TÉCNICO LISBOA

Towards a dependable consensus: Broadcast primitives (cont.) and read/write protocols

Highly dependable systems – 2023/24

Lecture 4

Lecturers: Miguel Matos and Rodrigo Miragaia Rodrigues

Byzantine Consistent Broadcast: specification

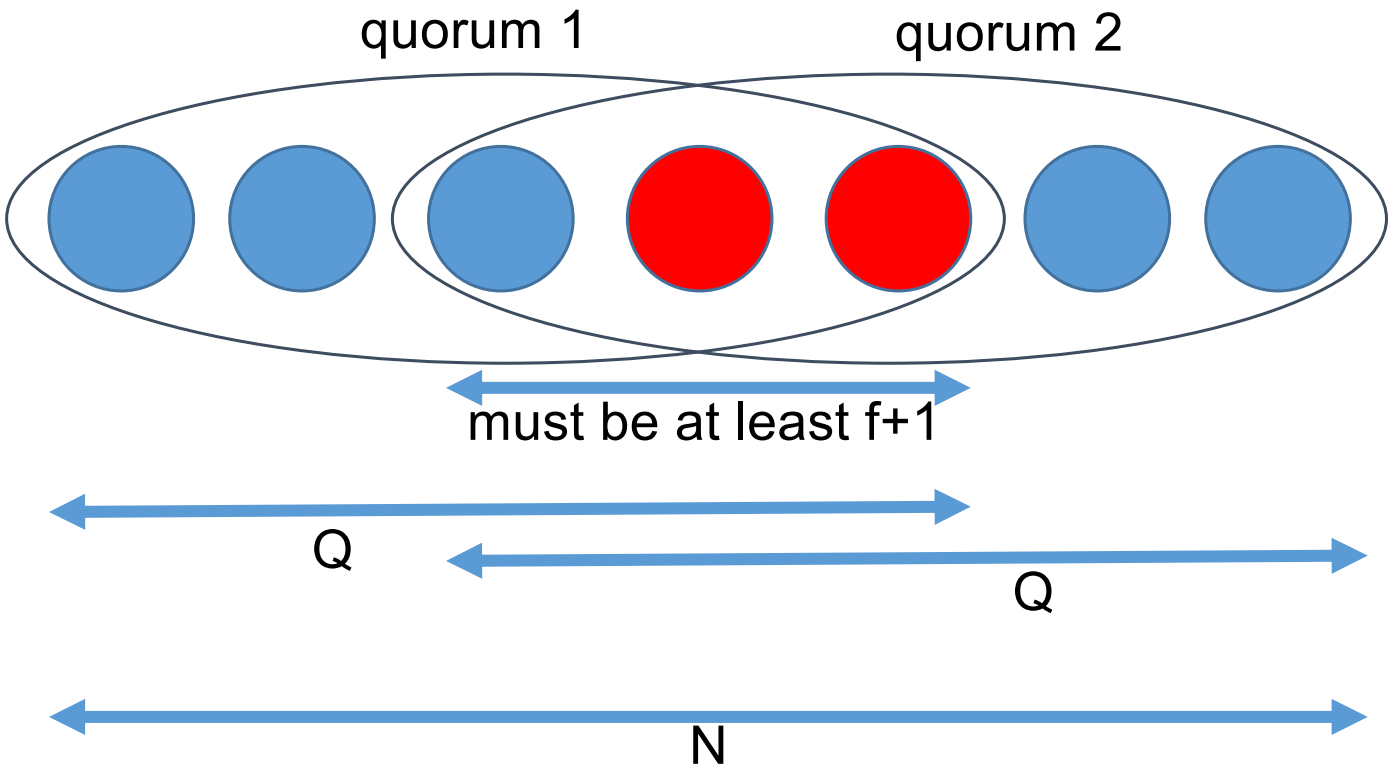
- **BCB1:** *Validity:* If a correct process p broadcasts a msg m, then every correct process eventually delivers m.
- **BCB2:** *No duplication:* Every correct process delivers at most one message.
- **BCB3:** *Integrity:* If some correct process delivers a message m with sender p and process p is correct, then m was previously broadcast by p.

  BCB4: *Consistency:* If some correct process delivers a message m and another correct process delivers a message m′, then m = m′.

Note: "Correct" now means non-Byzantine-faulty

Derivation of Byzantine quorum sizes

- Safety: every pair of quorums must intersect in at least one correct process



Solving a system of two equations

- Note that f is a system parameter (constant)

$$\begin{cases} 2Q - N = f+1 \\ N - f = Q \end{cases}$$

- Thus, solving these equations for N and Q yields:

$$N = 3f+1$$

$$Q = 2f+1$$

- Generically, for N>3f, we have $Q = \lceil (N+f)/2 \rceil$

Authenticated Echo Broadcast

- exploits *Byzantine quorums* (uses N,Q derived before)
  - two rounds of message exchanges.
  - 1st round: sender disseminates msg to all processes.
  - 2nd round:
    - every process acts as a witness for m
    - resends m in an ECHO message to all others.
    - deliver only when received more than a quorum (Q) of ECHOs
      - Byzantine processes cannot cause a correct process to *bcb*-deliver a message m′ != m.
- Relies on authenticated links

Authenticated Echo Broadcast

- **Implements:**

  ByzantineConsistentBroadcast, with sender s.

- **Uses:**

  AuthPerfectPointToPointLinks, **instance** *al*.

- **upon event** ⟨ *bcb*, Init ⟩ **do**

  sentecho := FALSE;

  delivered := FALSE;

  echos := $[\perp]^N$;

Authenticated Echo Broadcast

- **upon event** ⟨*bcb*, Broadcast | m ⟩ **do** *// only process s*

  forall q ∈ Π **do**

  trigger ⟨*al*, Send | q, [SEND, m] ⟩;

- **upon event** ⟨*al*, Deliver | p, [SEND, m] ⟩

  such that p = s **and** *sentecho* = FALSE **do**

  sentecho := TRUE;

  forall q ∈ Π **do**

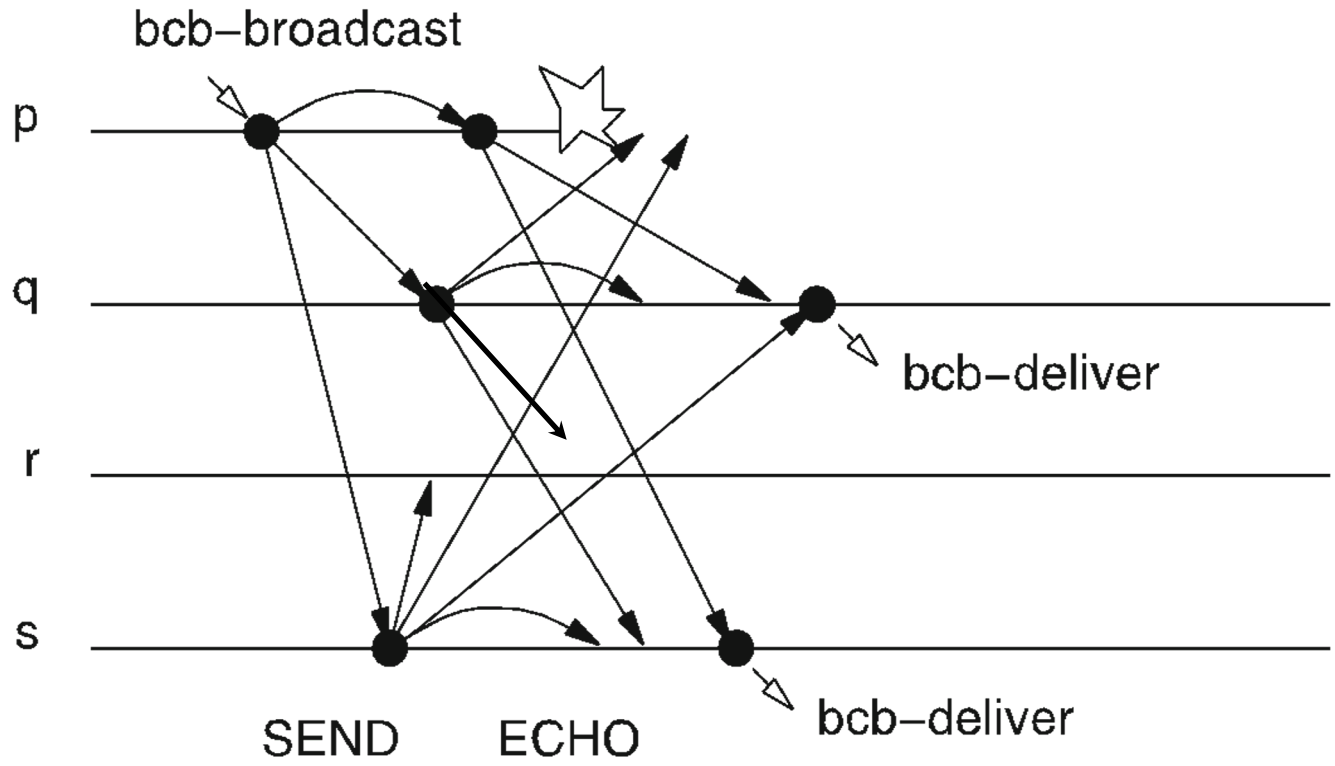
  trigger ⟨*al*, Send | q, [ECHO, m] ⟩;

Authenticated Echo Broadcast

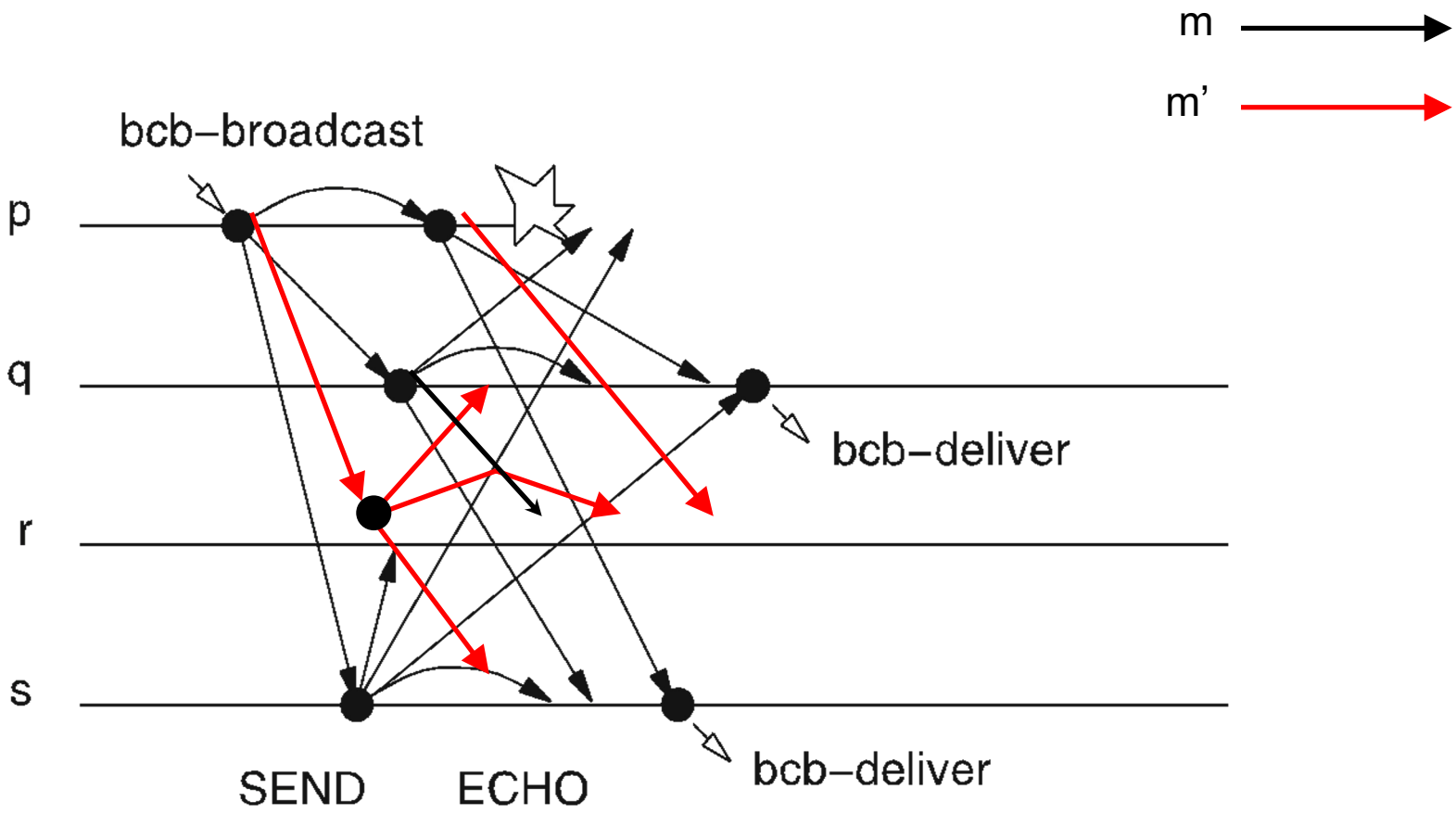
- **upon event** ⟨*al*, Deliver | p, [ECHO, m] ⟩ **do**
  if *echos*[p] = ⊥ **then**
    echos[p] := m;

- **upon exists** m != ⊥ such that :
  #{p ∈ Π |*echos*[p] = m} > (N+f)/2 **and** *delivered* = FALSE **do**
  delivered := TRUE;
  trigger ⟨ *bcb*, Deliver | s, m ⟩;

Authenticated Echo Broadcast



Authenticated Echo Broadcast



r can never deliver m', if q and s deliver m
r should get 3 ECHOES, but can get at most 2 (including its own)

Correctness

BCB1: *Validity:* If a correct process p broadcasts a msg m, then every correct process eventually delivers m.

Proof (sketch)

- if the sender is correct, then every correct process *al*-sends an ECHO message
- every correct process *al*-delivers at least N − f of them.
- N − f > (N + f)/2 under the assumption that N > 3f

  implies that every correct process also *bcb*-delivers the message m contained in the ECHO messages

Correctness

- **BCB2:** *No duplication:* Every correct process delivers at most one message.
- **Proof (sketch):** enforced by the *delivered* variable

Correctness

- **BCB3:** *Integrity:* If some correct process delivers a message m with sender p and process p is correct, then m was previously broadcast by p.

- **Proof (sketch):** by the Authenticated Perfect Link's properties

Correctness

- **BCB4:** *Consistency:* If some correct process delivers a message m and another correct process delivers a message m′, then m = m′.
- For a correct process p to *bcb*-deliver some m, it needs to receive ECHO messages for m from a Byzantine quorum.

Correctness

- Two Byzantine quorums overlap in at least one correct process.
- Assume, by contradiction, that a different correct process p′ *bcb*-delivers some message m′ != m.
  - p′ has received a Byzantine quorum of ECHO messages for m′
  - the correct process in the intersection of the two Byzantine quorums sent different ECHO messages to p and to p′

Contradiction found!

Signed echo broadcast

- Exploits digital signatures:
  - more powerful than MACs
  - allow a third process to verify the authenticity of a message sent from a 1st process s to a 2nd process r
  - avoid quadratic number of message exchanges of prev. algorithm

Signed echo broadcast

- High level idea:
  - Witnesses authenticate a request not by sending an ECHO message to all processes
  - but by signing a statement which they return to the sender
  - Sender collects a Byzantine quorum of these signed statements and relays them in a third communication step to all processes.

Signed Echo Broadcast

- **Implements:**

  ByzantineConsistentBroadcast, **instance** *bcb.*

- **Uses:**

  AuthPerfectPointToPointLinks, **instance** *al*.

- **upon event** ⟨ *bcb*, Init ⟩ **do**

  sentecho := FALSE;

  sentfinal := FALSE;

  delivered := FALSE;

  echos := $[\perp]^N$; $\Sigma := [\perp]^N$;

Signed Echo Broadcast

- **upon event** ⟨ *bcb*, Broadcast | m ⟩ **do** *// only process s*

  forall q ∈ Π **do**

  trigger ⟨ *al*, Send | q, [SEND, m] ⟩;

- **upon event** ⟨ *al*, Deliver | p, [SEND, m] ⟩

  such that p = s **and** *sentecho* = FALSE **do**

  sentecho := TRUE;

  σ = sign(*self*, *bcb* ‖ *self* ‖ ECHO ‖ m);

  trigger ⟨ *al*, Send | p, [ECHO, m], σ ⟩;

algorithm instance identifier *bcb* *is* included in the input argument to the digital signature (else, a Byzantine process might reuse a signature issued by a correct process in a different context)

Signed Echo Broadcast

- *// only process s*

  upon event ⟨ *al*, Deliver | p, [ECHO, m, σ] ⟩ **do**

  if *echos*[p] = ⊥ ∧ verifysig(p, *bcb*‖p‖ECHO‖m, σ)

  then *echos*[p] := m; Σ[p] := σ;

- **upon exists** m != ⊥ such that :

  #{p ∈ Π |*echos*[p] = m}>(N+f)/2 **and** *sentfinal*= FALSE **do**

  sentfinal:= TRUE;

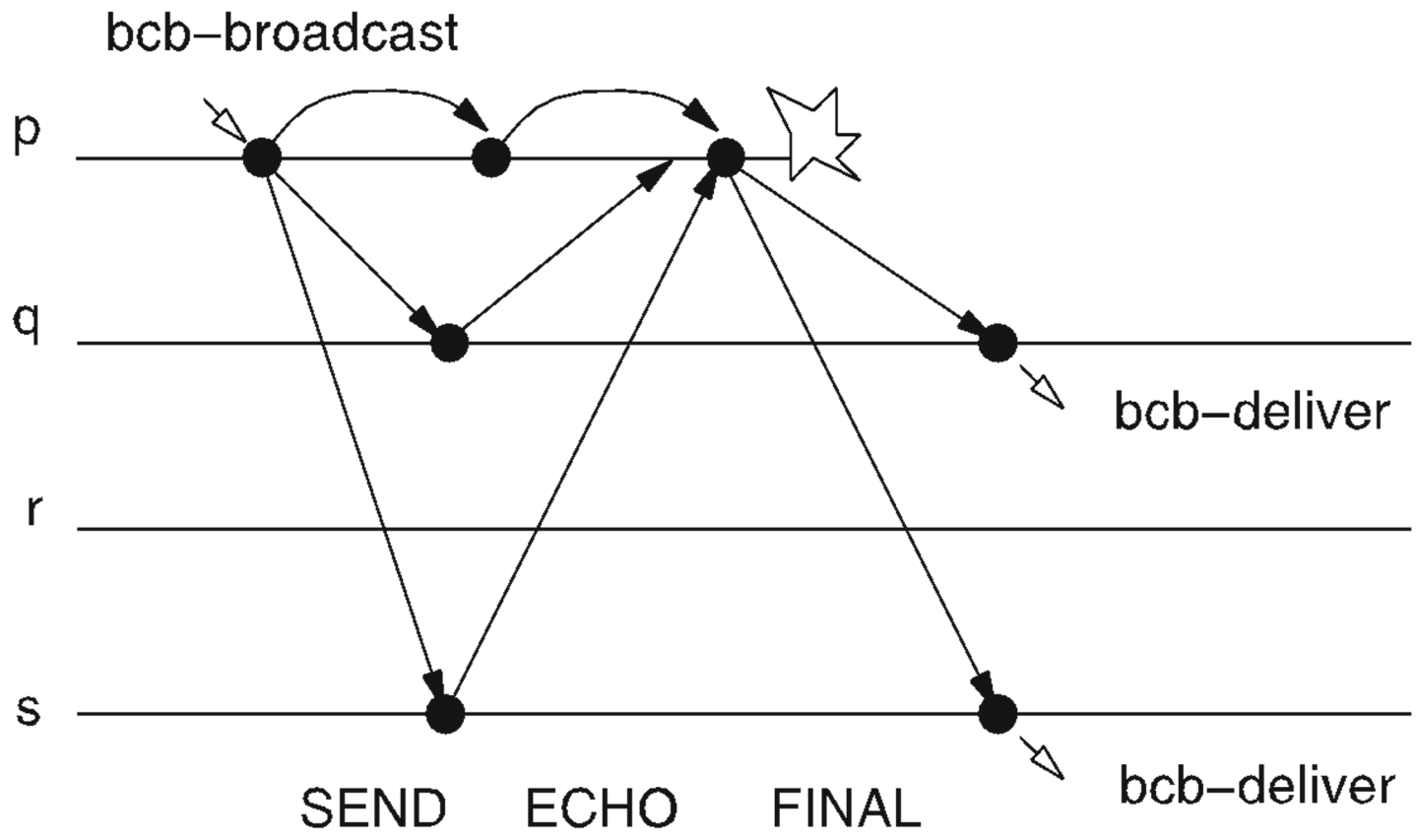
  forall q ∈ Π **do**

  trigger ⟨ *al*, Send | q, [FINAL, m, Σ] ⟩;

Signed Echo Broadcast

- **upon event** ⟨ *al*, Deliver | p, [FINAL, m, Σ] ⟩ **do**
  if (*delivered* = FALSE) **and**
  (#{p∈Π | Σ[p] !=⊥ ∧
  verifysig(p, *bcb*‖p‖ECHO‖m, Σ[p])} >(N+f)/2)
  delivered := TRUE;
  trigger ⟨ *bcb*, Deliver | s, m ⟩;

Signed Echo Broadcast



Correctness

- **BCB1-BCB3:** as in prev. algorithm

- **BCB4:** *Consistency:* If some correct process delivers a message m and another correct process delivers a message m′, then m = m′.

  - The digitally signed echo messages convey the same information as the ECHO message that a process sends directly to all others in the prev. algorithm

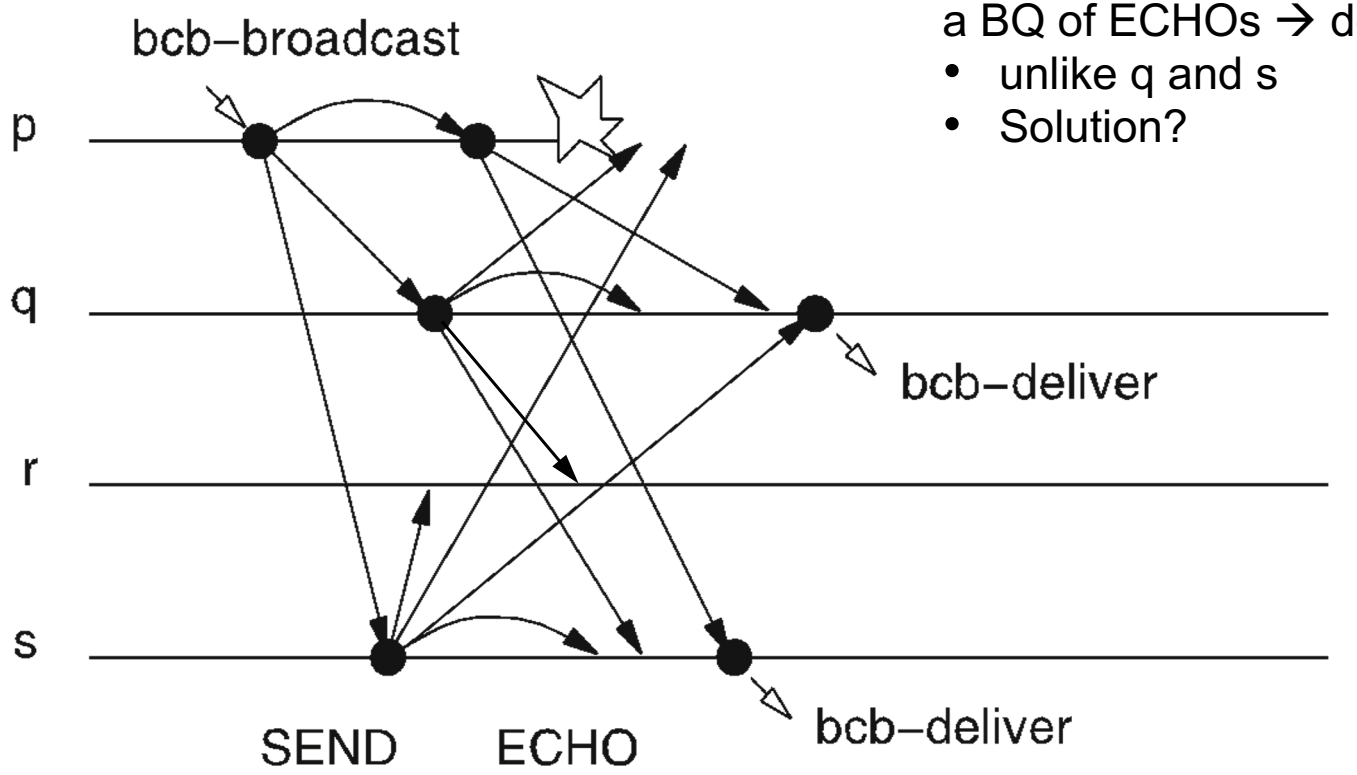
Byzantine Reliable Broadcast: specification

- **BRB1–BRB4 :** same as in BCB1-BCB4
- **BRB5:** *Totality:* If some message is delivered by any correct process, every correct process eventually delivers a message.
- **Note**
- Both BCB and BRB are defined for a designated sender, a given message
  - not for a stream of messages, unlike in crash model
- *Totality* + *Consistency* are equivalent to *Agreement* of Reliable Broadcast in the crash model

Byzantine Reliable Broadcast

- Specification of Byzantine Reliable Broadcast
  - Ensures that if a correct process delivers m, then every correct process delivers m
    - **independently of whether the sender is faulty**
  - Stronger spec than Byzantine Consistent Broadcast
- Algorithm: Authenticated Double-Echo Broadcast
- Extends Authenticated Echo Broadcast. How?

Authenticated Echo Broadcast



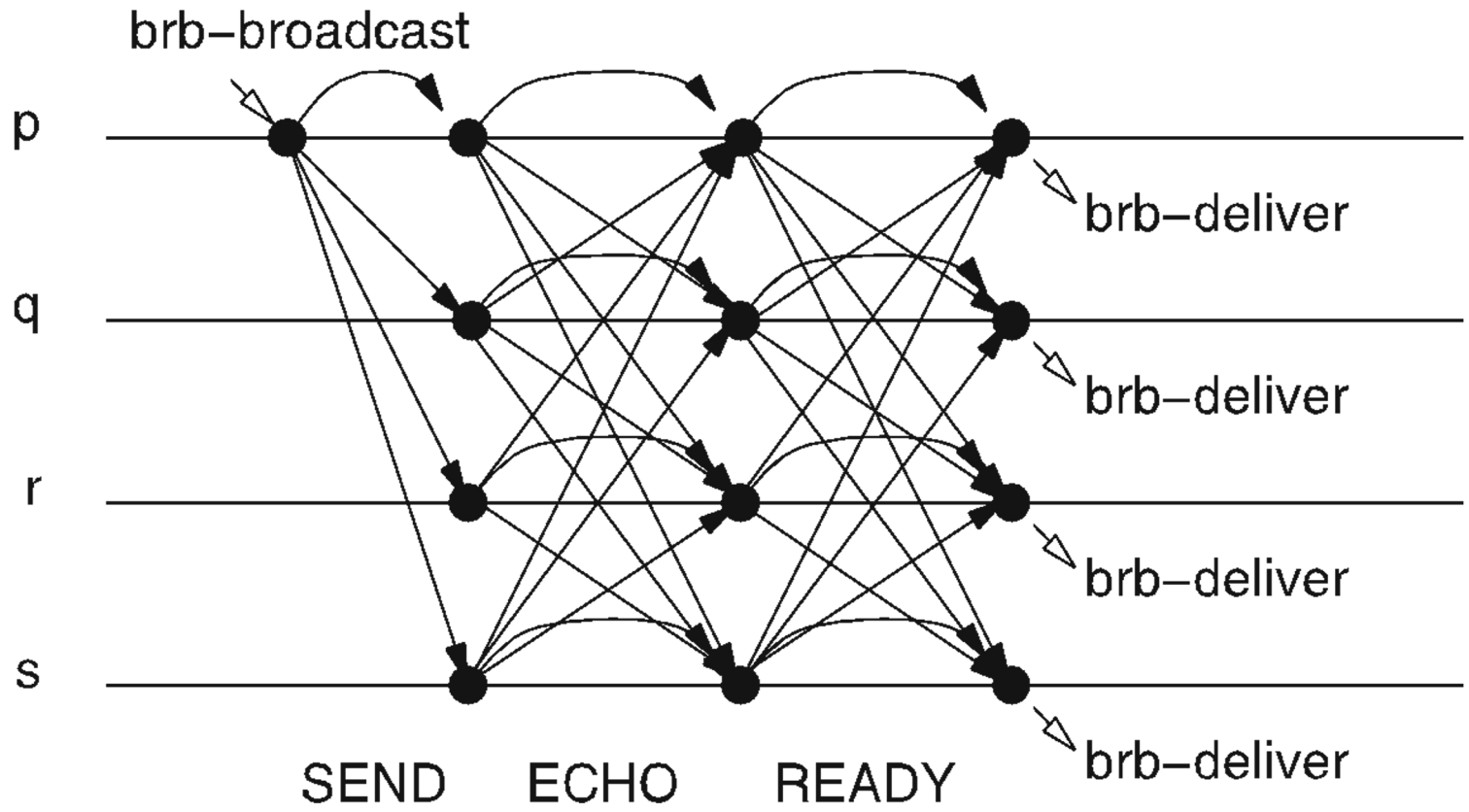
PROBLEM:

- process r never gets to receive

a BQ of ECHOs → does not deliver m

- unlike q and s
- Solution?

Authenticated Double-Echo Broadcast



Authenticated Double Echo Broadcast

- **Implements:**

  ByzantineReliableBroadcast, with sender s.

- **Uses:**

  AuthPerfectPointToPointLinks, **instance** *al*.

- **upon event** ⟨ *brb*, Init ⟩ **do**

  sentecho := FALSE;

  sentready := FALSE;

  delivered := FALSE;

  echos := $[\perp]^N$;

  readys := $[\perp]^N$;

Authenticated Double Echo Broadcast

- **upon event** ⟨ *brb*, Broadcast | m ⟩ **do** *// only process s*

  forall q ∈ Π **do**

  trigger ⟨ *al*, Send | q, [SEND, m] ⟩;

- **upon event** ⟨ *al*, Deliver | p, [SEND, m] ⟩

  such that p = s **and** *sentecho* = FALSE **do**

  sentecho := TRUE;

  forall q ∈ Π **do**

  trigger ⟨ *al*, Send | q, [ECHO, m] ⟩;

Authenticated Double Echo Broadcast

- **upon event** ⟨ *al*, Deliver | p, [ECHO, m] ⟩ **do**
  if *echos*[p] = ⊥ **then** *echos*[p] := m;
- **upon exists** m != ⊥ such that :
  |{p ∈ Π |*echos*[p] = m}| > (N+f)/2 **and** *sentready*=FALSE **do**
  sentready:= TRUE
  forall q ∈ Π **do**
  trigger ⟨ *al*, Send| q, [READY, m] ⟩;
- **upon event** ⟨ *al*, Deliver | p, [READY, m] ⟩ **do**
  if *readys*[p] = ⊥ **then** *readys*[p] := m;

Retransmission step: inform other correct processes about existence of byzantine quorum for m

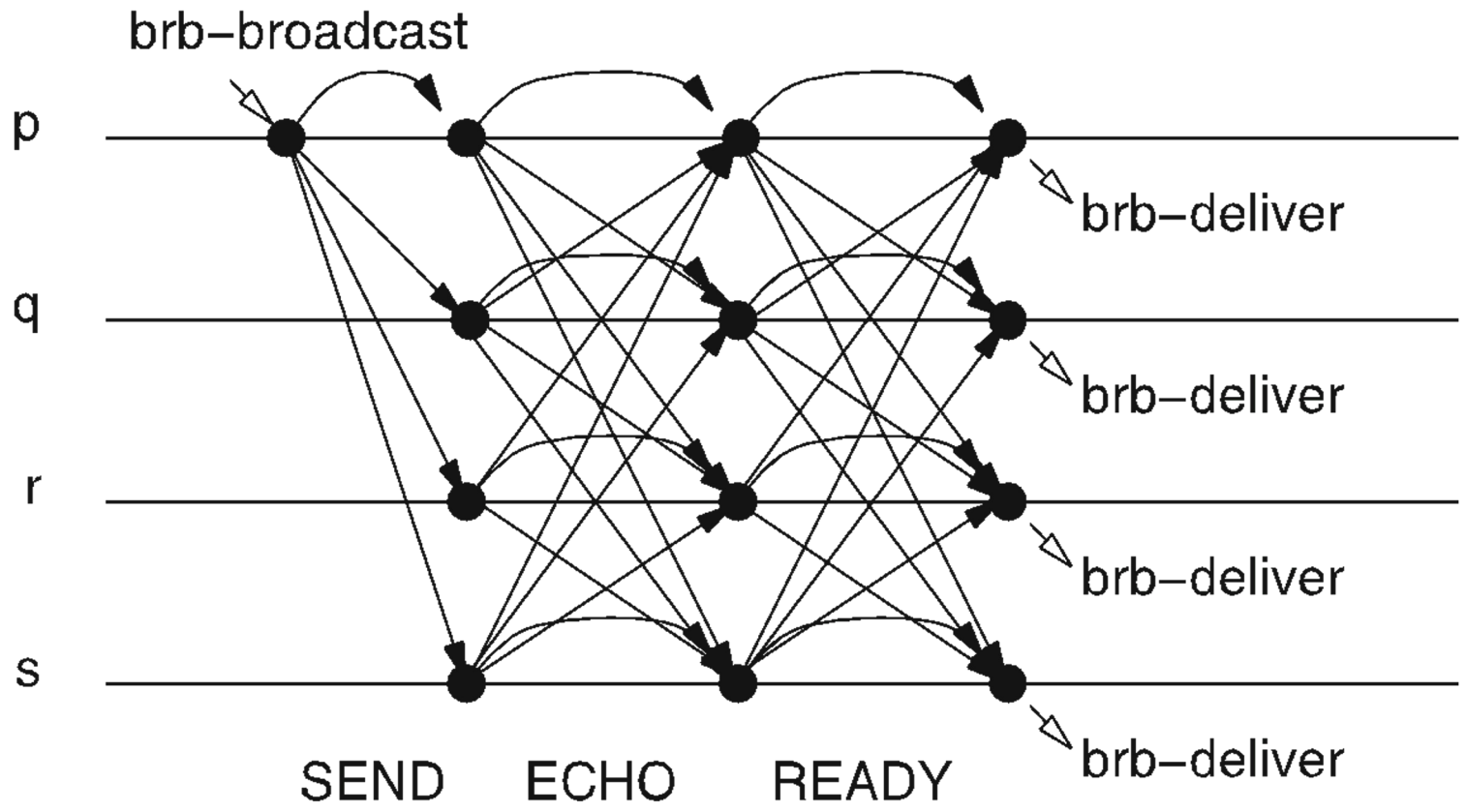
Authenticated Double Echo Broadcast

Amplification step:
Why is it needed?

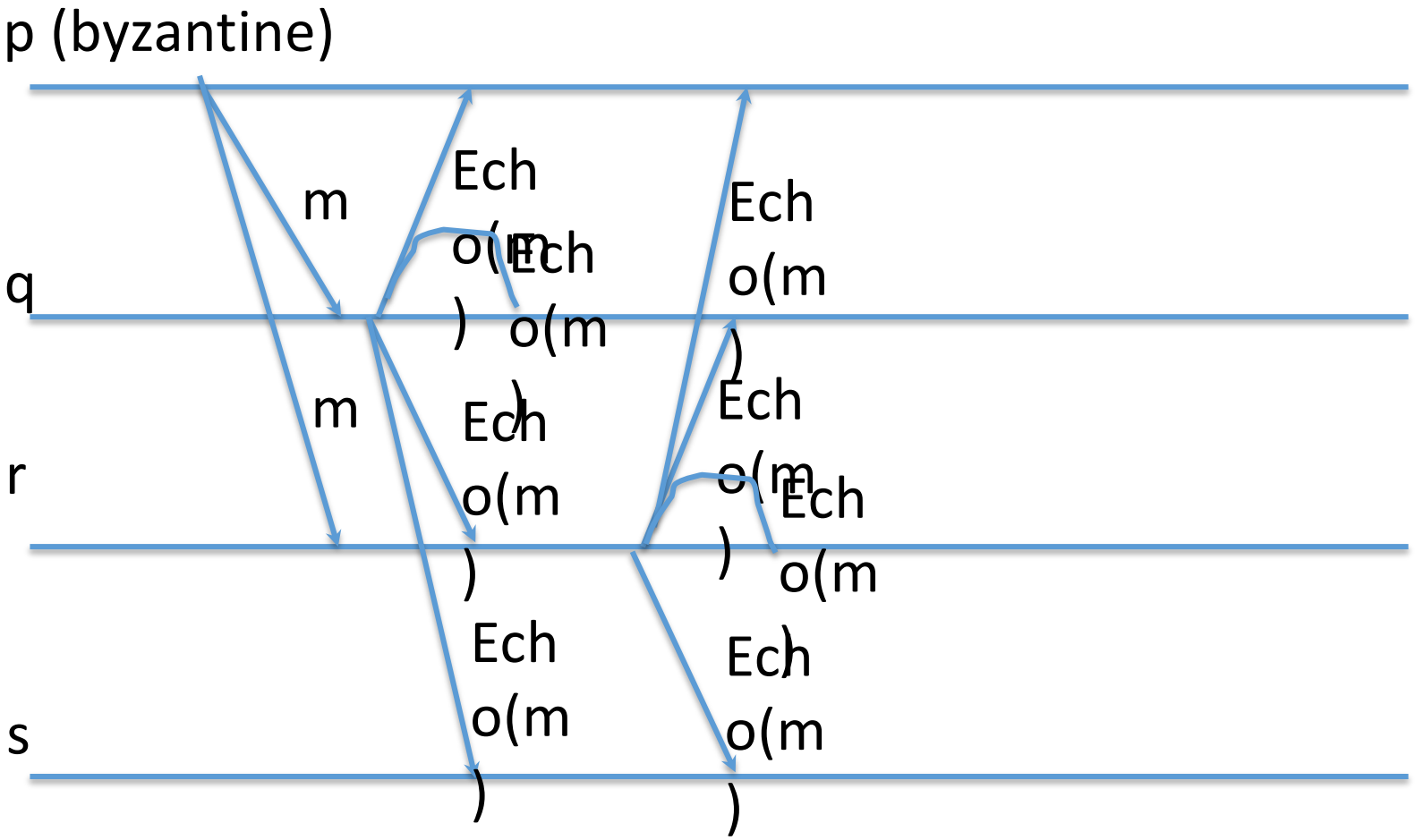
- **upon exists** m !=⊥ such that
  |{p∈Π|*readys*[p]=m}| > f **and** *sentready* = FALSE **do**
  sentready := TRUE;
  forall q ∈ Π **do**
    trigger ⟨ *al*, Send | q, [READY, m] ⟩;

- **upon exists** m !=⊥such that
  |{p∈Π|*readys*[p]=m}| > 2f **and** *delivered* = FALSE **do**
  delivered := TRUE;
  trigger ⟨ *brb*, Deliver | s, m ⟩;

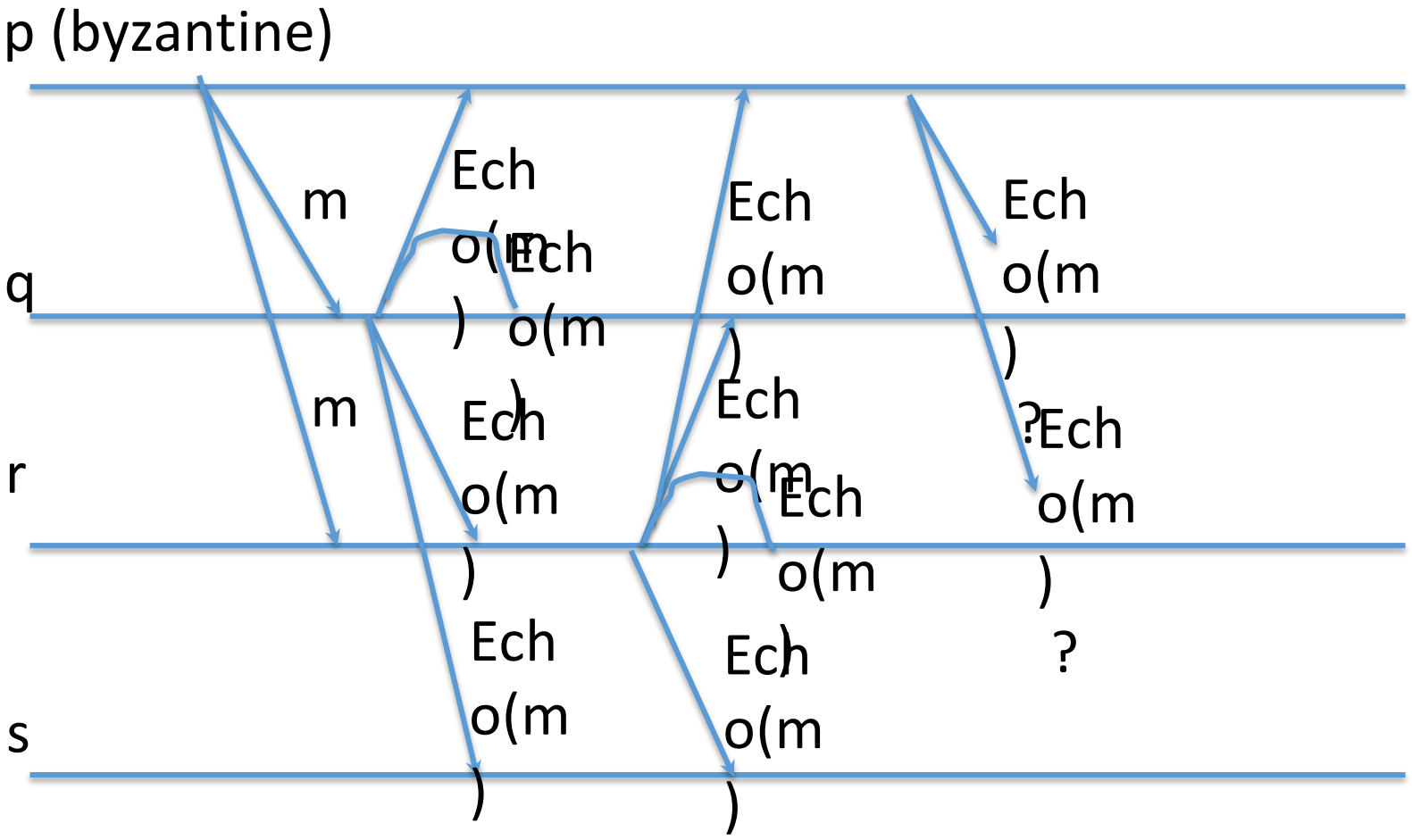
Authenticated Double-Echo Broadcast



Is it possible that only q or only r send a ready msg?

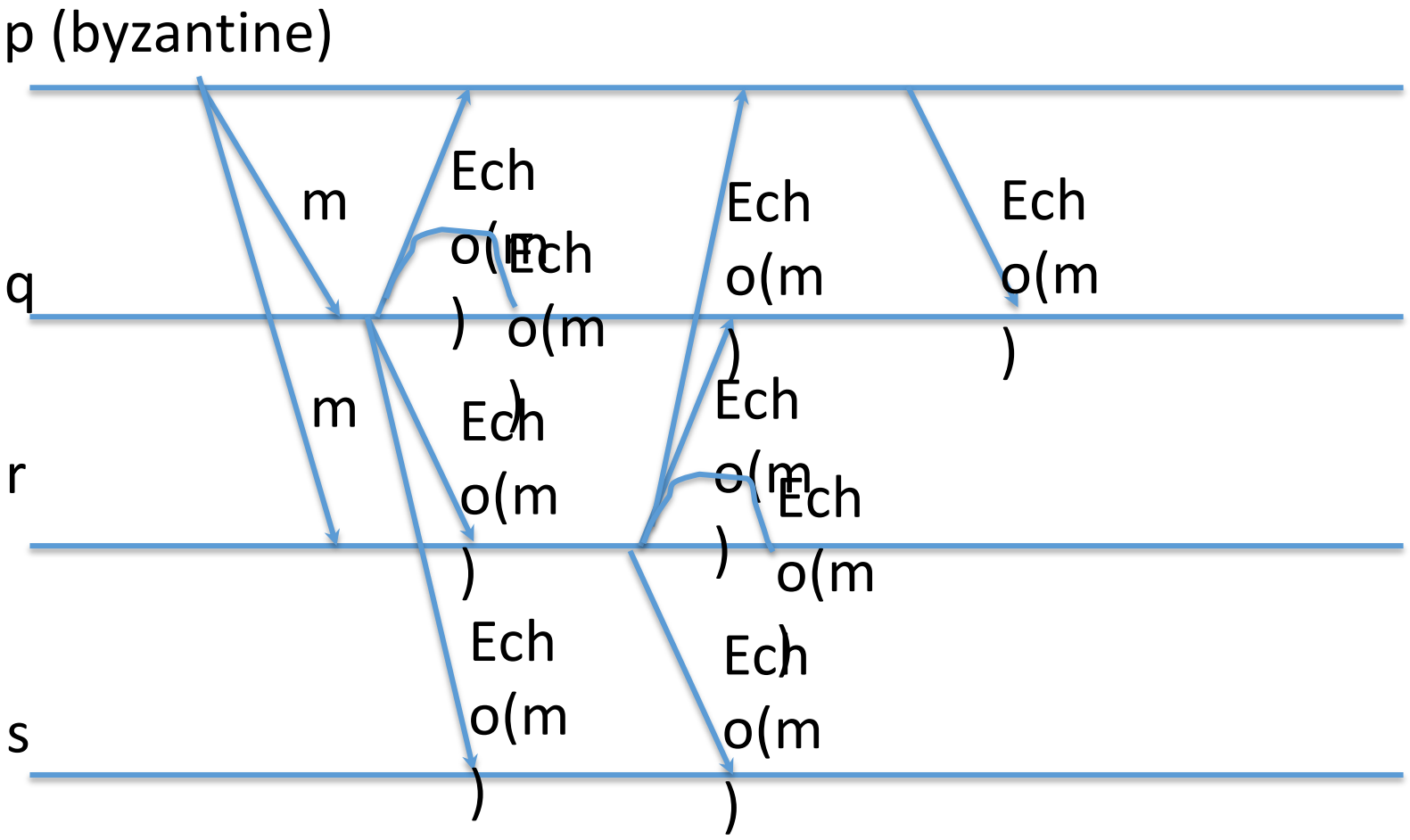


Is it possible that only q or r send a ready msg?



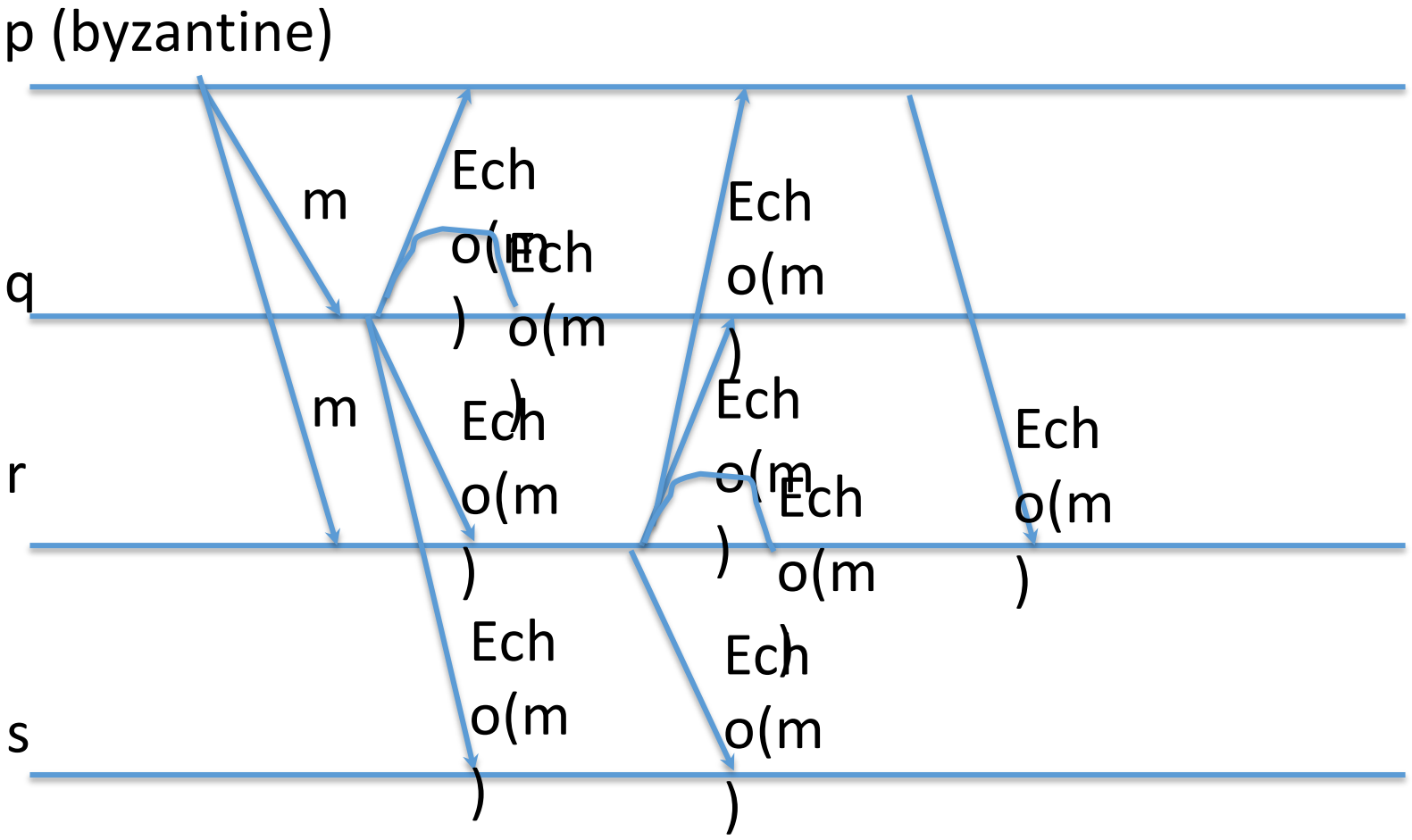
Yes, depending on whether p sends an echo to q and/or r

Is it possible that only q or r send a ready msg?



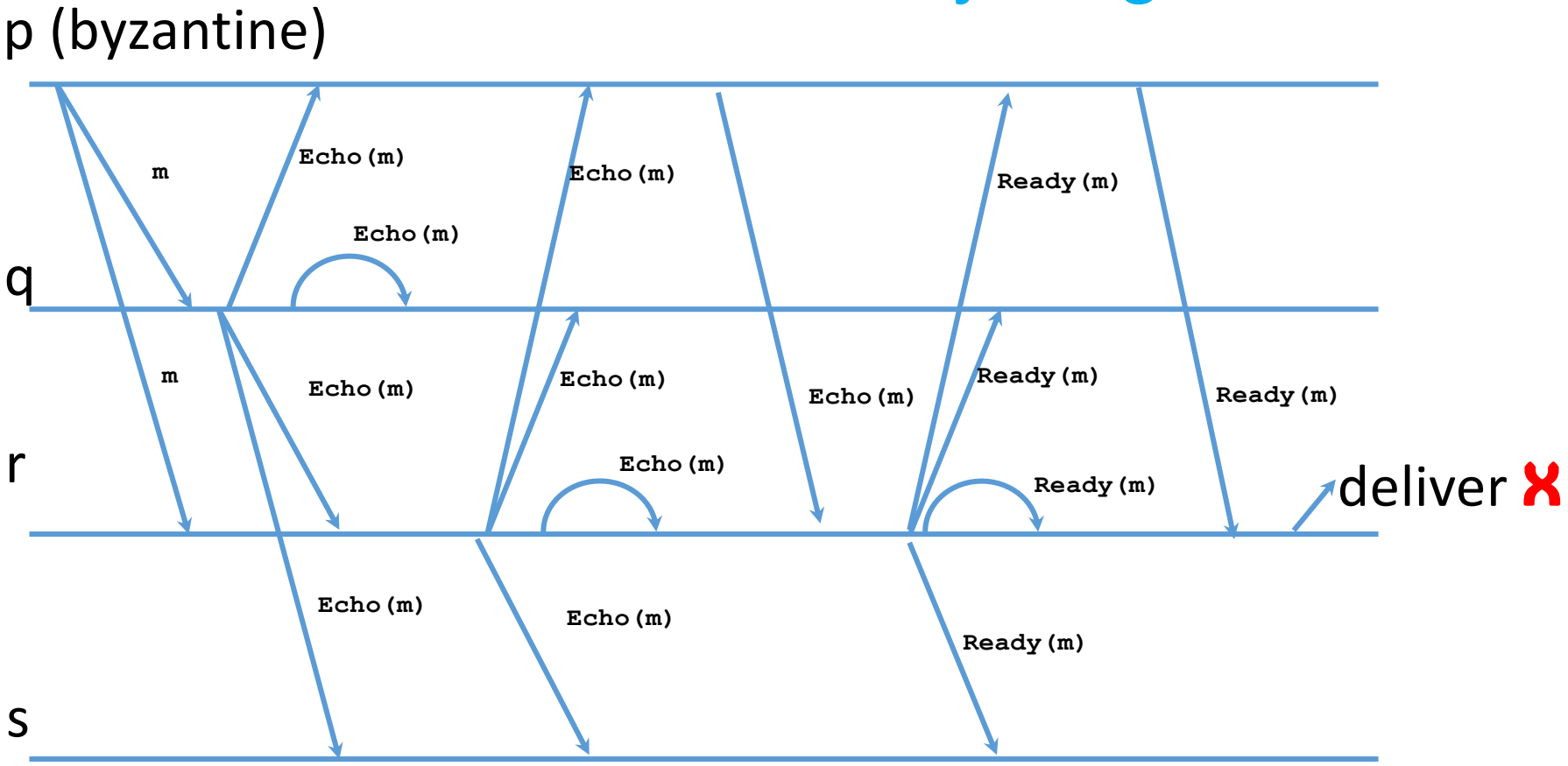
Yes, depending on whether p sends an echo to q and/or r

Is it possible that only q or r send a ready msg?



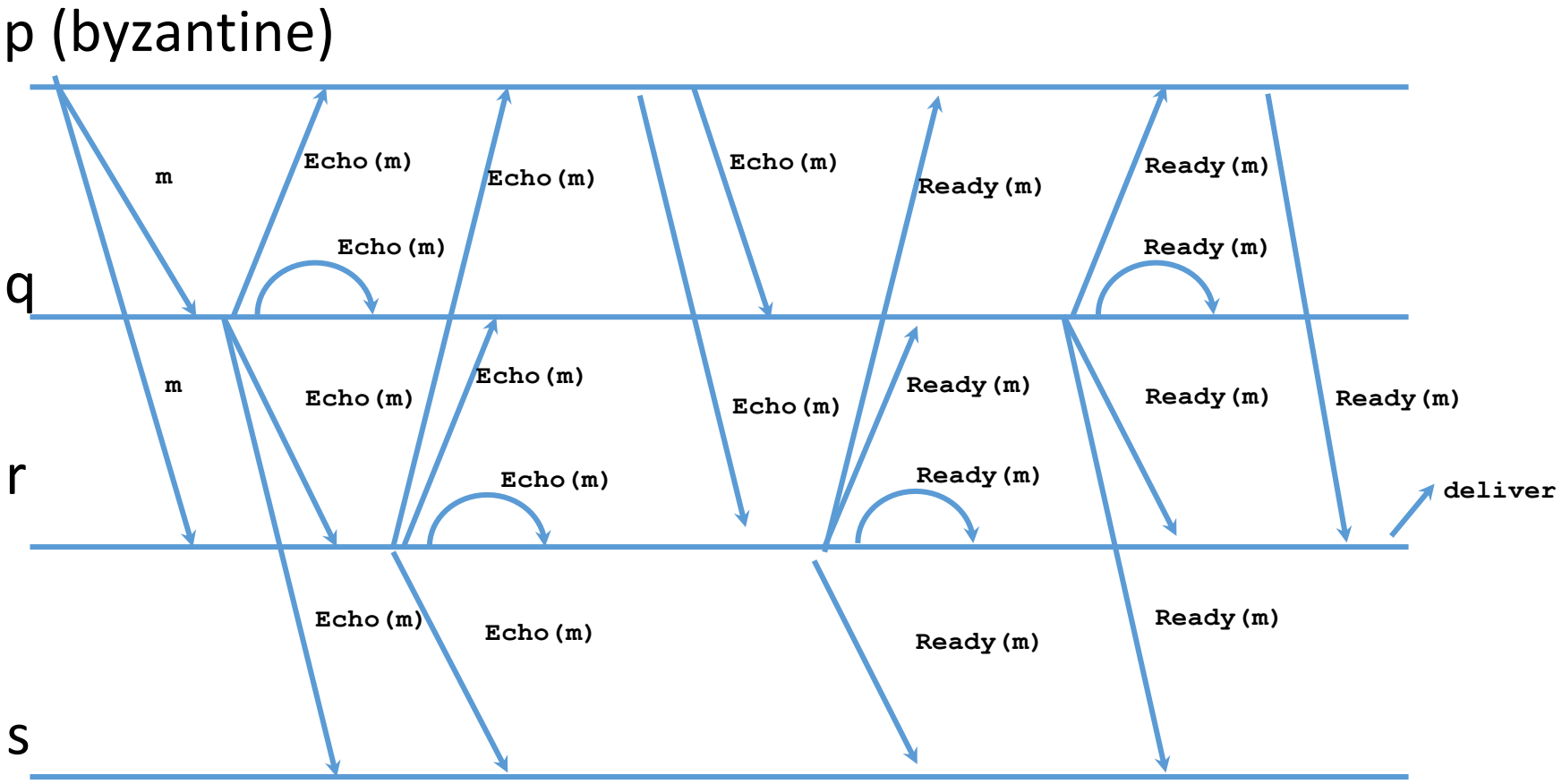
Yes, depending on whether p sends an echo to q and/or r

Is it ok to deliver after f+1 Ready msgs?



Need to wait for 2f+1 Ready msgs to secure totality

Why is the amplification phase needed?



- r receives 3 Ready(m) msgs and delivers m...
- but q does not, as p does not send Ready(m)

Correctness (1/3)

- Validity, No duplication, Integrity:
  - same as in Authenticated Echo Broadcast

Correctness (2/3)

- Consistency: If some correct process delivers a message m and another correct process delivers a message m’, then m = m’.
  - if some correct process broadcasts ready messages for m and another correct process broadcasts ready for m', then m=m'
  - Proof by contradiction, same as in previous proof
  - Directly implies consistency because cannot have more than f ready messages for different messages

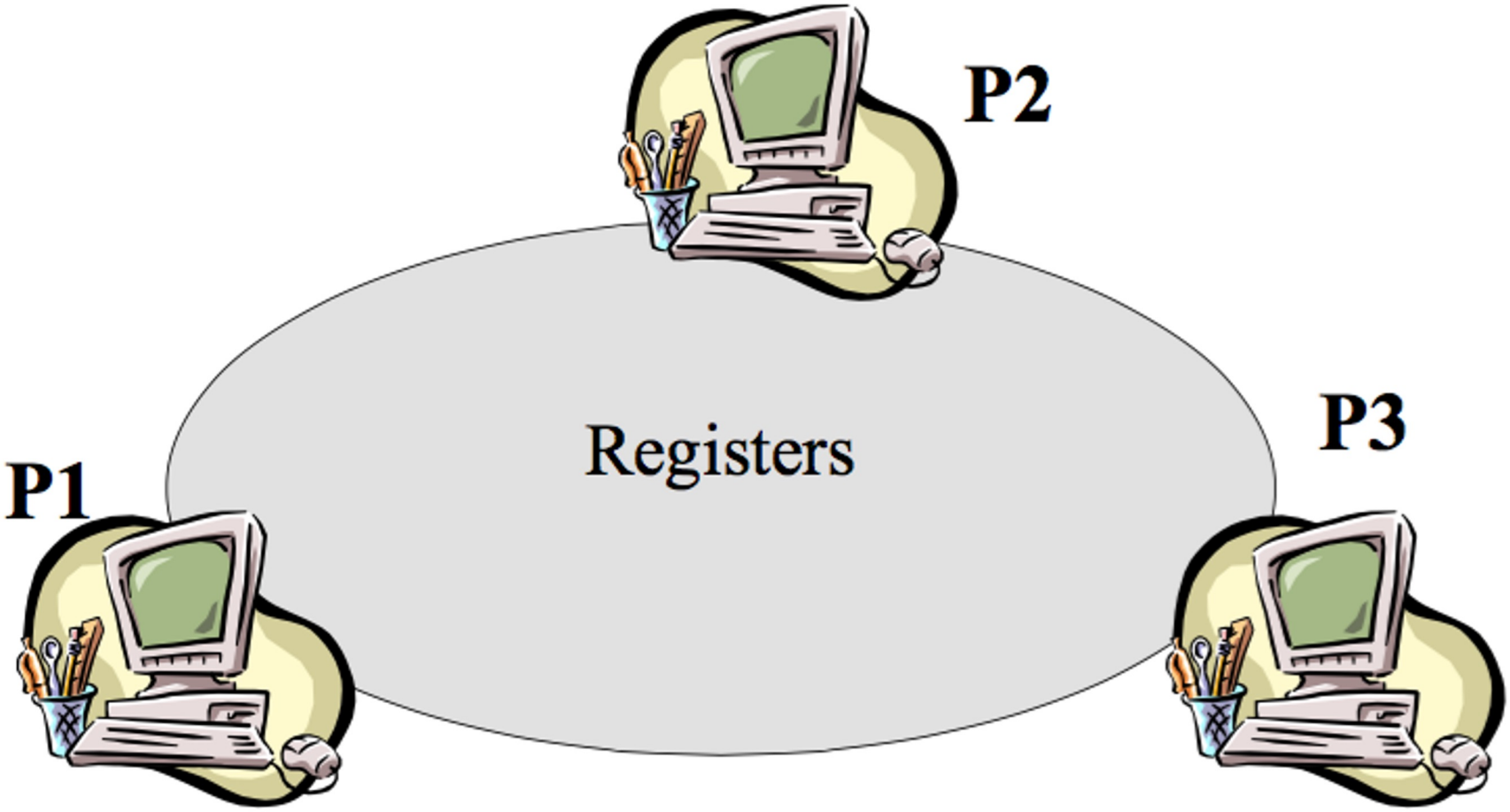
Correctness (3/3)

- Totality:
  - if a correct process delivers, then it received at least 2f+1 ready messages, at least f+1 from correct process.
  - These f+1 will eventually be delivered to all correct processes, triggering amplification step, and consequently sufficient ready messages for totality

Byzantine broadcast channels

- Byzantine broadcast works only for a single message
- It is simple to implement a broadcast channel
  - Either consistent or reliable broadcast channels
- Idea: use multiple instances of Byzantine broadcast

New abstraction: read/write registers



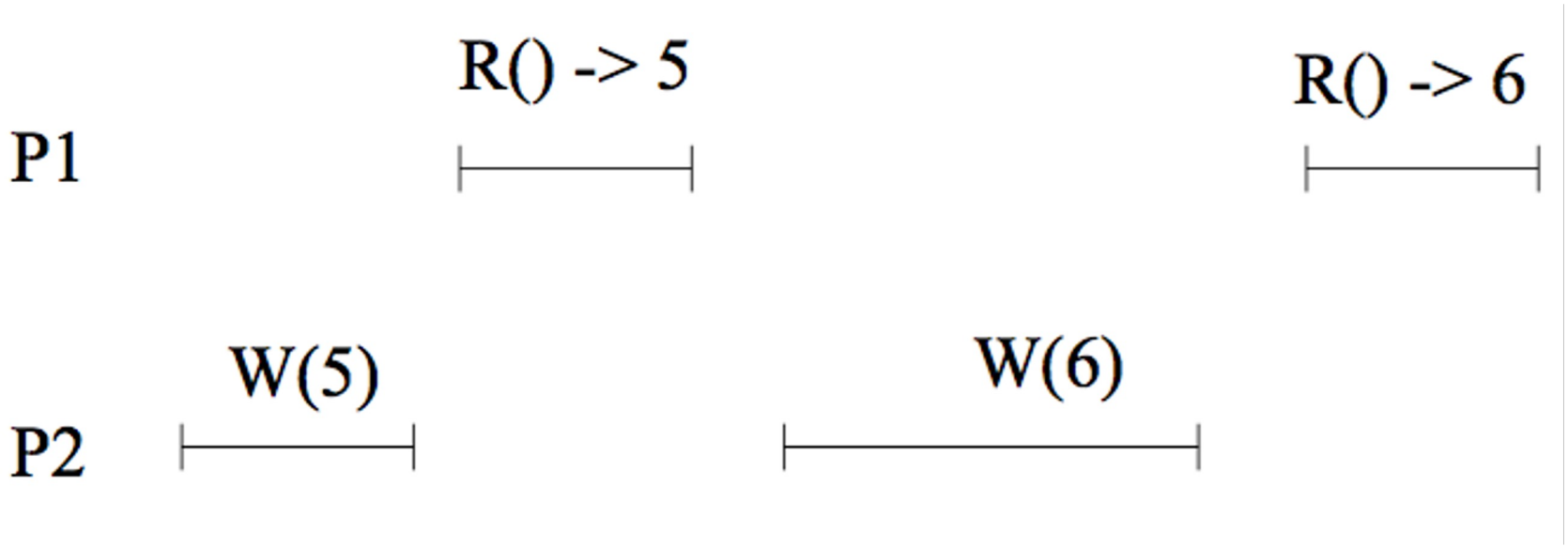
P1,P2,P3 are both clients and replicas (without loss of generality)

Assumptions

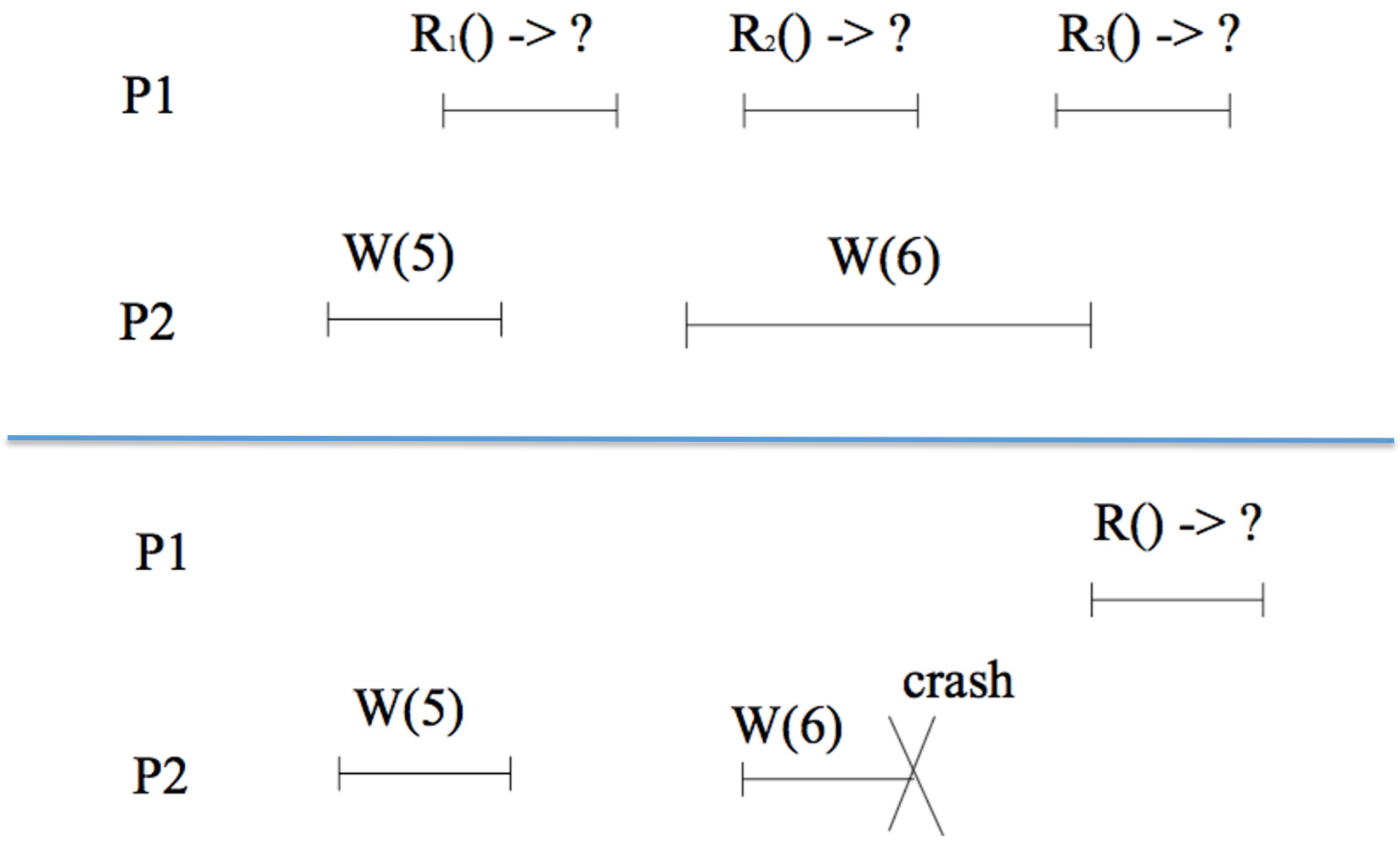
- Single register, shared by all processes
- Interface:
  - **`Read()`** → returns **`value`**
  - **`Write(value)`** → returns "**`ack`**"
- Values written are integers
- Initial value of register is zero
- Every written value is unique
  - this can be enforced by associating a process id and a timestamp with the value

Sequential specification

- **`Read()`** returns the most recent **`value`** that was written



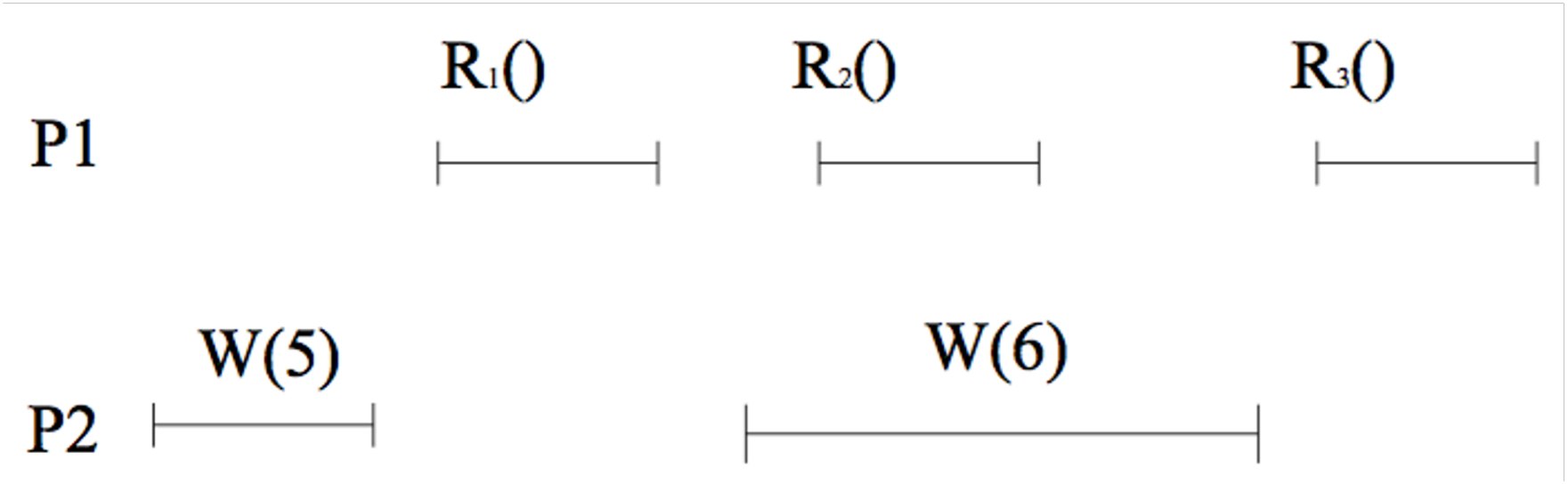
How to specify concurrent / faulty execution?



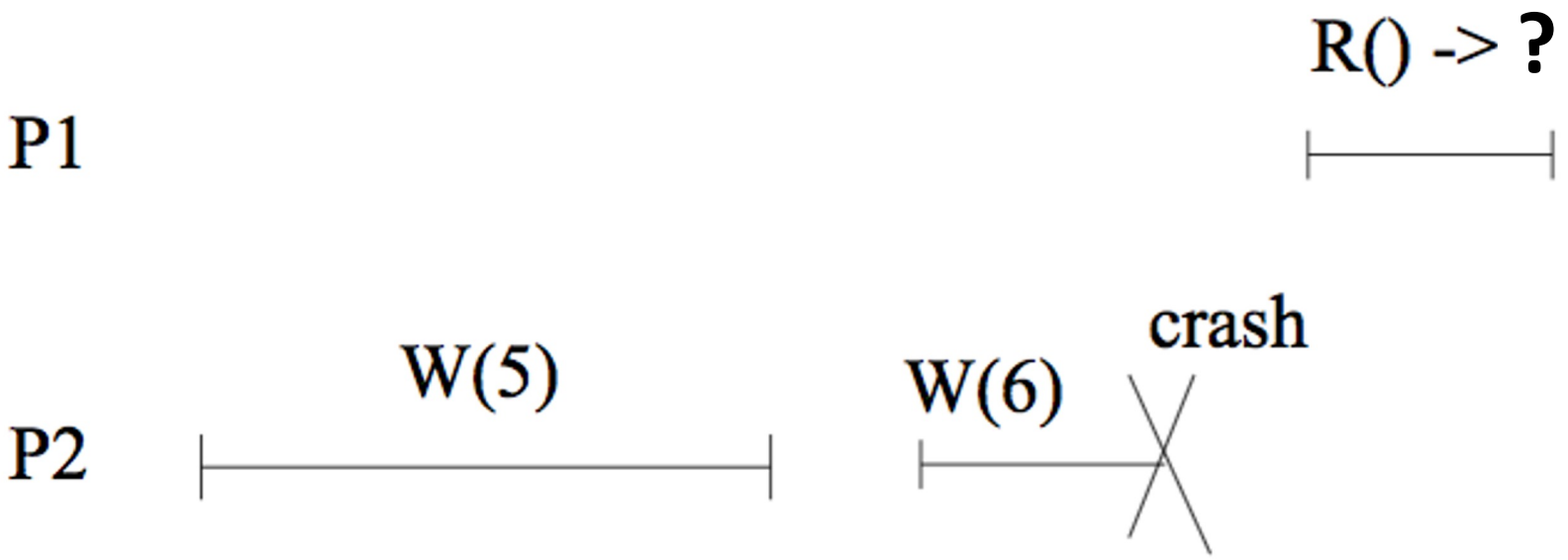
Specification #1: Regular register

- Initially, we will assume a single writer
- [Liveness] If a correct process invokes an operation, then the operation eventually completes.
- [Safety] Read must return:
  - the last value written if there is no concurrent write operation, otherwise
  - either the last value or any value concurrently written
- When a process crashes in the middle of reading/writing, the operation interval does not have an upper limit

Exercise: which values are admissible for R1, R2, R3?



Exercise: which values are admissible?



(On a side note, there exists a weaker specification: safe register)

- Similar to regular register, but:
- [Safety] Read must return:
  - the last value written if there is no concurrent write operation, otherwise
  - **any** value if there are concurrent writes
- In this course we will skip the design of safe registers

Implementation of a (1,N)-Regular register

- Uses BestEffortBroadcast + Perfect-p2p-links
- Each process maintains:

  <ts,val> /* Current value and associated timestamp, ts */

  readlist /* List of returned values, for reading */

  rid /* id of current read operation */
- Writer process maintains:

  wts /* Next timestamp to be written */

  acks /* How may writes have been acknowledged */
- Algorithm must tolerate up to *f* crash faults. How?

Implements:

(1, N)-RegularRegister, **instance** *onrr*.

Uses:

BestEffortBroadcast, **instance** *beb*;

PerfectPointToPointLinks, **instance** *pl*.

upon event ⟨ *onrr*, *Init* ⟩ **do**

$(ts, val) := (0, \perp)$;

$wts := 0$;

$acks := 0$;

$rid := 0$;

$readlist := [\perp]^N$;

upon event ⟨ *onrr*, *Write* | v ⟩ **do**

$wts := wts + 1$;

$acks := 0$;

trigger ⟨ *beb*, *Broadcast* | [WRITE, wts, v] ⟩;

upon event ⟨ *beb*, *Deliver* | p, [WRITE, ts', v'] ⟩ **do**

if $ts' > ts$ **then**

$(ts, val) := (ts', v')$;

trigger ⟨ *pl*, *Send* | p, [ACK, ts'] ⟩;

upon event ⟨ *pl*, *Deliver* | q, [ACK, ts'] ⟩ **such that** $ts' = wts$ **do**

$acks := acks + 1$;

if $acks > N/2$ **then**

$acks := 0$;

trigger ⟨ *onrr*, *WriteReturn* ⟩;

upon event $\langle$ *onrr*, *Read* $\rangle$ **do**
    $rid := rid + 1$;
    $readlist := [\perp]^N$;
    trigger $\langle$ *beb*, *Broadcast* | [READ, *rid*] $\rangle$;

upon event $\langle$ *beb*, *Deliver* | p, [READ, r] $\rangle$ **do**
    trigger $\langle$ *pl*, *Send* | p, [VALUE, r, ts, val] $\rangle$;

upon event $\langle$ *pl*, *Deliver* | q, [VALUE, r, ts', v'] $\rangle$ **such that** $r = rid$ **do**
    $readlist[q] := (ts', v')$;
    if $\#(readlist) > N/2$ **then**
        $v := highestval(readlist)$;
        $readlist := [\perp]^N$;
        trigger $\langle$ *onrr*, *ReadReturn* | v $\rangle$;

Is this execution valid?

Yes, reads might return the last value written of the value concurrently being written

Homework: come up with an execution that leads to this trace of outputs

Acknowledgements

- Rachid Guerraoui, EPFL