# Distributed algorithms in the shared memory model (part 2)

.

Emmanuelle Anceaume

emmanuelle.anceaume@irisa.fr

Slides : http://people.irisa.fr/Emmanuelle.Anceaume/

# Fault tolerant simulations of read/write objects

We have seen how we can build high level shared objects by using low level ones:

By relying on mutual exclusion section, where objects are updated in critical sections

Unfortunately those constructions are very sensitive to delays or failures

If some process blocks in the CS, then it blocks all the other processes which are waiting for the CS

What we would like is to build/simulate high level shared objects from low level ones without using locks

# Constructing atomic registers from safe ones

**Theorem**:

A multivalued MWMR atomic register can be wait-freely implemented with binary SRSW safe registers

Wait-free: any processor that is ready to write() or read() must do it without waiting for the other processors.

# Constructing atomic registers from safe ones

Assumptions:

n processors, $p_1,\dots p_i$, $p_j$, $\dots p_n$

Notations:

- the operations of the base registers:

  read() and write()

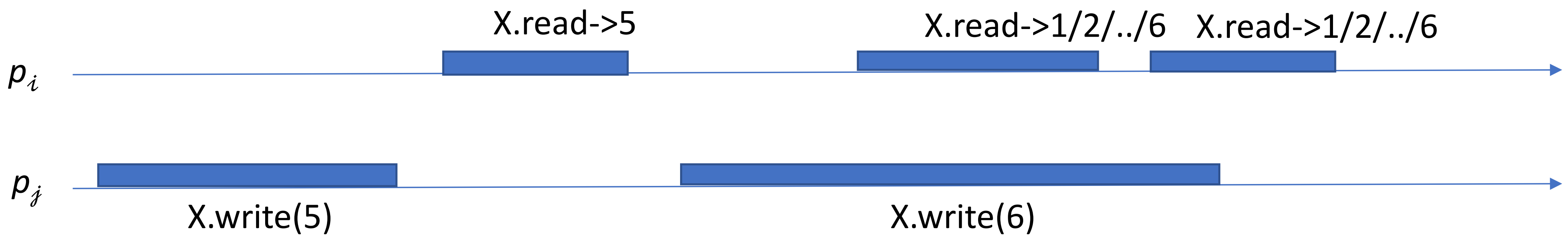
- the operations to be implemented:

  Read() and Write()

# Read/Write safe register

Properties:

- a read() not concurrent with any write() obtains the correct value, i.e., the most recently written one
- a read() that overlaps a write() returns any possible values of the register

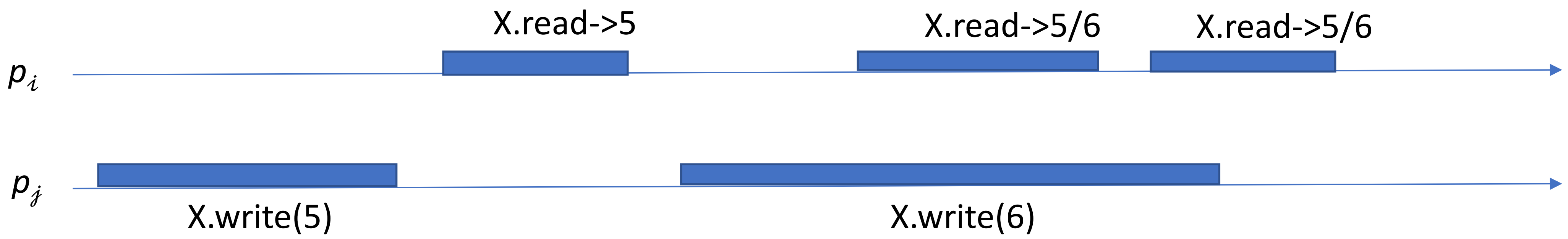



[1,..6]-valued safe register X

# Regular read/write register

Properties:

- a read() not concurrent with any write() obtains the correct value, i.e., the most recently written one
- a read() that overlaps a write() returns either the old or the new written value

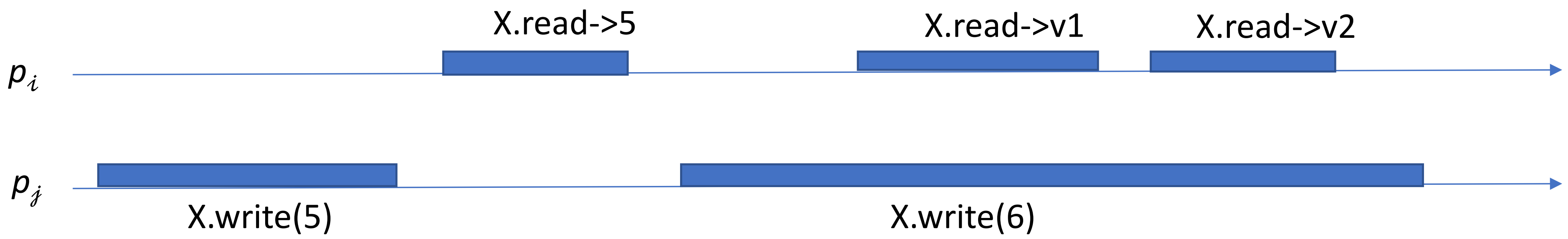



[1,..6]-valued regular register X

# Atomic read/write register

Properties:

- A read() not concurrent with any write() obtains the correct value, i.e., the most recently written one
- Concurrent reads() and writes() behave as if they occur in some definite order, i.e., as if reads() and writes() occur sequentially




v1=5 and v2=5 reflect a correct execution
v1=5 and v2=6 reflect a correct execution
V1=6 and v2=5 reflect an incorrect execution

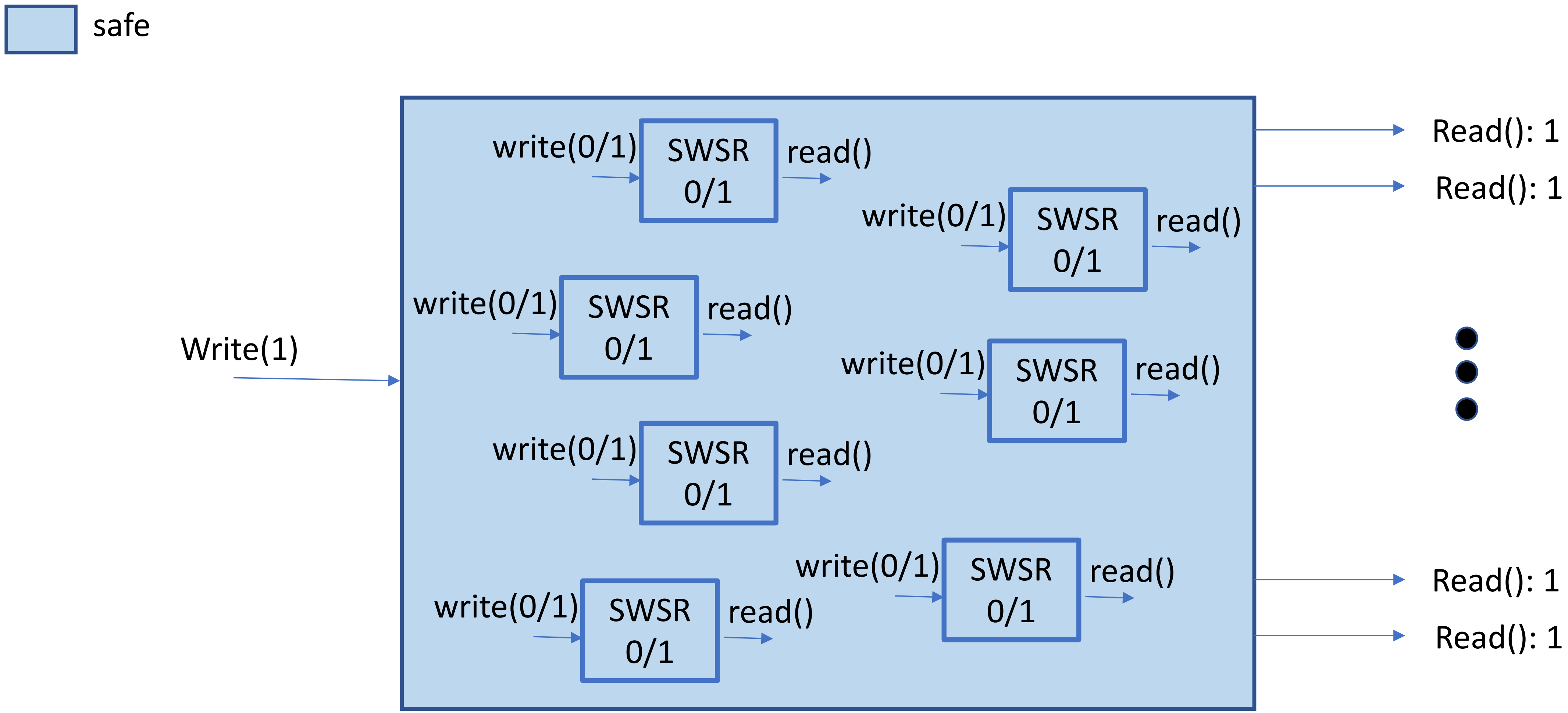
[1,..6]-valued atomic register X

# Constructing atomic from safe registers

Binary SWSR safe register

- Can easily be constructed
- The writer set a voltage level either high or low and the reader tests the level of voltage
- This is safe: setting a level high when it is already high can cause a perturbation of the level

# Construction 1:
# binary SWMR safe from binary SWSR safe registers



# Construction 1:
# binary SWMR safe from binary SWSR safe registers

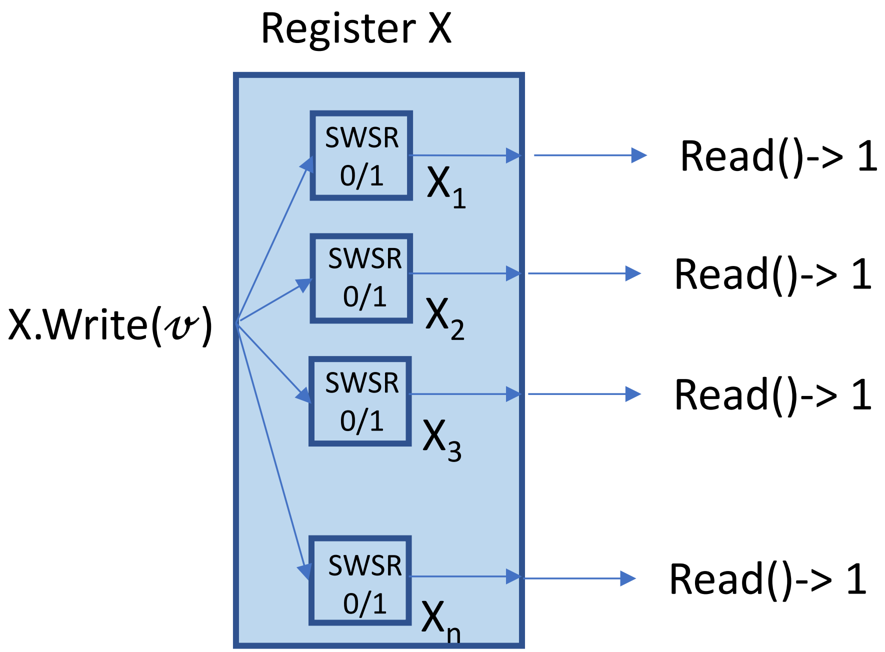
X: binary SWMR safe register we want to build

Let $X_1, \dots X_n$ be $n$ binary SWSR safe registers

The writer maintains a copy of $X_1, \dots X_n$ for each reader

```
when p invokes X.Write(v):
        for each i in {1, .., n}
                Xi.write(v)

when pi invokes X.Read():
        return  Xi.Read()
```




Note that each binary SWSR safe register has the same size (bits number) as the register we want to build (i.e., 1 bit)

# Construction 1:
# binary SWMR safe from binary SWSR safe registers

If the $X_i$ are binary SWSR safe registers, then X is a binary safe SWMR register

- Any Read() by $p_i$ that does not overlap a Write() does not overlap a write() of $X_i$
  - Thus if $X_i$ is safe, then this read() gets the correct value, and thus Read() gets the correct value.
- Any Read() by $p_i$ that overlaps a Write() may overlap a write() of $X_i$
  - Thus if $X_i$ is safe, then this read() may get 0/1 , and thus Read() gets 0/1
  - This satisfies the definition of safe registers

# Construction 1:
# binary SWMR safe from binary SWMR safe registers

1. Construction 1 allows us to build a binary SWMR regular register from binary SWSR regular registers

2. Construction 1 does not allow us to build binar SWMR atomic register from binary SWSR atomic ones

Construction 1 allows us to build a binary SWMR regular register from binary SWSR regular registers

1. Recall that with a regular register, any read concurrent with a write may return either the value of the concurrent write or the value of the last terminated write
2. We only consider binary registers, i.e., 0/1 values

Any Read() by $p_i$ that does not overlap a Write() does not overlap a write() of $X_i$

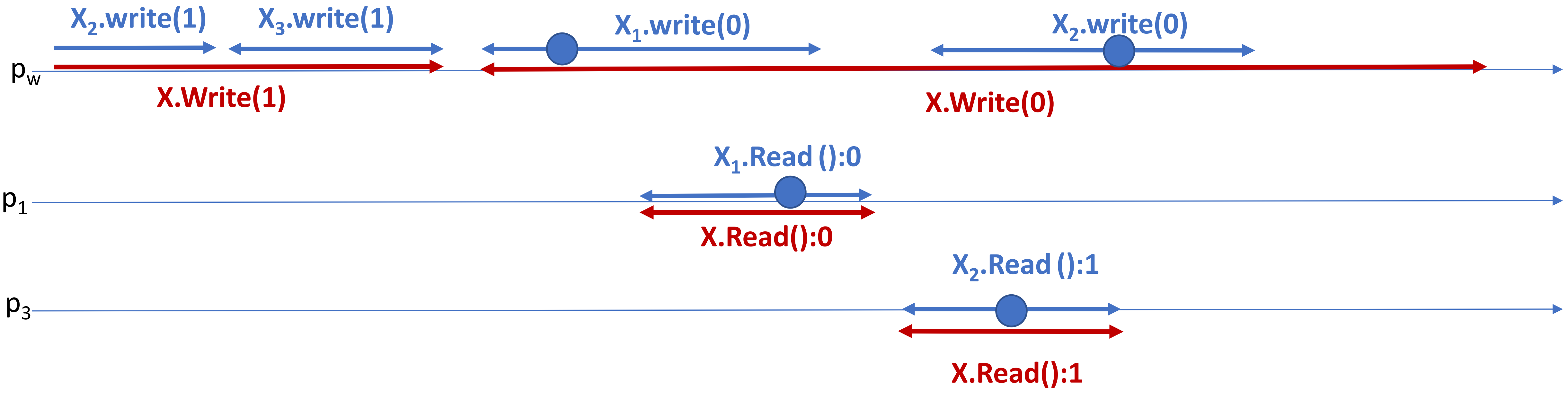
- Thus if $X_i$ is regular, then the read() gets the correct value, and thus Read() returns the correct value.

Any Read() by $p_i$ that overlaps a Write() may overlap a write() of $X_i$

- 1. overlap: Thus if $X_i$ is regular, then the read() may get either the concurrent or the preceding written value, and thus the Read() will return either the concurrent or the preceding written value
- 2. no overlap: the read() return the last preceding written value. So does the Read()

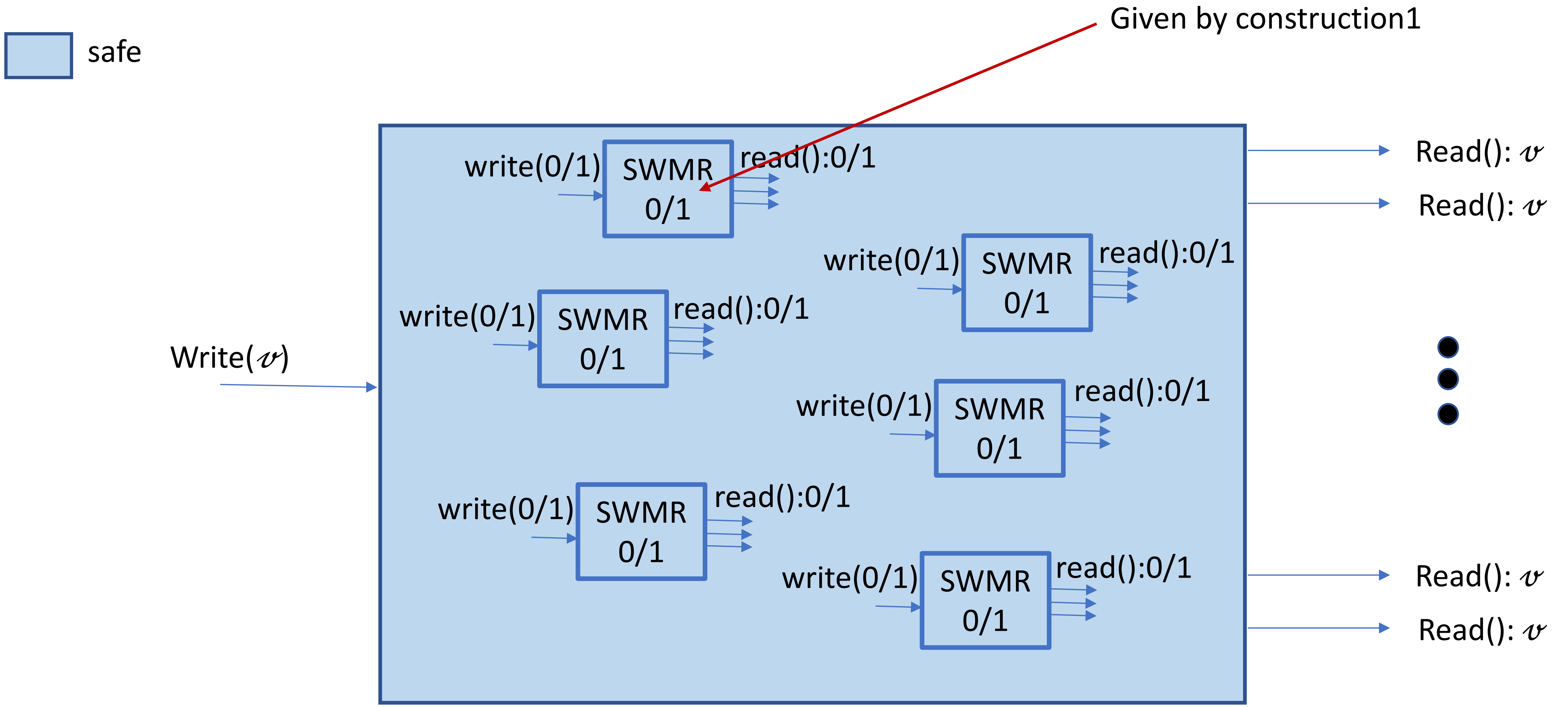
## Construction 1 does not allow us to build a binary SWMR atomic register from binary SWSR atomic registers

1. Recall that an atomic register is a regular register with no new/old inversion: that is a read operation cannot return a value which is older than the value returned by a previous read




Suppose that initially X contains the value 1 (i.e. $X_1$ = 1 and $X_2$ = 1)

# Construction 2:
# multivalued SWMR safe from binary SWMR safe registers



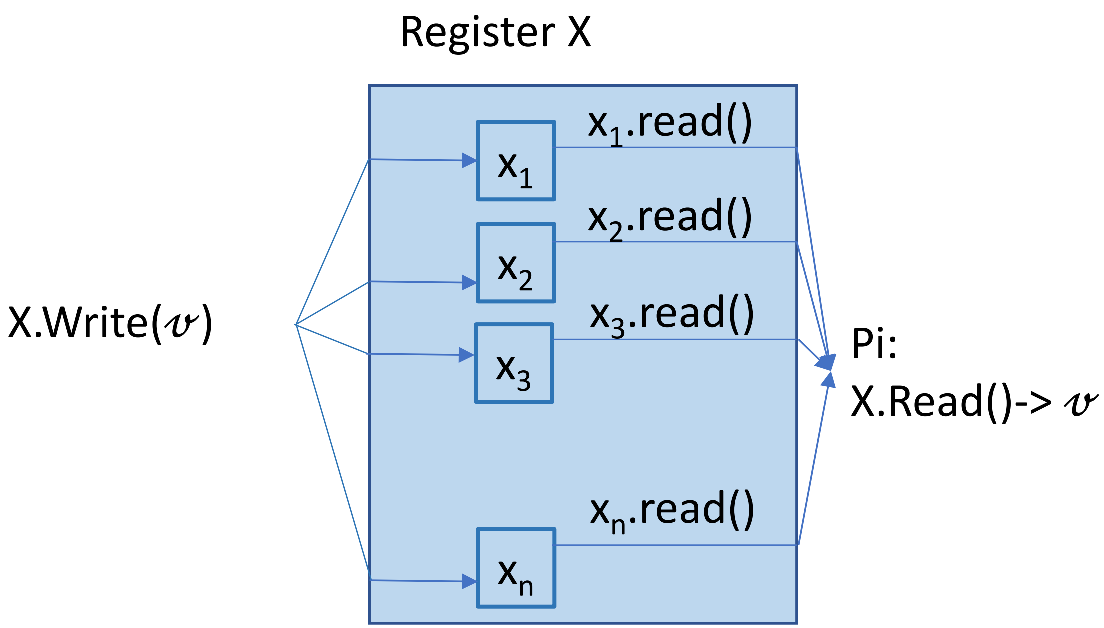
# Construction 2: multivalued SWMR safe from binary SWMR safe registers

X: r-valued-SWMR safe register we want to build

Let $X_1, \ldots X_n$ be $n$ binary MRSW safe registers, where n is the smallest integer such that $r \leq 2^n$

when p invokes X.Write($v = v_1\ v_2\ v_3 \ldots v_n$):
    for each $i$ in {1, .., n}
        $X_i$.write($v_i$)

when $p_i$ invokes X.Read():
    for each $i$ in {1, .., n}
        $x_i = X_i$.Read()
    return $x_1\ x_2\ x_3 \ldots x_n$



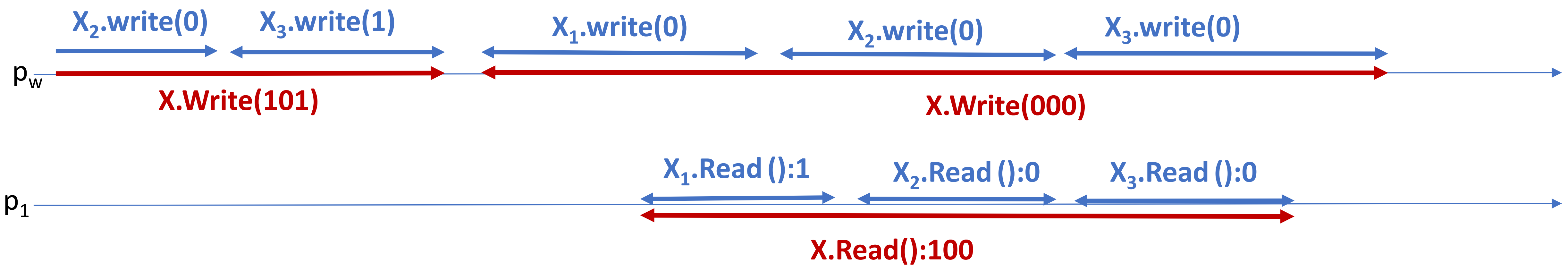

Note that the size of register we want to build is log(n) bits while each base register is a 1 bit size

# Construction 2: multivalued SWMR safe from binary SWMR safe registers

If the $X_i$ are safe SWMR binary registers, then X is a safe SWMR multivalued register

- Any Read() by $p_i$ that does not overlap a Write() does not overlap a write() of $X_i$
  - Thus if $X_i$ is safe, then this read() gets the correct value, and thus Read() gets the correct value.
- Any Read() by $p_i$ that overlaps a Write() may overlap the writes() of $X_1$, $X_2$,…$X_n$
  - Thus if $X_1$, $X_2$,…X are safe, then each of those reads() may get 0/1 , and thus Read() return any binary string from 0…0 to 1…1 which belong to the domain value of X
  - This satisfies the definition of safe registers

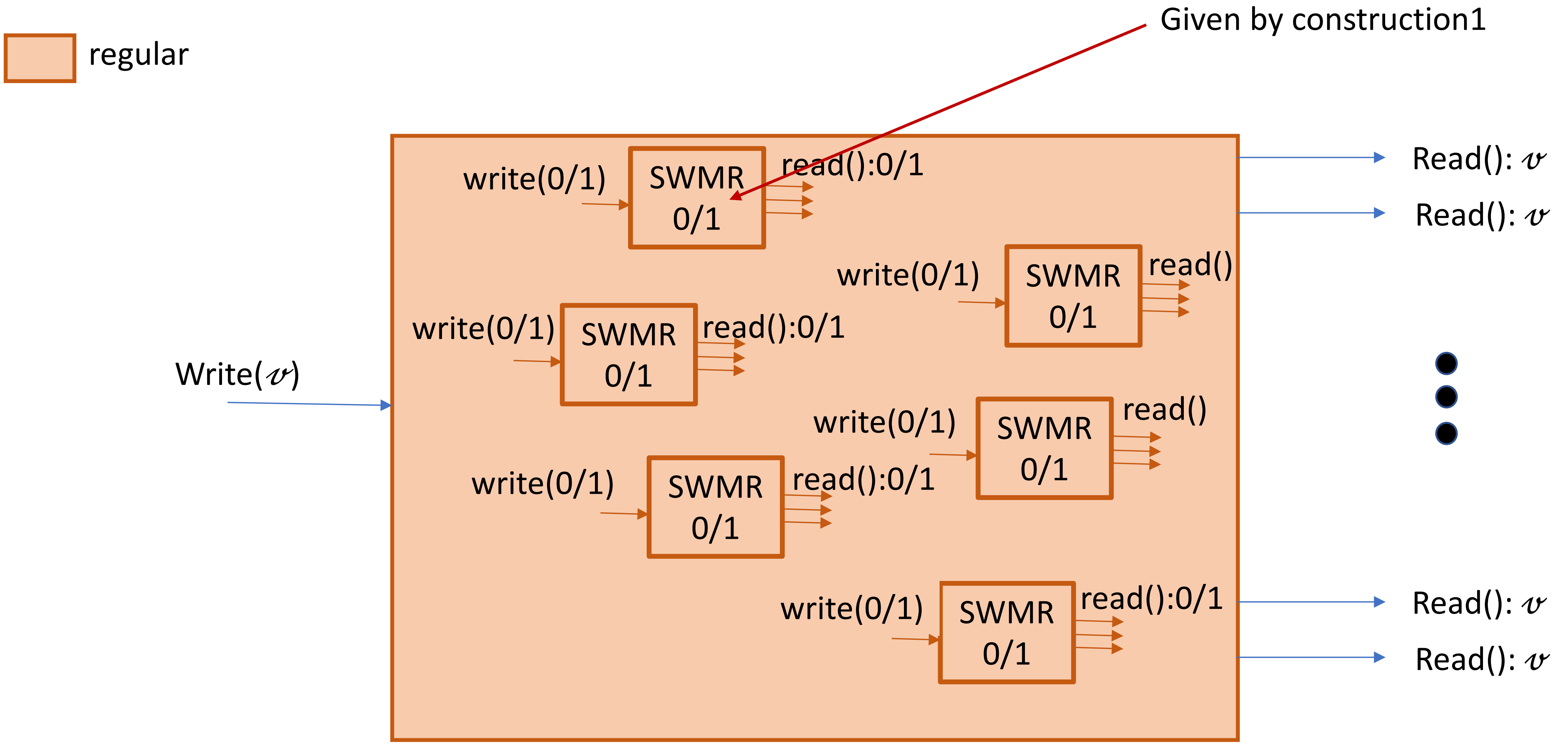
Construction 2 cannot implement a multivalued SWMR regular register even if the n binary SWMR registers are regular




Any value between 000 and 101 can be returned !

Thus construction 2 cannot implement a multivalued SWMR atomic register even if the n binary SWMR registers are atomic

# Construction 3: multivalued SWMR regular from binary SWMR regular registers



# Construction 3:
# multivalued SWMR regular from binary SWMR regular registers

- The construction employs a unary-encoding:
  value $v$ is denoted by "0" in bits 0 through $v$-1 and "1" in bit $v$

- The construction uses n binary SWMR regular registers to code n distinct values
- The idea is to write in one direction and to read in the opposite direction

  - To write value $v$, the writer first sets bit $x_v$ to "1" and
    then sets bits $x_{v-1}, x_{v-2}, \ldots, x_0$ to "0"

  - A reader reads the bits from left to right ($x_0$, ,…, $x_{v-2}$, $x_{v-1}$) until it finds a "1".

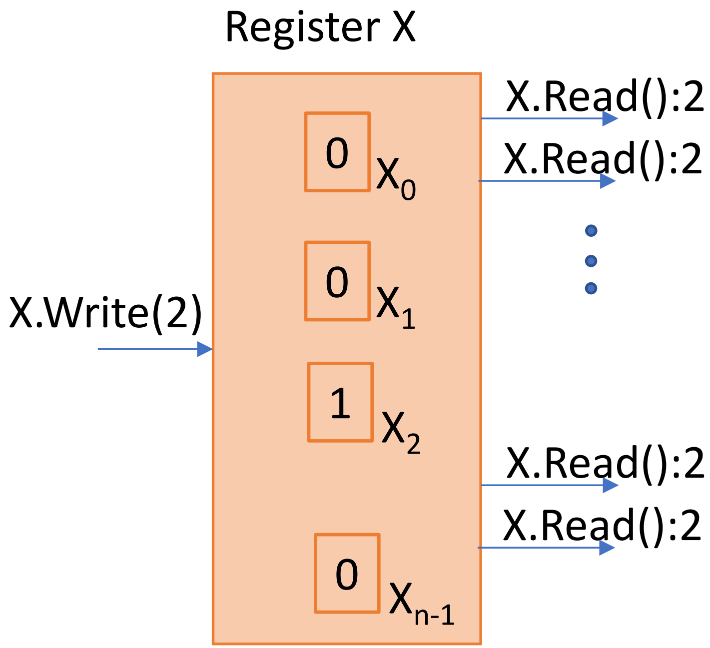
# Construction 3:
# multivalued SWMR regular from binary SWMR regular registers

X: n-SWMR regular register we want to build

Let $X_0, \ldots X_{n-1}$ be *n* binary SWMR regular registers

```
when p invokes X.Write(v):
    X_v.write(1)
    for each i in {v-1, .., 0}
       X_i.write(0)

when p_i invokes X.Read():
    j:=0
    while (X_j.Read() ≠ 1) do
         j:=j+1
    return j
```




Note that the size of register we want to build is n bits while each base register is a 1 bit size

# Construction 3:
# multivalued SWMR regular from binary SWMR regular registers

Q: Does the while stops ?

Yes: a "0" is written only if a "1" is written to its right

Q: would it be OK if "0s" were written first?

No: the while loop could never stop

Q: Is it true that when a reader reads a "1" then

- this "1" has been written either by a concurrent write()
- or by the preceding write()?

Yes: base registers are regular

```
when p invokes X.Write(v):
    X_v.write(1)
    for each i in {v-1, .., 0}
        X_i.write(0)

when p_i invokes X.Read():
    j:=0
    while (X_j.Read() ≠ 1) do
        j:=j+1
    return j
```

## Construction 3 implements a n-values SWMR regular register from n SWMR binary ones

Proof

1. Consider a read() op not concurrent with a write() op.

- When X.write($v$) terminates, the first base register equal to 1 is $X_v$ *(i.e.,* $X_j = 0$ *for* $0 \leq j \leq v-1$)
- As a read scans base registers starting from $X_0$, *then* $X_1$*….it necessarily reads until* $X_v$, *and returns* $v$, the correct value

```
when p invokes X.Write(v):
    X_v.write(1)
    for each i in {v-1, .., 0}
        X_i.write(0)

when p_i invokes X.Read():
    j:=0
    while (X_j.Read() ≠ 1) do
        j:=j+1
    return j
```

## Construction 3 implements a n-values SWMR regular register from n SWMR binary ones

Proof

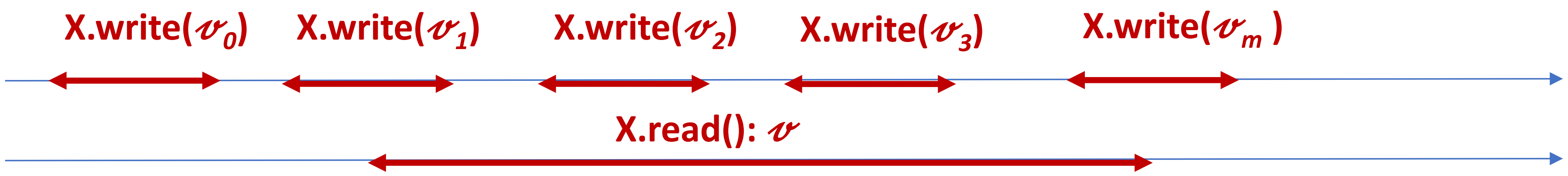
2. Consider now that X.Read() is concurrent with one or more X.Write() operations X.Write($v_1$), …, X.Write($v_m$), and let $v_0$ be the last written value terminated before X.Read()

- As read() op terminates, the number of concurrent write is finite
- We need to show that the value $v$ returned by X.Read() is either $v_0, v_1, \ldots, v_m$.

*1.* $v < v_0$

1. X.write($v_0$) has set to "1" base register $X_{v0}$ *and to* "0" all the base registers $X_{v0-1, \ldots} X_0$
2. By assumption, base registers are regular, a write from "0" to "0" can only return "0"

- Combining 1. and 2.: if X.read() returns a value $v$ smaller than $v_0$, it means that the value has necessarily be written by a concurrent write ($v_1, \ldots, v_m$), and thus X.read() satisfies the regularity property

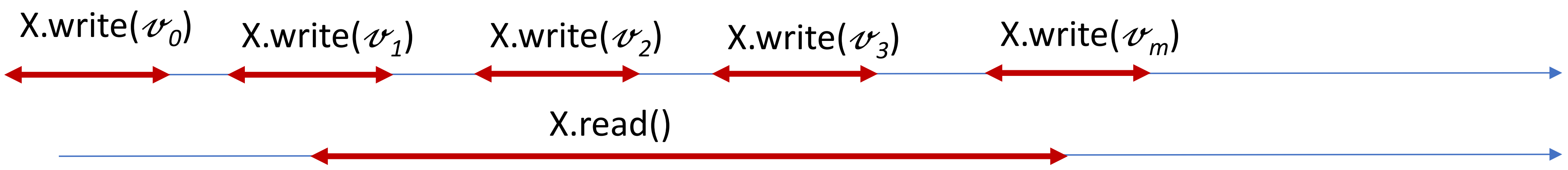


## Construction 3 implements a n-values SWMR regular register from n SWMR binary ones

Proof

- We need to show that the value $v$ returned by X.read() is either $v_0$, $v_1$, …, $v_m$.

2. $v = v_0$
   - X.read() trivially satisfies the regularity property



# Construction 3:
# multivalued SWMR regular from binary SWMR regular registers

## Construction 3 implements a n-valued SWMR regular register from n SWMR binary ones

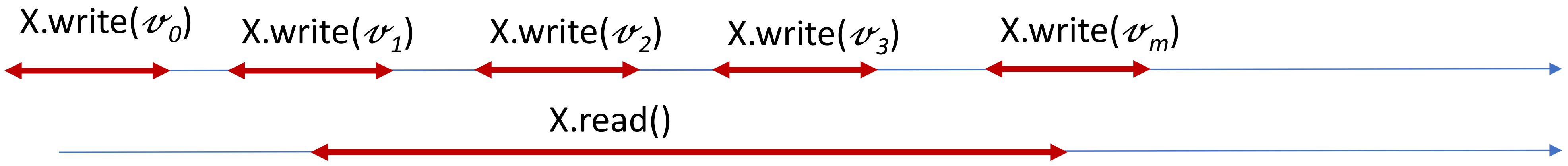
Proof

- We need to show that the value $v$ returned by X.read() is either $v_0$, $v_1$, or $v_m$.

3. $v > v_0$
   - $X_{v0}$.read() returns 0 (since $v > v_0$)
   - Let X.write($v'$) issued after X.write($v_0$) and concurrent with X.read() (by assumption X.write($v_0$) is the last terminated op before X.read())
     We have $X_{v'} = 1$ and $X_{v'-1}, \ldots, X_0 = 0$. 3 cases need to be considered
     1. $v = v'$: regularity trivially satisfied as X.write($v'$) and X.read() are concurrent operations
     2. $v_0 < v < v'$
        As $X_v$ was set to "0" by X.write($v'$), there must have a X.write($v$) issued after X.write($v'$) that updated $X_v$ from "0" to "1". The value returned by X.read() is consequently a value written by a concurrent write op. Thus regularity is satisfied

```
when p invokes X.Write(v):
    X_v.write(1)
    for each i in {v-1, .., 0}
        X_i.write(0)

when p_i invokes X.Read():
    j:=0
    while (X_j.Read() ≠ 1) do
        j:=j+1
    return j
```



# Construction 3:
# multivalued SWMR regular from binary SWMR regular registers

## Construction 3 implements a n-valued SWMR regular register from n SWMR binary ones

Proof

- We need to show that the value $v$ returned by X.read() is either $v_0$, $v_1$, or $v_m$.

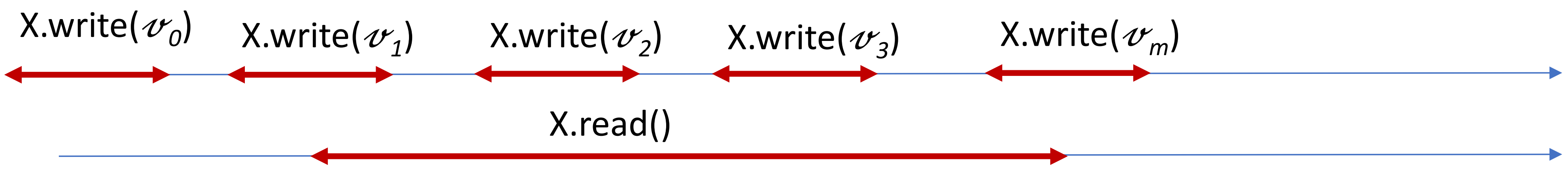
*3.* $v > v_0$

*3.* $v_0 < v' < v$

*Thus it means that X.read() missed the value 1 in $X_{v'}$.*
*This can only be due to a X.write($v''$) operation issued after X.write($v'$) and concurrent with X.read() such that $v'' > v'$. By the write op, we have $Xv'' = 1$ and all the registers $X_{v''-1}, \ldots, X_{v'}$ are set to 0.*
*We are in the same situation as above (case 3.2), where we replace $v_0$ by $v'$ and X.write($v'$) by X.write($v''$). As the number of values between $v'$ is finite and the read operations terminate, then it terminates either in case 3.1 or in case 3.2*

when p invokes X.Write($v$):
 $X_v$.write(1)
 for each $i$ in {$v$-1, .., 0}
  $X_i$.write(0)

when $p_i$ invokes X.Read():
 $j$:=0
 while ($X_j$.Read() ≠ 1) do
  $j$:=$j$+1
 return $j$

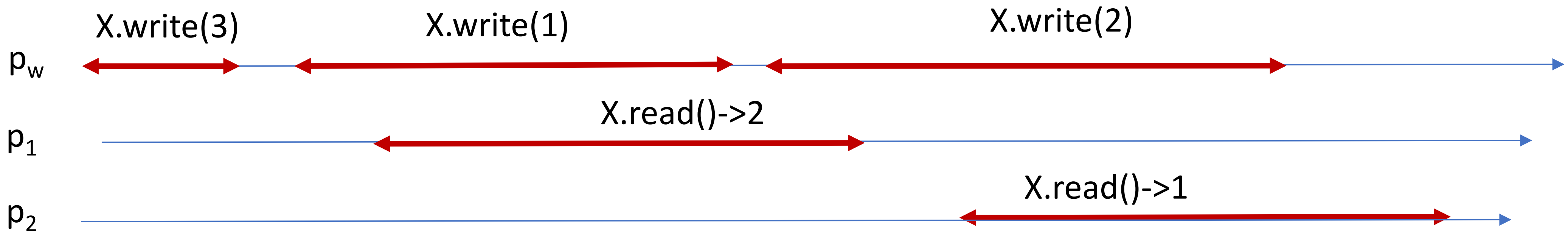


# Construction 3:
# multivalued SWMR regular from binary SWMR regular registers

**Construction 3 does not implement a multivalued SWMR atomic register from binary atomic SWMR ones**

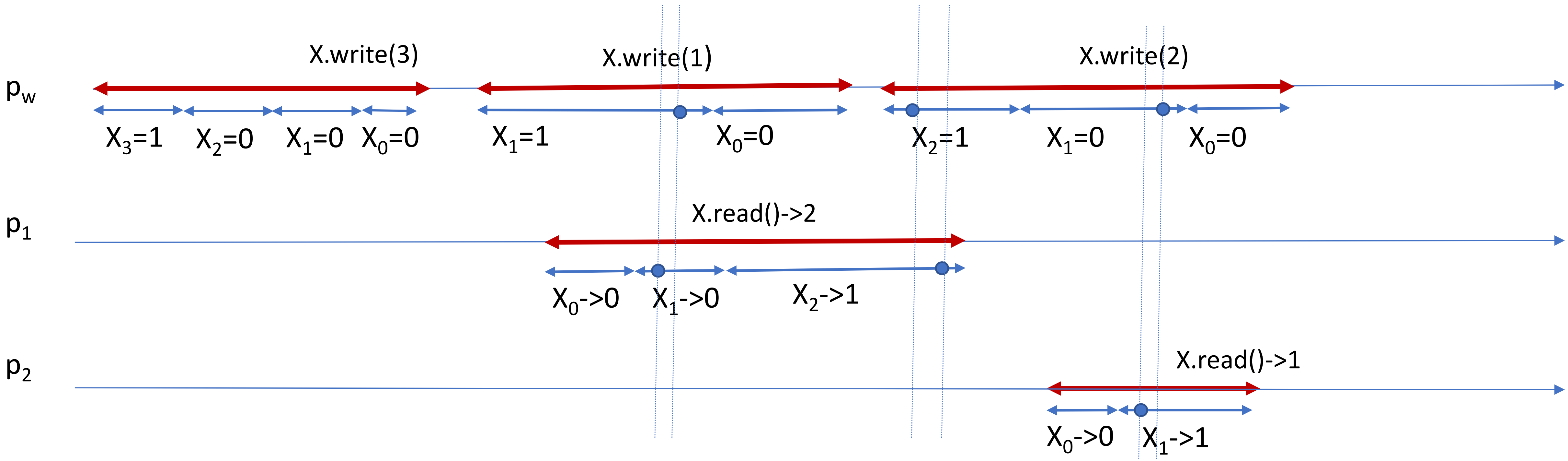
Proof

Counter-example




We show that such an new/old inversion execution is feasible

# Construction 3:
# multivalued SWMR regular from binary SWMR regular registers

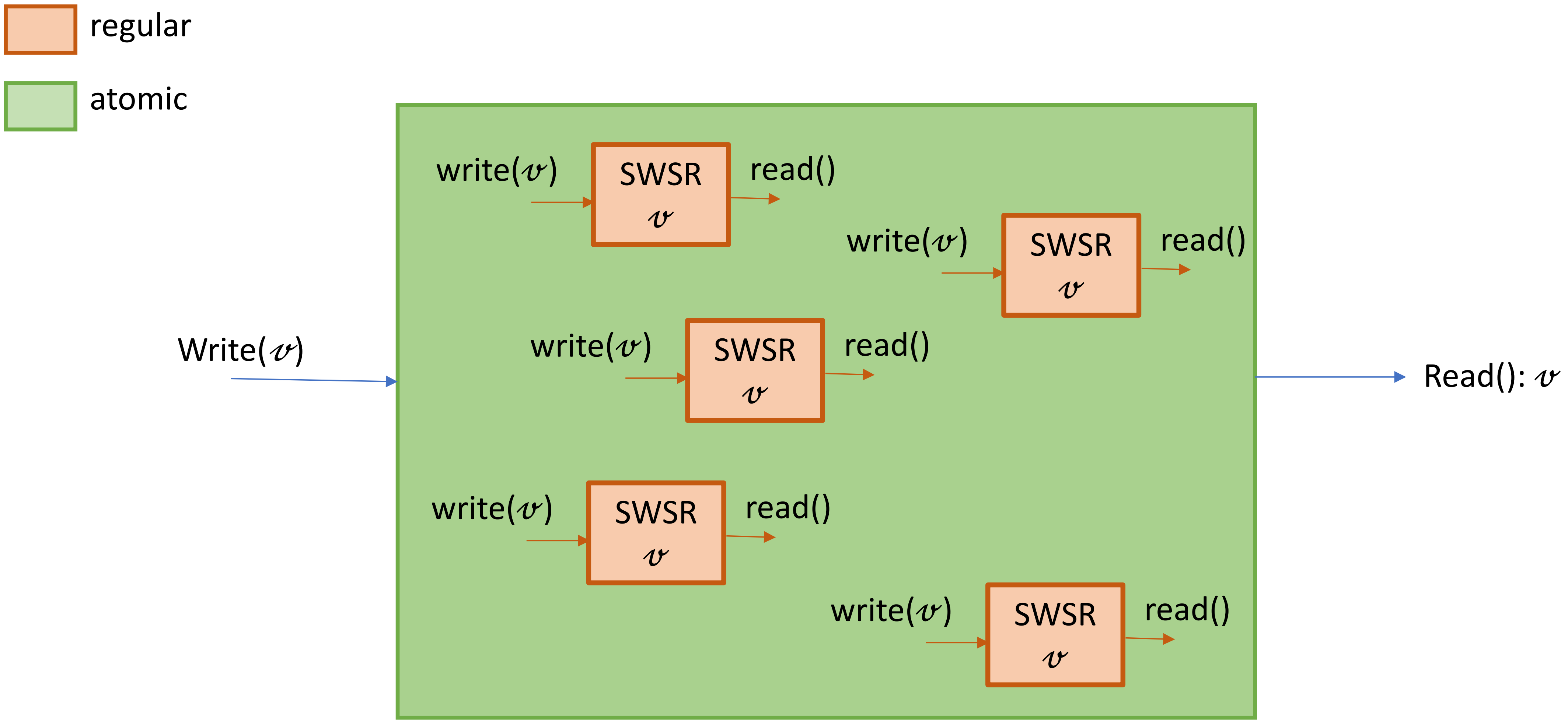


# Construction 4

So far we have seen how to construct multivalued safe SWMR registers from binary SWsR ones and multivalued regular SWMR from from binary SWSR ones.

What about atomic registers ?

# Construction 4:
# multivalued SWSR atomic from multivalued SWSR regular registers

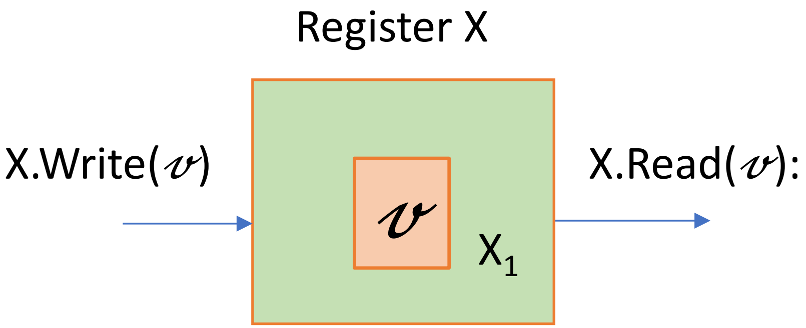


# Construction 4: multivalued SRSW atomic from multivalued SRSW regular registers

Atomic: a Read() operation cannot return an older value than the previously returned one

Construction 4 uses

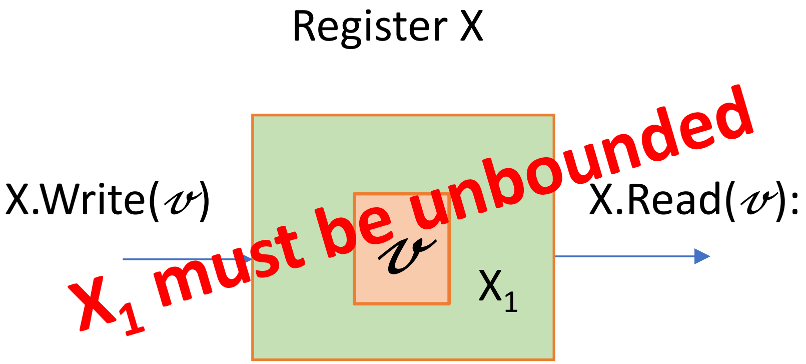
- 1 SWSR regular register $X_1$ and
- the writer maintains a local variable $t_w$ that will represent the time (to prevent new/old inversion) and
- the reader maintains two local variables $t_r$ and $x_r$. Initially $t_r$=0 and $x_r$=nil



# Construction 4: multivalued SRSW atomic from multivalued SWSR regular registers

```
X.Write(v): { // code executed by p_i
    t_w := t_w+1
    X_1.write(v,t_w)
    return
}

X.Read(): { // code executed by p_i
    (v,t) <- X_1.read()
    if (t>t_r) then
        x_r := v
        t_r := t
    return x_r
}
```



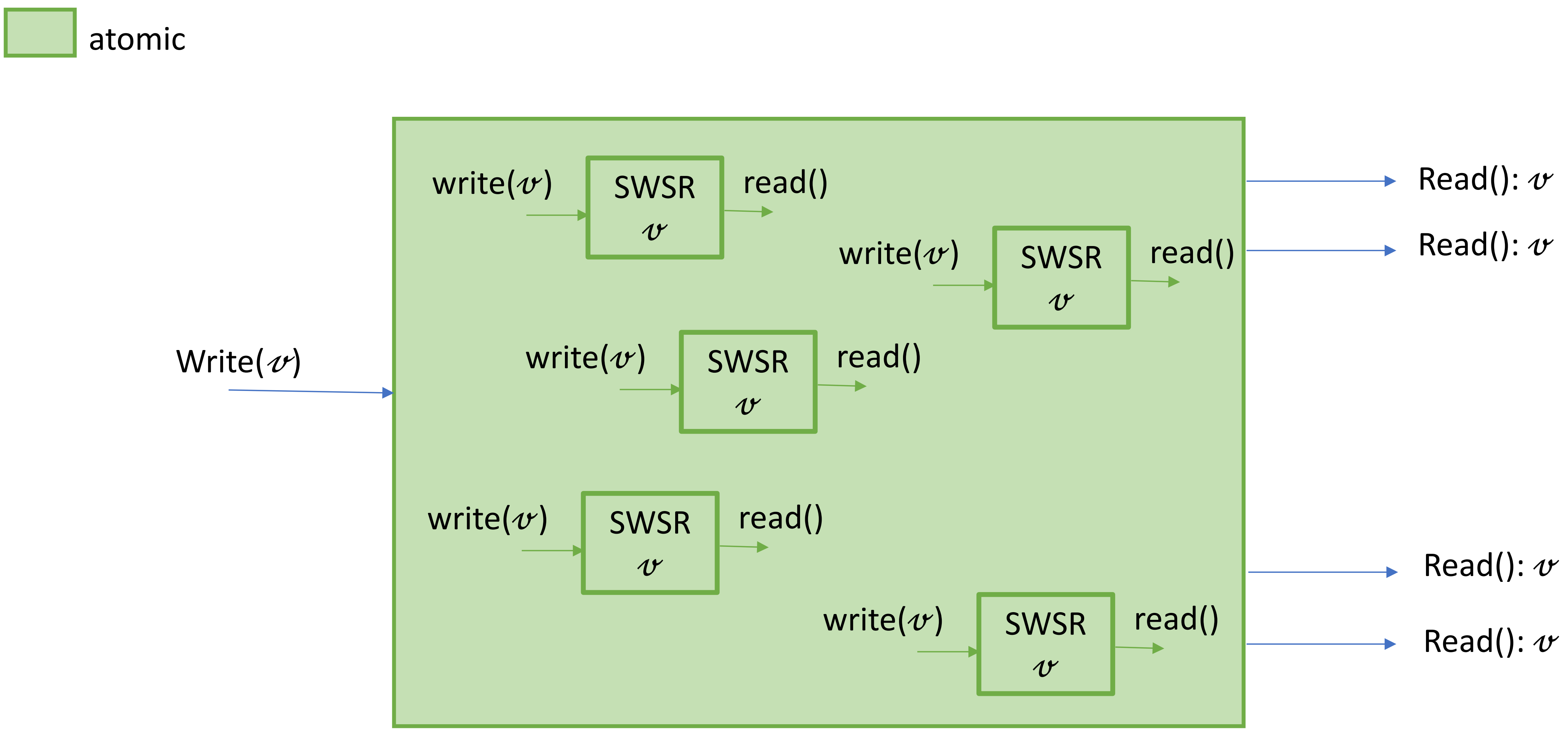
# Construction 4:
# multivalued SWSR atomic from multivalued SWSR regular registers

**Construction 4 does not allow us to implement a SWMR atomic register**

- We might think that by using multiple SWSR regular registers we might be able to build a SWMR atomic register
- For instance by associating 1 SWSR to each reader and have the writer writes in all of them
- But a fast reader might first see a new concurrently written value while a second reader may read an older value. This is because readers do not know the timestamps of each other and
- time does not grow at the same rate at each reader

**Note that the same argument holds if base registers were atomic SWSR registers**

# Construction 5: multivalued SWMR atomic from multivalued SWSR atomic registers



# Construction 5: multivalued SWMR atomic from multivalued SWSR atomic registers

- Idea: All the readers must help each other !

# Construction 5: multivalued SWMR atomic from multivalued SWSR atomic registers

- Idea: All the readers must help each other !

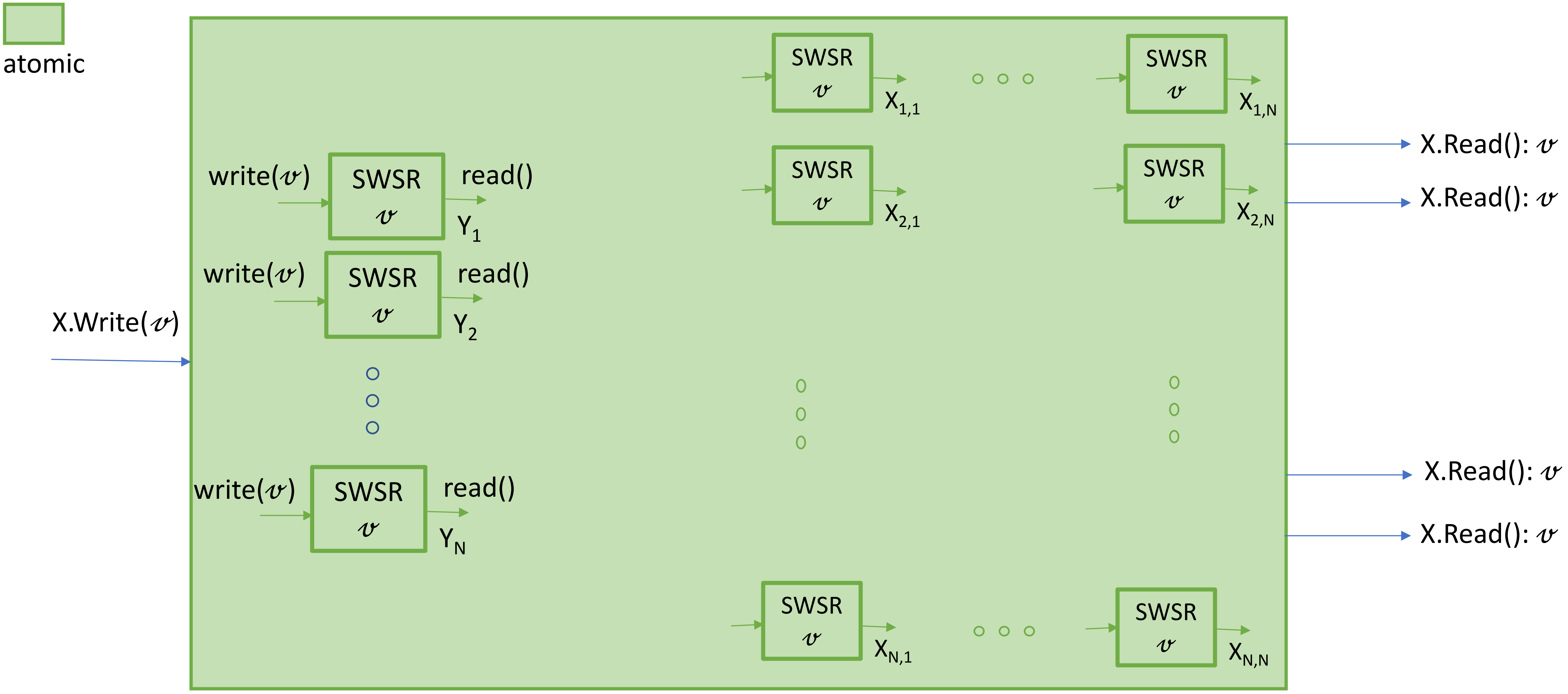
  a Read() operation ”informs” all the other readers of the read value, and then returns that value

# Construction 5:
# multivalued SWMR atomic from multivalued SWSR atomic registers

- Requires:

  1. to know the number N of readers
  2. to use N x N SWSR atomic registers: $X_{k,j}$ ($p_k$ is the reader and $p_j$ is the writer of $X_{k,j}$) to allow all the readers to communicate with each other the new values
  3. to use N SWSR atomic registers: $Y_j$ to write the new values

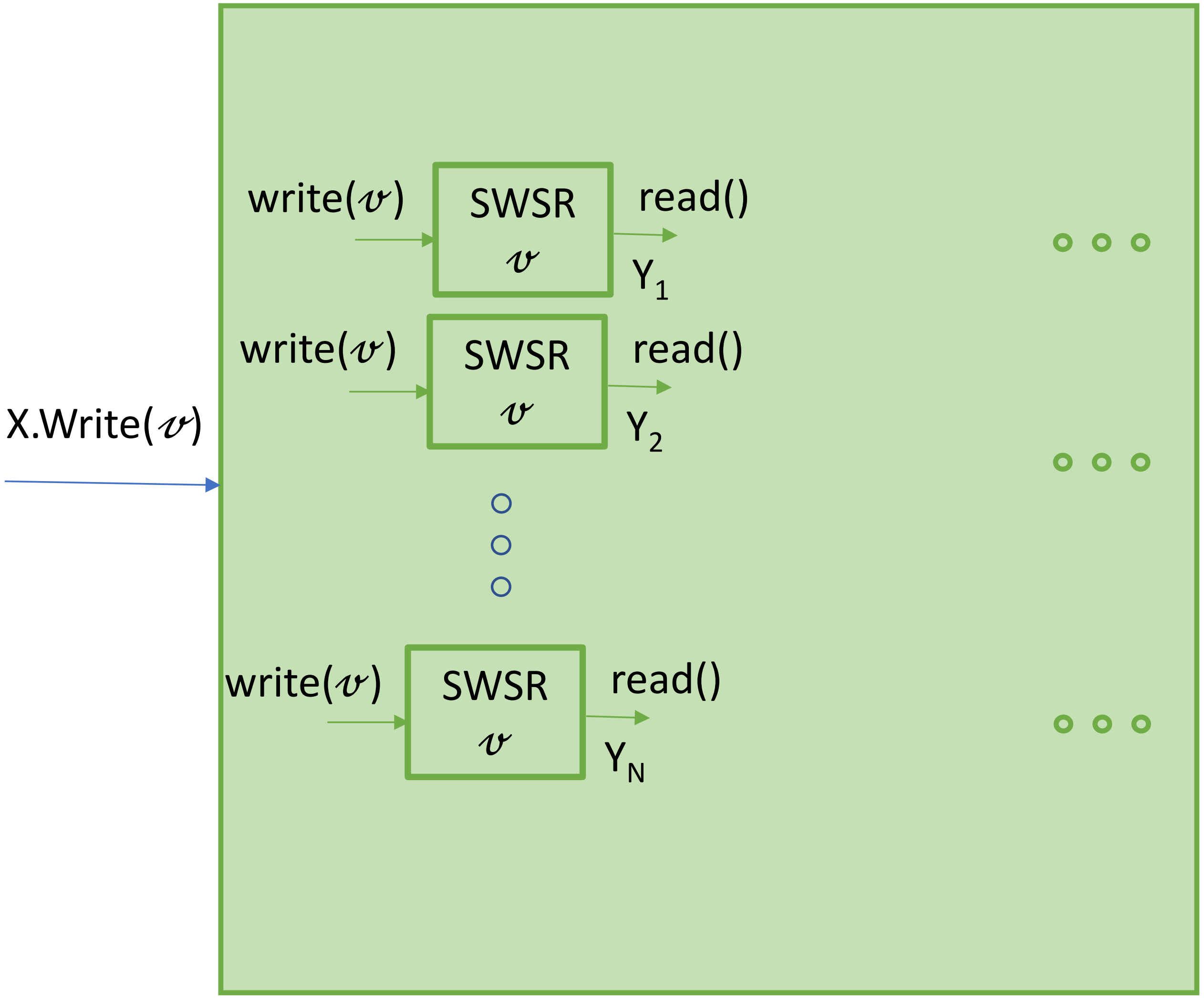
# Construction 5: multivalued SWMR atomic from multivalued SWSR atomic registers



# Construction 5:
# multivalued SWMR atomic from multivalued SRSW atomic registers

X.Write($v$): { *// code executed by $p_i$*

  $t_w := t_w + 1$
  for j=1 to N
    $Y_j$.write($v$, $t_w$)
  *return*
}



# Construction 5:
# multivalued SWMR atomic from multivalued SWSR atomic registers

X.Read() { // *code executed by $p_i$*

for j=1 to N

$(t[j],x[j]) := X_{i,j}.read()$

$(t[0],x[0]) := Y_i.read()$

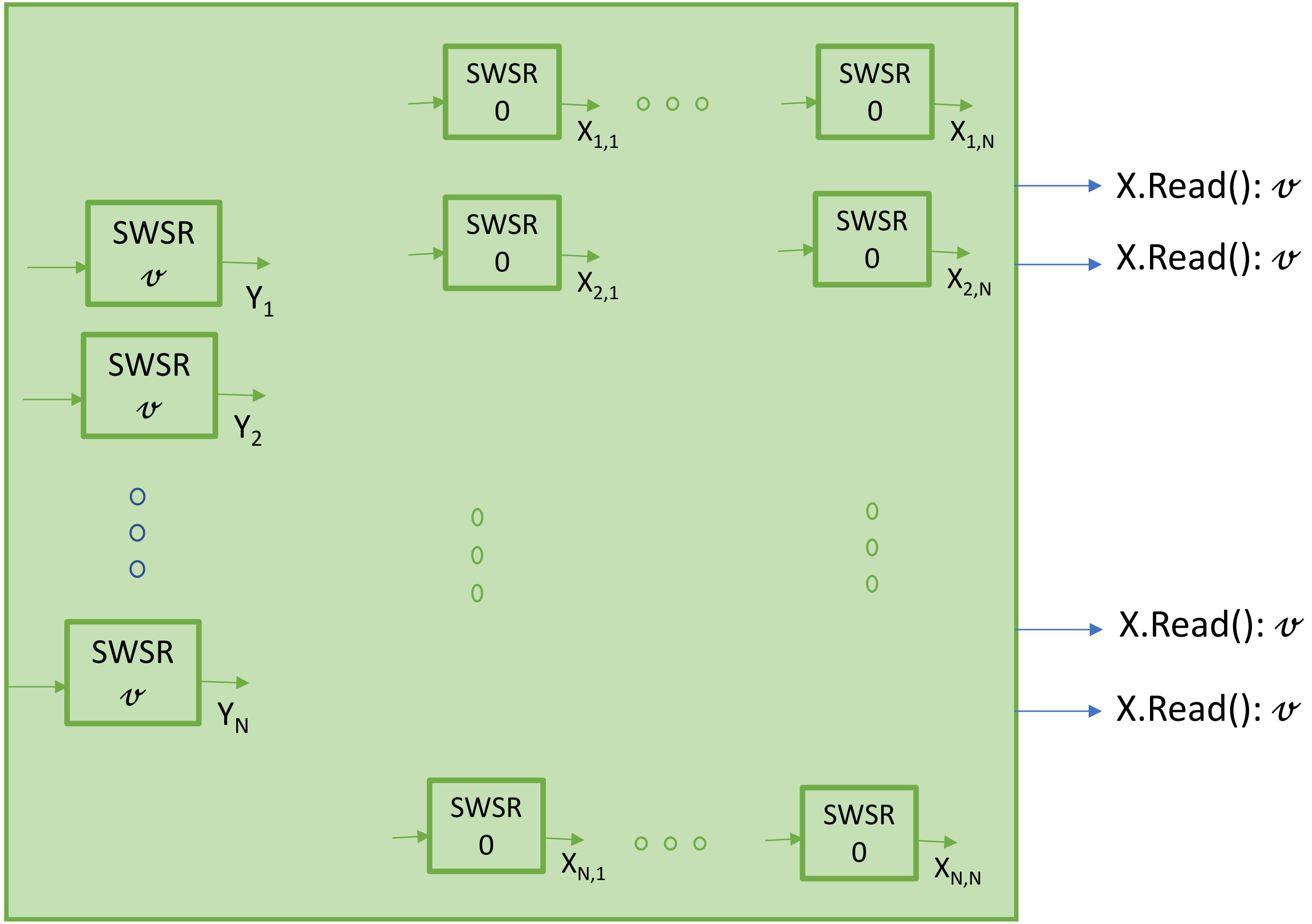
$(t_{max}, v) :=$ *tuple with largest t*

for j=1 to N

$X_{j,i}.write(t_{max}, v)$

*return* $v$

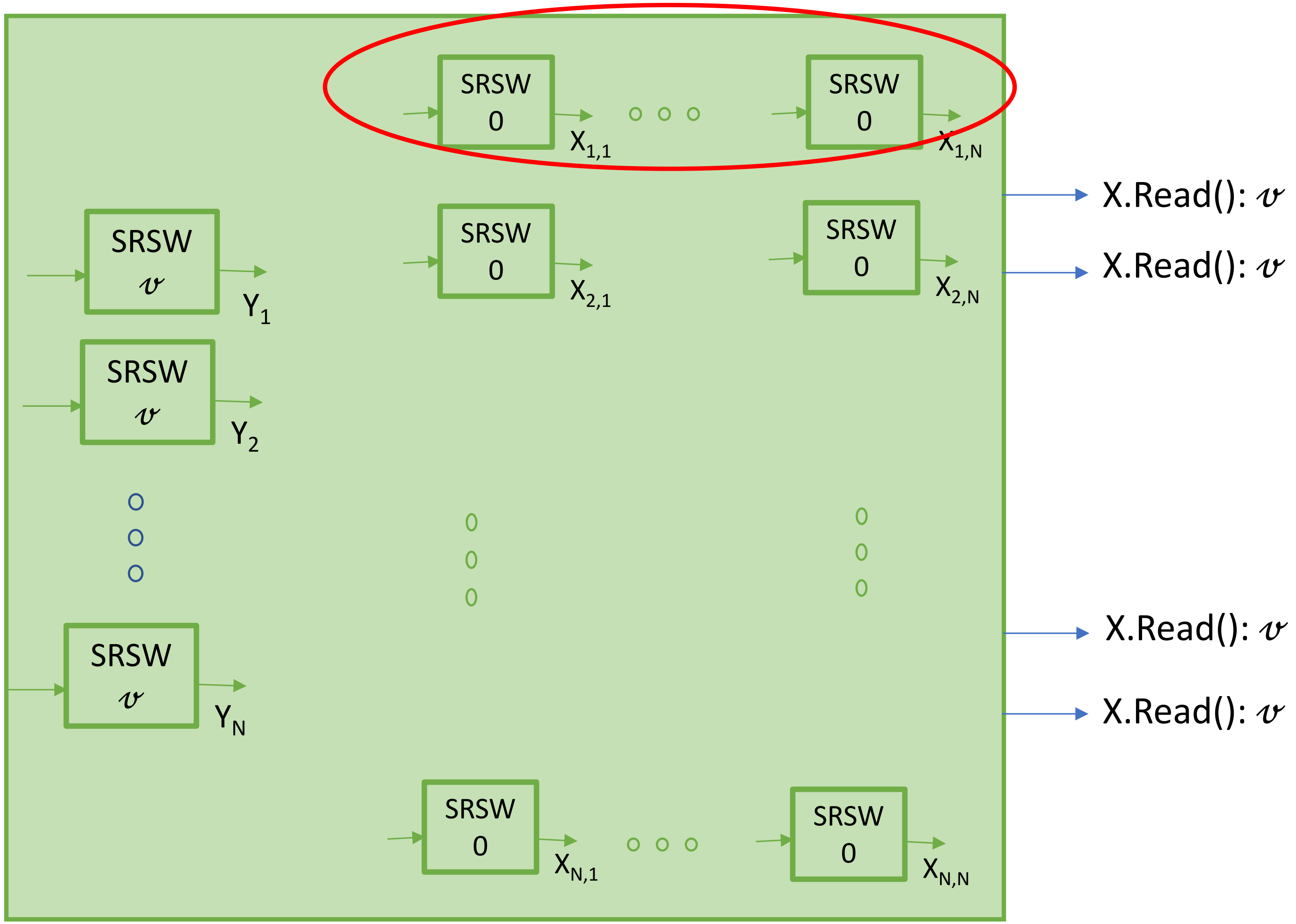
}



# Construction 5: multivalued SWMR atomic from multivalued SWSR atomic registers

```
X.Read() {     // code executed by p_i

    for j=1 to N
        (t[j],x[j]) := X_{i,j}.read()
    (t[0],x[0]) := Y_i.read()
    (t_max, v) := tuple with largest t
    for j=1 to N
        X_{j,i}.write(t_max, v)
    return v
}
```



# Construction 5:
# multivalued SWMR atomic from multivalued SWSR atomic registers

X.Read() { *// code executed by $p_i$*

for j=1 to N

*$(t[j],x[j]) := X_{i,j}.read()$*

*$(t[0],x[0]) := Y_i.read()$*

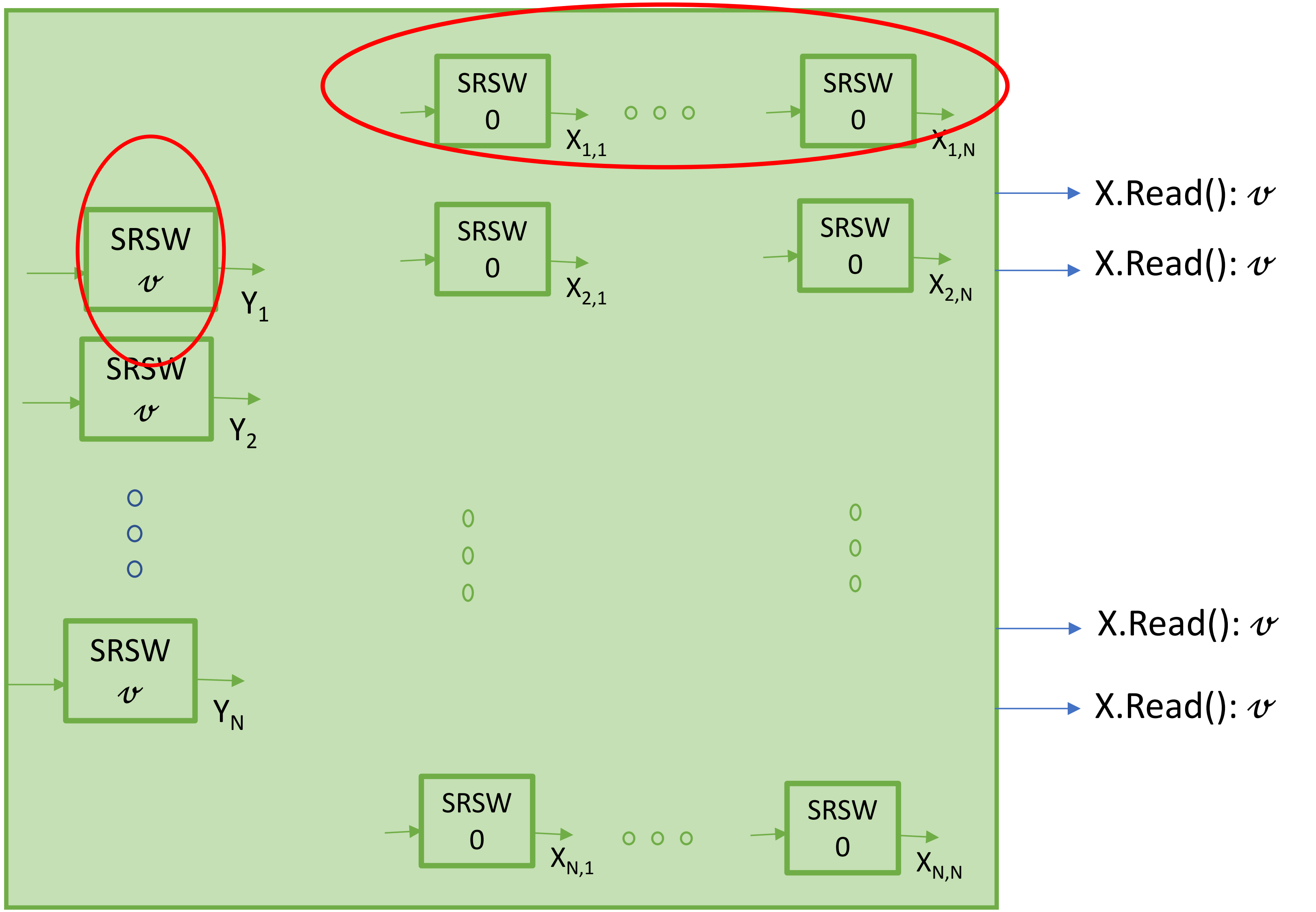
*$(t_{max}, v)$ := tuple with largest t*

for j=1 to N

*$X_{j,i}.write(t_{max}, v)$*

*return $v$*

}



# Construction 5:
# multivalued SWMR atomic from multivalued SWSR atomic registers

X.Read() { // *code executed by $p_i$*

for j=1 to N

*(t[j],x[j]) := $X_{i,j}$.read()*

*(t[0],x[0]) := $Y_i$.read()*

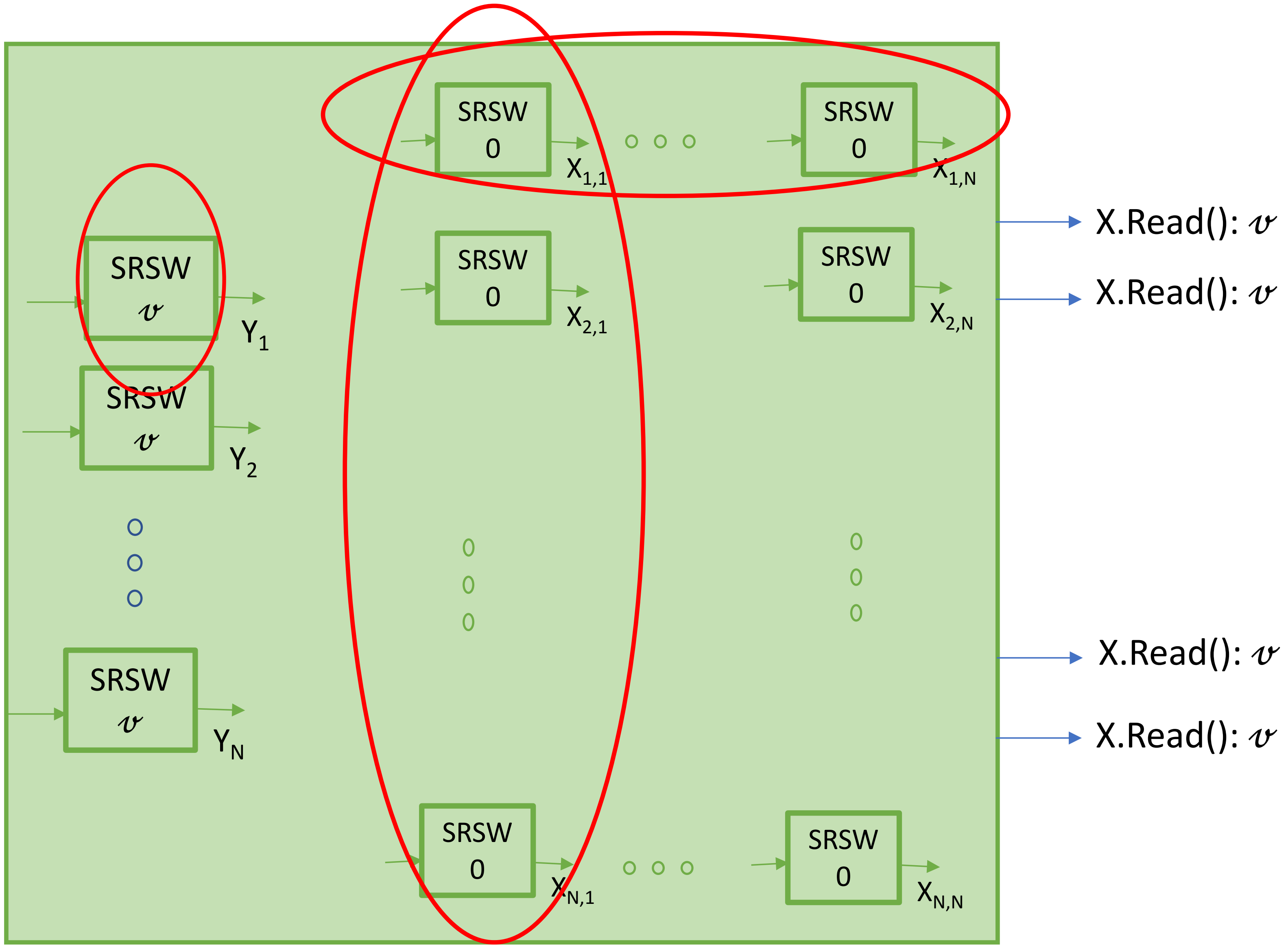
*($t_{max}$, $v$) := tuple with largest t*

for j=1 to N

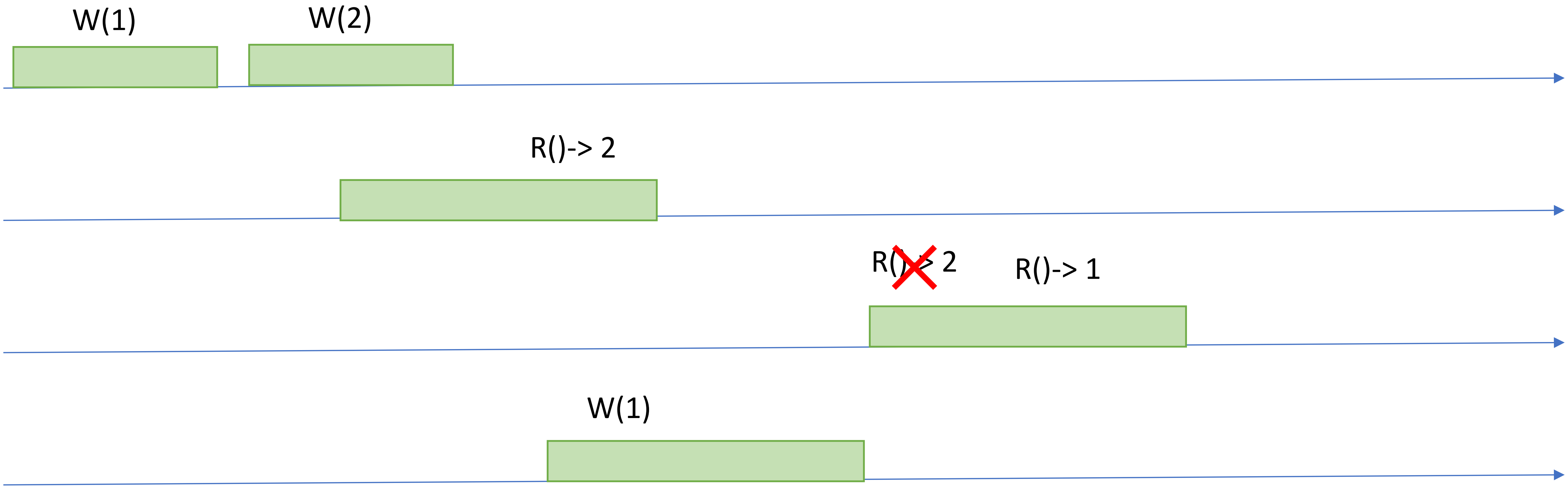
*$X_{j,i}$.write($t_{max}$, $v$)*

*return $v$*

}



# Construction 6:
# multivalued MWMR atomic from multivalued SWMR atomic registers

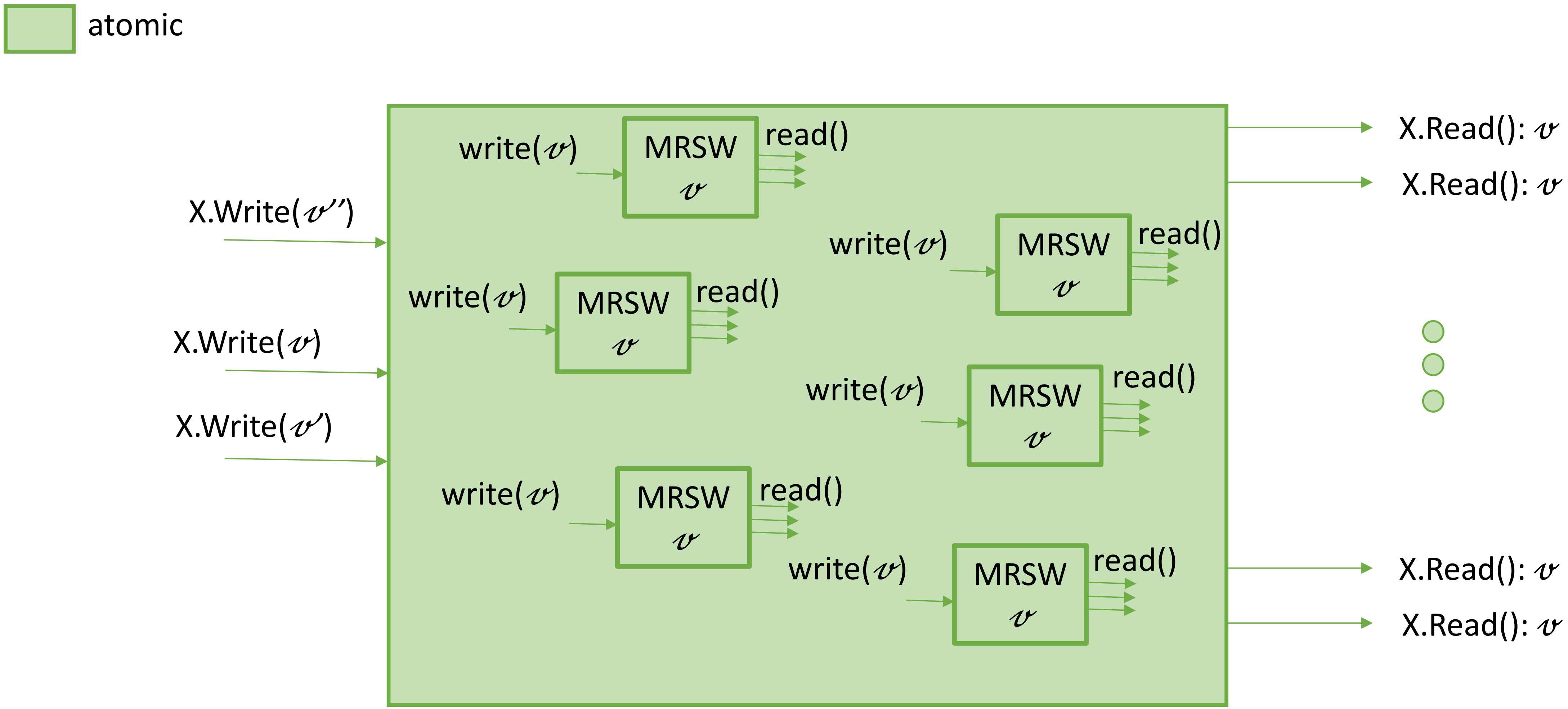



In Construction 5, timestamp is relative to a single writer and thus increases at its own rate

# Construction 6: multivalued MWMR atomic from multivalued SWMR atomic registers

- Hint:
  - All writers must determine the “current time”, i.e., the largest timestamp ever used by one of them
  - Write() operation: determine the current time and then apply the write to reader registers

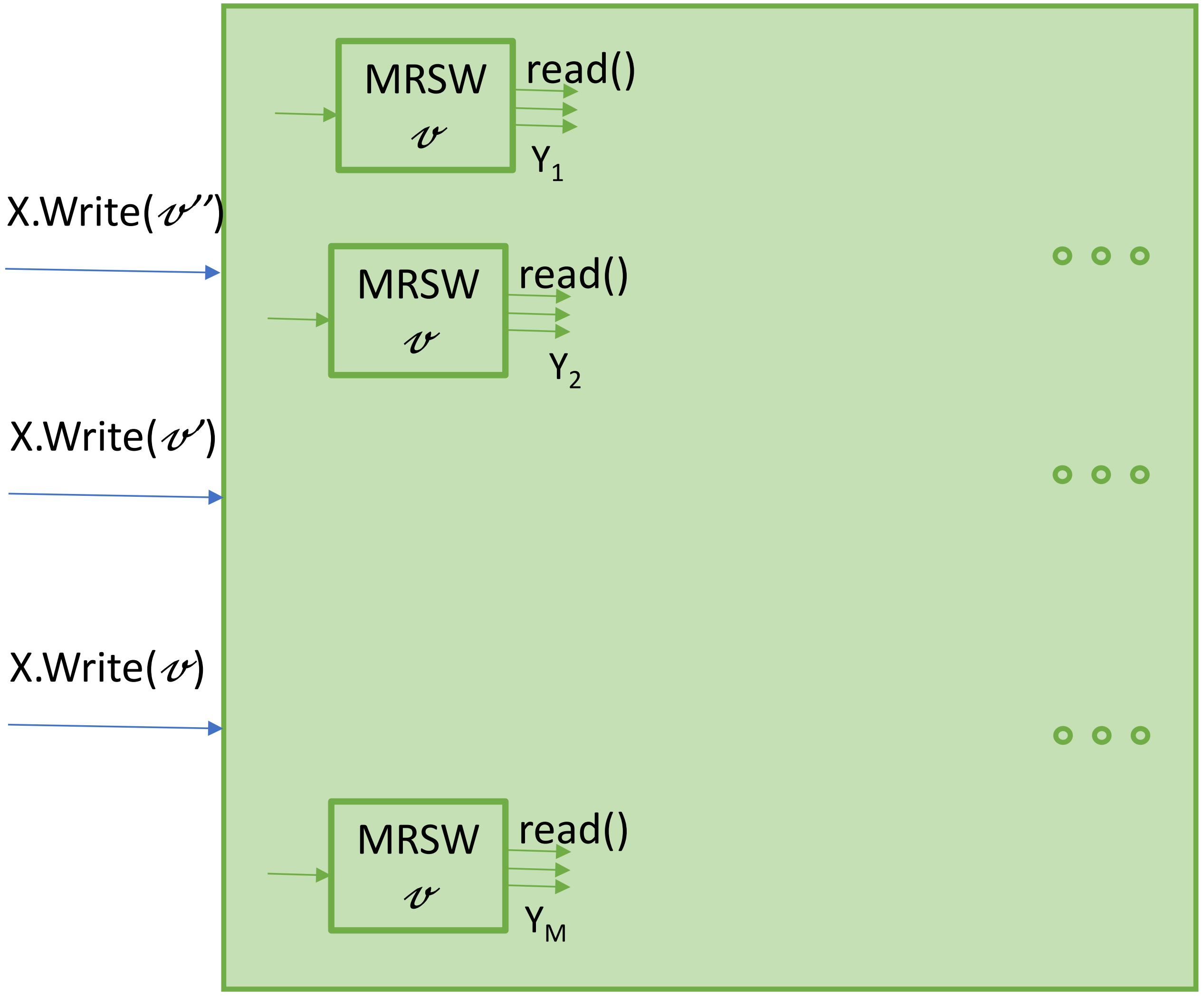
# Construction 6: multivalued MWMR atomic from multivalued SWMR atomic registers



# Construction 6:
# multivalued MWMR atomic from multivalued SWMR atomic registers

X.Write($v$):{ *// code executed by $p_i$*

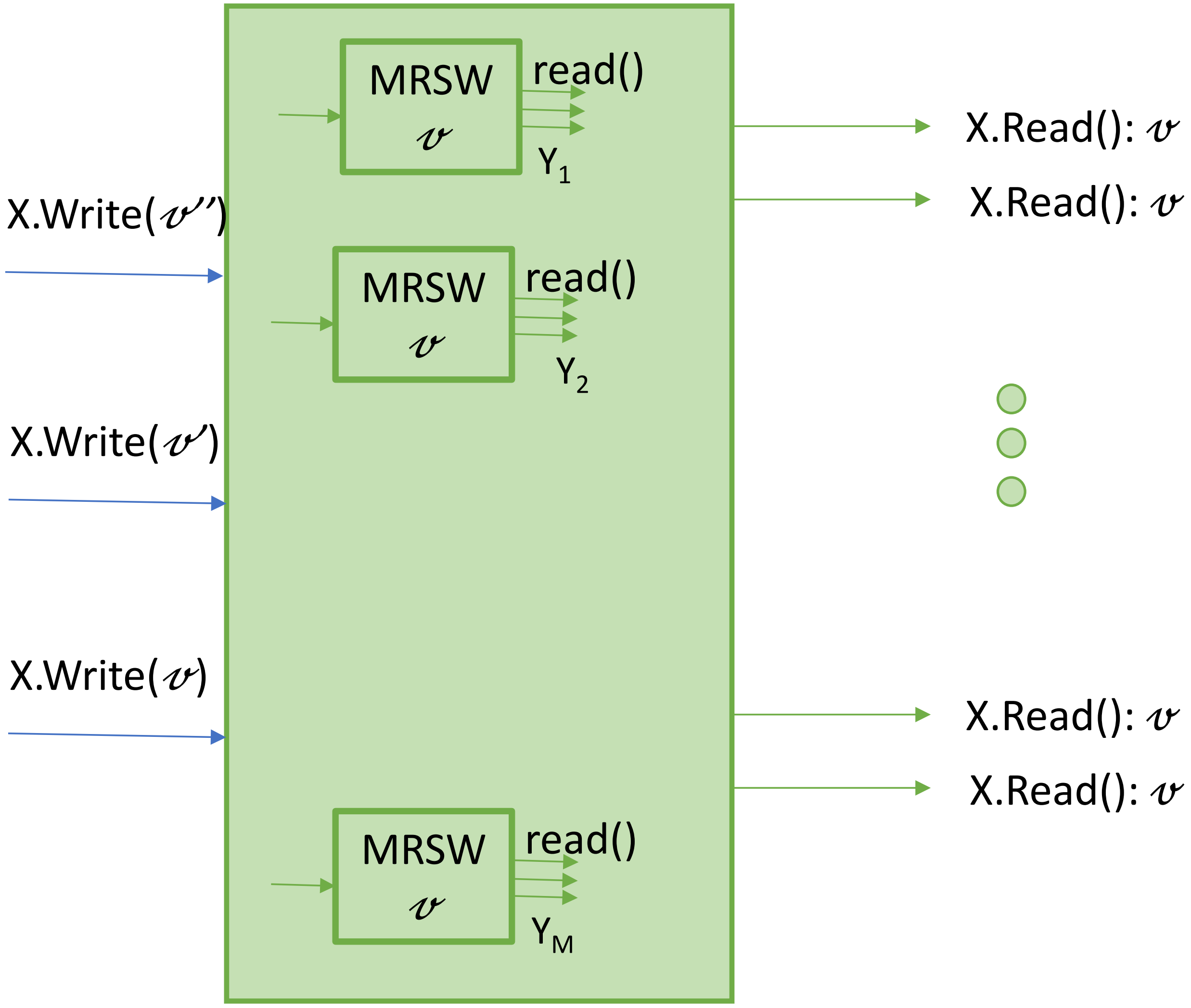
for $j$=1 to M
    t[$j$],x[$j$]:= $Y_j$.read()
*($t_{max}$,$x$) := tuple with largest t*
*$t_{max:=}$ $t_{max+1}$*
*$Y_i$.write($t_{max}$,$v$)*
return;
}



# Construction 6:
# multivalued MWMR atomic from multivalued SWMR atomic registers

X.Read():{ *// code executed by* $p_i$

for $j$=1 to M
  t[$j$],x[$j$]:= $Y_j$.read()
*($t_{max}$,$v$) := tuple with largest t*
*return* $v$



# Construction 3’: multivalued SWMR atomic from binary SWMR atomic registers

**Construction 3’ implements a multivalued SWMR atomic register from binary atomic SWMR ones**

- We have seen that construction 3 does not work to build a multivalued atomic register even from binary registers
- Actually, by just modifying the read operation, one can build such an SWMR atomic register
  - 1rst phase : identical to the one of construction 3: reads from register $X_0$ to register h ($X_h$=1)
  - 2nd : reads the registers in the opposite direction from $X_{h-1}$ to $X_l$ ($X_l$ =1, $0 \leq l \leq h-1$) or to $X_0$
  - The returned value is $l$ if different 0, h otherwise

# Construction 3':
# multivalued SWMR atomic from binary SWMR atomic registers

**Construction 3' implements a multivalued SWMR atomic register from binary atomic SWMR ones**

```
when p invokes X.Write(v):
    X_v.write(1)
    for each i in {v-1, .., 0}
        X_i.write(0)

when p_i invokes X.Read():
    h:=0
    while (X_h.Read() ≠ 1) do
        h:=h+1
    j = h
    for ℓ:= h -1 to 0
        if X_ℓ.Read() = 1
            j = ℓ
    return j
```

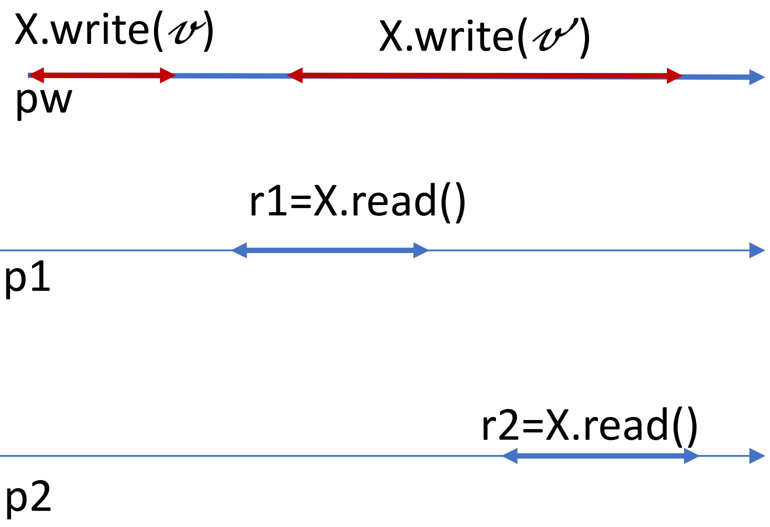
We first prove that Register X is regular

Let X.read() be an operation that returns $j$. We have two cases

- Case 1: $j = h$
  *Same proof as for construction 3. It follows that $j$ is either the value returned by the last preceding write or by a concurrent one. Thus the read() is regular*
- Case 1: $j \leq h$
  *Thus $X_j$ was equal to 0 during the ascending phase of the read and $X_j$ was equal to 1 during the descending phase*
  *By assumption, base registers are atomic. This means that a concurrent write operation has written* X.Write($j$) between those two readings. Thus the value $j$ *has been written by a concurrent operation. Thus X is regular*

# Construction 3’: multivalued SWMR atomic from binary SWMR atomic registers

## Construction 3’ implements a multivalued SWMR atomic register from binary atomic SWMR ones




We now prove that there are no new/old inversions

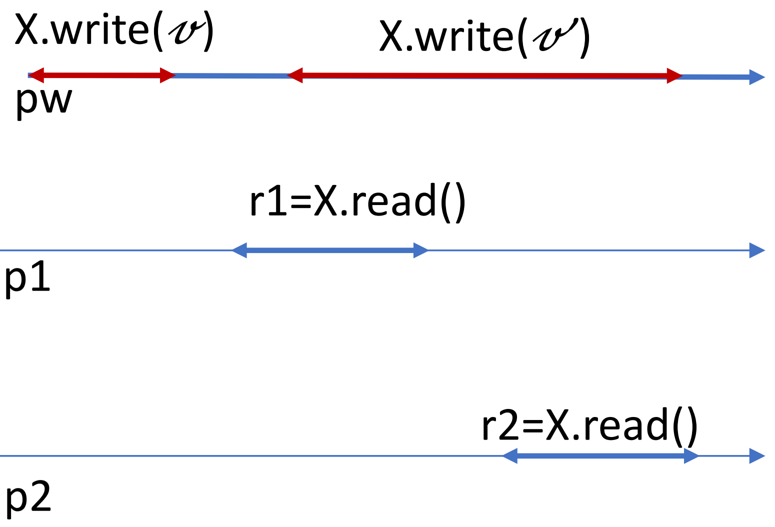
Consider the example. There are two reads concurrent with the second write such that the first read returns r1 and the second one returns r2.

- X is regular (from above) thus both reads can return r1 and r2
- Case 1: r1 = *$v$, then r2 can return either $v$ or $v'$. No new/old inversion*
- Case 2: r1 = *$v'$, then if r2 returns $v'$, no new/old inversion.*
  *We show that r2 returns $v''$ with $v''$ written by a more recent concurrent write. Two cases*
  *Case 2.1: $v'' < v'$: Thus a write op has written $X_{v''}$ =1 after the descending phase of r1 and before the read() of $X_{v''}$ by r2. This write is after* X.Write($v'$) (we have a single writer). Thus r2 obtains a value which is more recent than *$v'$ and thus more recent than $v$ thus there are no new/old inversion*
  *Case 2.2: $v' < v''$ :*

# Construction 3’: multivalued SWMR atomic from binary SWMR atomic registers

## Construction 3’ implements a multivalued SWMR atomic register from binary atomic SWMR ones

We now prove that there are no new/old inversions (continued)




*Case 2.2: $v' < v''$ : In this case, r2 has read $X_{v''}$=1 during its ascending phase and $X_{v'}$=0 during its descending phase (otherwise it would have returned $v'$).*
*As r1 precedes r2 and writes are sequential it means that there is a write operation X.write($v'''$) with $v''' > v'$ issued after X.write($v'$) that has updated $X_{v'}$=0 .*

*Case 2.2.1: If that operation is X.write($v''$) then the value read by r2 is posterior to $v'$ thus there is no new/old inversion.*

*Case 2.2.2. On the other hand, if X.write($v'''$) is different from X.write($v''$) then X.write($v'''$) is before X.write($v''$) otherwise X.write($v''$) would have updated $X_{v'}$=0. Thus we have X.write($v'$) -> X.write($v'''$) -> X.write($v''$). Thus value $v''$ is more recent than value $v'$ and thus we do not have a new/old inversion.*

# Bibliography


- Mutual exclusion part comes from
  - Hagit Attiya and Jennifer Welch’s book, “Distributed computing, Fundamentals, simulations, and advanced topics”, Wiley series.
  - Michel Raynal’s book, Concurrent programming: Algorithms, Principles, and Foundations, Springer
- Constructions part comes from
  - On interprocess communication, Part I and II. Distributed computing. Vol 1, Number 2, pages 77 – 101.
  - Michel Raynal’s book, Concurrent programming: Algorithms, Principles, and Foundations, Springer