# The concept of a $\boxed{\text{TIMED REGISTER}}$ and its application to indulgent synchronization

Michel RAYNAL and Gadi TAUBENFELD

IRISA, Université de Rennes, France

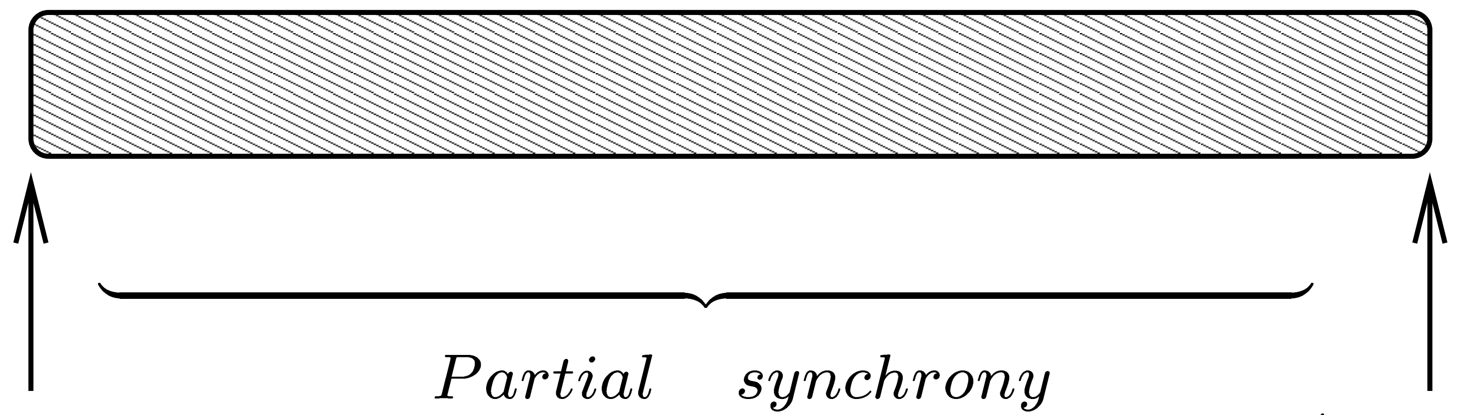
Interdisciplinary Center, Herzliya, Israel

raynal@irisa.fr tgadi@idc.ac.il

## Summary

- Basic timing-based SM model
- Concept of a Timed Register
- Indulgent synchronization
- Indulgent agreement
- Timed register in perspective
- Conclusions

## Models of computation



*Synchronous*
(Known bounds)
Extremely powerful
Unrealistic

*Asynchronous*
(No bounds)
Less powerful
Realistic

## Goal

- Observation: Many systems exhibit a significant degree of synchrony in practice, but few guarantee to do so
- Goal
  - ⋆ Exploit synchrony when it is available
  - ⋆ In any case guarantee correctness regardless of the timing behavior of the system

## Part I

BASIC TIMING-BASED SM MODEL

## Basic timing-based model

- A set $\Pi$ of processes $\{\ldots, p_i, \ldots\}$ ($i$ is the of $p_i$)
- Communication by accessing atomic memory locations
- Timing assumption: There is an upper bound $\Delta$ on the time that can elapse between any two consecutive memory accesses by the same process $p_i$

  It is important to notice that $\Delta$ is on any two consecutive accesses on any pair of registers by the same process $p_i$: the SM is considered as a "single register" from the timing assumption point of view
- Timing-based algorithms:
  Their safety and liveness properties rely on such a global bound $\Delta$

## Timing-based algorithm: an example

Fischer's mutual exclusion algorithm

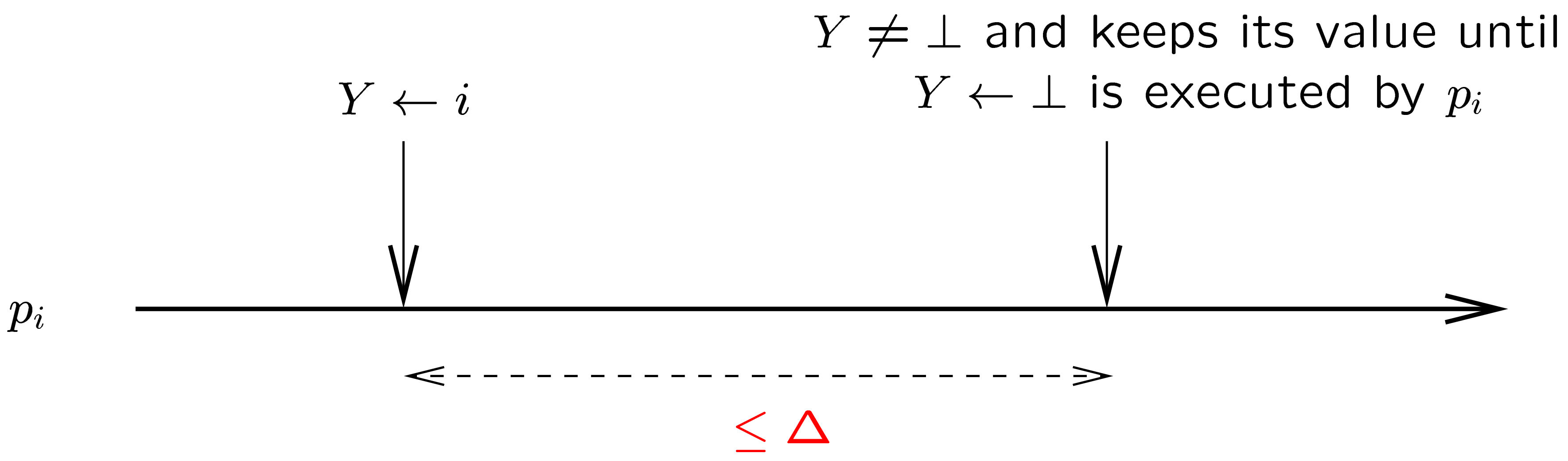
```
init $Y \leftarrow \bot$

operation enter_mutex($i$):
    repeat await ($Y = \bot$);
              $Y \leftarrow i$; delay($\Delta$)
    until ($Y = i$) end repeat

operation exit_mutex(): $Y \leftarrow \bot$
```

Simple and elegant

## Fischer's algorithm: illustration


$Y \leftarrow i$
$Y \neq \perp$ and keeps its value until
$Y \leftarrow \perp$ is executed by $p_i$
$p_i$
$\leq \Delta$

Mutex + Deadlock-free if the $\Delta$ always satisfied

No guarantee when the $\Delta$ assumption is violated

## Part II

The concept of a TIMED REGISTER

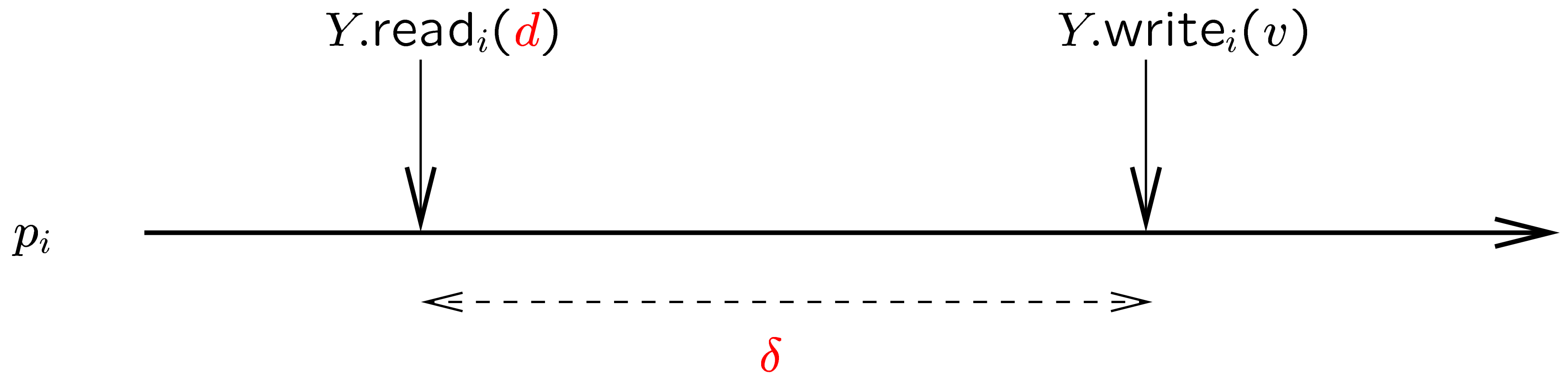
## Timed register: preliminaries

- Generalize the notion of atomic register by imposing time constraints on some write operations in order they be successful
- A write operation returns $true$ or $false$
- Context:
  - ⋆ Let $Y$ be a timed register and $p_i$ a process
  - ⋆ Let $Y$.read$_i(d)$ be a read of $Y$ by $p_i$
  - ⋆ $d$ is a duration defined as part of the read operation
  - ⋆ Let $Y$.write$_i(v)$ be the first write on $Y$ issued by $p_i$ after $Y$.read$_i()$
  - ⋆ Between these two operations:
    - $p_i$ can issue other operations on other registers
    - Other processes can access $Y$

## Timed register: definition

- Constraint:
  - ⋆ $Y$.write$_i(v)$ succeeds if $Y$.write$_i(v)$ and $Y$.read$_i(d)$ are separated by at most $d$ time units
  - ⋆ If the writer is successful it returns $true$, otherwise it returns $false$
- This constraint is local: it involves an ordered pair (read;write) on the same object by the same $p_i$
- Particular case: if a process $p_i$ always sets its constraint $d$ equal to $+\infty$ when it reads the timed register $Y$, that register behaves as a classical register wrt $p_i$ (all its writes are successful)

## Timed register: illustration



$d \leq \delta$: $Y$.write$_i(v)$ is successful

$d > \delta$: $Y$.write$_i(v)$ fails

During the period $\delta$:

- $p_i$ can access other timed or time-free registers
- Other processes can access the timed register $Y$

## Our timing-based model

- AIM: ensure that the writes can be successful
- Assumption $\Delta$: There is an upper bound $\Delta$ on the time that can elapse between a constraining read and the associated constrained write issued by the same $p_i$ on the same register (for any $p_i$)
- Differently from the basic timing model, here the assumption $\Delta$ is LOCAL
- $\Delta$ can be known or unknown

## Failures and indulgence

- Transient failures:
  when the bound $\Delta$ is violated intermittently
- Indulgent algorithm:
  - ⋆ Safety: never violated
  - ⋆ Liveness: asa there are no more failures
- Timed registers are universal objects in systems that eventually satisfy the $\Delta$ assumption

## A basic pattern

**init** $Y = \perp$; $\forall i$: $v_i \neq \perp$

$Y$ is not accessed outside the pattern

The pattern is executed at most once by each $p_i$

PATTERN:

```
while ($Y$.read($\delta$) $= \perp$) do $Y$.write($v_i$) end while;
    Here: $Y$ has a non-$\perp$ value
delay($\delta$).
    Here: $Y$ has its definitive value
```

Connection with compare&swap()

## Part III

INDULGENT SYNCHRONIZATION

## Indulgent mutual exclusion: algorithm

$Y \neq \perp \Leftrightarrow$ "processes are competing"
The algorithm is an instance of the basic pattern

```
operation enter_mutex($i$):
      repeat
         await ($Y$.read($\Delta$) $= \perp$);
         if $Y$.write($i$) then delay($\Delta$) end if
      until ($Y$.read($\infty$) $= i$) end repeat

operation exit_mutex():
      $Y$.write($\perp$)
```

## Indulgent mutual exclusion: properties

- Makes indulgent Fischer's algorithm
- Works for any number of processes
- It is symmetric (proc indexes are only compared)
- Uses a single timed register
- When the bound $\Delta$ is not known
- Assume a process does not crash in its critical section
- Easy extension to the $\ell$-mutex problem

## The adaptive Wait-free renaming problem

- Processes wants to acquire (and later release) new names from a small name space $[1..M]$
- If a single process (e.g., $p_n$) wants to acquire a new name, it cannot consider its index as its new name
- Resource allocation problem: the resources are the new names
- The best that can be done in a RW asynchronous SM system with up to $n-1$ crashes $M = 2p-1$ (where $p$ is the nb of participating processes)
- Consensus number of renaming is 2 (same as test&set)

## Adaptive Wait-free renaming algorithm: algorithm

Shared array $Y[1..n]$ such that
$Y[c]$ controls the assignment of the new name $c$
New name space $M = p$

```
operation get_name($i$):
        $c_i \leftarrow 1$;
        repeat
            while ($Y[c_i]$.read($\Delta$) $\neq \perp$) do $c \leftarrow c_i + 1$ end while;
            if $Y[c_i]$.write($i$) then delay($\Delta$) end if
        until ($Y[c_i]$.read($\infty$) $= i$) end repeat;
        return ($c_i$)

operation release_name($c_i$):
        $Y[c_i]$.write($\perp$)
```

## Part IV

INDULGENT AGREEMENT

## Indulgent test&set with known bound

Test&set object: elects a single winner
The algorithm is another instance of the basic pattern

```
operation $TS$.test&set():
    while ($Y$.read($\Delta$) $= \perp$) do
        if $Y$.write($i$) then delay($\Delta$) end if
    end while;
    if $Y$.read($\infty$) $= i$ then return(1) else return(0) end if

operation $TS$.reset():
    $Y$.write($\perp$)
```

- Can be extended when $\Delta$ is unknown

## The consensus problem

- Each process $p_i$ proposes a value $(v_i)$
- Properties:
  - ⋆ Validity: a decided value is a proposed value
  - ⋆ Agreement: no two processes decide different values
  - ⋆ Termination: every non-faulty process decides
- Wait-free: termination has to be ensured whatever the number of process crashes
- Consensus universality: Atomic registers and consensus objects allow wait-free implementing any object that has a sequential specification
- No solution in asynchronous RW SM systems

## Indulgent consensus with known bound

```
**operation** consensus($v_i$):
        **while** ($Y$.read($\Delta$) $= \perp$) **do** $Y$.write($v_i$) **end while**;
        delay($\Delta$);
        return($Y$.read($\infty$))
```

Simple, but not fast!

Aim: allow for fast decision in good circumstances

Good circumstances: Here, when
a single value is proposed and there is no timing failure

## Fast indulgent consensus with known bound

$X[1..b]$ of boolean values initialized to $[\mathit{false}, \ldots, \mathit{false}]$

$X[v] \Leftrightarrow$ the value $v$ has been proposed

```
operation consensus($v_i$):
        $X[v_i] \leftarrow \mathit{true}$;
        while($Y$.read($\Delta$) $= \perp$) do $Y$.write($v_i$) end while;
        if $(\exists v : v \neq v_i \wedge X[v])$ then delay($\Delta$) end if;
        return($Y$.read($\infty$))
```

When a single value is proposed: no process is delayed

No timing failure: 2/3 accesses to $Y$, $b$ accesses to $X[1..b]$ (4/5 accesses for boolean proposals)

## Fast indulgent consensus with unknown bound

Shared array of 1WnR atompic regsiters $DELAY[1..N]$

$DELAY[i]$: $p_i$'s curent approximation of $\Delta$

```
operation consensus($v_i$):
    $X[v_i] \leftarrow true$;
    while ($Y$.read($d_i$) $= \perp$) do
        if $\neg$($Y$.write($v_i$))
            then $d_i \leftarrow d_i + 1$; $DELAY[i] \leftarrow d_i$ end if
    end while;
    if ($\exists v : v \neq v_i \wedge X[v]$)
        then delay(max($\{DELAY[k]_{1 \leq k \leq n}\}$)) end if;
    return($Y$.read($\infty$))
```

## Part V

# TIMED REGISTERS in PERSPECTIVE

## Timed registers vs Sticky bits (Plotkin)

- A sticky bit is initialized to $\perp$, can then contain 0 or 1
- A write returns $\mathit{false}$ if the value it is trying to write disagree with the already written value, otherwise it returns $\mathit{true}$
- Sticky bits and timed registers are universal objects
- Sticky bits and timed registers have different types:
  - ⋆ Sticky bits are write-once objects that are not time-constrained
  - ⋆ Timed registers are not write-once, but their writes are time constrained

## Obstruction-freeness and abortable objects

- Obstruction-free property:
  - ⋆ Safety is never violated
  - ⋆ Liveness is guaranteed (only) when a process executes alone (i.e., in the absence of concurrency)
- Abortable objects:
  - ⋆ An abortable object behaves as an ordinary object when it is accessed sequentially, but an operation may return $\perp$ when the object is accessed concurrently

## Obstruction-free/abort. objects *vs* Timed Reg.

- Obstruction-freeness, abortable objects:
  - ⋆ In both cases, the "undesirable" behavior occurs in presence of concurrency
  - ⋆ These notions are contention-oriented
  - ⋆ Liveness can be ensured by using additional contention managers (e.g., $\Omega$, $\diamond\mathcal{P}$)
  - ⋆ The progress of a process depends on the others
- Timed registers:
  - ⋆ Independent of the concurrency degree, of the (speed of the) other processes
  - ⋆ This notion is time-oriented
  - ⋆ When satisfied, the assumption $\Delta$: ensures liveness
  - ⋆ The progress of a process depends only on itself

## Obstruction-free/abort. objects *vs* Timed Reg.

- Some duality
- Both a contention manager and the assumption $\Delta$ are SCHEDULERS that provide appropriate fairness rules in order operations issued by the processes terminate

## Conclusion

- Notion of timed register (new object type)
- Indulgent synchronization, Indulgent agreement
- Universal object when $\Delta$ is eventually satisfied
- Duality wrt obstruction-freedom, abortable objects

## Conclusion cont'd

- Consider other timed objects (queues, stacks, etc.)
- Address other problems (e.g., fast mutex)
- What when the timed registers are faulty?