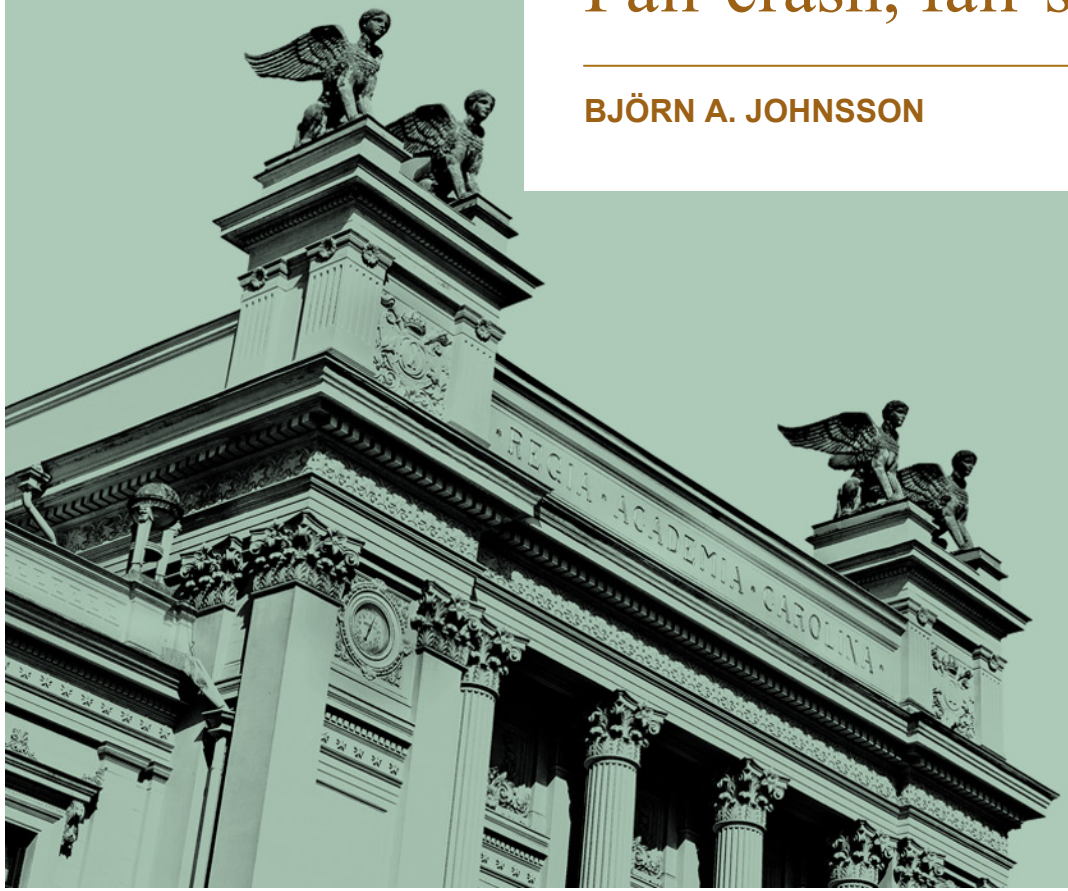

Registers – Shared Memory

Fail-crash, fail-silent

BJÖRN A. JOHNSSON

Introduction

- Analogy from multi-CPU computers.
- Over network: *emulation of shared-memory*.
- Benefit: use shared-memory where there really is none.
- Considered easier than message exchanges.

LUND
UNIVERSITY

Register Overview

- Process starts *read* operation with ⟨ *r*, *Read* ⟩
- Process starts *write* operation with ⟨ *r*, *Write* | *v* ⟩
- Process *completes* after reply event from register:
  - ⟨ *r*, *ReadReturn* | *v* ⟩
  - ⟨ *r*, *WriteReturn* ⟩
- Processes access registers in *sequential* manner
- Types: (1, 1), (1, *N*), (*N*, *N*)

Semantics

- *Liveness*: Every operation eventually completes.
- *Safety*: Every read operation returns the value written by the **last** write operation.

> A process p invokes a write operation on a register with a value v and completes this write. Later on, some other process q invokes a write operation on the register with a new value w, and then q crashes before the operation completes. Hence, q does not get any indication that the operation has indeed taken place before it crashes, and the operation has failed. Now, if a process r subsequently invokes a read operation on the register, what is the value that r is supposed to return? Should it be v or w?

LUND UNIVERSITY

Concurrency

- *Serial* (or *sequential*) exec: one operation after another
- *Concurrent* exec: what happens to def. of “last”?
- Three abstractions: *safe*, *regular*, and *atomic*.

Algorithm Overview

- (1, *N*) Regular Register
  - Read-One Write-All
  - Majority Voting Regular Register
- (1, *N*) Atomic Register
  - ~~(1, *N*) Regular → (1, 1) Atomic → (1, *N*) Atomic Register~~
  - Read-Impose Write-All
  - Read-Impose Write-Majority
- (*N*, *N*) Atomic Register
  - ~~(1, *N*) Atomic→ (*N*, *N*) Atomic Register~~
  - Read-Impose Write-Consult-All
  - Read-Impose Write-Consult-Majority

Repetition

Distributed-System Models

- Fail-stop
  - crash-stop, perfekt links, perfect failure detector (*P*)
- Fail-silent
  - crash-stop, perfekt links, no failure detector

(1, *N*) Regular Register

Module 4.1: Interface and properties of a $(1, N)$ regular register

Module:

Name: $(1, N)$-RegularRegister, **instance** *onrr*.

Events:

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

Request: $\langle$ *onrr*, *Write* | v $\rangle$: Invokes a write operation with value v on the register.

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

Indication: $\langle$ *onrr*, *WriteReturn* $\rangle$: Completes a write operation on the register.

Properties:

ONRR1: *Termination:* If a correct process invokes an operation, then the operation eventually completes.

ONRR2: *Validity:* A read that is not concurrent with a write returns the last value written; a read that is concurrent with a write returns the last value written or the value concurrently written.

initially ⊥

LUND UNIVERSITY

(1, *N*) Regular Register

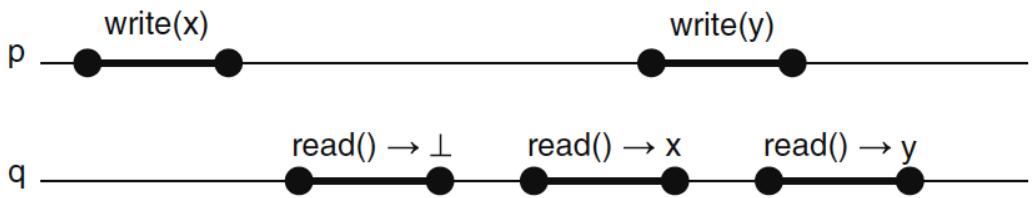

Figure 4.1: A register execution that is not regular because of the first read by process q

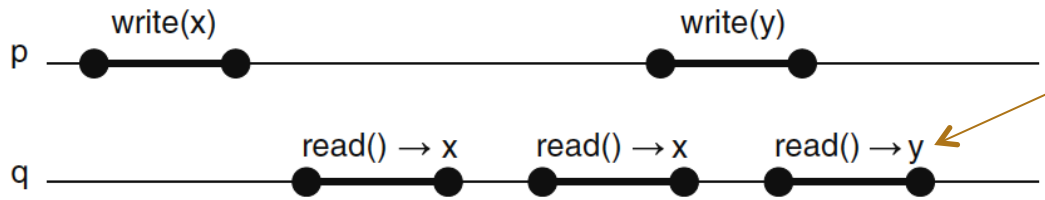

Figure 4.2: A regular register execution

(1, N) Regular Register

Read-One Write-All

Algorithm 4.1: Read-One Write-All

Implements:

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

Uses:

BestEffortBroadcast, **instance** *beb*;
PerfectPointToPointLinks, **instance** *pl*;
PerfectFailureDetector, **instance** $\mathcal{P}$.

upon event $\langle$ *onrr*, *Init* $\rangle$ **do**

$val := \bot$;
$correct := \Pi$;
$writeset := \emptyset$;

upon event $\langle$ $\mathcal{P}$, *Crash* | p $\rangle$ **do**

$correct := correct \setminus \{p\}$;

LUND UNIVERSITY

(1, *N*) Regular Register

Read-One Write-All

upon event ⟨ *onrr*, *Read* ⟩ **do**
    trigger ⟨ *onrr*, *ReadReturn* | *val* ⟩;

upon event ⟨ *onrr*, *Write* | *v* ⟩ **do**
    trigger ⟨ *beb*, *Broadcast* | [WRITE, *v*] ⟩;

upon event ⟨ *beb*, *Deliver* | *q*, [WRITE, *v*] ⟩ **do**
    val := *v*;
    trigger ⟨ *pl*, *Send* | *q*, ACK ⟩;

upon event ⟨ *pl*, *Deliver* | *p*, ACK ⟩ **do**
    writeset := *writeset* ∪ {*p*};

upon *correct* ⊆ *writeset* **do**
    writeset := ∅;
    trigger ⟨ *onrr*, *WriteReturn* ⟩;

Read-One Write-All

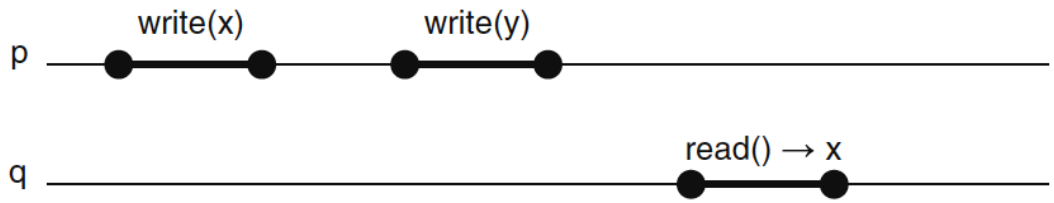

Figure 4.3: A non-regular register execution

Majority Voting Regular Register

Algorithm 4.2: Majority Voting Regular Register

Implements:

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

Uses:

BestEffortBroadcast, **instance** *beb*;
PerfectPointToPointLinks, **instance** *pl*.

upon event $\langle$ *onrr*, *Init* $\rangle$ **do**

$(ts, val) := (0, \perp)$;
$wts := 0$;
$acks := 0$;
$rid := 0$;
$readlist := [\perp]^N$;

upon event $\langle$ *onrr*, *Write* | v $\rangle$ **do**

$wts := wts + 1$;
$acks := 0$;
trigger $\langle$ *beb*, *Broadcast* | [WRITE, wts, v] $\rangle$;

upon event $\langle$ *beb*, *Deliver* | p, [WRITE, ts', v'] $\rangle$ **do**

if $ts' > ts$ **then**
$(ts, val) := (ts', v')$;
trigger $\langle$ *pl*, *Send* | p, [ACK, ts'] $\rangle$;

Majority Voting Regular Register

upon event $\langle$ *pl*, *Deliver* | q, [ACK, ts'] $\rangle$ **such that** $ts' = wts$ **do**
    $acks := acks + 1$;
    if $acks > N/2$ **then**
        $acks := 0$;
        trigger $\langle$ *onrr*, *WriteReturn* $\rangle$;

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**
        $v := highestval(readlist)$; ← returns pair with greatest time stamp
        $readlist := [\perp]^N$;
        trigger $\langle$ *onrr*, *ReadReturn* | v $\rangle$;

LUND UNIVERSITY

(1, N) Atomic Register

Module 4.2: Interface and properties of a $(1, N)$ atomic register

Module:

Name: $(1, N)$-AtomicRegister, **instance** *onar*.

Events:

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

Request: $\langle$ *onar*, *Write* | v $\rangle$: Invokes a write operation with value v on the register.

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

Indication: $\langle$ *onar*, *WriteReturn* $\rangle$: Completes a write operation on the register.

Properties:

ONAR1–ONAR2: Same as properties ONRR1–ONRR2 of a $(1, N)$ regular register (Module 4.1).

! **ONAR3:** *Ordering:* If a read returns a value v and a subsequent read returns a value w, then the write of w does not precede the write of v.

LUND UNIVERSITY

(1, *N*) Atomic Register

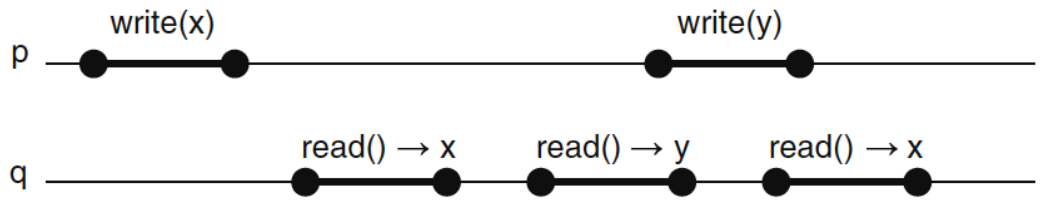

Figure 4.4: A register execution that is not atomic because of the third read by process q

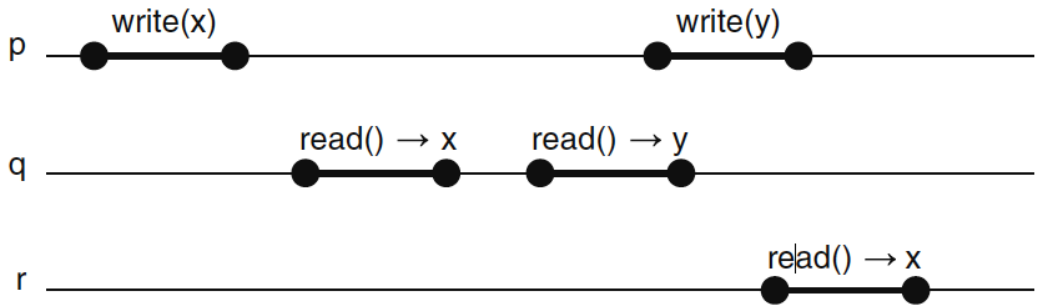

Figure 4.5: Violation of atomicity in the "Read-One Write-All" regular register algorithm

(1, *N*) Atomic Register

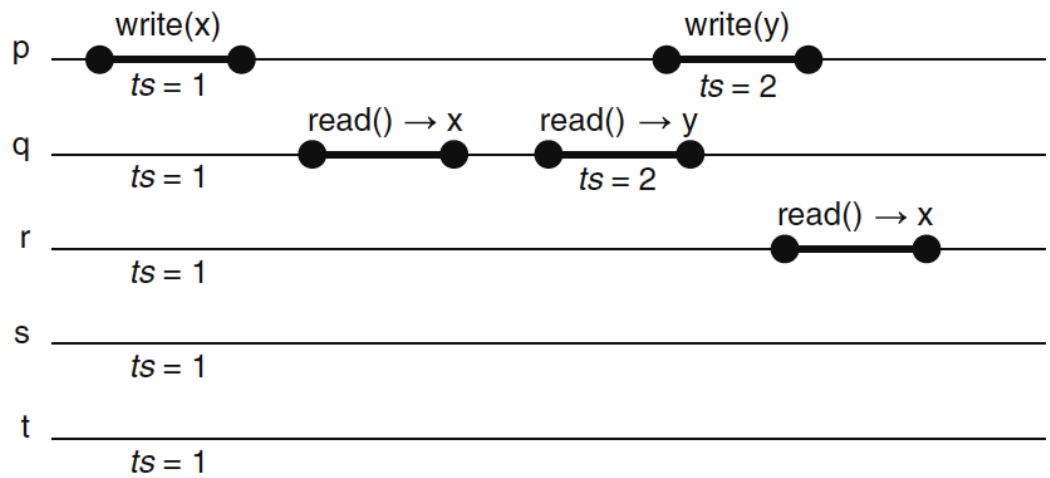

Figure 4.6: Violation of atomicity in the "Majority Voting" regular register algorithm

(1, *N*) Regular → (1, 1) Atomic Register

Algorithm 4.3: From $(1, N)$ Regular to $(1, 1)$ Atomic Registers

Implements:

$(1, 1)$-AtomicRegister, **instance** *ooar*.

Uses:

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

upon event $\langle$ *ooar*, *Init* $\rangle$ **do**

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

$wts := 0$;

upon event $\langle$ *ooar*, *Write* | v $\rangle$ **do**

$wts := wts + 1$;

trigger $\langle$ *onrr*, *Write* | (wts, v) $\rangle$;

upon event $\langle$ *onrr*, *WriteReturn* $\rangle$ **do**

trigger $\langle$ *ooar*, *WriteReturn* $\rangle$;

upon event $\langle$ *ooar*, *Read* $\rangle$ **do**

trigger $\langle$ *onrr*, *Read* $\rangle$;

upon event $\langle$ *onrr*, *ReadReturn* | (ts', v') $\rangle$ **do**

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

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

trigger $\langle$ *ooar*, *ReadReturn* | val $\rangle$;

(1, 1) Atomic → (1, *N*) Atomic Register

Algorithm 4.4: From $(1, 1)$ Atomic to $(1, N)$ Atomic Registers

Implements:

$(1, N)$-AtomicRegister, **instance** *onar*.

Uses:

$(1, 1)$-AtomicRegister (multiple instances).

upon event $\langle$ *onar*, *Init* $\rangle$ **do**

$ts := 0$;
acks $:= 0$;
writing := FALSE;
readval $:= \perp$;
readlist $:= [\perp]^N$;
forall $q \in \Pi, r \in \Pi$ **do**
Initialize a new instance *ooar*.$q.r$ of $(1, 1)$-AtomicRegister
with writer r and reader q;

upon event $\langle$ *onar*, *Write* | v $\rangle$ **do**

$ts := ts + 1$;
writing := TRUE;
forall $q \in \Pi$ **do**
trigger $\langle$ *ooar*.q.*self*, *Write* | (ts, v) $\rangle$;

(1, 1) Atomic → (1, *N*) Atomic Register

```
upon event ⟨ ooar.q.self, WriteReturn ⟩ do
    acks := acks + 1;
    if acks = N then
        acks := 0;
        if writing = TRUE then
            trigger ⟨ onar, WriteReturn ⟩;
            writing := FALSE;
        else
            trigger ⟨ onar, ReadReturn | readval ⟩;

upon event ⟨ onar, Read ⟩ do
    forall r ∈ Π do
        trigger ⟨ ooar.self.r, Read ⟩;

upon event ⟨ ooar.self.r, ReadReturn | (ts′, v′) ⟩ do
    readlist[r] := (ts′, v′);
    if #(readlist) = N then
        (maxts, readval) := highest(readlist);
        readlist := [⊥]^N;
        forall q ∈ Π do
            trigger ⟨ ooar.q.self, Write | (maxts, readval) ⟩;
```

Read-Impose Write-All

Algorithm 4.5: Read-Impose Write-All

Implements:
(1, N)-AtomicRegister, **instance** *onar*.

Uses:
BestEffortBroadcast, **instance** *beb*;
PerfectPointToPointLinks, **instance** *pl*;
PerfectFailureDetector, **instance** $\mathcal{P}$.

upon event ⟨ *onar*, *Init* ⟩ **do**
$(ts, val) := (0, \perp)$;
correct := Π;
writeset := $\emptyset$;
readval := $\perp$;
reading := FALSE;

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

upon event ⟨ *onar*, *Read* ⟩ **do**
! *reading* := TRUE;
readval := val;
trigger ⟨ *beb*, *Broadcast* | [WRITE, ts, val] ⟩;

same for ⟨ *onar*, *Write* ⟩ !

(1, *N*) Atomic Register

Read-Impose Write-All

upon event ⟨ *onar*, *Write* | v ⟩ **do**
  trigger ⟨ *beb*, *Broadcast* | [WRITE, $ts + 1, v$] ⟩;

upon event ⟨ *beb*, *Deliver* | p, [WRITE, ts', v'] ⟩ **do**
  if $ts' > ts$ **then**
    $(ts, val) := (ts', v')$;
  trigger ⟨ *pl*, *Send* | p, [ACK] ⟩;

upon event ⟨ *pl*, *Deliver* | p, [ACK] ⟩ **then**
  writeset := *writeset* ∪ $\{p\}$;

upon *correct* ⊆ *writeset* **do**
  writeset := ∅;
  if *reading* = TRUE **then**
    reading := FALSE;
    trigger ⟨ *onar*, *ReadReturn* | *readval* ⟩;
  else
    trigger ⟨ *onar*, *WriteReturn* ⟩;

same for ⟨ *onar*, *Read* ⟩ !

Read-Impose Write-Majority

Algorithm 4.6: Read-Impose Write-Majority (part 1, read)

Implements:

(1, N)-AtomicRegister, **instance** *onar*.

Uses:

BestEffortBroadcast, **instance** *beb*;
PerfectPointToPointLinks, **instance** *pl*.

upon event ⟨ *onar*, *Init* ⟩ **do**

$(ts, val) := (0, \perp)$;
$wts := 0$;
$acks := 0$;
$rid := 0$;
$readlist := [\perp]^N$;
$readval := \perp$;
$reading :=$ FALSE;

LUND UNIVERSITY

Read-Impose Write-Majority

upon event $\langle$ *onar*, *Read* $\rangle$ **do**
    $rid := rid + 1$;
    $acks := 0$;
    $readlist := [\bot]^N$;
    $reading :=$ TRUE;
    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**
        $(maxts, readval) :=$ *highest*$(readlist)$;
        $readlist := [\bot]^N$;
        trigger $\langle$ *beb*, *Broadcast* | [WRITE, *rid*, *maxts*, *readval*] $\rangle$;

Read-Impose Write-Majority

upon event $\langle$ *onar*, *Write* | v $\rangle$ **do**
    $rid := rid + 1$;
    $wts := wts + 1$;
    $acks := 0$;
    trigger $\langle$ *beb*, *Broadcast* | [WRITE, rid, wts, v] $\rangle$;

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

upon event $\langle$ *pl*, *Deliver* | q, [ACK, r] $\rangle$ **such that** $r = rid$ **do**
    $acks := acks + 1$;
    if $acks > N/2$ **then**
        $acks := 0$;
        if *reading* $=$ TRUE **then**
            reading := FALSE;
            trigger $\langle$ *onar*, *ReadReturn* | *readval* $\rangle$;
        else
            trigger $\langle$ *onar*, *WriteReturn* $\rangle$;

LUND UNIVERSITY

(N, N) Atomic Register

Module 4.3: Interface and properties of an (N, N) atomic register

Module:

Name: (N, N)-AtomicRegister, **instance** *nnar*.

Events:

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

Request: $\langle$ *nnar*, *Write* | v $\rangle$: Invokes a write operation with value v on the register.

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

Indication: $\langle$ *nnar*, *WriteReturn* $\rangle$: Completes a write operation on the register.

Properties:

NNAR1: *Termination:* Same as property ONAR1 of a $(1, N)$ atomic register (Module 4.2).

NNAR2: *Atomicity:* Every read operation returns the value that was written most recently in a hypothetical execution, where every failed operation appears to be complete or does not appear to have been invoked at all, and every complete operation appears to have been executed at some instant between its invocation and its completion.

LUND UNIVERSITY

(*N*, *N*) Atomic Register

The hypothetical serial execution mentioned before is called a *linearization* of the actual execution. More precisely, a linearization of an execution is defined as a sequence of complete operations that appear atomically, one after the other, which contains at least all complete operations of the actual execution (and possibly some operations that were incomplete) and satisfies the following conditions:

1. every read returns the last value written; and
2. for any two operations o and o', if o precedes o' in the actual execution, then o also appears before o' in the linearization.

We call an execution *linearizable* if there is a way to *linearize* it like this. With this notion, one can reformulate the *atomicity* property of an (N, N) atomic register in Module 4.3 as:

NNAR2': *Atomicity:* Every execution of the register is linearizable.

LUND UNIVERSITY

(1, *N*) Atomic→ (*N*, *N*) Atomic Register

Algorithm 4.8: From $(1, N)$ Atomic to (N, N) Atomic Registers

Implements:

(N, N)-AtomicRegister, **instance** *nnar*.

Uses:

$(1, N)$-AtomicRegister (multiple instances).

upon event $\langle$ *nnar*, *Init* $\rangle$ **do**

val := $\perp$;
writing := FALSE;
readlist := $[\perp]^N$;
forall $q \in \Pi$ **do**
Initialize a new instance *onar.q* of $(1, N)$-AtomicRegister
with writer q;

(1, *N*) Atomic→ (*N*, *N*) Atomic Register

```
upon event ⟨ nnar, Write | v ⟩ do
    val := v;
    writing := TRUE;
    forall q ∈ Π do
        trigger ⟨ onar.q, Read ⟩;

upon event ⟨ nnar, Read ⟩ do
    forall q ∈ Π do
        trigger ⟨ onar.q, Read ⟩;

upon event ⟨ onar.q, ReadReturn | (ts′, v′) ⟩ do
    readlist[q] := (ts′, rank(q), v′);
    if #(readlist) = N then
        (ts, v) := highest(readlist);
        readlist := [⊥]^N;
        if writing = TRUE then
            writing := FALSE;
            trigger ⟨ onar.self, Write | (ts + 1, val) ⟩;
        else
            trigger ⟨ nnar, ReadReturn | v ⟩;

upon event ⟨ onar.self, WriteReturn ⟩ do
        trigger ⟨ nnar, WriteReturn ⟩;
```

LUND UNIVERSITY

(*N*, *N*) Atomic Register

Read-Impose Write-Consult-All

Algorithm 4.9: Read-Impose Write-Consult-All

Implements:

(N, N)-AtomicRegister, **instance** *nnar*.

Uses:

BestEffortBroadcast, **instance** *beb*;
PerfectPointToPointLinks, **instance** *pl*;
PerfectFailureDetector, **instance** $\mathcal{P}$.

upon event $\langle$ *nnar*, *Init* $\rangle$ **do**

! $(ts, \underline{wr}, val) := (0, 0, \perp)$;
$correct := \Pi$;
$writeset := \emptyset$;
$readval := \perp$;
$reading :=$ FALSE;

upon event $\langle$ $\mathcal{P}$, *Crash* | p $\rangle$ **do**

$correct := correct \setminus \{p\}$;

(*N*, *N*) Atomic Register

Read-Impose Write-Consult-All

upon event ⟨ *nnar*, *Read* ⟩ **do**
    reading := TRUE;
    readval := *val*;
    trigger ⟨ *beb*, *Broadcast* | [WRITE, *ts*, *wr*, *val*] ⟩;

upon event ⟨ *nnar*, *Write* | *v* ⟩ **do**
    trigger ⟨ *beb*, *Broadcast* | [WRITE, $ts + 1$, *rank*(*self*), *v*] ⟩;

upon event ⟨ *beb*, *Deliver* | *p*, [WRITE, ts', wr', v'] ⟩ **do**
    if (ts', wr') is larger than (ts, wr) **then**
        $(ts, wr, val) := (ts', wr', v')$;
    trigger ⟨ *pl*, *Send* | *p*, [ACK] ⟩;

upon event ⟨ *pl*, *Deliver* | *p*, [ACK] ⟩ **then**
    writeset := *writeset* $\cup \{p\}$;

upon *correct* $\subseteq$ *writeset* **do**
    writeset := $\emptyset$;
    if *reading* = TRUE **then**
        reading := FALSE;
        trigger ⟨ *nnar*, *ReadReturn* | *readval* ⟩;
    else
        trigger ⟨ *nnar*, *WriteReturn* ⟩;

LUND UNIVERSITY

Read-Impose Write-Consult-Majority

Algorithm 4.10: Read-Impose Write-Consult-Majority (part 1, read and consult)

Implements:

(*N*, *N*)-AtomicRegister, **instance** *nnar*.

Uses:

BestEffortBroadcast, **instance** *beb*;
PerfectPointToPointLinks, **instance** *pl*.

upon event ⟨ *nnar*, *Init* ⟩ **do**

! $(ts, wr, val) := (0, 0, \perp)$;
$acks := 0$;
$writeval := \perp$;
$rid := 0$;
$readlist := [\perp]^N$;
$readval := \perp$;
$reading :=$ FALSE;

Read-Impose Write-Consult-Majority

upon event $\langle$ *nnar*, *Read* $\rangle$ **do**
  $rid := rid + 1$;
  $acks := 0$;
  $readlist := [\perp]^N$;
  $reading :=$ TRUE;
  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, wr, val] $\rangle$;

upon event $\langle$ *pl*, *Deliver* | q, [VALUE, r, ts', wr', v'] $\rangle$ **such that** $r = rid$ **do**
  $readlist[q] := (ts', wr', v')$;
  if $\#(readlist) > N/2$ **then**
    $(maxts, rr, readval) :=$ *highest*(*readlist*);
    $readlist := [\perp]^N$;
    if $reading =$ TRUE **then**
      trigger $\langle$ *beb*, *Broadcast* | [WRITE, *rid*, *maxts*, rr, *readval*] $\rangle$;
    else
      trigger $\langle$ *beb*, *Broadcast* | [WRITE, *rid*, *maxts* + 1, *rank*(*self*), *writeval*] $\rangle$;

factors in *rank*

LUND UNIVERSITY

Read-Impose Write-Consult-Majority

upon event $\langle$ *nnar*, *Write* | v $\rangle$ **do**
    $rid := rid + 1$;
    $writeval := v$;
    $acks := 0$;
    $readlist := [\bot]^N$;
    trigger $\langle$ *beb*, *Broadcast* | [READ, *rid*] $\rangle$;

upon event $\langle$ *beb*, *Deliver* | p, [WRITE, r, ts', wr', v'] $\rangle$ **do**
    if (ts', wr') is larger than (ts, wr) **then**
        $(ts, wr, val) := (ts', wr', v')$;
    trigger $\langle$ *pl*, *Send* | p, [ACK, r] $\rangle$;

upon event $\langle$ *pl*, *Deliver* | q, [ACK, r] $\rangle$ **such that** $r = rid$ **do**
    $acks := acks + 1$;
    if $acks > N/2$ **then**
        $acks := 0$;
        if *reading* = TRUE **then**
            reading := FALSE;
            trigger $\langle$ *nnar*, *ReadReturn* | *readval* $\rangle$;
        else
            trigger $\langle$ *nnar*, *WriteReturn* $\rangle$;

LUND UNIVERSITY

LUND
UNIVERSITY