Power and Limit of Dist Computing Shared Memory Models

Sergio Rajsbaum[†] **Michel Raynal**[⋆]

[†] Dept of Mathematics, UNAM, Mexico City, Mexico

[⋆] Institut Universitaire de France & IRISA Rennes, France

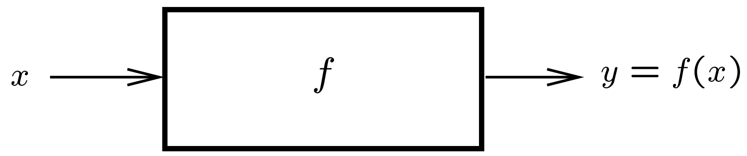
Table of content

Part I

Sequential Computing vs Distributed Computing

Sequential computing

- Power and limit of sequential computing
- Notion of a function: $y = f(x)$



- Notion of a computable function
- Several formalisms:
  Turing machine,
  Post system,
  Church's lambda calculus, etc.

What is distributed computing about?

- Real-time: masters On-time computing
- Parallelism: provides Efficiency
- Distributed computing:

$\boxed{\text{masters Uncertainty}}$

(We are -more or less- implicitly using a lot of heuristics!)

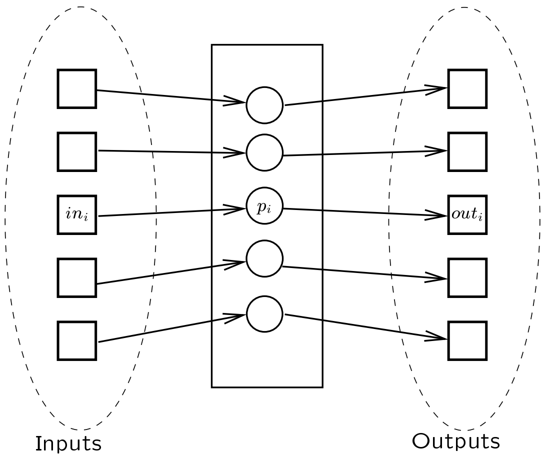
- Raynal M., Communication and Agreement Abstractions for Fault-Tolerant Asynchronous Distributed Systems. *Morgan & Claypool Publishers*, 251 pages, 2010 (ISBN 978-1-60845-293-4)

Uncertainty is created by

- Multiple loci of control
- Asynchrony (vs Synchrony)
- Failures (Failure models)
- Process mobility (and related stuff)
- Low computing capacity, bandwidth
- Dynamicity
- Etc., etc., ..., etc.!

The notion of a task in DC

The DC counterpart of a function



The notion of a task in DC

- A task T is a triple $(\mathcal{I}, \mathcal{O}, \Delta)$
  - ⋆ $\mathcal{I}$: set of input vectors (of size n)
  - ⋆ $\mathcal{O}$: set of output vectors (of size n)
  - ⋆ Δ: relation from $\mathcal{I}$ into $\mathcal{O}$: $\forall I \in \mathcal{I} : \Delta(I) \subseteq \mathcal{O}$
- $I[i]$: private input of p_i
- $O[i]$: private output of p_i
- $\forall I \in \mathcal{I}$:
  $\Delta(I)$ defines the set of legal output vectors that can be decided from the input vector I

Solving a task

A distributed algorithm A is a set of n local automata (Turing machines) that cooperate through specific communication objects (e.g., message-passing network, shared memory, etc.)

An algorithm A solves a task T if in any run

- $\exists\ I \in \mathcal{I}$ such that each p_i starts with (proposes) $in_i = I[i]$
- $\exists\ O \in \mathcal{O}$ such that $O[j] = out_j$ for each process p_j that computes (decides) an output out_j

Examples of tasks

- Consensus and k-set agreement
  - ⋆ Binary consensus:

    $\mathcal{I} = \{$all vectors of 0 and 1$\}$

    $\mathcal{O} = \{\{0,\dots,0\},\{1,\dots,1\}\}$

    Let $X_0 = \{0,\dots,0\}$ and $X_0 = \{1,\dots,1\}$

    $\Delta($any vector but $X_O, X_1) = \mathcal{O}$
    $\Delta(X_0) = \{0,\dots,0\}$ and $\Delta(X_1) = \{1,\dots,1\}$.
- Renaming
- Weak symmetry breaking
- k-Simultaneous consensus
- Etc.

Type of a task

- Colorless: if a value is decided by a process it can be decided by any other process in the same run
- Colored: otherwise

Part II

Base read/write wait-free model

Processes, atomic registers and wait-freedom

- n sequential asynchronous processes: $p_1, \ldots, p_n$
- Communicate through atomic read/write registers
- Up to $n-1$ processes may crash in a run (wait-free)
  - ⋆ If a process crashes in a run it is faulty in that run, otherwise it is correct in that run
  - ⋆ An object implementation is *wait-free* if any invocation of any of its operations always terminates (i.e., whatever the behavior of the other processes) when the invoking process is correct

- Herlihy M.P., Wait-Free Synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124-149, 1991

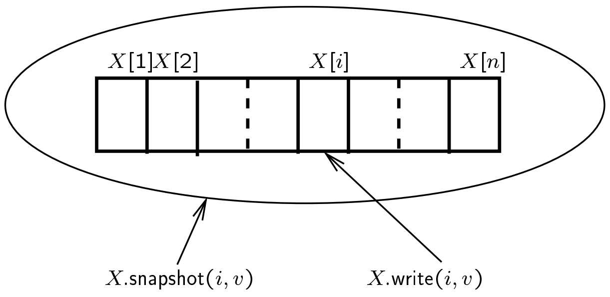
The fundamental issue

- Impossibility for a given process p_i to know if another process p_j has crashed or is only very slow (generates a lot of impossibilities)
- Due to the net effect of asynchrony and crashes the DC model is weaker than a Turing machine!

  A lot of tasks cannot be wait-free solved in this DC model while they can in a reliable DC system
- $\Rightarrow$ a single reliable Turing machine is strictly stronger than a set of crash-prone asynchronous Turing machines that communicate by reading/writing a shared memory (or by exchanging messages through a reliable network)

A useful abstraction: snapshot object (1)

- Array $X[1..n]$ of 1WMR registers (only p_i can write $X[i]$)
- Two operations: X.write(i, v) and X.snapshot$()$



- Afek Y., Attiya H., Dolev D., Gafni E., Merritt M. and Shavit N., Atomic Snapshots of Shared Memory. *Journal of the ACM*, 40(4):873-890, 1993

- Anderson J., Multi-writer Composite Registers. *Distributed Computing*, 7(4):175-195, 1994

A useful abstraction: snapshot object (2)

- The invocations of X.write() and X.snapshot() are atomic (linearizable): they appear as if they have been issued sequentially in and order that respects their real-time execution order
- Can be wait-free built on top of atomic R/W registers
- Hence, same computational power as the base R/W wait-free model

Part III

The iterated snapshot model

- Borowsky E. and Gafni E., Immediate Atomic Snapshots and Fast Renaming. *Proc. 12th ACM Symposium on Principles of Distributed Computing (PODC'93)*, pp. 41-51, 1993

- Borowsky E. and Gafni E., A Simple Algorithmically Reasoned Characterization of Wait-free Computations. *Proc. 16th ACM Symposium on Principles of Distributed Computing (PODC'97)*, ACM Press, pp. 189-198, 1997

Aim

- Focus on round-based computations
- In an asynchronous round: a process p_i writes + snapshots and proceeds to the next round
- Algorithm skeleton for process p_i:

$r_i \leftarrow 0$;
loop forever $r_i \leftarrow r_i + 1$;
local computations; compute v_i;
$\boxed{sm_i \leftarrow WS[r_i].\mathsf{write\_snapshot}(v_i)}$;
local computations
end loop.

One-shot write-snapshot object

- Simple and elegant round-by-round structure
- A new write-snapshot object is associated with each round: $WS[1]$, $WS[2]$, ...
- Write and snapshot are pieced together into a single operation denoted $\mathsf{write\_snapshot}()$
- Let sm_i denote the value returned by $\mathsf{write\_snapshot}()$
- sm_i is a set of pairs (k, v_k) that means that p_k has written v_k into the corresponding write-snapshot object

The $WS.\mathsf{write\_snapshot}()$ operation

- Self-inclusion. $\forall i : (i, v_i) \in sm_i$
- Containment. $\forall i, j : sm_i \subseteq sm_j \ \vee \ sm_j \subseteq sm_i$
- Immediacy.
$\forall i, j : [(i, v_i) \in sm_j \ \wedge \ (j, v_j) \in sm_i] \ \Rightarrow \ (sm_i = sm_j)$.
- Termination.
Any invocation of $WS.\mathsf{write\_snapshot}()$ by a correct process terminates

A fundamental result

- A task is *bounded* if its set of input vectors $\mathcal{I}$ is finite
- Borowsky-Gafni'97
  Let T be a bounded task.

  T can be wait-free solved in the base read/write shared memory model $\Leftrightarrow$ T can be wait-free solved in the iterated model
- Hence, the iterated model restricts the set of interleavings but does not restricts the computational power!

- Borowsky E. and Gafni E., A Simple Algorithmically Reasoned Characterization of Wait-free Computations. *Proc. 16th ACM Symposium on Principles of Distributed Computing (PODC'97)*, ACM Press, pp. 189-198, 1997

The underlying mathematical structure

- Topology:
  simplicial complexes (set closed under containment)
- Easy way to capture all runs
- Examples with two and three processes

- Borowsky E. and Gafni E., Generalized FLP Impossibility Results for t-Resilient Asynchronous Computations. *Proc. 25th ACM Symposium on Theory of Computing (STOC'93)*, ACM Press, pp. 91-100, 1993

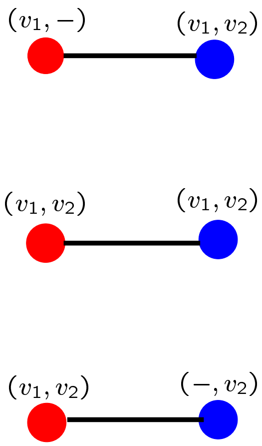
Herlihy M.P. and Shavit N., -The Topological Structure of Asynchronous Computability. *Journal of the ACM*, 46(6):858-923, 1999

An example with two processes (1)

p_1: write_snapshot(v_1)
p_2: write_snapshot(v_2)

p_1: write_snapshot(v_1)
p_2: write_snapshot(v_2)

p_1: write_snapshot(v_1)
p_2: write_snapshot(v_2)

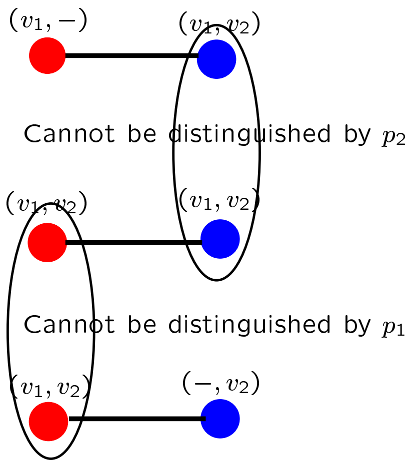


An example with two processes (2)

p_1: write_snapshot(v_1)
p_2: write_snapshot(v_2)

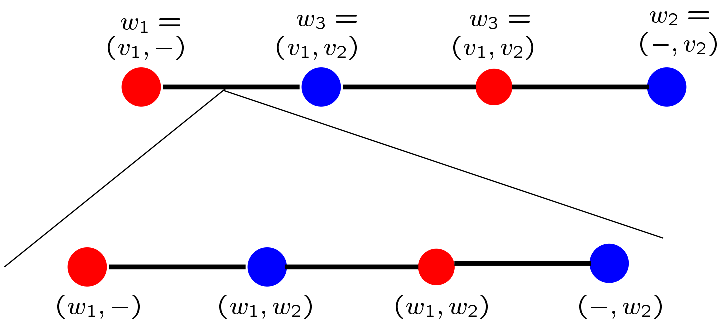
p_1: write_snapshot(v_1)
p_2: write_snapshot(v_2)

p_1: write_snapshot(v_1)
p_2: write_snapshot(v_2)

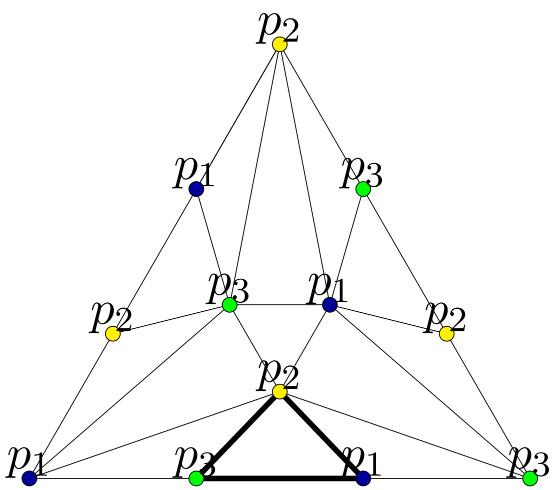


From the first to the second round

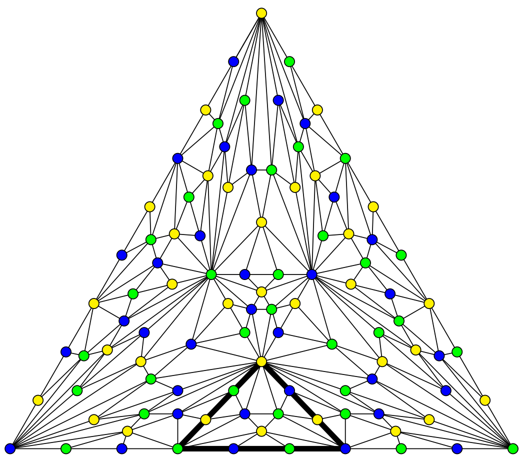
- Complex: subdivision of an edge (simplexes)
- More rounds: finer subdivision



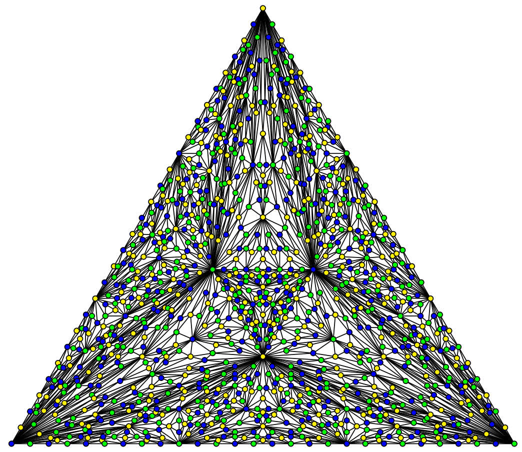
Three processes: First round



Three processes: Second round



[Three processes: Third round, and then etc.



Iterated model with failure detectors

- A failure detector is a distributed oracle that provides processes with additional information related to failed processes and can consequently enrich the computability power of a the underlying system
- Example: Ω
  - ⋆ Each process has a read-only variable $leader_i$ such that after an arbitrary but finite period the variables $leader_i$ of all correct processes p_i contain the same id of a correct process
  - ⋆ Weakest failure detector that allows consensus to be solved in the base R/W wait-free model

- Chandra T., Hadzilacos V. and Toueg S., The Weakest Failure Detector for Solving Consensus. *Journal of the ACM*, 43(4):685-722, 1996

A question

- Let the base read/write wait-free shared memory model and the write-snapshot iterated model both enriched with the same failure detector.
- Question:
  Have both systems the same computability power?

- Rajsbaum S., Raynal M. and Travers C., An Impossibility about Failure Detectors in the Iterated Immediate Snapshot Model. *Information Processing Letters*, 108(3):160-164, 2008

The answer to the previous question

- Theorem:
  The base read/write wait-free shared memory model and the write-snapshot iterated model both enriched with the same failure detector do not have the same computational power
- Actually the additional power offered by a failure detector cannot be used in the iterated model enriched with the very same failure detector
- Why?
  Because the iterated model is too much structured to benefit from the help of a failure detector
- How to circumvent this negative result?
  The iterated restricted model

- Rajsbaum S., Raynal M. and Travers C., The Iterated Restricted Immediate Snapshot (IRIS) Model. *14th Int'l Computing and Combinatorics Conference (COCOON'08)*, Springer-Verlag LNCS 5092, pp.487-496, 2008

The property to obtain the power of Ω

$$PR(\Omega) \equiv (\exists \ell, \exists r : \forall r' \geq r : sm_\ell^{r'} = \{\ell\}).$$

- $\exists$ a round r and a process p_ℓ such that, at every round $r' \geq r$, any invocation $sm_i \leftarrow IS[r']$.write_snapshot() sees ℓ in its view sm_i (i.e., $\ell \in sm_i$)
- Said differently, whatever the concurrency degree among the $IS[r']$.write_snapshot() invocations issued by processes during r', the invocation issued by p_ℓ is always (set-)linearized alone and before the other invocations

$\boxed{\text{IWS} + PR(\Omega) \simeq \text{Base model} + \Omega}$

IWS + $PR(\Omega)$ at work: consensus

- Each process p_i proposes a value v_i
- Each correct process p_i
  - ⋆ Decides a value (termination) such that
  - ⋆ A decided value is a proposed value (validity) and
  - ⋆ No two processes decide different values (agreement)

IWS + $PR(\Omega)$ at work: algorithm

$r_i \leftarrow 0$; $est_i \leftarrow v_i$; $dec_i \leftarrow \perp$;
loop forever
    $r_i \leftarrow r_i + 1$;
    $sm_i \leftarrow WS[r_i]$.write_snapshot($\langle i, est_i, dec_i \rangle$);
    $r_i \leftarrow r_i + 1$;
    $tm_i \leftarrow WS[r_i]$.write_snapshot(sm_i);
    if $(\exists sm : (sm \in tm_i) \wedge (\langle -, -, dec \rangle \in sm$ with $dec \neq \perp))$
      then if $dec_i = \perp$ **then** $est_i \leftarrow dec$; $dec_i \leftarrow dec$ **end if**
      else if $(\exists sm : (sm \in tm_i) \wedge (sm = \{\langle -, est, - \rangle\}))$
          then $est_i \leftarrow est$;
            if $tm_i = \{sm\} \wedge sm = \{\langle i, est, - \rangle\}$
              then $dec_i \leftarrow est_i$ **end if**
        end if
    end if
end loop.

Part IV

From Wait-freedom to adversaries

Main idea

- A progress condition characterizes (defines) the set of runs for which every correct process has to return from its operation invocations (termination)
  - ⋆ Wait-freedom
  - ⋆ Obstruction-freedom, Non-blocking
  - ⋆ t-Resilience
  - ⋆ Core and survivors sets
  - ⋆ Adversaries
- The implementation of an object satisfies a given progress condition if ...

Wait-freedom

- An operation invocation by a correct process always terminates (i.e., whatever the behavior of the other processes)
- The less restrictive (most powerful) progress condition

- Herlihy M.P., Wait-Free Synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124-149, 1991

Obstruction-freedom

- A correct process is required to terminate its operation invocation if it executes "alone during a long enough period"
-
- Obstruction-freedom: progress cond. wrt concurrency This means progress has to be guaranteed in concurrency-free setting
- Weaker than wait-freedom

- Herlihy M.P., Luchangco V. and Moir M., Obstruction-free Synchronization: Double-ended Queues as an Example. *Proc. 23th Int'l IEEE Conference on Distributed Computing Systems (ICDCS'03)*, pp. 522-529, 2003

- Imbs D., Raynal M. and Taubenfeld G., On asymmetric progress conditions. *29th Annual ACM Symposium on Principles of Distributed Computing (PODC'10)*, ACM Press, pp. 55-64, 2010

- Taubenfeld G., *The Computational Structure of Progress Conditions (DISC'10)*, Springer Verlag LNCS 6343, pp. 221-235, 2010

t-Resilience

- Considers models including an upper bound t on the number of faulty processes
- Any operation by a correct process has to terminate in all runs where at most t process crashes (i.e., at least $(n-t)$ processes are correct)
- Wait-freedom is $(n-1)$-resilience

Core and survivors sets (1)

- t-resilience considers implicitly that failure are independent
- Core and survivors sets allow for dependent failures
- A core C is a minimal set of processes such that in any run at least one process in C does not fail
- A survivor set S is a minimal set of processes such that there is a run in which the set of correct processes is exactly the set S
- Cores and survivor sets are dual notions

Core and survivors sets (1)

- Example with 4 processes p_1, p_2, p_3 and p_4
  - ⋆ Let us define two cores: $\{p_1, p_2\}$ and $\{p_3, p_4\}$
  - ⋆ Survivor sets are $\{p_1, p_3\}$, $\{p_1, p_4\}$, $\{p_2, p_3\}$ and $\{p_3, p_4\}$

- Junqueira F. and Marzullo K., Designing Algorithms for Dependent Process Failures. *Future Directions in Distributed Computing*, Springer-Verlag, #LNCS 2584, pp. 24-28, 2003

Adversaries

- The most general notion
- An adversary A is a set of sets of processes such that each correct process has to terminate its operations when the set of correct processes is exactly a set of A
- Example with 4 processes p_1, p_2, p_3 and p_4
  - ⋆ $A = \{\{p_1, p_2\}, \{p_3, p_4\}, \{p_1, p_3, p_4\}\}$
  - ⋆ Termination is required when the set of correct processes is exactly a set of A
  - ⋆ No requirement when the set of correct processes is (for example) $\{p_1, p_4\}$ or $\{p_1, p_2, p_3\}$

- Delporte-Gallet C., Fauconnier H., Guerraoui R. and Tielmann A., The Disagreement Power of an Adversary. *Proc. 23th Int'l Symposium on Distributed Computing (DISC'09)*, Springer-Verlag LNCS #5805, pp. 8-21, 2009

Part V

The Borowsky-Gafni (BG) simulation

- Borowsky E. and Gafni E., Immediate Atomic Snapshots and Fast Renaming. *Proc. 12th ACM Symposium on Principles of Distributed Computing (PODC'93)*, pp. 41-51, 1993

- Gafni E., The Extended BG Simulation and the Characterization of t-Resiliency. *Proc. 41th ACM Symposium on Theory of Computing (STOC09)*, ACM Press, pp. 85-92, 2009

- Imbs D. and Raynal M., Visiting Gafni's Reduction Land: from the BG Simulation to the Extended BG Simulation. *Proc. 11th Int'l Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS'09)*, Springer-Verlag LNCS #5873, pp. 369-383, 2009

Aim

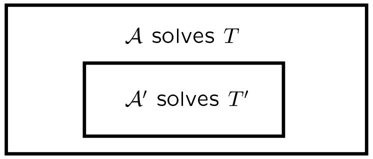
- Answers the fundamental question
  Which tasks are t-resiliently solvable?
- To that end
  - ⋆ Simulates t-resilient adversaries in the wait-free model
  - ⋆ Reduces (questions about) t-resilience
    to (questions about) wait-freedom
  - ⋆ Characterizes t-resilience in terms of wait-freedom

Motivations

- Consensus:
  - ⋆ Consensus is not solvable in a read/write model with $n = 2$ processes where any of them may crash
  - ⋆ What about consensus in a system with $n = 100$ and $t = 1$?
- (n, k)-set agreement (assume $t < 12 \leq n$):
  - ⋆ Given an algorithm that solves $(14, 4)$-set agreement
  - ⋆ It is possible to solve $(12, 5)$-set agreement?
- The BG simulation allows for reductions that answer these questions!

Reduction

- A task T can be reduced to a task T' if there is an algorithm $\mathcal{A}$ that solves T using as a subroutine any algorithm $\mathcal{A}'$ that solves T'
- Notation $T' \succeq T$



- T is impossible $\Rightarrow$ T' is impossible

Borowsky-Gafni's simulation

- We are looking for an algorithm $\mathcal{A}$ that wait-free solves T over $t+1$ processes
- We know/assume an algorithm $\mathcal{A}'$ that t-resiliently solves T over n processes
- The Borowsky-Gafni's simulation algorithm provides us with such an algorithm $\mathcal{A}$

A more general simulation

- Let $ASM(n, t, x)$ be a read/write shared memory system enriched with object with consensus number x
- The BG shows that $ASM(n, t, 1) \simeq ASM(t + 1, t, 1)$
- Question: when do we have $ASM(n_1, t_1, x_1) \simeq ASM(n_2, t_2, x_2)$?
- Answer: the multiplicative power of consensus numbers:

$$(ASM(n_1, t_1, x_1) \simeq ASM(n_2, t_2, x_2) \Leftrightarrow \lfloor \frac{t_1}{x_1} \rfloor = \lfloor \frac{t_2}{x_2} \rfloor$$

- Particular case: $ASM(n, t, 1) \simeq ASM(n, t \times x, x)$

- Imbs D. and Raynal M., The Multiplicative Power of Consensus Numbers. *Proc. 29th ACM Symposium on Principles of Distributed Computing (PODC'10)*, ACM Press, pp. 26-35, 2010

Part V

Conclusion

- The aim was to understand the power, subtleties and limits of crash-prone asynchronous distributed shared memory models
- A "Holy Grail" quest: have a view as clear as what we have in sequential computing wrt to computability, complexity and languages hierarchy

The only slide to remember

Failures do modify

- Our view of synchronization
- The way synchronization has to be solved