DISTRIBUTED SYSTEMS

05

Fault-tolerant Registers

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 4.

Introduction

Three fundamental abstractions:

- Interpreters,
- Storage,
- Communication channels.

Introduction

Three fundamental abstractions:

- Interpreters,
- Storage,
- Communication channels.

Introduction

- A shared register is an abstraction of shared memory.
- Conceptually, it stores some value and provides two operations for processes accessing it:
  - *Read* – returns the stored value;
  - *Write* – replaces the stored value with a new one.
- This simple abstraction can be used to implement, for instance:
  - shared memory;
  - shared block storage;
  - shared files;
  - ...

Introduction

Our assumptions for today:

- The population of processes is relatively small (think of 3-4 up to a dozen processes).
- Each process maintains a copy of the register value.
- Only a (selected) member of the population may invoke the operations on the register (under some further assumptions).
- All processes are pairwise connected with communication links.
- We consider various models of failures, communication, and timing.

Interface and Semantics

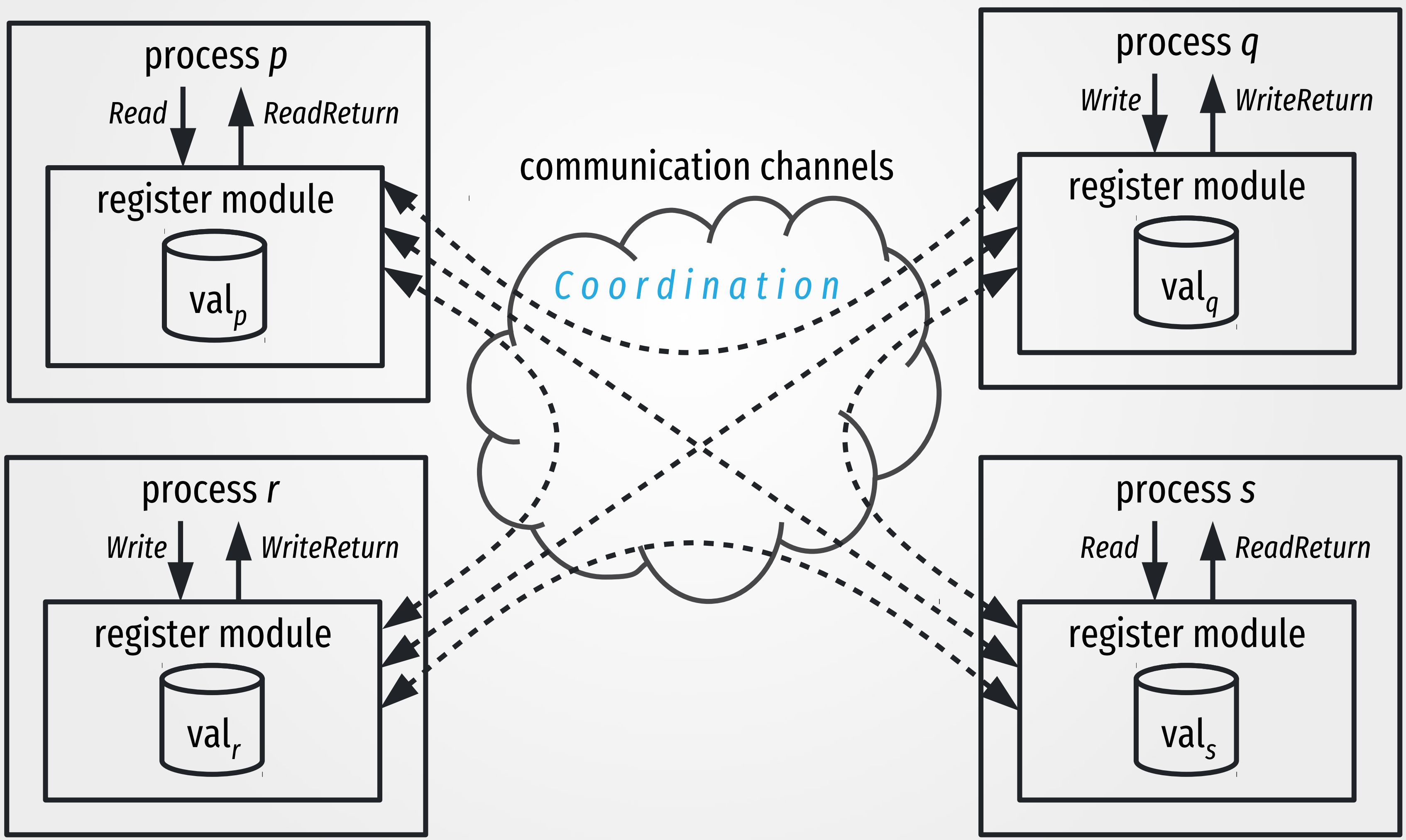
Reads:

- A process invokes a read operation on a register, r, by triggering a request event ⟨*r*, *Read*⟩.
- The register signals that it has terminated a read operation by triggering an indication event ⟨*r*, *ReadReturn* | *v*⟩, where *v* is the returned value (presumably the current value of the register).

Writes:

- A process invokes a write operation on a register, r, by triggering a request event ⟨*r*, *Write* | *v*⟩, where *v* is an input parameter containing the written value.
- The register signals that it has terminated a write operation by triggering an indication event ⟨*r*, *WriteReturn*⟩.

Interface and Semantics



Interface and Semantics

- Each process accesses a register in a sequential manner: it invokes no further operations until the previous one completes.
- The values *v* can be of an arbitrary domain.
- The initial value is $\perp$ and this value cannot be written.
- For simplicity, we assume that the written values are unique.
- Some of registers restrict the set of processes that may read from and write to a register:
  - ***(1, 1) register*** – has one writer and one reader;
  - ***(1, N) register*** – has one writer but any process can read;
  - ***(N, N) register*** – any process can be a writer or a reader.

Interface and Semantics

If:

- a register were to be accessed by only one process and
- the process did not experience any failures,

then we could define the semantics of a register as follows:

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

Interface and Semantics

If:

- a register were to be accessed by only one process and
- the process did not experience any failures,

then we could define the semantics of a register as follows:

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

Question: Under what assumption on register accesses could we extend this definition to multiple processes (but still without failures)?

Interface and Semantics

If:

- a register were to be accessed by only one process and
- the process did not experience any failures,

then we could define the semantics of a register as follows:

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

Question: Under what assumption on register accesses could we extend this definition to multiple processes (but still without failures)?

- If we assume that a process does not invoke an operation on a register if some process has invoked an operation and has not received a reply.

Interface and Semantics

- If a processes may fail, say by crashing, then an operations started by this process may fail as well.
- If a process invokes an operation and subsequently does not fail, then it should eventually get a reply to this invocation (i.e., complete the operation)...
- ... even if other processes fail.
- Algorithms implementing such a register are referred to as ***fault-tolerant***, ***robust***, or ***wait-free***.

Interface and Semantics

How do we define the notion of *last* under failures?

Interface and Semantics

Example:

- Process *p* invokes and completes a write with value *v*.
- Later, process *q* invokes a write with value *w* but crashes before the operation completes.
- Process *q* thus does not get any indication that its write has completed, and the operation fails.
- If another process, *r*, subsequently invokes a read operation on the register:
  - The operation has to complete (robustness)...
  - ... but what value should it return: *v*, or *w*, or yet something else?

Interface and Semantics

Both values may be valid, depending on what happens.

- The value returned has indeed been written by the last process that completed its write, even if some other process invoked a write later but crashed.
- In this case, no future read should return the value written by the failed write.
- Everything happens as if the failed operation has never been invoked.

- The value returned was the input parameter of the last write operation that was invoked, even if the writer process crashed before completing the operation.
- Everything happens as if the failed operation has been completed.

Interface and Semantics

- The underlying difficulty is that the failed write operation (by process *q*) does not complete and is therefore “concurrent” to the later read operation (by process *r*) and actually any subsequent operation.
- The same problem occurs even if process *q* does not fail but its operation is merely delayed.

Interface and Semantics

When multiple processes access a register, executions are most often not serial and clearly not sequential:

- What should we expect a read operation to return when it is concurrent with some write operation?
- What is the meaning of the “last” write in this context?
- If two write operations are invoked concurrently, what is the “last” value written?
- Can a subsequent read return one of the values and then a later read the other value?

Interface and Semantics

We specify three register abstractions that differ in the way these questions are addressed:

- ***safe register*** – may return an arbitrary value when a write is concurrently ongoing;
- ***regular register*** – ensures a minimal guarantee in the face of concurrent or failed operations and may only return the previous or the newly written value;
- ***atomic register*** – is even stronger and provides a strict form of consistency in the face of concurrency and failures.

Considered Failure Model

Crash-stop failures

Completeness and Precedence

For the purpose of further reasoning, we will more precisely define the notions for:

- ***completeness*** of the execution of an operation;
- ***precedence*** between different operation executions.

Operations:

- $\langle r, \mathit{Read} \rangle$
- $\langle r, \mathit{Write} \mid v \rangle$

invocation of a read/write operation.

- $\langle r, \mathit{ReadReturn} \mid v \rangle$
- $\langle r, \mathit{WriteReturn} \rangle$

completion of a read/write operation.

Occur at a single indivisible point in time.

Completeness and Precedence

- An operation is said to be ***complete*** iff its invocation and completion events have *both* occurred.
  - The process invoking the operation does not crash before the operation terminates AND
  - The completion event occurs at the invoking process.
- An operation is said to ***fail*** iff when the process that has invoked it crashes *before* the corresponding completion event occurs.

Completeness and Precedence

- An operation, *o*, is said to ***precede*** another operation, *o'*, iff the completion event of *o* occurs before the invocation event of *o'*.
- If two operations are such that one precedes the other, then we say that the operations are ***sequential***.
- If neither one of two operations precedes the other, then we say that they are ***concurrent***.

Completeness and Precedence

- An operation, *o*, is said to ***precede*** another operation, *o'*, iff the completion event of *o* occurs before the invocation event of *o'*.
- If two operations are such that one precedes the other, then we say that the operations are ***sequential***.
- If neither one of two operations precedes the other, then we say that they are ***concurrent***.

Question: What does the precedence relation define?

Completeness and Precedence

- The precedence relation defines a partial order on read and write operations in an execution.

Completeness and Precedence

- The precedence relation defines a partial order on read and write operations in an execution.

Question: When is this order total?

Completeness and Precedence

- The precedence relation defines a partial order on read and write operations in an execution.

Question: When is this order total?

- If only one process invokes the operations because every process operates sequentially on a given register.
- When no two operations are concurrent and all operations are complete.

Completeness and Precedence

- When a read operation, o_r, returns a value, *v*, and *v* was the input parameter of some write operation, o_w, we say that o_r *reads from* o_w or that value *v is read from* o_w.
- When a write operation, o_w, with input parameter *v* completes, we say that value *v is written (by* o_w*)*.
- NB: Write operations are unique.

(1, N) Regular Register

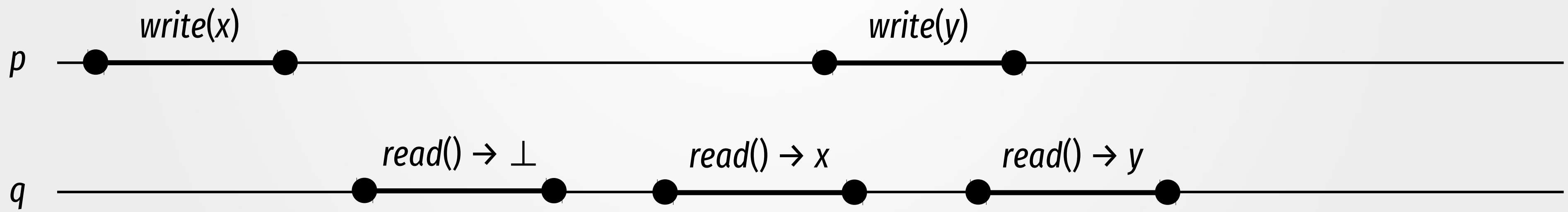
Module:
Name: (1, N)-RegularRegister, **instance** *onrr*.

Events:
Request: ⟨*onrr*, *Read*⟩: Invokes a read operation on the register.
Request: ⟨*onrr*, *Write* | *v*⟩: Invokes a write operation with value *v* on the register.
Indication: ⟨*onrr*, *ReadReturn* | *v*⟩: Completes a read operation on the register with return value *v*.
Indication: ⟨*onrr*, *WriteReturn*⟩: 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.

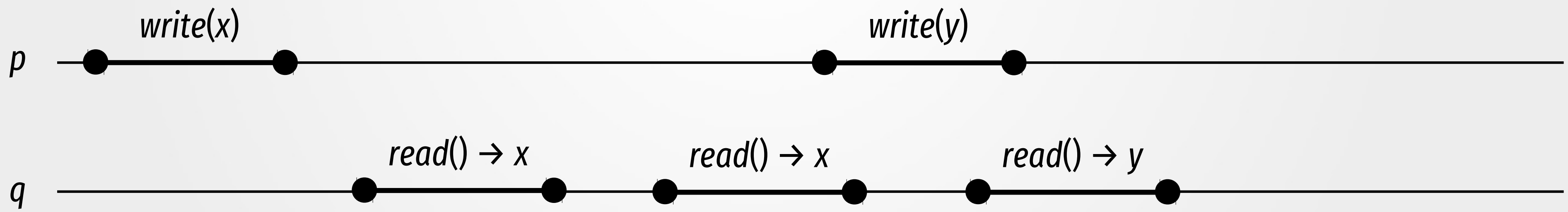
(1, N) Regular Register

Question: Is the following execution possible for a (1, N) Regular Register?



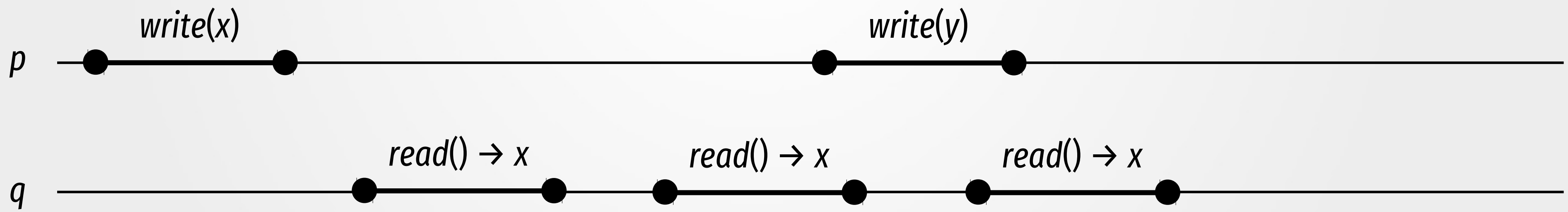
(1, N) Regular Register

Question: What about this one?



(1, N) Regular Register

Question: And this one?



(1, N) Regular Register

Algorithm: Read-One Write-All

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

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

upon event ⟨*onrr*, *Init*⟩ **do**
val := ⊥;
correct := Π;
writeset := ∅;

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

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*⟩;

(1, N) Regular Register

Correctness of the algorithm – Cheatsheet:

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*.

(1, N) Regular Register

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*.

(1, N) Regular Register

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.

(1, N) Regular Register

Correctness of the algorithm:

- Termination:
  - Read:
    - Straightforward because the invoking process returns its local value.
  - Write:
    - Any process that crashes is detected (completeness of $\mathcal{P}$).
    - Any process that does not crash eventually delivers the write (validity of BEB) and sends an acknowledgment.
    - The originator thus delivers the acknowledgment from every correct process (reliable delivery of PL).

(1, N) Regular Register

Correctness of the algorithm (cont.):

- Validity (induction on the number of writes; below just the inductive step):
  - Assume first that there is no concurrency and that all previous operations are complete:
    - Assume that *v* is the last value written and consider a read invoked by some process *p*.
    - Because of the accuracy property of $\mathcal{P}$, at the moment when the read is invoked all correct processes, in particular process *p*, store value *v* as their local values.
    - Process *p* thus returns *v*, which is the last value written.
  - Assume that the read is concurrent with some write of a value, *v*, and that the value written before is *v'*:
    - Like previously, when it invokes the write of *v*, every process has *v'* as its local value (we have only 1 writer).
    - Because of no creation of BEB and PL, no value is stored by any process unless the writer has invoked a write operation with this value.
    - At the time of the read, every process either still stores *v'* or has BEB-delivered the write message with *v* and thus stores *v*.
    - Therefore, the return value is either *v'* or *v*.

(1, N) Regular Register

Performance of the algorithm:

- The work of the algorithm is:
  - $2 \cdot N$ for a write,
  - 0 for a read.
- The span of the algorithm is:
  - 2 for a write,
  - 0 for a read.

(1, N) Regular Register

Algorithm: Read-One Write-All

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

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

upon event ⟨*onrr*, *Init*⟩ **do**
val := ⊥;
correct := Π;
writeset := ∅;

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

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]⟩;

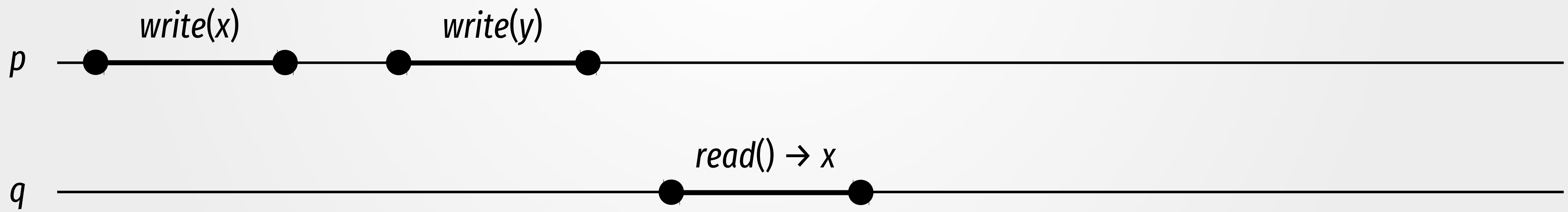
upon event ⟨*pl*, *Deliver* | *p*, [Ack]⟩ **do**
writeset := *writeset* ∪ { *p* };

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

Question: What happens if the failure detector is not perfect?

(1, N) Regular Register

- During the second write, process *p* falsely suspects that process *q* has crashed.



(1, N) Regular Register

Question: How to get rid of the failure detector?

(1, N) Regular Register

```
Algorithm: Majority Voting Regular Register

Implements:
   (1, N)-RegularRegister, instance onrr.

Uses:
   BestEfforBroadcast, instance beb.
   PerfectLinks, instance pl.

upon event ⟨onrr, Init⟩ do
   (ts, val) := (0, ⊥);
   wts := acks := rid := 0;
   readlist := [⊥]^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 ⟨onrr, Read⟩ do
   rid := rid + 1;
   readlist := [⊥]^N;
   trigger ⟨beb, Broadcast | [Read, rid]⟩;

upon event ⟨beb, Deliver | p, [Read, r]⟩ do
   trigger ⟨pl, Send | p, [Value, r, ts, val]⟩;

upon event ⟨pl, Deliver | q, [Value, r, ts', v']⟩
      such that r = rid do
   readlist[q] := (ts', v');
   if #(readlist) > N / 2 then
      v := highestval(readlist);
      readlist := [⊥]^N;
      trigger ⟨onrr, ReadReturn | v⟩;
```

Function *highestval* returns the value with the largest timestamp.

(1, N) Regular Register

Question: What do we lose here?

(1, N) Regular Register

Question: What do we lose here?

A majority of processes must be correct.

(1, N) Regular Register

Correctness of the algorithm:

- Termination – follows from the properties of BEB (validity) and PL (reliable delivery), and from the assumption that a majority of processes in the system are correct.
- Validity (like previously):
  - Consider a read operation, invoked by some process *q*, that is not concurrent with any write:
    - Assume that the last value written by the writer, process *p*, is *v* and the associated timestamp is *wts*.
    - This means that at the moment when the read is invoked some majority of processes, *C*, store *wts* and *v* in their variables *ts* and *val*, respectively, and that, due to the algorithm and no creation of BEB and PL, there is no larger timestamp in the system.
    - Before returning from the read operation, process *q* consults some majority of processes, *S*.
    - Since $C \cap S \neq \emptyset$, at least one process in *S* has its timestamp and value equal *wts* and *v*, respectively.
    - Given the definition of function *highestval*, process *q* returns *v* as the read return value.
  - Consider the case when a read by some process, *q*, is concurrent with some write of value *v* with timestamp *wts*, whereas the previous write was for value *v'* and had its timestamp equal to *wts* – 1:
    - If any process returns the pair (*wts*, *v*) to the reader, *q*, then *q* returns *v*, which is a valid result.
    - Otherwise, at least one reply from more a majority of processes is (*wts* - 1, *v'*), and thus q returns *v'*, which is also a valid reply.

(1, N) Regular Register

Performance of the algorithm:

- The work of the algorithm is $2 \cdot N$ for both a read and a write.
- The span of the algorithm is 2 for both a read and a write.

(1, N) Regular Register

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

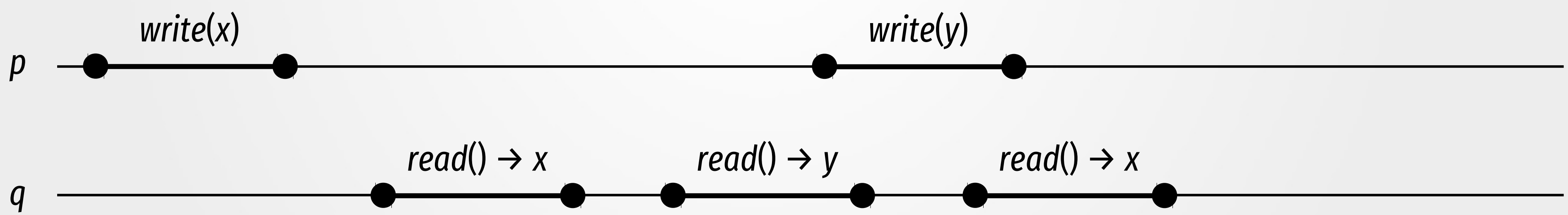
Events:
Request: ⟨*onrr*, *Read*⟩: Invokes a read operation on the register.
Request: ⟨*onrr*, *Write* | *v*⟩: Invokes a write operation with value *v* on the register.
Indication: ⟨*onrr*, *ReadReturn* | *v*⟩: Completes a read operation on the register with return value *v*.
Indication: ⟨*onrr*, *WriteReturn*⟩: 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.

Question: Can it display some nonintuitive behaviors?

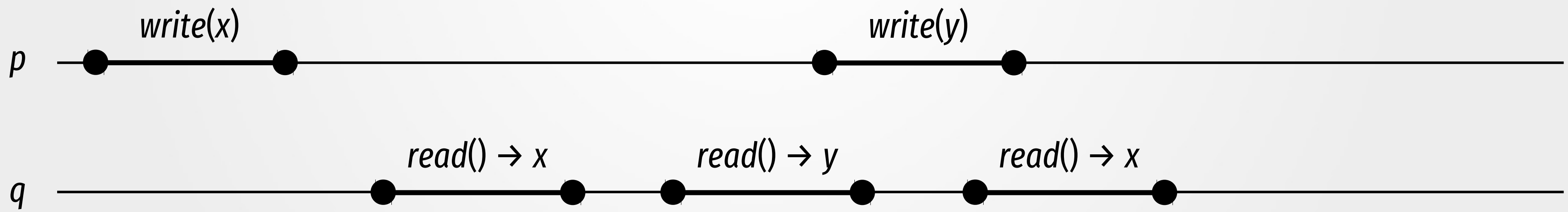
(1, N) Regular Register

- In the presence of concurrency, we may observe a “time travel” effect.



(1, N) Regular Register

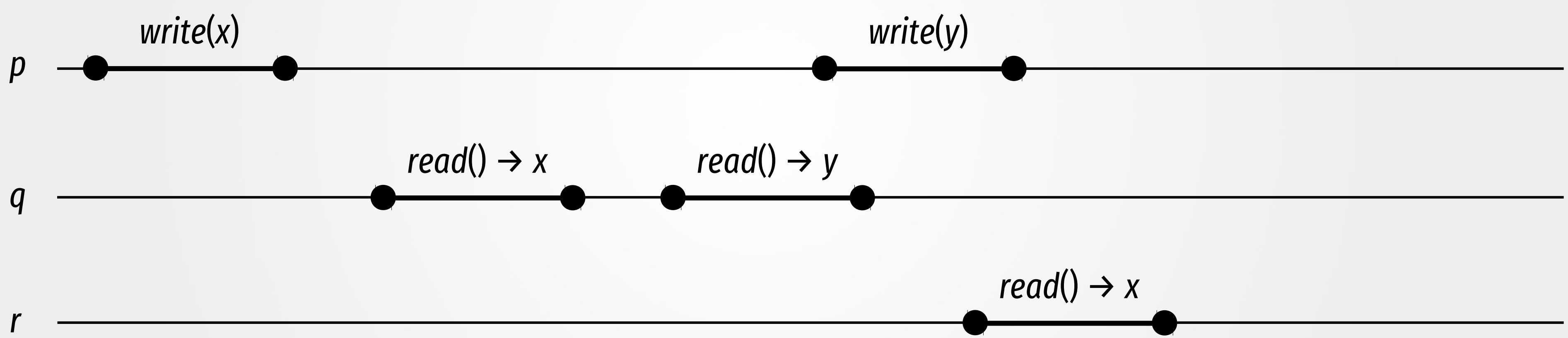
- In the presence of concurrency, we may observe a “time travel” effect.



Question: Does it actually occur for the presented algorithms?

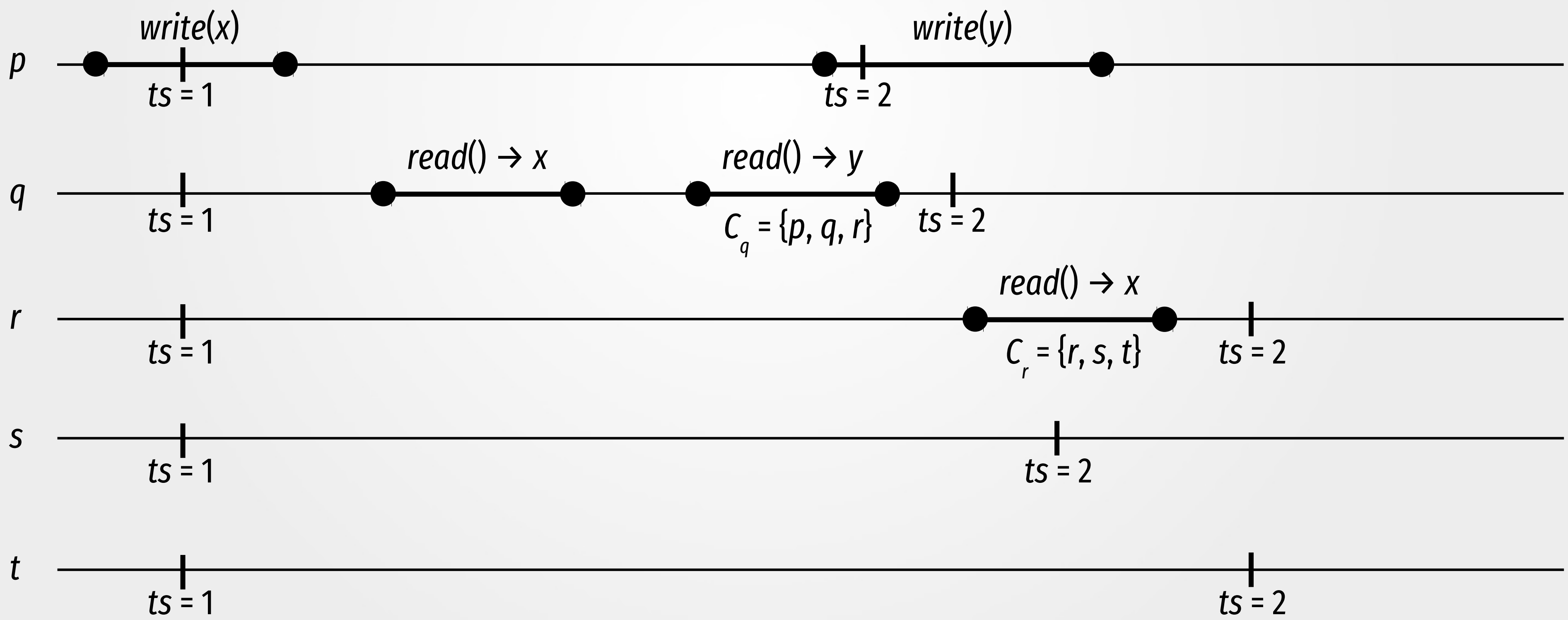
(1, N) Regular Register

- A possible execution in the Read One Write All algorithm (global time travel):



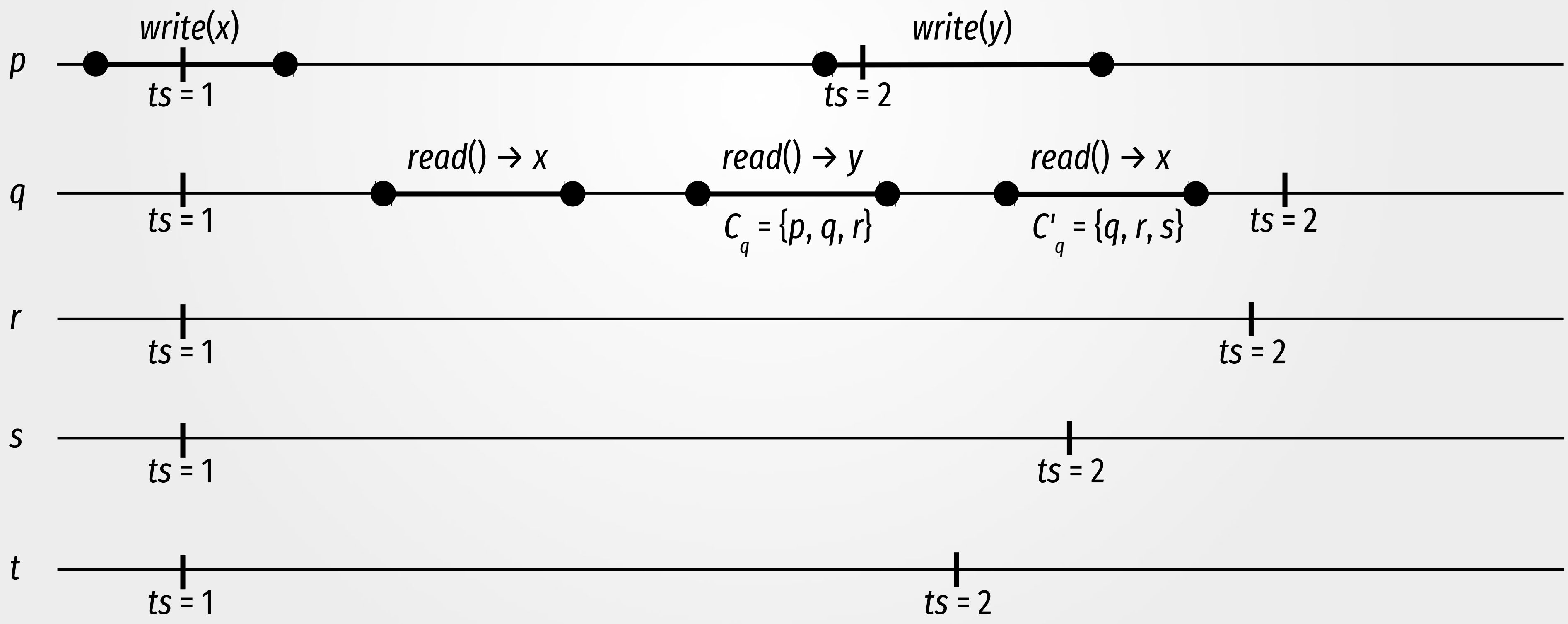
(1, N) Regular Register

- A possible execution in the Majority Voting Regular Register algorithm (global time travel):



(1, N) Regular Register

- A possible execution in the Majority Voting Regular Register algorithm (local time travel):



(1, N) Regular Register

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

Events:
Request: ⟨*onrr*, *Read*⟩: Invokes a read operation on the register.
Request: ⟨*onrr*, *Write* | *v*⟩: Invokes a write operation with value *v* on the register.
Indication: ⟨*onrr*, *ReadReturn* | *v*⟩: Completes a read operation on the register with return value *v*.
Indication: ⟨*onrr*, *WriteReturn*⟩: 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.

Question: How to prevent time travel?

(1, N) Atomic Register

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

Events:
Request: ⟨*onar*, *Read*⟩: Invokes a read operation on the register.
Request: ⟨*onar*, *Write* | *v*⟩: Invokes a write operation with value *v* on the register.
Indication: ⟨*onar*, *ReadReturn* | *v*⟩: Completes a read operation on the register with return value *v*.
Indication: ⟨*onar*, *WriteReturn*⟩: Completes a write operation on the register.

Properties:
ONAR1: *termination*: If a correct process invokes an operation, then the operation eventually completes.
ONAR2: *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.
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*.

(1, N) Atomic Register

Module:

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

Events:

Request: ⟨*onar*, *Read*⟩: Invokes a read operation on the register.
Request: ⟨*onar*, *Write* | *v*⟩: Invokes a write operation with value *v* on the register.
Indication: ⟨*onar*, *ReadReturn* | *v*⟩: Completes a read operation on the register with return value *v*.
Indication: ⟨*onar*, *WriteReturn*⟩: Completes a write operation on the register.

Properties:

ONAR1: *termination*: If a correct process invokes an operation, then the operation eventually completes.
ONAR2: *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.
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*.

Question: How to enforce the ordering property?

(1, N) Atomic Register

- Idea: Decorate a regular register.

(1, N) Atomic Register

- Idea: Decorate a regular register.
- Not that straightforward because the ordering property concerns reads by any processes...

(1, N) Atomic Register

- Idea: Decorate a regular register.
- Not that straightforward because the ordering property concerns reads by any processes...
- ..., so for a while assume there is only one reader: we will first develop a (1, 1) Atomic Register.

(1, N) Atomic Register

- Idea: Decorate a regular register.
- Not that straightforward because the ordering property concerns reads by any processes...
- ..., so for a while assume there is only one reader: we will first develop a (1, 1) Atomic Register.
- (Note: The reader and the writer can be different processes.)

(1, 1) Atomic Register

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

Implements:
(1, 1)-AtomicRegister, **instance** *ooar*.

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

upon event ⟨*ooar*, *Init*⟩ **do**
  (*ts*, *val*) := (0, ⊥);
  wts := 0;

upon event ⟨*ooar*, *Write* | *v*⟩ **do**
  wts := *wts* + 1;
  trigger ⟨*onrr*, *Write* | (*wts*, *v*)⟩;

upon event ⟨*onrr*, *WriteReturn*⟩ **do**
  trigger ⟨*ooar*, *WriteReturn*⟩;

upon event ⟨*ooar*, *Read*⟩ **do**
  trigger ⟨*onrr*, *Read*⟩;

upon event ⟨*onrr*, *ReadReturn* | (*ts'*, *v'*)⟩ **do**
  if *ts'* > *ts* **then**
    (*ts*, *val*) := (*ts'*, *v'*);
  trigger ⟨*ooar*, *ReadReturn* | *val*⟩;

(1, 1) Atomic Register

Correctness of the algorithm:

- Termination – follows from termination of RR.
- Validity (follows from validity of RR and monotonicity of the reader's timestamps):
  - Consider a read operation, invoked by the reader, process *q*, that is not concurrent with a write:
    - Assume that the last value written by the writer, process *p*, is *v* and the associated timestamp is *ts'*.
    - The reader's timestamp stored by process *q* is either *ts'*, if *q* has already read the last write in some previous read, or a strictly smaller value.
    - Because of validity of RR, in both cases, process *q* will return *v* as the result.
  - Consider the case when a read by the reader, process, *q*, is concurrent with some write of value *v* with timestamp *ts'*, whereas the previous write was for value *v'* and had its timestamp equal to *ts'* – 1:
    - The reader's timestamp stored by process *q* is at most *ts'*.
    - Therefore, from validity of RR, process q will return either *v* or *v'*, and they both are valid results.
- Ordering:
  - Assume that process *p* writes *v* with timestamp *ts* and then writes *w* with timestamp *ts'*.
  - Suppose that process *q* returns *w* for some read and consider any subsequent read by *q*.
  - The reader timestamp stored by *q* is either *ts'* or a larger value.
  - The algorithm precludes returning any value with a smaller timestamp, in particular value *v*.

(1, 1) Atomic Register

Performance of the algorithm – the same as of the underlying regular register.

(1, N) Atomic Register

Question: How to transform a (1, 1) register to a (1, N) register?

- Idea: Use a N x N matrix of (1, 1) registers, each having precisely one writer and one reader.
- This works because in a (1, 1) register, the reader and the writer need not be the same process.

(1, N) Atomic Register

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

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

Uses:
(1, 1)-AtomicRegister, **multiple instances**.

```
upon event ⟨onar, Init⟩ do
  ts := acks := 0;
  writing := FALSE;
  readval := ⊥;
  readlist := [⊥]^N;
  forall q ∈ Π, r ∈ Π do
    Initialize a new instance, ooar.q.r of
      (1, 1)-AtomicRegister with writer r and reader q;

upon event ⟨onar, Write | v⟩ do
  ts := ts + 1;
  writing := TRUE;
  forall q ∈ Π do
    trigger ⟨ooar.q.self, Write | (ts, v)⟩;

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

Function *highest* returns the (timestamp, value) pair with the largest timestamp.

(1, N) Atomic Register

Correctness of the algorithm:

- Termination – follows from termination of (1, 1) AR.
- Validity – follows from validity of (1, 1) AR and the choice of a value with the largest timestamp.
- Ordering:
  - Consider an ONAR-write operation of a value *v* with timestamp *tsv* that precedes an ONAR-write with value *w* and timestamp *tsw* (i.e., *tsv* < *tsw*).
  - ONAR-reading value *w* by process *r* means that the process has written (*tsw*, *w*) to *N* underlying (1, 1) atomic registers with identifiers *ooar.q.r* for all $q \in \Pi$.
  - Because of ordering of the (1, 1) atomic registers, every subsequent ONAR-read operation reads at least one of the underlying registers that contains (*tsw*, *w*) or a pair containing a larger timestamp.
  - Because of the definition of function *highest*, the ONAR-read returns a value associated with a timestamp that is at least *tsw*. In other words, there is no way for the algorithm to return *v*.

(1, N) Atomic Register

Performance of the algorithm:

- Write: every ONAR-write requires *N* OOAR-writes.
- Read: every ONAR-read requires *N* OOAR-reads and *N* OOAR-writes.

(1, N) Atomic Register

Performance of the algorithm:

- Write: every ONAR-write requires N OOAR-writes.
- Read: every ONAR-read requires N OOAR-reads and N OOAR-writes.

Recall that OOAR can be implemented by ONRR, whose performance is as follows:

- The work of the algorithm is $2 \cdot N$ for both a read and a write.
- The span of the algorithm is 2 for both a read and a write.

(1, N) Atomic Register

Performance of the algorithm:

- Write: every ONAR-write requires N OOAR-writes.
- Read: every ONAR-read requires N OOAR-reads and N OOAR-writes.

Recall that OOAR can be implemented by ONRR, whose performance is as follows:

- The work of the algorithm is $2 \cdot N$ for both a read and a write.
- The span of the algorithm is 2 for both a read and a write.

Therefore, the end-to-end performance of the algorithm is as follows:

- The work of the algorithm is:
  - $2 \cdot N^2$ for a write,
  - $4 \cdot N^2$ for a read.
- The span of the algorithm is:
  - 2 for a write,
  - 4 for a read.

(1, N) Atomic Register

Performance of the algorithm:

- Write: every ONAR-write requires N OOAR-writes.
- Read: every ONAR-read requires N OOAR-reads and N OOAR-writes.

Recall that OOAR can be implemented by ONRR, whose performance is as follows:

- The work of the algorithm is $2 \cdot N$ for both a read and a write.
- The span of the algorithm is 2 for both a read and a write.

Therefore, the end-to-end performance of the algorithm is as follows:

- The work of the algorithm is:
  - $2 \cdot N^2$ for a write,
  - $4 \cdot N^2$ for a read.
- The span of the algorithm is:
  - 2 for a write,
  - 4 for a read.

Question: Can we improve on this with a monolithic algorithm?

(1, N) Atomic Register

Algorithm: Read-Impose Write-All

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

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

upon event ⟨*onar*, *Init*⟩ **do**
  (*ts*, *val*) := (0, ⊥);
  correct := *Π*;
  writeset := ∅;
  readval := ⊥;
  reading := FALSE;

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

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⟩ **do**
  writeset := *writeset* ∪ { *p* };

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

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

(1, N) Atomic Register

Correctness of the algorithm – exercise.

(1, N) Atomic Register

Performance of the algorithm:

- The work of the algorithm is $2 \cdot N$ for both a read and a write.
- The span of the algorithm is 2 for both a read and a write.

(1, N) Atomic Register

Algorithm: Read-Impose Write-All

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

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

upon event ⟨*onar*, *Init*⟩ **do**
  (*ts*, *val*) := (0, ⊥);
  correct := *Π*;
  writeset := ∅;
  readval := ⊥;
  reading := FALSE;

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

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⟩ **do**
  writeset := *writeset* ∪ { *p* };

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

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

Question: No failure detector?

(1, N) Atomic Register

Algorithm: Read-Impose Write-Majority (part 1)

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

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

upon event ⟨*onar*, *Init*⟩ **do**
(*ts*, *val*) := (0, ⊥);
wts := *acks* := *rid* := 0;
readlist := $[\perp]^N$;
readval := ⊥;
reading := FALSE;

upon event ⟨*onar*, *Read*⟩ **do**
rid := *rid* + 1;
acks := 0;
readlist := $[\perp]^N$;
reading := TRUE;
trigger ⟨*beb*, *Broadcast* | [Read, *rid*]⟩;

upon event ⟨*beb*, *Deliver* | *p*, [Read, *r*]⟩ **do**
trigger ⟨*pl*, *Send* | *p*, [Value, *r*, *ts*, *val*]⟩;

upon event ⟨*pl*, *Deliver* | *q*, [Value, *r*, *ts'*, *v'*]⟩
such that *r* = *rid* **do**
readlist[*q*] := (*ts'*, *v'*);
if #(*readlist*) > *N* / 2 **then**
(*maxts*, *readval*) := *highest*(*readlist*);
readlist := $[\perp]^N$;
trigger ⟨*beb*, *Broadcast* | [Write, *rid*, *maxts*, *readval*]⟩;

(1, N) Atomic Register

Algorithm: Read-Impose Write-Majority (part 2)

upon event ⟨*onar*, *Write* | *v*⟩ **do**
    rid := *rid* + 1;
    wts := *wts* + 1;
    acks := 0;
    trigger ⟨*beb*, *Broadcast* | [Write, *rid*, *wts*, *v*]⟩;

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

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

(1, N) Atomic Register

Correctness of the algorithm:

- Termination and validity – a similar reasoning as in the Majority Voting Regular Register.
- Ordering:
  - Suppose that a read operation, *or*, by process *r* reads a value, *v*, from a write operation, *ov*, by process *p* (the only writer) and that a read operation, *oq*, by process *q* reads a different value, *w*, from a write operation, *ow*, also by process *p*, and that *or* precedes *oq*.
  - Assume by contradiction that *ow* precedes *ov*.
  - According to the algorithm, the timestamp, *tsv*, associated with value *v* is strictly larger than the timestamp, *tsw*, associated with value *w*.
  - Given that read operation *or* precedes read operation *oq*, at the time when *oq* was invoked, a majority of processes have stored a timestamp value in their *ts* variables that is at least *tsv* (the write-back part of the algorithm).
  - Therefore, process *q* could not have read *w* because the timestamp associated with *w*, *tsw*, is strictly smaller than *tsv* – contradiction!

(1, N) Atomic Register

Performance of the algorithm:

- The work of the algorithm is:
  - $2 \cdot N$ for a write,
  - $4 \cdot N$ for a read.
- The span of the algorithm is:
  - 2 for a write,
  - 4 for a read.

(1, N) Atomic Register

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

Events:
Request: ⟨*onar*, *Read*⟩: Invokes a read operation on the register.
Request: ⟨*onar*, *Write* | *v*⟩: Invokes a write operation with value *v* on the register.
Indication: ⟨*onar*, *ReadReturn* | *v*⟩: Completes a read operation on the register with return value *v*.
Indication: ⟨*onar*, *WriteReturn*⟩: Completes a write operation on the register.

Properties:
ONAR1: *termination*: If a correct process invokes an operation, then the operation eventually completes.
ONAR2: *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.
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*.

Question: What happens when there are multiple writers? Which properties are affected?

(N, N) Atomic Register

- We need to formulate an appropriate *validity* property.
- Issues:
  - If two processes have written different values *v* and *v'* concurrently before some other process invokes a read operation, then what should this read return?
  - Assuming it is possible for the reader to return either *v* or *v'*, do we allow a concurrent reader, or even a reader that comes later, to return the other value?
  - What about a failed write operation?
  - If a process writes a value, *v*, and crashes before completing the write, does a reader have to return *v* or can it return an older value?

(N, N) Atomic Register

- Idea: Link together reads and writes in a stricter way than previously.
  - Ensure that every failed write appears either as if it has never been invoked or as if it has completed.
  - A failed read may appear as it has never been invoked.
  - Even in the face of concurrency, the values returned by reads must be explainable in that they could have been returned by a hypothetical serial execution, where every operation takes place at an indivisible point time, which lies between moments of the invocation event and the completion event of this operation.
- An (N, N) AR is a strict generalization of a $(1, N)$ AR: every execution of a $(1, N)$ AR is also an execution of an (N, N) AR but not the other way around.

(N, N) Atomic Register

Module:

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

Events:

Request: ⟨*nnar*, *Read*⟩: Invokes a read operation on the register.
Request: ⟨*nnar*, *Write* | *v*⟩: Invokes a write operation with value *v* on the register.
Indication: ⟨*nnar*, *ReadReturn* | *v*⟩: Completes a read operation on the register with return value *v*.
Indication: ⟨*nnar*, *WriteReturn*⟩: Completes a write operation on the register.

Properties:

NNAR1: *termination*: If a correct process invokes an operation, then the operation eventually completes.
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.

(N, N) Atomic Register

- The hypothetical serial execution is called a *linearization* of the actual execution.
- More precisely, a ***linearization*** of an execution is 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 incomplete ones) and satisfies all of the following conditions:
  1. Every read returns the last value written.
  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 linearlize it.
- Given this, we can reformulate the *atomicity* property of (N, N) AR.

(N, N) Atomic Register

Module:

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

Events:

Request: ⟨*nnar*, *Read*⟩: Invokes a read operation on the register.
Request: ⟨*nnar*, *Write* | *v*⟩: Invokes a write operation with value *v* on the register.
Indication: ⟨*nnar*, *ReadReturn* | *v*⟩: Completes a read operation on the register with return value *v*.
Indication: ⟨*nnar*, *WriteReturn*⟩: Completes a write operation on the register.

Properties:

NNAR1: *termination*: If a correct process invokes an operation, then the operation eventually completes.
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.~~ ⇨ Every execution of the register is linearizable.

(N, N) Atomic Register

Question: How to transform a (1, N) register to a (N, N) register?

- Idea: Use an *N*-element vector of (1, N) registers, each having precisely one writer and coordinate writers to enforce a happened-before relation.

(N, N) Atomic Register

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

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

Uses:
(1, N)-AtomicRegister, **multiple instances**.

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

upon event ⟨*nnar*, *Write* | *v*⟩ **do**
  val := *v*;
  writing := TRUE;
  forall $q \in \Pi$ **do**
    trigger ⟨*onar.q*, *Read*⟩;

upon event ⟨*nnar*, *Read*⟩ **do**
  forall $q \in \Pi$ **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 := $[\perp]^N$;
    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*⟩;

Function *highest* returns the (timestamp, value) pair corresponding to a triple (timestamp, rank, value) in a collection that has the largest (timestamp, rank) pair among all triples in the collection (pair comparison is done lexicographically).

(N, N) Atomic Register

Correctness of the algorithm:

- Termination – follows from the same property of (1, N) Atomic Registers.
- Atomicity (we have to demonstrate that NNAR-Read and NNAR-Write operations are linearizable):
  - The algorithm uses a total order on (timestamp, rank, value) tuples: enforced by function *highest*.
  - From the algorithm, the timestamps written by two serial NNAR-Write operations are strictly increasing.
  - We can construct the linearlization of an arbitrary execution as follows:
    - Include all NNAR-Write operations according to the total order of their associated (timestamp, rank) pairs.
    - Consider NNAR-Read operations in the order in which their responses occur in the actual execution.
    - For each such operation, *or*, take the timestamp, *ts*, and the rank of the writer, *q*, associated with the value, *v*, returned.
    - Find the NNAR-Write operation, *ow*, during which process *q* ONAR-wrote (*ts*, *v*) to instance *onar.q*.
    - Place operation *or* after operation *ow* into the linearization immediately before the subsequent NNAR-Write operation.
  - The first condition of linearizability ("Every read returns the last value written") holds from the construction of the linearization: each read returns the value of the latest preceding write.

(N, N) Atomic Register

Correctness of the algorithm (cont.):

- Atomicity (cont.):
  - We have to show the second condition of linearizability ("For any two operations, *o* and *o'*, if *o* precedes *o'* in the actual execution, then *o* also appears before *o'* in the linearization").
  - To this end, consider two operations *o1* and *o2* in the actual execution such that *o1* precedes *o2*.
  - Four cases to consider:
    - *o1* and *o2* are both writes:
      - They are in the correct order, which is enforced by the (timestamp, rank) pairs.
    - *o1* is a read and *o2* is a write:
      - The algorithm for the write first reads the underlying (1, *N*) registers, selects the highest (timestamp, rank) pair, and increments the timestamp by one for writing.
      - Consequently, *o1* occurs before *o2* in the linearization according to its construction.
    - *o1* is a write and *o2* is a read:
      - The algorithm for the read first reads the underlying (1, *N*) registers, selects the highest (timestamp, rank) pair, and returns the associated value.
      - Therefore, *o2* returns a value associated with the timestamp generated by *o1* or by a later write.
      - The construction of linearization thus places *o2* after *o1*.

(N, N) Atomic Register

Correctness of the algorithm (cont.):

- Atomicity (cont.):
  - Four cases to consider (cont.):
    - *o1* and *o2* are both reads:
      - Suppose that *o1* selects (*ts1*, *r1*) as the highest (timestamp, rank) pair and returns the associated value, *v1*, whereas *o2* selects (*ts2*, *r2*) as the highest (timestamp, rank) pair.
      - Since *o2* occurs after *o1* in the actual execution and since any intermediate write does not decrease the timestamp value, we have *ts2* ≥ *ts1*.
      - If *ts2* > *ts1*, then o1 appears before o2 in the linearization according to its construction.
      - Otherwise (*ts2* = *t1* [*]), consider processes *p1* and *p2* with ranks *r1* and *r2*, respectively.
        - If *r1* < *r2*, then the write of *p1* is placed in the linearization before the write by *p2*, and hence also *o1* is placed before *o2* in the linearization.
        - If *r1* = *r2*, then also *o1* is placed before *o2* in the linearization, because according to its construction, reads are considered in the order their completion events are signaled in the actual execution and are placed immediately before the succeeding write.
        - Consider thus the last case, *r1* > *r2* [**].
          - When *o2* is invoked, the underlying (1, *N*) register instance, *onar.p1*, still contains the pair (*ts1*, *v1*) – [*].
          - Read *o2* would thus have selected (*ts1*, *r1*) as the highest (timestamp, rank) pair.
          - This, however, contradicts [**] because it would mean that *r1* = *r2*.
          - Therefore, this case is impossible.

(N, N) Atomic Register

Performance of the algorithm:

- Write: every NNAR-write requires N ONAR-reads and 1 ONAR-write.
- Read: every NNAR-read requires N ONAR-reads.

Depending on the underlying algorithm for ONAR, the end-to-end performance is thus as follows:

Read-Impose Write-All:

- Work:
  - Write: $2 \cdot (N^2 + N)$,
  - Read: $2 \cdot N^2$.
- Span:
  - Write: 4,
  - Read: 2.

Read-Impose Write-Majority:

- Work:
  - Write: $4 \cdot N^2 + 2 \cdot N$,
  - Read: $4 \cdot N^2$.
- Span:
  - Write: 6,
  - Read: 4.

(N, N) Atomic Register

Performance of the algorithm:

- Write: every NNAR-write requires N ONAR-reads and 1 ONAR-write.
- Read: every NNAR-read requires N ONAR-reads.

Depending on the underlying algorithm for ONAR, the end-to-end performance is thus as follows:

Read-Impose Write-All:

- Work:
  - Write: $2 \cdot (N^2 + N)$,
  - Read: $2 \cdot N^2$.
- Span:
  - Write: 4,
  - Read: 2.

Read-Impose Write-Majority:

- Work:
  - Write: $4 \cdot N^2 + 2 \cdot N$,
  - Read: $4 \cdot N^2$.
- Span:
  - Write: 6,
  - Read: 4.

Question: Can we improve on this with a monolithic algorithm?

(N, N) Atomic Register

Algorithm: Read-Impose Write-Consult-All

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

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

upon event ⟨*nnar*, *Init*⟩ **do**
  (*ts*, *wr*, *val*) := (0, 0, ⊥);
  correct := *Π*;
  writeset := ∅;
  readval := ⊥;
  reading := FALSE;

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

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'*) > (*ts*, *wr*) **then**
    (*ts*, *wr*, *val*) := (*ts'*, *wr'*, *v'*);
  trigger ⟨*pl*, *Send* | *p*, [Ack]⟩;

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

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

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

(N, N) Atomic Register

Correctness of the algorithm:

- Termination – follows from completeness of $\mathcal{P}$, validity of BEB, and reliable delivery of PL.
- Atomicity – exercise – like in the previous algorithm plus accuracy of $\mathcal{P}$.

(N, N) Atomic Register

Performance of the algorithm:

- The work of the algorithm is $2 \cdot N$ for both a write and a read.
- The span of the algorithm is 2 for both a write and a read.

(N, N) Atomic Register

Algorithm: Read-Impose Write-Consult-All

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

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

```
upon event ⟨nnar, Init⟩ do
   (ts, wr, val) := (0, 0, ⊥);
   correct := Π;
   writeset := ∅;
   readval := ⊥;
   reading := FALSE;

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

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') > (ts, wr) then
      (ts, wr, val) := (ts', wr', v');
   trigger ⟨pl, Send | p, [Ack]⟩;

upon event ⟨pl, Deliver | p, [Ack]⟩ do
   writeset := writeset ∪ { p };

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

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

Question: No failure detector?

(N, N) Atomic Register

```
Algorithm: Read-Impose Write-Consult-Majority (part 1)

Implements:
   (N, N)-AtomicRegister, instance nnar.

Uses:
   BestEfforBroadcast, instance beb.
   PerfectLinks, instance pl.

upon event ⟨nnar, Init⟩ do
   (ts, wr, val) := (0, 0, ⊥);
   acks := rid := 0;
   readlist := [⊥]^N;
   writeval := readval := ⊥;
   reading := FALSE;

upon event ⟨nnar, Read⟩ do
   rid := rid + 1;
   acks := 0;
   readlist := [⊥]^N;
   reading := TRUE;
   trigger ⟨beb, Broadcast | [Read, rid]⟩;

upon event ⟨beb, Deliver | p, [Read, r]⟩ do
   trigger ⟨pl, Send | p, [Value, r, ts, wr, val]⟩;

upon event ⟨pl, Deliver | q, [Value, r, ts', wr', v']⟩
      such that r = rid do
   readlist[q] := (ts', wr', v');
   if #(readlist) > N / 2 then
      (maxts, rr, readval) := highest(readlist);
      readlist := [⊥]^N;
      if reading = TRUE then
         trigger ⟨beb, Broadcast |
               [Write, rid, maxts, rr, readval]⟩;
      else
         trigger ⟨beb, Broadcast |
               [Write, rid, maxts + 1, rank(self), writeval]⟩;
```

(N, N) Atomic Register

Algorithm: Read-Impose Write-Consult-Majority (part 2)

upon event ⟨*nnar*, *Write* | *v*⟩ **do**
    rid := *rid* + 1;
    writeval := *v*;
    acks := 0;
    readlist := $[\perp]^N$;
    trigger ⟨*beb*, *Broadcast* | [Read, *rid*]⟩;

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

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

(N, N) Atomic Register

Correctness of the algorithm:

- Termination – follows from the correct majority assumption, validity of BEB, and reliable delivery of PL.
- Atomicity – exercise – like in the previous algorithms.

(N, N) Atomic Register

Performance of the algorithm:

- The work of the algorithm is $4 \cdot N$ for both a write and a read.
- The span of the algorithm is 4 for both a write and a read.

Considered Failure Model

Crash-recovery failures

Precedence Revisited

- A process accesses every register in a sequential manner, which in the crash-stop model implies strictly alternating between invocation events and completion events.
- It may happen that a process crashes:
  - right after a register implementation has triggered a completion event of an operation but
  - before the invoking higher-level modules have reacted to this event.
- When the process recovers:
  - from the perspective of the register implementation, the operation is complete but
  - from the perspective of the invoking higher-level software, it is not.
- We thus revisit the notion of *precedence* to cover also incomplete operations.

Precedence Revisited

Was:

- An operation, o, is said to ***precede*** another operation, o', iff the completion event of o occurs before the invocation event of o'.

Is:

- An operation, o, is said to ***precede*** another operation, o', iff any of the two following conditions hold:
  1. the completion event of o occurs before the invocation event of o' or
  2. the operations are invoked by the same process and the invocation event of o' occurs after the invocation event of o.

(1, N) Logged Regular Register

Module:

Name: (1, N)-LoggedRegularRegister, **instance** *lonrr*.

Events:

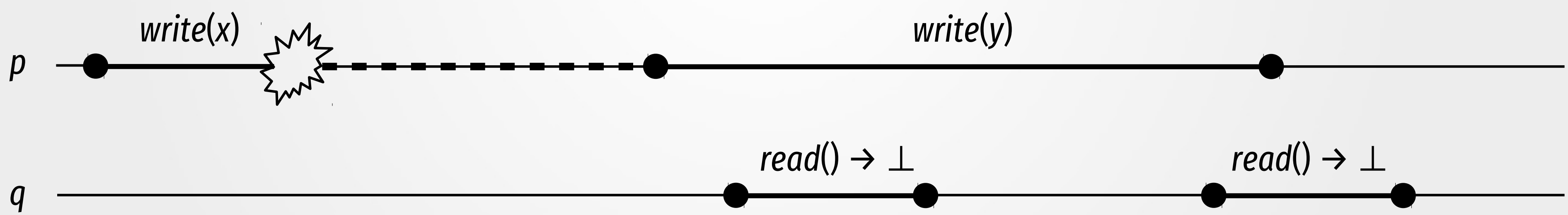
Request: ⟨*lonrr*, *Read*⟩: Invokes a read operation on the register.
Request: ⟨*lonrr*, *Write* | *v*⟩: Invokes a write operation with value *v* on the register.
Indication: ⟨*lonrr*, *ReadReturn* | *v*⟩: Completes a read operation on the register with return value *v*.
Indication: ⟨*lonrr*, *WriteReturn*⟩: Completes a write operation on the register.

Properties:

LONRR1: *termination*: If a correct process invokes an operation and never crashes afterward, then the operation eventually completes.
LONRR2: *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.

(1, N) Logged Regular Register

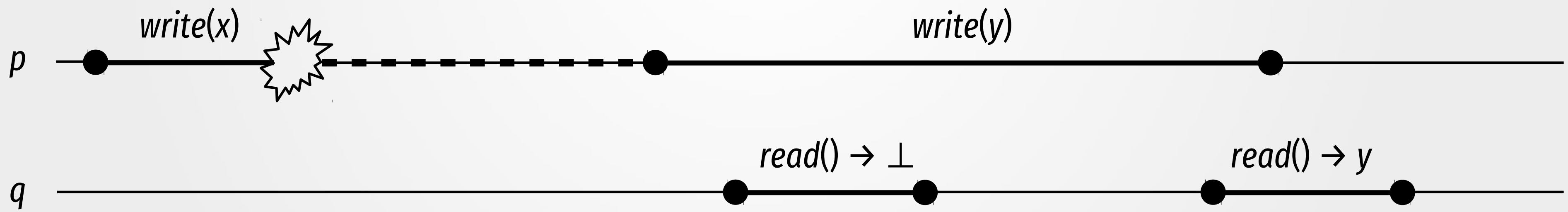
- Process *p* crashes before completing *write*(*x*) and after recovery invokes *write*(*y*).
- The first read by process *q* occurs after invoking write(y), the second – before completing it.



Question: Is the above execution possible?

(1, N) Logged Regular Register

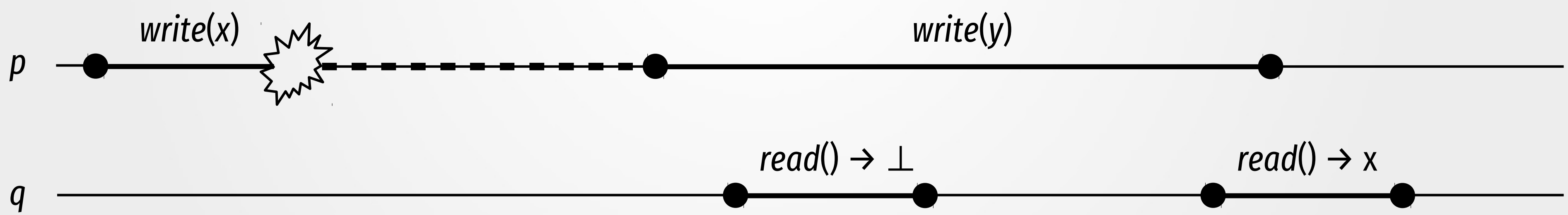
- Process *p* crashes before completing *write*(*x*) and after recovery invokes *write*(*y*).
- The first read by process *q* occurs after invoking write(y), the second – before completing it.



Question: What about this one?

(1, N) Logged Regular Register

- Process *p* crashes before completing *write*(*x*) and after recovery invokes *write*(*y*).
- The first read by process *q* occurs after invoking write(y), the second – before completing it.



Question: And this one?

(1, N) Logged Regular Register

Module:
Name: (1, N)-LoggedRegularRegister, **instance** *lonrr*.

Events:
Request: ⟨*lonrr*, *Read*⟩: Invokes a read operation on the register.
Request: ⟨*lonrr*, *Write* | *v*⟩: Invokes a write operation with value *v* on the register.
Indication: ⟨*lonrr*, *ReadReturn* | *v*⟩: Completes a read operation on the register with return value *v*.
Indication: ⟨*lonrr*, *WriteReturn*⟩: Completes a write operation on the register.

Properties:
LONRR1: *termination*: If a correct process invokes an operation and never crashes afterward, then the operation eventually completes.
LONRR2: *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.

Question: Why not through a log, like in reliable broadcast?

(1, N) Logged Regular Register

- In communication abstractions, the receiver could not anticipate if and how many messages it will deliver.
- Here, both indication events occur only if the process has invoked the corresponding request.
- (1, N) register operations are ***idempotent***:
  - Executing the same operation multiple times in succession has the same effect as executing it once.
- Even if an indication event is lost, the operation can be restarted without any harm.

(1, N) Logged Regular Register

Algorithm: Logged Majority Voting (part 1)

Implements:
(1, N)-LoggedRegularRegister, **instance** *lonrr*.

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

```
upon event ⟨lonrr, Init⟩ do
   (ts, val) := (0, ⊥);
   wts := rid := 0;
   acklist := readlist := [⊥]^N;  writing := reading := FALSE;
   store(wts, ts, val, rid, writing);

upon event ⟨lonrr, Recovery⟩ do
   retrieve(wts, ts, val, rid, writing);
   acklist := readlist := [⊥]^N;  reading := FALSE;
   if writing = TRUE then
      rid := rid + 1;
      trigger ⟨sbeb, Broadcast | [Write, rid, ts, val]⟩;

upon event ⟨lonrr, Write | v⟩ do
   wts := wts + 1;
   (ts, val) := (wts, v);
   rid := rid + 1;
   acklist := [⊥]^N;  writing := TRUE;
   store(wts, ts, val, rid, writing);
   trigger ⟨sbeb, Broadcast | [Write, rid, ts, val]⟩;

upon event ⟨sbeb, Deliver | p, [Write, r, ts', v']⟩ do
   if ts' > ts then
      (ts, val) := (ts', v');
      store(ts, val);
   trigger ⟨sl, Send | p, [Ack, r]⟩;

upon event ⟨sl, Deliver | q, [Ack, r]⟩ such that r = rid do
   acklist[q] := Ack;
   if #(acklist) > N / 2 ∧ writing = TRUE then
      acklist := [⊥]^N;  writing := FALSE;
      store(writing);
      trigger ⟨lonrr, WriteReturn⟩;
```

(1, N) Logged Regular Register

Algorithm: Logged Majority Voting (part 2)

upon event ⟨*lonrr*, *Read*⟩ **do**
  $rid := rid + 1$;
  $readlist := [\perp]^N$; $reading :=$ TRUE;
  trigger ⟨*sbeb*, *Broadcast* | [Read, *rid*]⟩;

upon event ⟨*sbeb*, *Deliver* | *p*, [Read, *r*]⟩ **do**
  trigger ⟨*sl*, *Send* | *p*, [Value, *r*, *ts*, *val*]⟩;

upon event ⟨*sl*, *Deliver* | *q*, [Value, *r*, *ts'*, *v'*]⟩
    such that $r = rid$ **do**
  $readlist[q] := (ts', v')$;
  if $\#(readlist) > N / 2 \land reading =$ TRUE **then**
    $v := highestval(readlist)$;
    $readlist := [\perp]^N$; $reading :=$ FALSE;
    trigger ⟨*lonrr*, *ReadReturn* | *v*⟩;

Function *highestval* returns the value with the largest timestamp.

(1, N) Logged Regular Register

Correctness of the algorithm – Cheatsheet:

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*.

(1, N) Logged Regular Register

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*.

(1, N) Logged Regular Register

Correctness of the algorithm:

- Termination (like in the Majority Voting Regular Register) – follows from the properties of SBEB (best-effort validity) and SL (stubborn delivery), and the assumption that a majority of the processes are correct.
- Validity (also like in the Majority Voting Regular Register):
  - Consider a read operation, invoked by some process *q*, that is not concurrent with any write:
    - Assume that the last value written by the writer, process *p*, is *v* and the associated timestamp is *wts*.
    - Because p logs every timestamp and increments the timestamp for every write, at the moment when the read is invoked, some majority of processes, *C*, have logged *wts* and *v* in their variables *ts* and *val*, respectively, and that, due to the algorithm and no creation of SBEB and SL, there is no larger timestamp than *wts* in the system.
    - Before returning from the read operation, process *q* consults some majority of processes, *S*.
    - Since $C \cap S \neq \emptyset$, at least one process in *S* has its timestamp and value equal *wts* and *v*, respectively.
    - Given the definition of function *highestval*, process *q* returns *v* as the read return value.

(1, N) Logged Regular Register

Correctness of the algorithm (cont.):

- Validity (cont.):
  - Consider the case when a read by some process, *q*, is concurrent with some write of value *v* with timestamp *wts*, whereas the last written value was *v′* and its timestamp was equal to *wts′* (where *wts′* = *wts* - 1):
    - If process *p* crashed during the previous write before it logged *v′*, then no other process ever sees *v′*, and thus *v′* cannot be the last written value.
    - Therefore, suppose that either *p* logged *v′* and *wts′* during the previous write or that *v′* is the initial value $\perp$.
      - In the first case, the write of *v′* is eventually completed because in particular upon recovery, process *p* reattempts it.
      - In the second case, every correct process eventually stores *v′* and *wts′* = 0 as a result of its initialization.
    - In any case, if any process returns the pair (*wts*, *v*) to the reader, process *q*, then *q* returns *v*, which is a valid result, because *wts* is the largest timestamp in the system (no creation of SBEB and SL).
    - Otherwise, at least one reply from more a majority of processes is (*wts′*, *v′*), and thus q returns *v′*, which is also a valid reply.

(1, N) Logged Regular Register

Performance of the algorithm:

- The work of the algorithm is $2 \cdot N$ for both a read and a write.
- The span of the algorithm is 2 for both a read and a write.

Plus up to 3 log operations on each write.

Other Logged Registers

- The communication pattern of the Logged Majority Voting algorithm is similar to the Majority Voting Regular Register for the crash-stop model.
- In the same way, one can port the Read-Impose Write-Majority and Read-Impose Write-Consult-Majority algorithms to the crash-recovery model, thereby obtaining (1, N) Logged Atomic and (N, N) Logged Atomic registers.

Summary

We have:

- introduced the various semantics of fault-tolerant registers;
- presented algorithms implementing them under various failure types and extra assumptions;
- discussed the notion of *linearization* and its application to proving algorithm correctness;
- introduced the notion of system *quiescence*.

Digression

Fault-tolerance and concurrency can make implementing even seemingly simple functionality a real challenge.

Digression

Be explicit

Get all of the assumptions out on the table.

Next Lecture

- Will be about distributed algorithms for the third of the three fundamental abstractions: interpreters.
- More specifically, we will discuss the so-called consensus problem.