreement – follows directly from properties of reliable broadcast.
- FIFO delivery:
  - A process FRB-delivers RB-delivered message from a given process, *p*, if and only if this is the next message in the message sequence of *p*.

FIFO Reliable Broadcast

Performance of the algorithm is the same as of the underlying reliable broadcast algorithm.

FIFO Reliable Broadcast

Module:
Name: FIFOReliableBroadcast, **instance** *frb*.

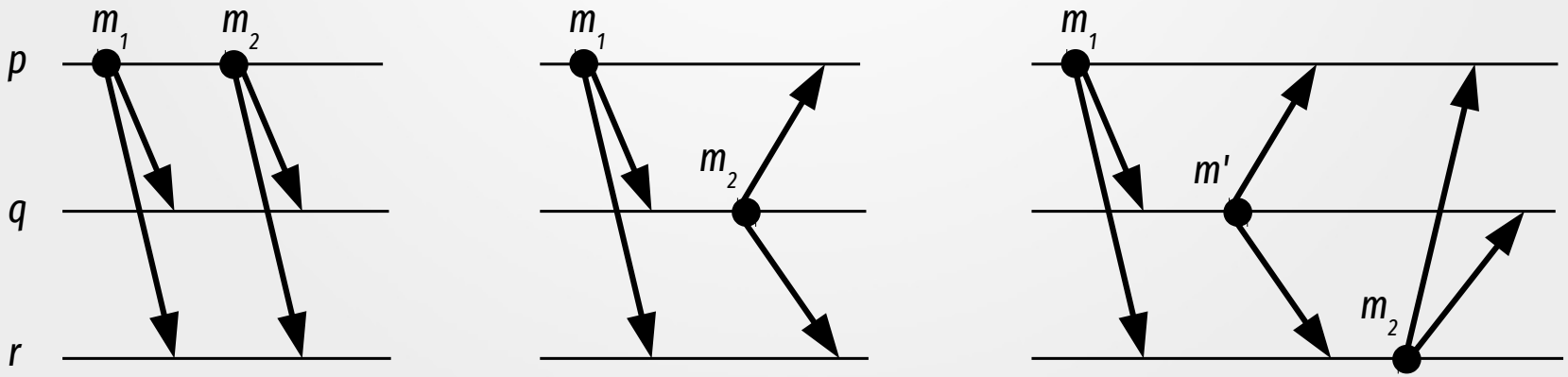
Events:
Request: ⟨*frb*, *Broadcast* | *m*⟩: Broadcasts a message, *m*, to all processes.
Indication: ⟨*frb*, *Deliver* | *p*, *m*⟩: Delivers a message, *m*, broadcast by process *p*.

Properties:
FRB1: *validity*: If a correct process, *p*, broadcasts a message, *m*, then process *p* eventually delivers *m*.
FRB2: *no duplication*: No message is delivered more than once.
FRB3: *no creation*: If a process delivers a message, *m*, with sender *s*, then *m* must have been previously broadcast by process *s*.
FRB4: *agreement*: If a message, *m*, is delivered by some correct process, then *m* is eventually delivered by every correct process.
FRB5: *FIFO delivery*: If some process broadcasts message m_1 before it broadcasts message m_2, then no correct process delivers m_2 unless it has already delivered m_1.

Question: What about messages originating at different processes? Do we have any ordering guarantees?

Causal-order Reliable Broadcast

- Idea: Order messages to *ensure* that their delivery respects all cause-effect relations.
- We say that a message, m_1, may have ***potentially caused*** another message, m_2, denoted as $m_1 \rightarrow m_2$, if any of the following apply:
  - some process p broadcasts m_1 before it broadcasts m_2;
  - some process p delivers m_1 and subsequently broadcasts m_2;
  - there exists some message m' such that $m_1 \rightarrow m'$ and $m' \rightarrow m_2$.
- We will refer to an abstraction ensuring this order as **causal(ly)-order(ed) reliable broadcast.**



Causal-order Reliable Broadcast

Module:
Name: CausalOrderReliableBroadcast, **instance** *crb*.

Events:
Request: ⟨*crb*, *Broadcast* | *m*⟩: Broadcasts a message, *m*, to all processes.
Indication: ⟨*crb*, *Deliver* | *p*, *m*⟩: Delivers a message, *m*, broadcast by process *p*.

Properties:
FRB1: *validity*: If a correct process, *p*, broadcasts a message, *m*, then process *p* eventually delivers *m*.
FRB2: *no duplication*: No message is delivered more than once.
FRB3: *no creation*: If a process delivers a message, *m*, with sender *s*, then *m* must have been previously broadcast by process *s*.
FRB4: *agreement*: If a message, *m*, is delivered by some correct process, then *m* is eventually delivered by every correct process.
FRB5: *causal delivery*: For any message m_1 that potentially causaly precedes message m_2, no process delivers m_2 unless it has already delivered m_1.

Causal-order Reliable Broadcast

Algorithm: No-Waiting Causal Broadcast

Implements:
CausalOrderReliableBroadcast, **instance** *crb*.

Uses:
ReliableBroadcast, **instance** *rb*.

upon event ⟨*crb*, *Init*⟩ **do**
delivered := ∅;
past := [ ];

upon event ⟨*crb*, *Broadcast* | *m*⟩ **do**
trigger ⟨*rb*, *Broadcast* | [Data, *past*, *m*]⟩;
append(*past*, (*self*, *m*));

upon event ⟨*rb*, *Deliver* | *p*, [Data, *mpast*, *m*]⟩ **do**
if *m* ∉ *delivered* **then**
forall (*s*, *n*) ∈ *mpast* **do**
if *n* ∉ *delivered* **then**
trigger ⟨*crb*, *Deliver* | *s*, *n*⟩;
delivered := *delivered* ∪ { *n* };
if (*s*, *n*) ∉ *past* **then**
append(*past*, (*s*, *n*));
trigger ⟨*crb*, *Deliver* | *p*, *m*⟩;
delivered := *delivered* ∪ { *m* };
if (*p*, *m*) ∉ *past* **then**
append(*past*, (*p*, *m*));

Causal-order Reliable Broadcast

Correctness of the algorithm:

- Validity, no creation, no duplication, agreement – follows directly from properties of reliable broadcast and the fact that a message is CRB-delivered immediately after being RB-delivered.
- Causal delivery:
  - Each message carries also its causal past.
  - Before CRB-delivering the message, the processes CRB-delivers any message from the causal past that it has not yet CRB-delivered.

Causal-order Reliable Broadcast

Performance of the algorithm is the same as of the underlying reliable broadcast algorithm.(*)

Causal-order Reliable Broadcast

Performance of the algorithm is the same as of the underlying reliable broadcast algorithm.(*)

Question: What is the (*) about?

Causal-order Reliable Broadcast

Performance of the algorithm is the same as of the underlying reliable broadcast algorithm.(*)

Question: What is the (*) about?

(*) Carrying all causally preceding messages, a message can become arbitrarily long.

Causal-order Reliable Broadcast

Performance of the algorithm is the same as of the underlying reliable broadcast algorithm.(*)

Question: What is the (*) about?

(*) Carrying all causally preceding messages, a message can become arbitrarily long.

Question: Can be the causal past be garbage-collected?

Causal-order Reliable Broadcast

Performance of the algorithm is the same as of the underlying reliable broadcast algorithm.(*)

Question: What is the (*) about?

(*) Carrying all causally preceding messages, a message can become arbitrarily long.

Question: Can be the causal past be garbage-collected?

Idea: Acknowledge deliveries of each message to ensure that no more will be necessary.

Causal-order Reliable Broadcast

Algorithm: Garbage-Collection of Causal Past

Implements:
CausalOrderReliableBroadcast, **instance** *crb*.

Uses:
ReliableBroadcast, **instance** *rb*.
PerfectFailureDetector, **instance** $\mathcal{P}$.

upon event ⟨*crb*, *Init*⟩ **do**
  delivered := ∅;
  past := [ ];
  correct := *Π*;
  forall *m* **do**
    ack[*m*] := ∅;

upon event ⟨*crb*, *Broadcast* | *m*⟩ **do**
  trigger ⟨*rb*, *Broadcast* | [Data, *past*, *m*]⟩;
  append(*past*, (*self*, *m*));

upon event ⟨$\mathcal{P}$, *Crash* | *p*⟩ **do**
  correct := *correct* \ { *p* };

upon event ⟨*rb*, *Deliver* | *p*, [Data, *mpast*, *m*]⟩ **do**
  if *m* ∉ *delivered* **then**
    forall (*s*, *n*) ∈ *mpast* **do**
      if *n* ∉ *delivered* **then**
        trigger ⟨*crb*, *Deliver* | *s*, *n*⟩;
        delivered := *delivered* ∪ { *n* };
        if (*s*, *n*) ∉ *past* **then**
          append(*past*, (*s*, *n*));
    trigger ⟨*crb*, *Deliver* | *p*, *m*⟩;
    delivered := *delivered* ∪ { *m* };
    if (*p*, *m*) ∉ *past* **then**
      append(*past*, (*p*, *m*));

upon exists *m* ∈ *delivered* **such that** *self* ∉ *ack*[*m*] **do**
  ack[*m*] := *ack*[*m*] ∪ { *self* };
  trigger ⟨*rb*, *Broadcast* | [Ack, *m*]⟩;

upon event ⟨*rb*, *Deliver* | *p*, [Ack, *m*]⟩ **do**
  ack[*m*] := *ack*[*m*] ∪ { *p* };

upon exists *m* ∈ *delivered* **such that** *correct* ⊆ *ack*[*m*] **do**
  forall (*s'*, *m'*) ∈ *past* **such that** *m'* = *m* **do**
    remove(*past*, (*s'*, *m*));

Causal-order Reliable Broadcast

Question: Can we do this better?

Question: Can we get rid of the failure detector?

Causal-order Reliable Broadcast

- Idea: Associate a fixed-size logical timestamp with each message such that, for any m_1 and m_2, $ts(m_1) < ts(m_2)$ if and only if $m_1 \rightarrow m_2$.
- More specifically, $ts(m)$ is a vector of N nonnegative numbers, each corresponding to a process.
- The value of $ts(m)[rank(p)]$ equals the number of messages originating at process p that have potentially caused m.
- Relation $ts(m) < ts(m')$ holds if and only if all of the following hold:
  - For all $k \in \{1, ..., N\}$, $ts(m)[k] \leq ts(m')[k]$ and
  - There exists $k \in \{1, ..., N\}$, such that $ts(m)[k] < ts(m')[k]$.
- This mechanism is referred to as ***vector clocks***.

Causal-order Reliable Broadcast

Algorithm: Waiting Causal Broadcast

Implements:
CausalOrderReliableBroadcast, **instance** *crb*.

Uses:
ReliableBroadcast, **instance** *rb*.

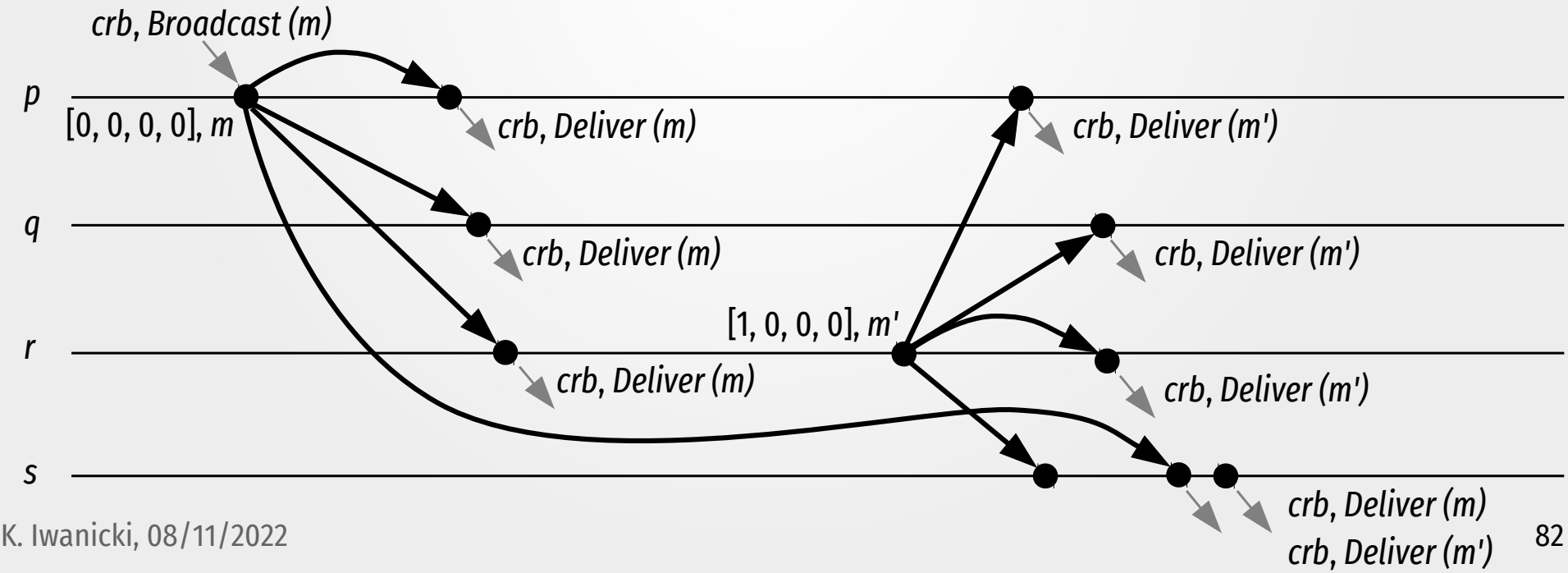
upon event ⟨*crb*, *Init*⟩ **do**
V := [0]^N;
lsn := 0;
pending := ∅;

upon event ⟨*crb*, *Broadcast* | *m*⟩ **do**
W := *V*;
W[*rank*(*self*)] := *lsn*;
lsn := *lsn* + 1;
trigger ⟨*rb*, *Broadcast* | [Data, *W*, *m*]⟩;

upon event ⟨*rb*, *Deliver* | *p*, [Data, *W*, *m*]⟩ **do**
pending := *pending* ∪ { (*p*, *W*, *m*) };
while exists (*p'*, *W'*, *m'*) ∈ *pending*
such that *W'* ≤ *V* **do**
pending := *pending* \ { (*p'*, *W'*, *m'*) };
V[*rank*(*p'*)] := *V*[*rank*(*p'*)] + 1;
trigger ⟨*crb*, *Deliver* | *p'*, *m'*⟩;

Causal-order Reliable Broadcast

- In the absence of failures, the algorithm works as the previous one.



Causal-order Reliable Broadcast

Correctness of the algorithm:

- Validity:
  - Consider a message, m, that is CRB-broadcast by process p.
  - From the validity property of reliable broadcast, process p eventually RB-delivers m.
  - Vector W that is RB-delivered together with m is equal to vector V of p when p RB-broadcast m.
  - Since the elements of V may only increase, it holds that $W \leq V$.
  - Therefore, process p CRB-delivers m immediately.
- No creation and no duplication – follow directly from properties of reliable broadcast.
- Agreement:
  - Consider a message m that is CRB-delivered by some correct process p.
  - Because of the agreement property of the underlying reliable broadcast, every correct process eventually RB-delivers m.
  - For the same reason, every correct process eventually RB-delivers every message that causally precedes m.
  - In effect, every correct process eventually CRB-delivers m.

Causal-order Reliable Broadcast

Correctness of the algorithm (cont.):

- Causal delivery:
  - Vector *V* at process *p* stores the number of CRB-delivered messages with sender *q* in element *V*[*rank*(*q*)].
  - Process *p* assigns a sequence number (starting at 0) to every message that it RB-broadcasts in element *rank*(*p*) of the attached vector.
  - When *p* RB-broadcasts a message, *m*, with attached vector *W* computed like this, then *W*[*rank*(*q*)] messages from sender *q* causally precede *m*.
  - But every RB-receiver of *m* also counts the number of messages that it has CRB-delivered from sender q and waits until V[rank(q)] such messages have been CRB-delivered before CRB

Causal-order Reliable Broadcast

Performance of the algorithm is the same as of the underlying reliable broadcast algorithm.

Considered Failure Model

Crash-recovery failures

Stubborn Best-effort Broadcast

Module:
Name: StubbornBestEfforBroadcast, **instance** *sbeb*.

Events:
Request: ⟨*sbeb*, *Broadcast* | *m*⟩: Broadcasts a message, *m*, to all processes.
Indication: ⟨*sbeb*, *Deliver* | *p*, *m*⟩: Delivers a message, *m*, broadcast by process *p*.

Properties:
SBEB1: *best-effort validity*: If a correct process broadcasts a message, *m*, and never crashes afterward, then every correct process delivers *m* an infinite number of times.
SBEB2: *no creation*: If a process delivers a message, *m*, with sender *s*, then *m* must have been previously broadcast by process *s*.

Stubborn Best-effort Broadcast

Algorithm: Basic Stubborn Broadcast

Implements:
StubbornBestEfforBroadcast, **instance** *sbeb*.

Uses:
StubbornLinks, **instance** *sl*.

upon event ⟨*sbeb*, *Broadcast* | *m*⟩ **do**
  forall $q \in \Pi$ **do**
    trigger ⟨*sl*, *Send* | *q*, *m*⟩;

upon event ⟨*sl*, *Deliver* | *p*, *m*⟩ **do**
  trigger ⟨*sbeb*, *Deliver* | *p*, *m*⟩;

Stubborn Best-effort Broadcast

Correctness of the algorithm – Cheatsheet:

Module:
Name: StubbornLinks, **instance** *sl*.

Events:
Request: ⟨*sl*, *Send* | *q*, *m*⟩: Requests to send message *m* to process *q*.
Indication: ⟨*sl*, *Deliver* | *p, m*⟩: Delivers message *m* from process *p*.

Properties:
SL1: *stubborn delivery*: If a correct process, *p*, sends a message, *m*, to a correct process, *q*, (and never crashes afterward), then *q* delivers *m* an infinite number of times.
SL2: *no creation*: If some process, *q*, delivers a message, *m*, with sender *p*, then *m* must have been previously sent to *q* by process *p*.

Stubborn Best-effort Broadcast

Correctness of the algorithm:

- Best-effort validity and no creation - trivially derived from the properties of stubborn links and the fact that a broadcast message is sent to every process.

Stubborn Best-effort Broadcast

Performance of the algorithm:

- The work of the algorithm is at least N, depending on the number of retransmissions.
- The span of the algorithm is at least 1, again, depending on the number of retransmissions.

Logged Best-effort Broadcast

Module:
Name: LoggedBestEfforBroadcast, **instance** *lbeb*.

Events:
Request: ⟨*lbeb*, *Broadcast* | *m*⟩: Broadcasts a message, *m*, to all processes.
Indication: ⟨*lbeb*, *Deliver* | *delivered*⟩: Notifies of potential updates to variable *delivered* in stable storage.

Properties:
LBEB1: *validity*: If a correct process broadcasts a message, m, and never crashes afterward, then every correct process eventually log-delivers *m*.
LBEB2: *no duplication*: No message is log-delivered more than once.
LBEB3: *no creation*: If a process log-delivers a message, *m*, with sender *s*, then *m* must have been previously broadcast by process *s*.

Logged Best-effort Broadcast

Algorithm: Logged Basic Broadcast

Implements:
LoggedBestEfforBroadcast, **instance** *lbeb*.

Uses:
StubbornLinks, **instance** *sl*.

upon event ⟨*lbeb*, *Init*⟩ **do**
delivered := ∅;
store(*delivered*);

upon event ⟨*lbeb*, *Recovery*⟩ **do**
retrieve(*delivered*);
trigger ⟨*lbeb*, *Deliver* | *delivered*⟩;

upon event ⟨*lbeb*, *Broadcast* | *m*⟩ **do**
forall $q \in \Pi$ **do**
trigger ⟨*sl*, *Send* | *q*, *m*⟩;

upon event ⟨*sl*, *Deliver* | *p*, *m*⟩ **do**
if $(p, m) \notin$ *delivered* **then**
delivered := *delivered* ∪ { (*p*, *m*) };
store(*delivered*);
trigger ⟨*lbeb*, *Deliver* | *delivered*⟩;

Logged Best-effort Broadcast

Correctness of the algorithm:

- No creation - follows directly from the same property of stubborn links.
- No duplication:
  - Before delivering any message, the log of the process is checked.
  - Messages in the log are not delivered again.
- Validity:
  - The broadcaster sends the message to every process in the system.
  - The stubborn delivery property of stubborn links guarantees that each such message is delivered an infinite number of times by the receiver unless it is faulty.

Logged Best-effort Broadcast

Performance of the algorithm:

- The work of the algorithm is at least N, depending on the number of retransmissions.
- The span of the algorithm is at least 1, again, depending on the number of retransmissions.

In addition, log operations are required.

Logged Best-effort Broadcast

Module:
Name: LoggedBestEfforBroadcast, **instance** *lbeb*.

Events:
Request: ⟨*lbeb*, *Broadcast* | *m*⟩: Broadcasts a message, *m*, to all processes.
Indication: ⟨*lbeb*, *Deliver* | *delivered*⟩: Notifies of potential updates to variable *delivered* in stable storage.

Properties:
LBEB1: *validity*: If a correct process broadcasts a message, m, and never crashes afterward, then every correct process eventually log-delivers *m*.
LBEB2: *no duplication*: No message is log-delivered more than once.
LBEB3: *no creation*: If a process log-delivers a message, *m*, with sender *s*, then *m* must have been previously broadcast by process *s*.

Question: What about reliability?

Logged Uniform Reliable Broadcast

Module:
Name: LoggedUniformReliableBroadcast, **instance** *lurb*.

Events:
Request: ⟨*lurb*, *Broadcast* | *m*⟩: Broadcasts a message, *m*, to all processes.
Indication: ⟨*lurb*, *Deliver* | *delivered*⟩: Notifies of potential updates to variable *delivered* in stable storage.

Properties:
LURB1: *validity*: If a correct process, p, broadcasts a message, m, and never crashes afterward, then *p* eventually log-delivers *m*.
LURB2: *no duplication*: No message is log-delivered more than once.
LURB3: *no creation*: If a process delivers a message, *m*, with sender *s*, then *m* must have been previously broadcast by process *s*.
LURB4: *uniform agreement*: If a message, *m*, is log-delivered by some process (correct or faulty), then *m* is eventually log-delivered by every correct process.

Logged Uniform Reliable Broadcast

Algorithm: Logged Majority-Ack Uniform Reliable Broadcast

Implements:
LoggedUniformReliableBroadcast, **instance** *lurb*.

Uses:
StubbornBestEfforBroadcast, **instance** *sbeb*.

upon event ⟨ *lurb*, *Init* ⟩ **do**
  delivered := ∅;
  pending := ∅;
  forall *m* **do**
    ack[*m*] := ∅;
  store(*pending*, *delivered*);

upon event ⟨ *lurb*, *Recovery* ⟩ **do**
  retrieve(*pending*, *delivered*);
  trigger ⟨ *lurb*, *Deliver* | *delivered* ⟩;
  forall (*s*, *m*) ∈ *pending* **do**
    trigger ⟨ *sbeb*, *Broadcast* | [Data, *s*, *m*] ⟩;

upon event ⟨ *lurb*, *Broadcast* | *m* ⟩ **do**
  pending := *pending* ∪ { (*self*, *m*) };
  store(*pending*);
  trigger ⟨ *sbeb*, *Broadcast* | [Data, *self*, *m*] ⟩;

upon event ⟨ *sbeb*, *Deliver* | *p*, [Data, *s*, *m*] ⟩ **do**
  if (*s*, *m*) ∉ *pending* **then**
    pending := *pending* ∪ { (*s*, *m*) };
    store(*pending*);
    trigger ⟨ *sbeb*, *Broadcast* | [Data, *s*, *m*] ⟩;
  if *p* ∉ *ack*[*m*] **then**
    ack[*m*] := *ack*[*m*] ∪ { *p* };
    if #(*ack*[*m*]) > *N* / 2 ∧ (*s*, *m*) ∉ *delivered* **then**
      delivered := *delivered* ∪ { (*s*, *m*) };
      store(*delivered*);
      trigger ⟨ *lurb*, *Deliver* | *delivered* ⟩;

Logged Uniform Reliable Broadcast

Correctness of the algorithm:

- No creation - follows directly from the same property of stubborn links.
- No duplication:
  - Before delivering any message, the delivery log of the process is checked.
  - Messages in the log are not delivered again.
- Validity:
  - Upon LURB-broadcasting message *m*, process *p* SBEB-broadcasts *m* to every process in the system and never crashes afterward.
  - The best-effort validity property of SBEB guarantees that *m* is SBEB-delivered by every correct process an infinite number of times.
  - After SBEB-delivering *m* for the first time, every correct process will SBEB-rebroadcast it. The same is true for every recovery.
  - Since a correct process eventually never crashes, at least the last SBEB-rebroadcast *m* from every correct process is eventually SBEB-delivered by *p*.
  - Assuming that the majority of processes is correct, *p* eventually collects SBEB-rebroadcasts of *m* from these processes and LURB-delivers *m*.

Logged Uniform Reliable Broadcast

Correctness of the algorithm:

- Uniform agreement:
  - Assume that some process, *p*, (correct or faulty) LURB-delivers a message, *m*.
  - To do so, it must have SBEB-delivered *m* from some majority, *C*, of processes.
  - From the no creation property of SBEB, the processes in *C* must have SBEB-broadcast *m*.
  - From the algorithm, the processes in *C* thus must have logged *m* in variable *pending*.
  - Assuming that a majority, *S*, of processes are correct, *S* and *C* must have at least one process, say *q*, in common.
  - From the algorithm every correct process, including *q*, SBEB-broadcasts *m* upon first reception and every recovery.
  - Therefore, from the best-effort validity of SBEB and the algorithm, every process is *S* will SBEB-deliver *m* from every process in *S*.
  - Since *S* represents a majority of processes, every process in *S* will LURB-deliver *m*.

Logged Uniform Reliable Broadcast

Correctness of the algorithm:

- Uniform agreement:
  - Assume that some process, *p*, (correct or faulty) LURB-delivers a message, *m*.
  - To do so, it must have SBEB-delivered *m* from some majority, *C*, of processes.
  - From the no creation property of SBEB, the processes in *C* must have SBEB-broadcast *m*.
  - From the algorithm, the processes in *C* thus must have logged *m* in variable *pending*.
  - Assuming that a majority, *S*, of processes are correct, *S* and *C* must have at least one process, say *q*, in common.
  - From the algorithm every correct process, including *q*, SBEB-broadcasts *m* upon first reception and every recovery.
  - Therefore, from the best-effort validity of SBEB and the algorithm, every process is *S* will SBEB-deliver *m* from every process in *S*.
  - Since *S* represents a majority of processes, every process in *S* will LURB-deliver *m*.

Note: A set containing a majority of processes is an example of a quorum under crash-stop or crash-recovery failures.

Logged Uniform Reliable Broadcast

Performance of the algorithm:

- The work of the algorithm is at least N^2, depending on the number of retransmissions.
- The span of the algorithm is at least 2, again, depending on the number of retransmissions.

In addition, log operations are required.

Summary

We have:

- introduced the various formulations of the reliable broadcast problem;
- presented algorithms addressing the problem under various failure types;
- showed a few possible message orderings;
- demonstrated algorithms guaranteeing them.

Digression

Defining “reliable” need not be straightforward.

Digression

Keep digging

Complex systems fail for complex reasons.

Digression

Keep digging

Complex systems fail for complex reasons.

Question: What other design principles did we make use of?

Next Lecture

- Will be about distributed algorithms for the second of the three fundamental abstractions: storage.
- More specifically, we will analyze various so-called registers.