Registers

- store values:
  - binary? multivalued?

- two operations:
  - `read`: one reader? many readers?
  - `write`: one writer? many writers?

- safety property?
  - (Liveness?: )

```java
public interface Register<T> {
    public T read();
    public void write(T v);
}
```

» SRSW: Single-reader single-writer
» MRSW: Multi-reader single-writer
» MRMW: Multi-reader multi-writer
Registres

- Always: if **operations don’t overlap** (no concurrency) every *read* returns the last *written* value (or else the initial value) (sequential specification of registers is clear)

  - `write(1001)`
  - `read(1001)`
  - OK
(Safe) Registers

- if operations overlap, read returns:
  - safe register: any value

« legal » value when concurrency

write(1001)

read(?????)

0000  1001  1111

$*&v
Regular Register

- if operations overlap, \textit{read} returns:
  - **Regular register**: previous written value or concurrently written value
Regular?

write(0) → read(1) → write(1) → read(0)
Regular?

write(0)  read(1)  write(1)  read(0)

Overlap: returns new value
Regular?

write(0) -> write(1) -> read(0)

Overlap: returns old value
Regular?

write(0)  read(1)  write(1)  read(0)  Regular!
(Atomic) registers

- Atomic = linearizable
  - the operations can be totally ordered preserving legality and precedence
  - Given an execution it is possible to reduce the time interval of each operation to a point (linearization point) (hence we get a totally ordered history) in such a way that satisfies the sequential specification of a registers.
    - (Linearizability is a general property we will present later with more details)
Atomic-linearizable

- Processes communicate by applying operations on and receiving response from shared objects

- A *shared objects* is define by:
  - states
  - operations
  - sequential specification
Register

• Operations: read, write

• state: value of the variable

• sequential specification:

\[
\{ \text{state} = y \}\text{write}(x)\{ \text{state} = x \} \quad \text{return ok}
\]

\[
\{ \text{state} = y \}\text{read}()\{ \text{state} = y \} \quad \text{return } y
\]
Example: Queue

- operations: enq(item x) ; item deq()
- states: sequence of items
- sequential specification:

\[
\begin{align*}
\{ state = f \} & \text{ enq}(x) \{ state = f.x \} \\
if (state \neq \emptyset) & \{ state = a.f \} \text{ deq()} \{ state = f \}
\end{align*}
\]

Invocation response

return ok

return a
Atomic register

Linearizable
Atomic register

write(1001)  read(1001)  write(1010)  read(1010)  read(1010)  read(1010)
Regular ≠ atomic

write(0) → write(1) → read(1) → read(0) → 0?

write(1) done
Register Space

MRMW

MRSW

SRSW

Safe

Regular

Atomic

m-valued

Boolean
Theorem: It is possible to implement multivalued MWMR atomic register from SWSR safe binary register

- From binary safe SRSW to binary safe MRSW
- From binary safe SWMR to binary regular SWMR
- From binary regular MRSW to multivalued regular MRSW
- From multivalued regular SRSW to multivalued atomic SRSW
- From multivalued atomic SRSW to multivalued atomic MRSW
- From multivalued atomic MRSW to multivalued atomic MRMW
Boolean safe MRSW from boolean safe SRMW
Boolean safe MRSW from boolean safe
SRMW

Let’s Write 1!
Boolean safe MRSW from boolean safe SRMW

Registers

0 or 1
Boolean safe MRSW from boolean safe SRMW
Boolean safe MRSW from boolean safe
SRMW

Registers
Boolean safe MRSW from boolean safe SRMW
Binary safe **SRSW** to binary safe **MRSW**

- **P0** is the only writer, **v** is the initial value

\[ \text{initially shared array } R \text{ of } n \text{ SWSR binary safe register init } v \]

**Code of P0**
```
write(w) {
    for (int i=0; i<n; i++) R[i].write(w);
    return (ok)
}
```

**Code for Pi**
```
read() {
    return R[i].read()
}
```

**Work also:**
- for multivalued regular

**doesn’t work for atomic register**
Boolean **regular** MRSW from boolean **safe** MRSW?
SWMR binary safe to SWMR binary regular

- P0 is the only writer and v is the initial value

```
initially shared R SWMR binary safe register init
local to P0 lw:=v //last written value

Code of P0
write(w){
    if w ≠ lw then
        lw:=w; R.write(w)
        return (ok)
    }

Code for all processes
read(){
    return R.read()
}
```
From binary regular MRSW to multivalued regular MRSW
Multivalued

Unary représentation: \text{bit}[i]

-> valeur i

initialement 0

\begin{array}{cccccccc}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\end{array}
Write

Write 5

1 0 0 0 0

0 1 2 3 4 5 6 7
Write..

Initially 0

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

0 1 2 3 4 5 6 7
Multi-valued Regular MRSW from boolean regular MRSW

```java
public class RegMRSWRegister implements Register{
    RegBoolMRSWRegister[M] bit;

    public void write(int x) {
        this.bit[x].write(true);
        for (int i=x-1; i>=0; i--)
            this.bit[i].write(false);
    }

    public int read() {
        for (int i=0; i < M; i++)
            if (this.bit[i].read())
                return i;
    }
}
```
Regular SWSR to atomic SWSR
Regular SWSR to atomic SWSR

Regular writer

Concurrent read

Regular reader

et pas 5678...

Issue?
Regular SWSR to atomic SWSR

Initially
1234

Reg write(5678)

Reg read(5678)

Same as Atomic

Regular writer

Regular reader

5678
Regular SWSR to atomic SWSR

Regular writer

Regular reader

Initialement
1234

Reg write(5678)

Reg read(1234)

Same as
Atomic

time
Regular SWSR to atomic SWSR

Régulier writer

Initialement 1234

Reg write(5678)

Reg read(5678)

Reg read(1234)

Write 5678 done

MPRI

Not Atomic...
Timestamp

Writer write value + timestamp

Reader save the last read (value, stamp) and returns new value only if the timestamp is greater.
Regular SWSR to atomic SWSR

writer

Reg write(2:00 5678)
read(2:00 5678)
old = 2:00
5678

Same as atomic

read(1:45 1234)

reader

time

1:45
1234

2:00
5678

MPRI
Atomic Single-Reader to Atomic Multi-Reader

<table>
<thead>
<tr>
<th>stamp</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:45</td>
<td>1234</td>
</tr>
<tr>
<td>1:45</td>
<td>1234</td>
</tr>
<tr>
<td>1:45</td>
<td>1234</td>
</tr>
<tr>
<td>1:45</td>
<td>1234</td>
</tr>
</tbody>
</table>

One per reader
**Scenario**

Writer begins its write

<table>
<thead>
<tr>
<th>Stamp</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>2:00</td>
<td>5678</td>
</tr>
<tr>
<td>1:45</td>
<td>1234</td>
</tr>
<tr>
<td>1:45</td>
<td>1234</td>
</tr>
<tr>
<td>1:45</td>
<td>1234</td>
</tr>
</tbody>
</table>
Scenario...

Yellow returns a previous value but is after Blue: not atomic.
# Multi-Reader

![Diagram of Multi-Reader]

One per process

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1:45</td>
<td>1234</td>
<td>1:45</td>
</tr>
<tr>
<td>2</td>
<td>1:45</td>
<td>1234</td>
<td>1:45</td>
</tr>
<tr>
<td>3</td>
<td>1:45</td>
<td>1234</td>
<td>1:45</td>
</tr>
</tbody>
</table>
**Multi-Reader**

Writer write a column

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2:00</td>
<td>5678</td>
<td>1:45</td>
</tr>
<tr>
<td>2</td>
<td>2:00</td>
<td>5678</td>
<td>1:45</td>
</tr>
<tr>
<td>3</td>
<td>2:00</td>
<td>5678</td>
<td>1:45</td>
</tr>
</tbody>
</table>

reader read a row

2:00, 5678
reader writes the column with what it has read

zzz...after second write

Yellow will will the new value in the column written by blue
Can Yellow miss the Blue’s update?...only with concurrency!

In that case read 1234 is ok
Without concurrency all is ok!

Bleu has finished its write 2:00 5678 in its column.
public class AtomicMRSWRegister<T> implements Register<T> {
    ThreadLocal<Long> lastStamp;

    private StampedValue<T>[] [] a_table; // each entry is SRSW atomic

    public AtomicMRSWRegister(T init, int readers) {
        lastStamp = new ThreadLocal<Long>() {
            protected Long initialValue() { return 0; }
        };
        a_table = (StampedValue<T>[][]) new StampedValue[readers][readers];
        StampedValue<T> value = new StampedValue<T>(init);
        for (int i = 0; i < readers; i++) {
            for (int j = 0; j < readers; j++) {
                a_table[i][j] = value;
            }
        }
    }

    public T read() {
        int me = ThreadID.get();
        StampedValue<T> value = a_table[me][me];
        for (int i = 0; i < a_table.length; i++) {
            value = StampedValue.max(value, a_table[i][me]);
        }
        return value;
    }

    public void write(T v) {
        long stamp = lastStamp.get() + 1;
        lastStamp.set(stamp);
        StampedValue<T> value = new StampedValue<T>(stamp, v);
        for (int i = 0; i < a_table.length; i++) {
            a_table[i][i] = value;
        }
    }
}