Distributed Systems
Master of Science in Engineering in Computer Science

AA 2018/2019

LECTURE 19: REGISTERS IN PRESENCE OF BYZANTINE PROCESSES

Safe Register Specification

Module 4.5: Interface and properties of a $(1, N)$ Byzantine safe register

Module:

Name: $(1, N)$-ByzantineSafeRegister, **instance** *bonsr*, with writer w.

Events:

Request: $\langle$ *bonsr*, *Read* $\rangle$: Invokes a read operation on the register.

Request: $\langle$ *bonsr*, *Write* $\mid v \rangle$: Invokes a write operation with value v on the register. Executed only by process w.

Indication: $\langle$ *bonsr*, *ReadReturn* $\mid v \rangle$: Completes a read operation on the register with return value v.

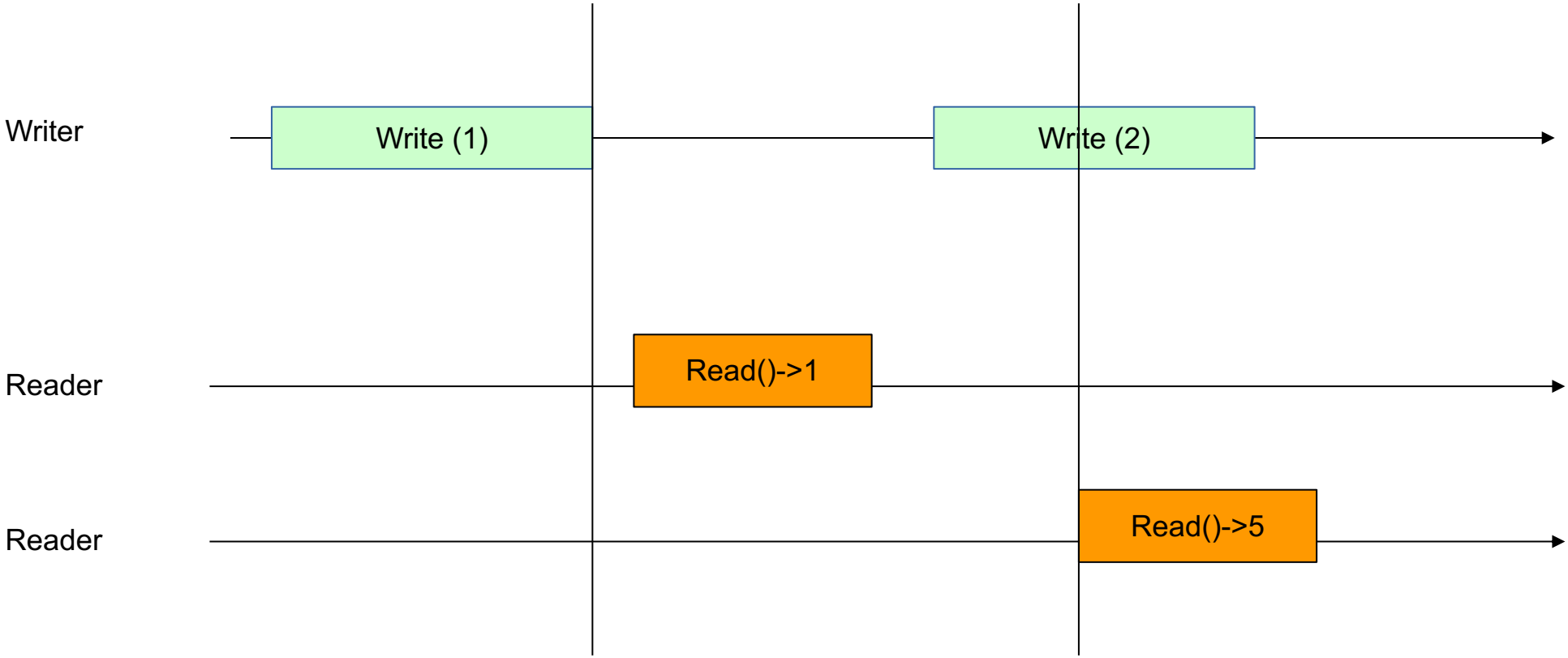
Indication: $\langle$ *bonsr*, *WriteReturn* $\rangle$: Completes a write operation on the register. Occurs only at process w.

Properties:

BONSR1: *Termination:* If a correct process invokes an operation, then the operation eventually completes.

BONSR2: *Validity:* A read that is not concurrent with a write returns the last value written.

Safe Register



Byzantine Tolerant Safe Register (1,n)

Module 4.5: Interface and properties of a $(1, N)$ Byzantine safe register

Module:

Name: $(1, N)$-ByzantineSafeRegister, **instance** *bonsr*, with writer w.

Events:

Request: $\langle$ *bonsr*, *Read* $\rangle$: Invokes a read operation on the register.

Request: $\langle$ *bonsr*, *Write* | v $\rangle$: Invokes a write operation with value v on the register. Executed only by process w.

Indication: $\langle$ *bonsr*, *ReadReturn* | v $\rangle$: Completes a read operation on the register with return value v.

Indication: $\langle$ *bonsr*, *WriteReturn* $\rangle$: Completes a write operation on the register. Occurs only at process w.

Properties:

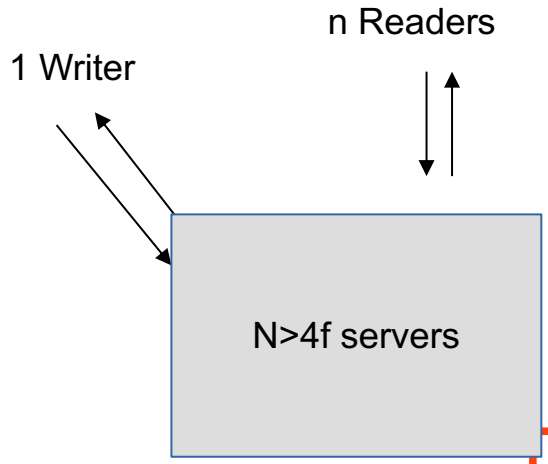
BONSR1: *Termination:* If a correct process invokes an operation, then the operation eventually completes.

BONSR2: *Validity:* A read that is not concurrent with a write returns the last value written.

Client-Server paradigm

Asynchronous System

N>4f



Do not confuse *n* with *N*

Safe Register Intuition

We have to assure that once writer returns from a write operation, then any following read operation returns the last written value.

- Write operation: sends <v,wts> to servers and waits for ACK messages.

  How many ACK messages?

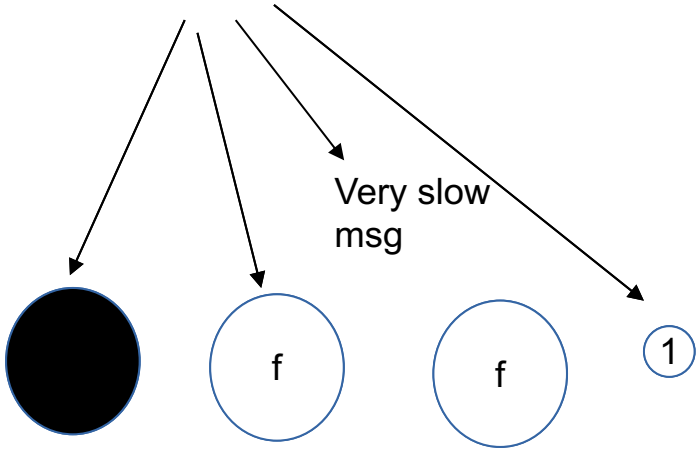
  Enough to be sure that enough correct servers deliver <v,wts>

- Read operation: sends a read request and waits for reply messages.

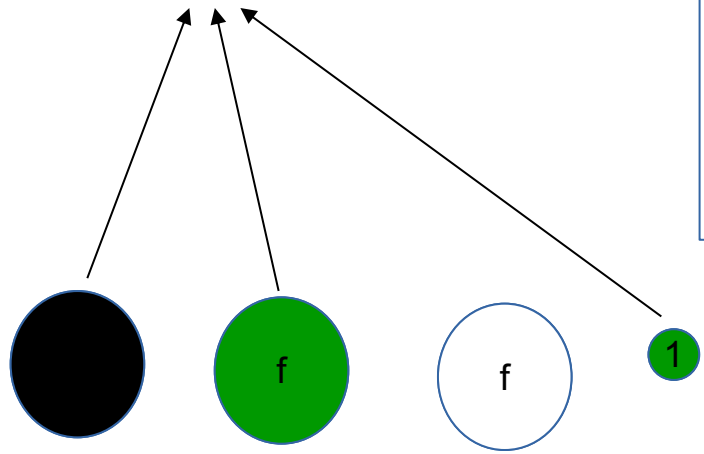
  How many reply messages?

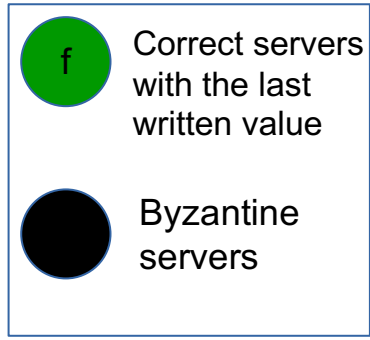
  Enough to be able to read newest value, not an old value or never written

1



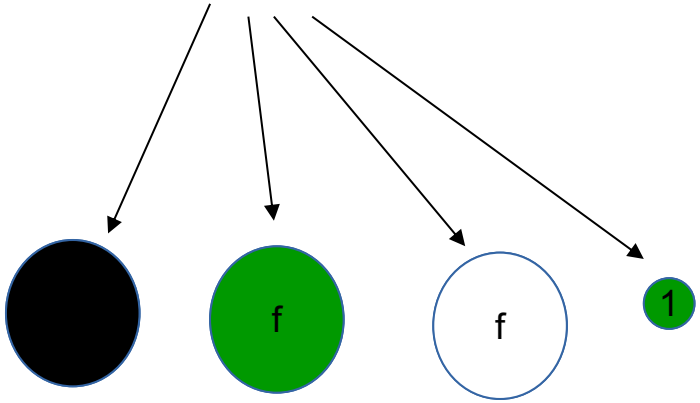
2



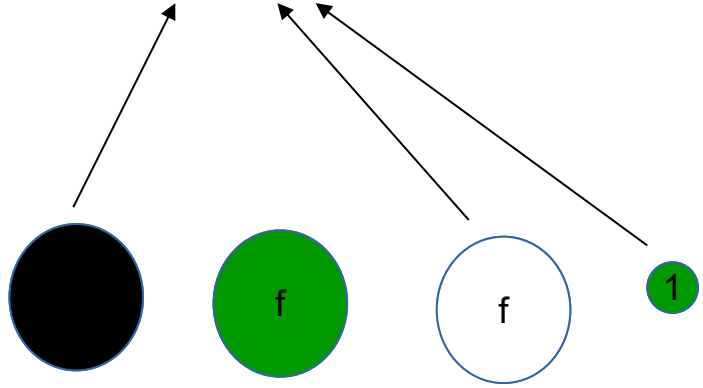


If we consider: N>3f
Quorum: (N+f)/2

3



4



Masking Quorum

The kind of quorum working for Byzantine Broadcast here is not enough.

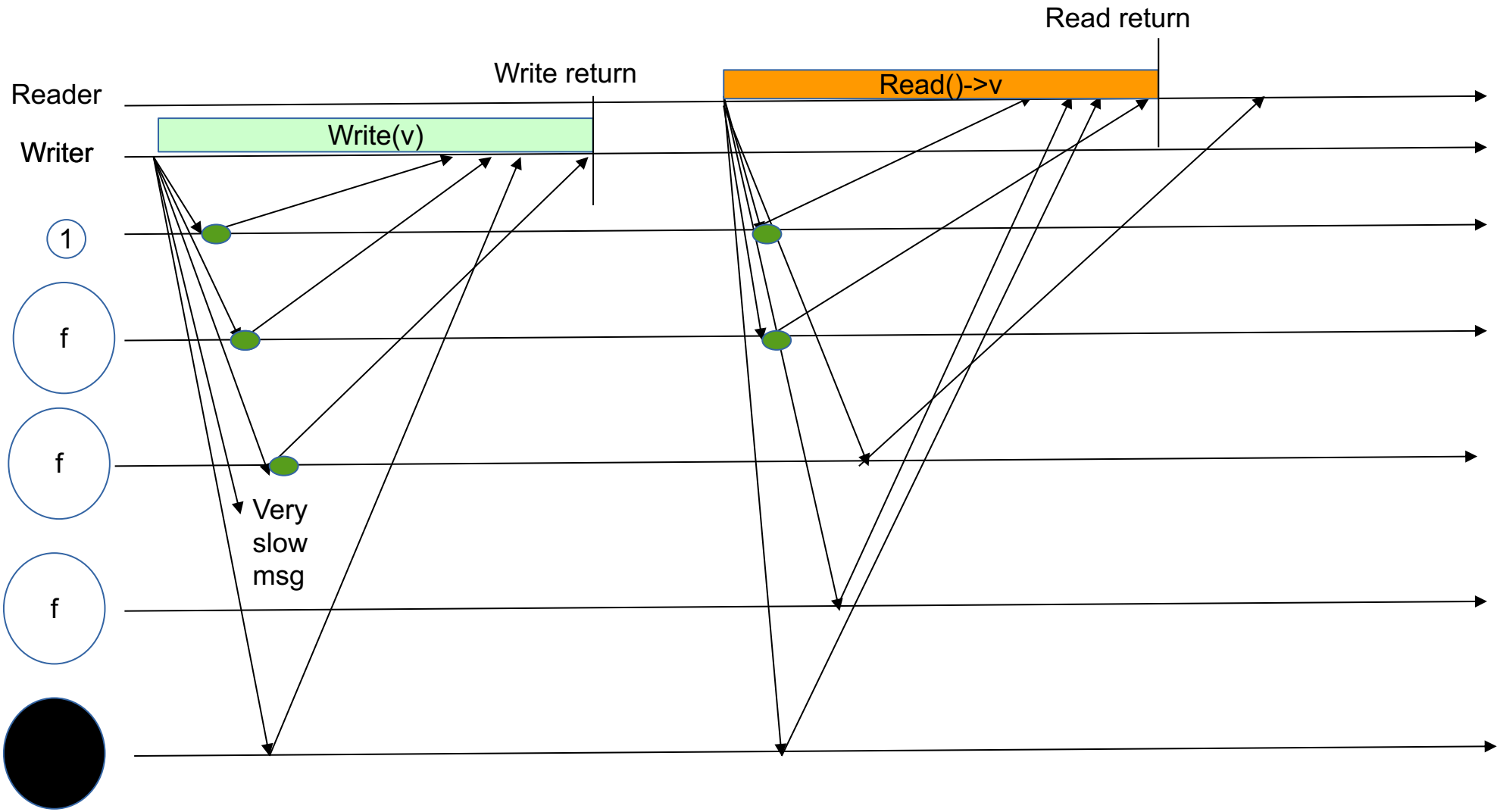
Safe Register has a stronger semantic with respect to broadcast. We require that a write operation is visible to all once it terminates.

To implement safe registers we use **Masking Quorums**:

$N>4f$

Quorum: $(N+2f)/2$

N>4f
Quorum: (N+2f)/2



Algorithm 4.14: Byzantine Masking Quorum

Implements:
$(1, N)$-ByzantineSafeRegister, **instance** *bonsr*, with writer w.
Uses:
AuthPerfectPointToPointLinks, **instance** *al*.

upon event $\langle$ *bonsr*, *Init* $\rangle$ **do**
$(ts, val) := (0, \perp)$;
$wts := 0$;
acklist $:= [\perp]^N$;
rid $:= 0$;
readlist $:= [\perp]^N$;

upon event $\langle$ *bonsr*, *Write* | v $\rangle$ **do** // only process w
$wts := wts + 1$;
acklist $:= [\perp]^N$;
forall $q \in \Pi$ **do**
trigger $\langle$ *al*, *Send* | q, [WRITE, wts, v] $\rangle$;

upon event $\langle$ *al*, *Deliver* | p, [WRITE, ts', v'] $\rangle$ **such that** $p = w$ **do**
if $ts' > ts$ **then**
$(ts, val) := (ts', v')$;
trigger $\langle$ *al*, *Send* | p, [ACK, ts'] $\rangle$;

byzhighestval (·): selects the value from the pair that occurs more than f time and with the highest timestamp. If no pair exists, the reader selects a default value v0 from the domain of the register.

Assumption
N>4f

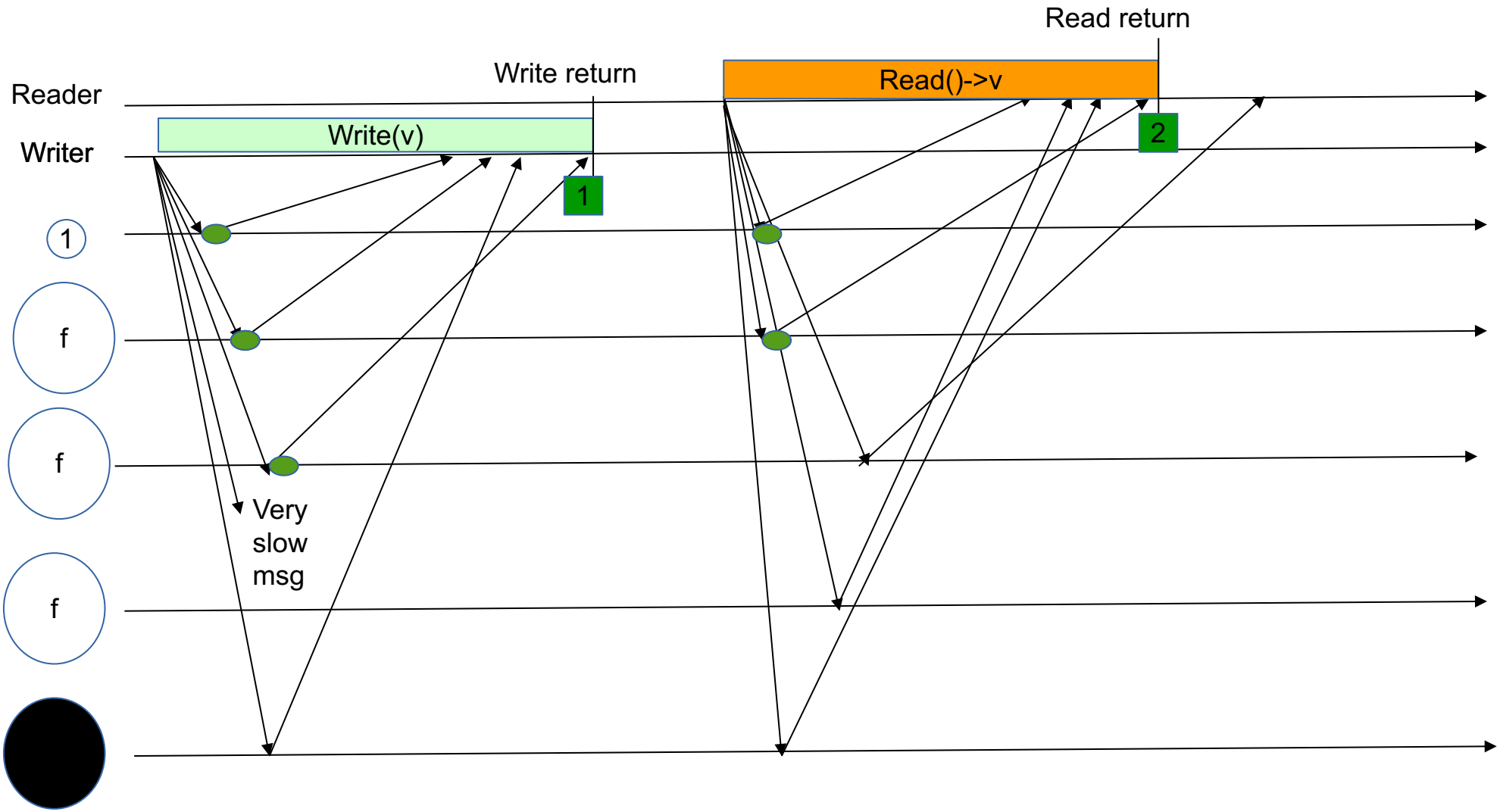
upon event $\langle$ *al*, *Deliver* | q, [ACK, ts'] $\rangle$ **such that** $ts' = wts$ **do**
1 *acklist*[q] := ACK;
if $\#(acklist) > (N + 2f)/2$ **then**
acklist $:= [\perp]^N$;
trigger $\langle$ *bonsr*, *WriteReturn* $\rangle$;

upon event $\langle$ *bonsr*, *Read* $\rangle$ **do**
rid := *rid* + 1;
readlist $:= [\perp]^N$;
forall $q \in \Pi$ **do**
trigger $\langle$ *al*, *Send* | q, [READ, *rid*] $\rangle$;

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

upon event $\langle$ *al*, *Deliver* | q, [VALUE, r, ts', v'] $\rangle$ **such that** $r = rid$ **do**
readlist[q] := (ts', v');
2 **if** $\#(readlist) > \frac{N+2f}{2}$ **then**
v := byzhighestval(*readlist*);
readlist $:= [\perp]^N$;
trigger $\langle$ *bonsr*, *ReadReturn* | v $\rangle$;

N>4f
Quorum: (N+2f)/2



Regular Registers

The specification does not change

We will discuss two implementations:

1. Using cryptography
2. Without cryptography

Regular Register
Implementation with cryptographic assumptions

BASIC IDEA -> evolution of Majority voting Algorithm

- the writer signs the timestamp/value pair
- Processes store it together with the signature
- The reader verifies the signature on each timestamp/value pair received in a VALUE message and ignores those with invalid signatures

A Byzantine process is prevented from returning an arbitrary timestamp value in the VALUE message, although it may include a signed value with an outdated timestamp

Regular Register
Implementation with cryptographic assumptions

Algorithm 4.15: Authenticated-Data Byzantine Quorum

Implements:
$(1, N)$-ByzantineRegularRegister, **instance** *bonrr*, with writer w.
Uses:
AuthPerfectPointToPointLinks, **instance** *al*.

upon event $\langle$ *bonrr*, *Init* $\rangle$ **do**
$(ts, val, \sigma) := (0, \perp, \perp)$;
$wts := 0$;
acklist $:= [\perp]^N$;
rid $:= 0$;
readlist $:= [\perp]^N$;

upon event $\langle$ *bonrr*, *Write* | v $\rangle$ **do** // only process w
$wts := wts + 1$;
acklist $:= [\perp]^N$;
$\sigma := sign(self, bonrr \| self \| \text{WRITE} \| wts \| v)$;
forall $q \in \Pi$ **do**
trigger $\langle$ *al*, *Send* | q, [WRITE, wts, v, σ] $\rangle$;

upon event $\langle$ *al*, *Deliver* | p, [WRITE, ts', v', σ'] $\rangle$ **such that** $p = w$ **do**
if $ts' > ts$ **then**
$(ts, val, \sigma) := (ts', v', \sigma')$;
trigger $\langle$ *al*, *Send* | p, [ACK, ts'] $\rangle$;

upon event $\langle$ *al*, *Deliver* | q, [ACK, ts'] $\rangle$ **such that** $ts' = wts$ **do**
acklist[q] := ACK;
if $\#(acklist) > (N + f)/2$ **then**
acklist $:= [\perp]^N$;
trigger $\langle$ *bonrr*, *WriteReturn* $\rangle$;

i.e. > 2f

Assumption
N>3f

upon event $\langle$ *bonrr*, *Read* $\rangle$ **do**
rid := *rid* + 1;
readlist $:= [\perp]^N$;
forall $q \in \Pi$ **do**
trigger $\langle$ *al*, *Send* | q, [READ, *rid*] $\rangle$;

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

upon event $\langle$ *al*, *Deliver* | q, [VALUE, r, ts', v', σ'] $\rangle$ **such that** $r = rid$ **do**
if $verifysig(q, bonrr \| w \| \text{WRITE} \| ts' \| v', \sigma')$ **then**
readlist[q] := (ts', v');
if $\#(readlist) > \frac{N+f}{2}$ **then**
$v := highestval(readlist)$;
readlist $:= [\perp]^N$;
trigger $\langle$ *bonrr*, *ReadReturn* | v $\rangle$;

i.e. > 2f

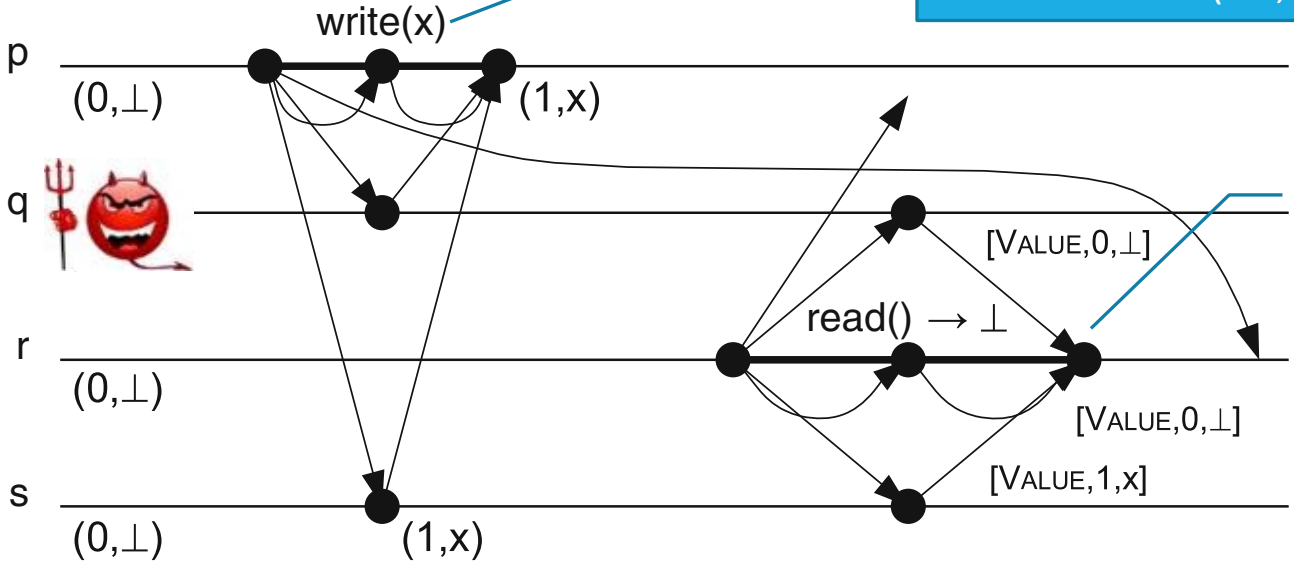
Regular Register
Implementation without cryptographic assumptions

the writer process p uses two phases to write a new
- a pre-write phase and
- a write phase

Assumption
N>3f

Why one phase is not enough?

In order to terminate, the write cannot wait more than N-f ack (i.e., 3)



The reader is not able to choose a value:
1. Value ⊥could be old given that process s provided value x with ts 1
2. Value x could be a fake value generated by a Byzantine process

Regular Register
Implementation without cryptographic assumptions

the writer process p uses two phases to write a new

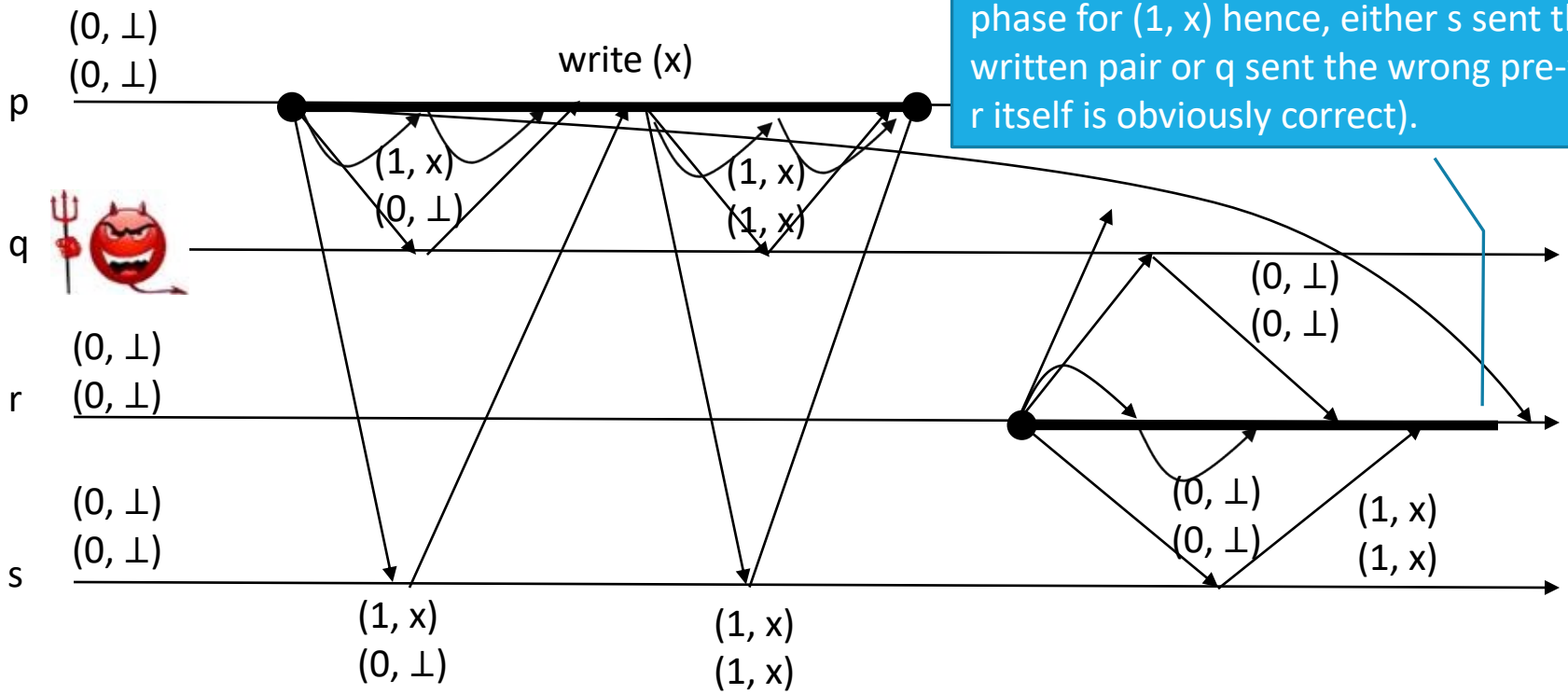
- a pre-write phase: the writer sends PREWRITE messages with the current timestamp/value pair. Then it waits until it receives PREACK messages from N − f processes

- a write phase: the writer sends ordinary WRITE messages, again containing the current time-stamp/value pair, and then waits until it receives ACK messages from N −f processes

Every process stores two timestamp/value pairs, one from the pre-write phase and one from the write phase

Intuition

r can infer that one of the three processes is faulty (the writer received N − f ack for the pre-write phase for (1, x) hence, either s sent the wrong written pair or q sent the wrong pre-written pair, as r itself is obviously correct).



Regular Register write Implementation without cryptographic assumptions

Algorithm 4.16: Double-Write Byzantine Quorum (part 1, write)

Implements:
$(1, N)$-ByzantineRegularRegister, **instance** *bonrr*, with writer w.

Uses:
AuthPerfectPointToPointLinks, **instance** *al*.

upon event $\langle$ *bonrr*, *Init* $\rangle$ **do**
$(pts, pval) := (0, \perp)$;
$(ts, val) := (0, \perp)$;
$(wts, wval) := (0, \perp)$;
preacklist := $[\perp]^N$;
acklist := $[\perp]^N$;
rid := 0;
readlist := $[\perp]^N$;

upon event $\langle$ *bonrr*, *Write* | v $\rangle$ **do** // only process w
$(wts, wval) := (wts + 1, v)$;
preacklist := $[\perp]^N$;
acklist := $[\perp]^N$;
forall $q \in \Pi$ **do**
trigger $\langle$ *al*, *Send* | q, [PREWRITE, $wts, wval$] $\rangle$;

upon event $\langle$ *al*, *Deliver* | p, [PREWRITE, $pts', pval'$] $\rangle$
such that $p = w \wedge pts' = pts + 1$ **do**
$(pts, pval) := (pts', pval')$;
trigger $\langle$ *al*, *Send* | p, [PREACK, pts] $\rangle$;

upon event $\langle$ *al*, *Deliver* | q, [PREACK, pts'] $\rangle$ **such that** $pts' = wts$ **do**
preacklist[q] := PREACK;
if $\#(preacklist) \geq N - f$ **then**
preacklist := $[\perp]^N$;
forall $q \in \Pi$ **do**
trigger $\langle$ *al*, *Send* | q, [WRITE, $wts, wval$] $\rangle$;

upon event $\langle$ *al*, *Deliver* | p, [WRITE, ts', val'] $\rangle$
such that $p = w \wedge ts' = pts \wedge ts' > ts$ **do**
$(ts, val) := (ts', val')$;
trigger $\langle$ *al*, *Send* | p, [ACK, ts] $\rangle$;

upon event $\langle$ *al*, *Deliver* | q, [ACK, ts'] $\rangle$ **such that** $ts' = wts$ **do**
acklist[q] := ACK;
if $\#(acklist) \geq N - f$ **then**
acklist := $[\perp]^N$;
trigger $\langle$ *bonrr*, *WriteReturn* $\rangle$;

Regular Register
read Implementation without cryptographic assumptions

Algorithm 4.17: Double-Write Byzantine Quorum (part 2, read)

upon event $\langle$ *bonrr*, *Read* $\rangle$ **do**
    $rid := rid + 1$;
    $readlist := [\perp]^N$;
    forall $q \in \Pi$ **do**
        trigger $\langle$ *al*, *Send* | q, [READ, rid] $\rangle$;

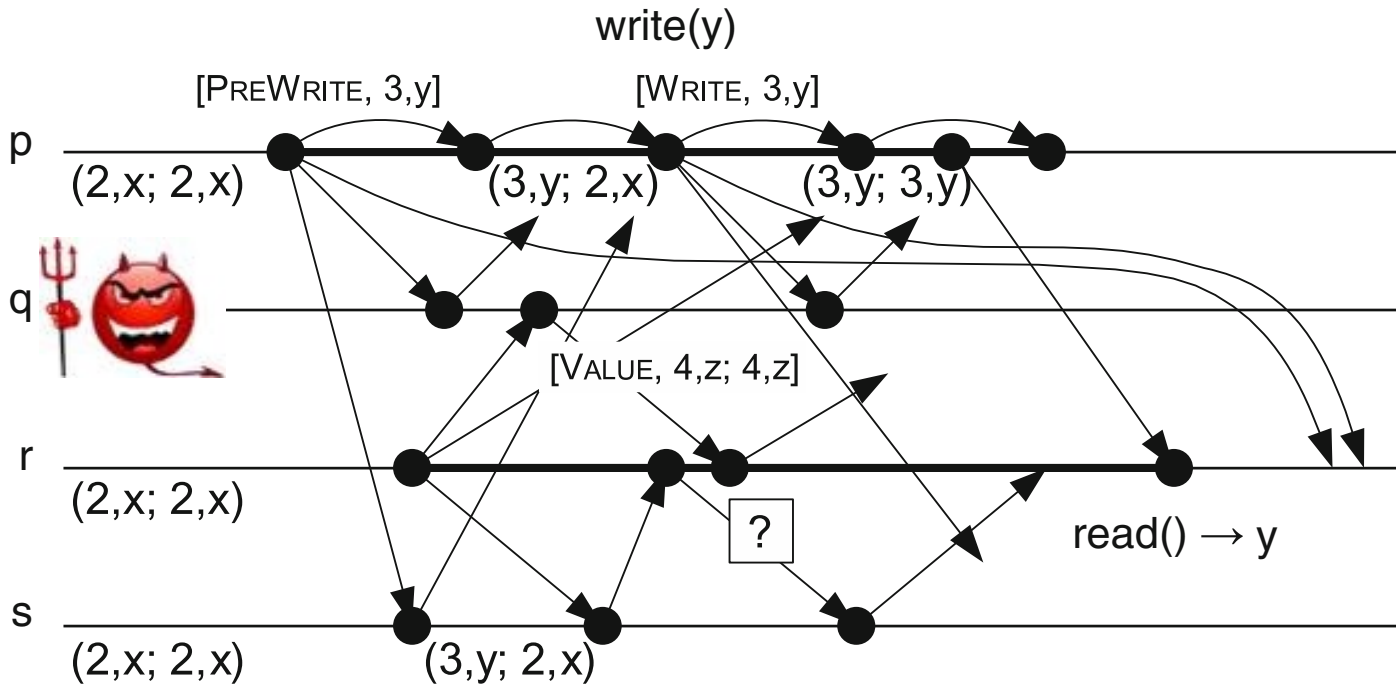
upon event $\langle$ *al*, *Deliver* | p, [READ, r] $\rangle$ **do**
    trigger $\langle$ *al*, *Send* | p, [VALUE, $r, pts, pval, ts, val$] $\rangle$;

upon event $\langle$ *al*, *Deliver* | q, [VALUE, $r, pts', pval', ts', val'$] $\rangle$ **such that** $r = rid$ **do**
    if $pts' = ts' + 1 \vee (pts', pval') = (ts', val')$ **then**
        $readlist[q] := (pts', pval', ts', val')$;
    if exists (ts, v) in an entry of *readlist* **such that** $authentic(ts, v, readlist) =$ TRUE
        and exists $Q \subseteq readlist$ **such that**
            $\#(Q) > \frac{N+f}{2} \wedge selectedmax(ts, v, Q) =$ TRUE **then**
        $readlist := [\perp]^N$;
        trigger $\langle$ *bonrr*, *ReadReturn* | v $\rangle$;
    else
        trigger $\langle$ *al*, *Send* | q, [READ, r] $\rangle$;

TRUE if *readlist* contains a pair (ts,v) that is found in the entries of more than f processes

readlist contains a pair (ts,v) such there is a Byzantine quorum (Q) of entries in *readlist* whose *highest* timestamp/value pair, selected among the pre-written or written pair of the entries, is (ts, v)

Example


write(y)
[PreWrite, 3,y]
[Write, 3,y]
p
(2,x; 2,x)
(3,y; 2,x)
(3,y; 3,y)
q
[Value, 4,z; 4,z]
r
(2,x; 2,x)
?
read() → y
s
(2,x; 2,x)
(3,y; 2,x)

Regular Register without cryptographic assumptions: Observations

Termination property must be relaxed in to *finite-write termination*

- Instead of requiring that *every* operation of a correct process eventually terminates, a *read* operation that is concurrent with *infinitely* many write operations may not terminate

It has been shown that such a relaxation is necessary

Exercise

Does the Regular Register implementation without cryptographic assumption satisfy also the atomic specification?

References

C. Cachin, R. Guerraoui and L. Rodrigues. Introduction to Reliable and Secure Distributed Programming, Springer, 2011

- Chapter 4 – Sections 4.6 and 4.7