Wait-free Concurrent Objects

An introduction for the sophomore

Michel RAYNAL

raynal@irisa.fr

IRISA
Campus Universitaire de Beaulieu, 35042 Rennes, France

Part I

To start: A partial, questionable, etc.

VIEW

of computer science and distributed computing

My view

- What is research?

  It is an adventure, both personal and collective, of intellectual nature

  (It is not -*only*- "joint venture" ...)

- What is teaching?

  "To teach is to think in front of students"

  H. Lebesgue (1875-1941)

  (It is not -*only*- "quiz exercise" ...)

My view

- We are a community with its way of thinking, its social rules, etc.
- We have our "family meetings" with close family, with cousins, etc. (e.g., PODC, DISC, OPODIS, SIROCCO, ICDCS, etc.)
- We have our history (and its "victories"), our great dates/events, our feasts and celebrations (e.g., Dijkstra prize)
- Etc.

"Our" community, a few years ago...

My view of our discipline (1)

- *Informatics*:
  The meeting point of mathematics and technology
- Its components:
  - ⋆ Computer science is to understand
  - ⋆ Computer engineering is to build
- "Time is to clock what spirit is to brain"

My view of our discipline (2)

- Informatics is a science of *abstraction*:

  We are concerned with
  - ⋆ Creating the right model for a problem, and
  - ⋆ Designing the appropriate mechanizable techniques to solve it
- Theory [see Fischer and Merritt, DC 2003]
  - ⋆ Provides framework for thinking about pbs, methods, and techniques
  - ⋆ Codifies knowledge (obtained in one pb domain) in order it can be transmitted (to others and applied to new problem domains)

My view of our discipline (3)

- When something works, we must know why it does work
- When something does not work, we must know why it does not work

Design principles and correctness
may be theoretical ... ... but ...

incorrectness has practical impact

M. Herlihy

My view of my job: in academia (1)

Voluntarily a little bit provocative!

- We are paid to teach and to think!
- Which means to understand problems

"Make it as simple as possible ... but no more"

(A. Einstein)

"I am sorry for having written such a long letter, I had no time to write a shorter one"

Blaise Pascal

My view of my job: in academia (2)

Still a little bit provocative!

- "Time to market" is important ... but is not our first concern ...
- Our real impact on industry are our students who work in industry and make industry to move to "new" knowledge and better practices
- "It is not in improving the candle technology that electricity was discovered, understood and mastered"

My view of my job: in academia (3)

Again a little bit provocative!

- A scholar learns how to write by reading great authors, ... not by reading the explanatory leaflet of washing machines
- Similarly, we have to give students a global view of our domain, teach them great problems, great solutions and ... results of great scientists (see ACM-IEEE curricula)
- So, our community has to produce nice textbooks (there are not enough)

Success stories

- FSA, Pushdown automata, Turing machines
  Hierarchy wrt to computability
- NP-completeness: Hierarchy wrt efficiency
- Safe, regular, atomic registers
  Lamport's register hierarchy
- Consensus number and Robustness
  Herlihy hierarchy synchronization power

My view of distributed computing (1)

- A birth certificate: *Time, clocks and the ordering of events in a distributed system*, Leslie Lamport, CACM 1978
- DC arises when one has to solve a problem in terms of entities (processes, agents, sensors, peers, actors, nodes, processors, ...) such that each entity has only a partial knowledge of the many parameters involved in the problem that has to be solved
- Finding models that are realistic while remaining abstract enough to be tractable was, is and remains a real challenge!

My view of distributed computing (2)

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

masters Uncertainty

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

DC is the value added on top of networks

My view of distributed computing (3)

Uncertainty is created by:

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

Part II

Before visiting
WAIT-FREE COMPUTING

MODEL and DEFINITIONS

- Raynal M., Synchronization is coming back, but is it the same? Invited talk, *Proc. 22nd Int'l IEEE Conference on Advanced Information Networking and Applications (AINA 2008)*, IEEE Computer Press, Okinawa (Japan), March 2008

Summary

- Computing Model: Processes and Concurrent Objects
- Safety versus Progress
- Linearizability
- Progress: From obstruction-freedom to wait-freedom
- Examples: Renaming, Store-collect, snapshot
- Hybrid algorithms (failure-free systems)
- Universality of Consensus
- Implementing a Consensus Object
- From a weaker to a stronger property
- Building t-Resilient objects and Graceful degradation
- Conclusion

Asynchronous process model

- A set Π of n processes $p_1, \ldots, p_n$
- Timing model: Asynchrony: No upper bound on the time required to execute a computation step
- Failure model: Process crash: a process behaves according to its specification until it possibly crashes, i.e., halts prematurely (after it has crashed a process is definitely stopped)
- Terminology wrt a run:
  - ⋆ A Correct process is a p_i that never crashes
  - ⋆ A Faulty process is a p_i that crashes

Asynchronous Base Communication Model

Shared memory = shared reliable, 1W*R, atomic registers

- Reliability means here that a register never crashes
- 1W*R means that each register has a single writer (statically defined), but can be read by all the process
- atomicity:
  - ⋆ The read and write operations appear as if they are executed sequentially
  - ⋆ Each operation appears as being executed instantaneously at some point of the time line between its start event and its end event

Asynchronous Base Communication Model cont'd

Reminder:

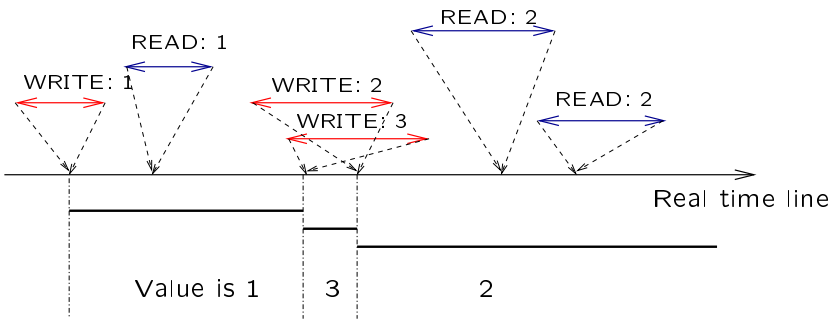
Shared memory can be implemented on top of a message-passing system prone to up to t crashes iff $t < n/2$

- Attiya H., Bar-Noy A. and Dolev D., Sharing Memory Robustly in Message-Passing Systems. *Journal of the ACM*, 42(1): 124-142, 1995

- Attiya H. and Welch J., *Distributed Computing: Fundamentals,Simulations and Advanced Topics,* (2d Edition), Wiley-Interscience, 414 pages, 2004

- Lynch, N. A., *Distributed Algorithms,* Morgan Kaufmann, 872 pages, 1997

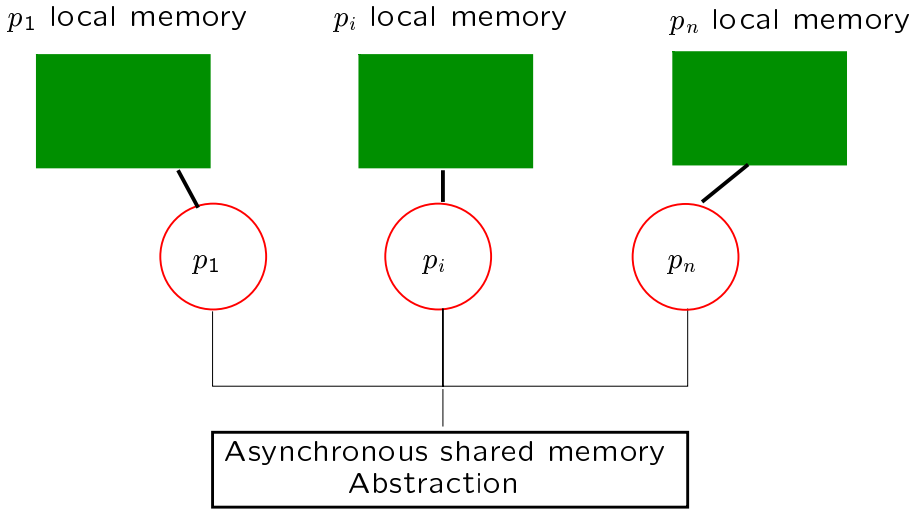
Atomic register



- Lamport L., On interprocess Communication (part 1: Basic Formalism, Part 2: Algorithms), *Distributed Computing*, 1(2):77-101, 1986

- Herlihy M.P. and Wing J.L., Linearizability: a Correctness Condition for Concurrent Objects. *ACM Trans. on Programming Languages and Systems*, 12(3):463-492, 1990

Structural view (Base level)



(could be shared disks, SAN)

SAFETY PROPERTY

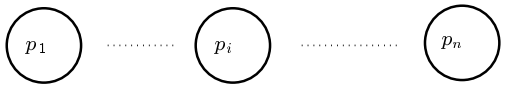
LINEARIZABILITY

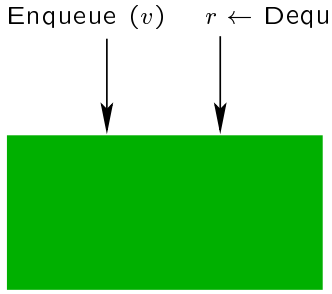
- Herlihy M.P. and Wing J.M., Linearizability: a correctness condition for concurrent objects. *ACM Toplas*, 12(3):463-492, 1990

- Lamport L., On interprocess Communication (part 1: Basic Formalism, Part 2: Algorithms), *Distributed Computing*, 1(2):77-101, 1986

Concurrent Object: example

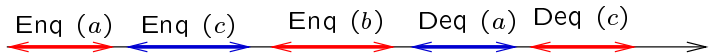
An object accessed by concurrent processes



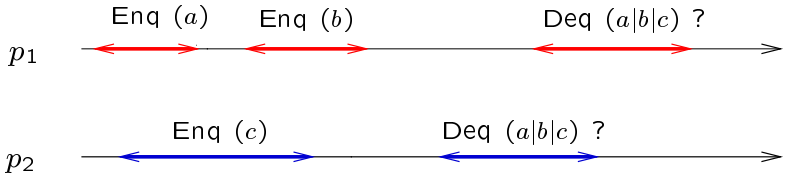


Sequential vs Concurrent (1)

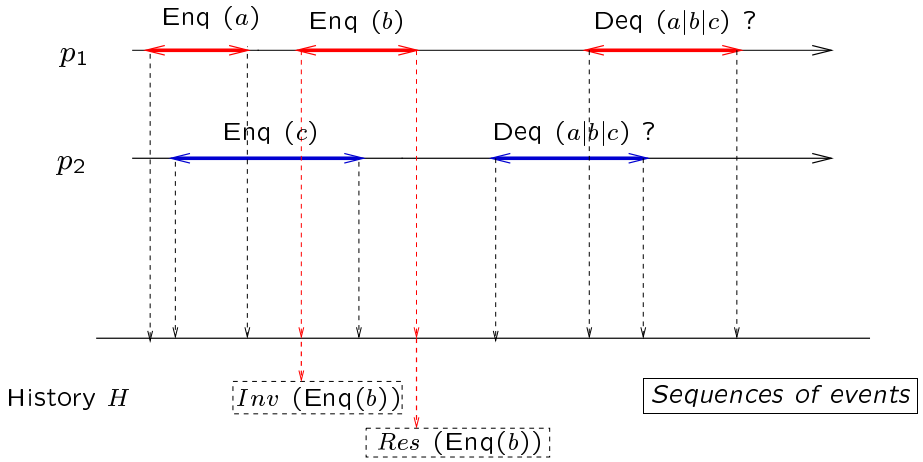
SEQUENTIAL:



CONCURRENT:



History (Run, Execution)



A history defines a partial order on the operations

Two types of Concurrent Objects

- A concurrent object encapsulates a particular synchro problem
- Concurrent objects with a sequential specification

  Queue, file, graph, tree, stack, ...

  ⋆ Fifo queue: producer/consumer problem
  ⋆ File (register): Readers/writers problem
- Concurrent objects with a not-sequential specification

  Rendezvous Object,
  Interactive Consistency,
  Non Blocking Atomic Commit Object, ...

Interactive Consistency

Each process p_i proposes a value v_i and decides on a vector of values such that:

- IC-Termination: Every correct process eventually decides on a vector
- IC-Validity: Any decided vector D is such that $D[i] \in \{v_i, \perp\}$, and is v_i if p_i does not crash
- IC-Agreement: No two processes decide different vectors

Pease L., Shostak R. and Lamport L., Reaching Agreement in Presence of Faults, *Journal of the ACM*, 27(2):228-234, 1980

A non-sequential Concurrent Object: NBAC

- NBAC-Termination: Every correct process eventually decides
- NBAC-Validity: A decided value is *commit* or *abort*. Moreover:
  - ⋆ NBAC-Justification: If a process decides *commit*, all processes have voted *yes*
  - ⋆ NBAC-Obligation: If all processes vote *yes* and there is no crash, then the decision value is *commit*
- NBAC-Agreement: No two processes decide differently

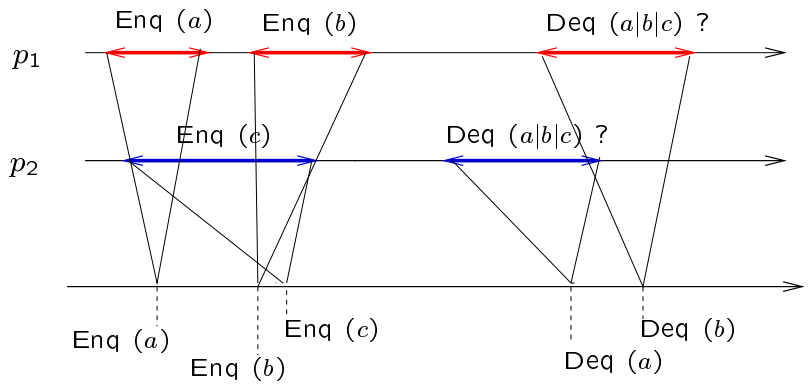
So, from *abort* we can conclude that a process has voted *no* or crashed. Moreover,*commit* can be decided despite crashes

Linearizability

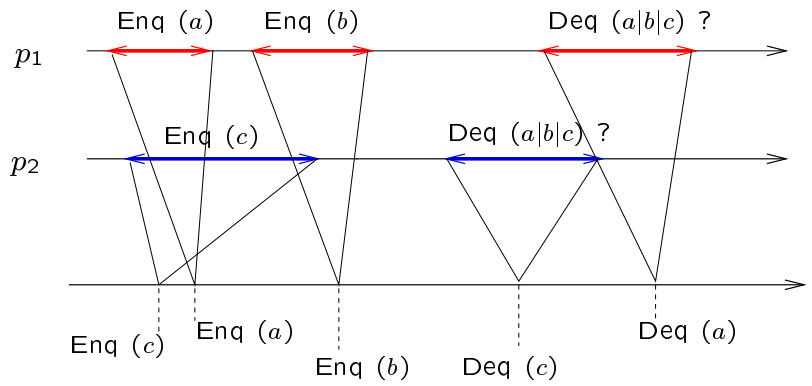
An execution (or "history") H is linearizable if the operations issued by the processes appear as if they have been executed in some sequential order that respects:

- The seq specification of the object
- The real-time order of the non-overlapping operations

Sequential vs Concurrent (2)



Sequential vs Concurrent (3)



Locality of the Linearizability property (1)

- A property P of a concurrent system is Local if the system as a whole satisfies P whenever each individual object satisfies P

  Locality means that, given two objects X and Y, each satisfying the property P, the composite object $[X, Y]$ satisfies the property P

- Linearizability is a local property (consistency criterion)

Locality of the Linearizability property (2)

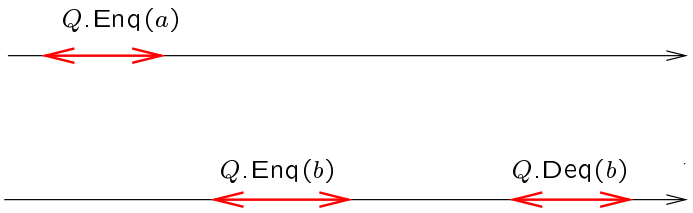
- Theorem: A history H is linearizable iff, for each object X, H restricted to the operations on X is linearizable

  Locality means that, given two linearizable objects X and Y, the composite object $[X, Y]$ is linearizable

- It follows that each object can be implemented independently from the others (this is a fundamental property from both theoretical and practical point of views)
- Sequential consistency, some forms of serializability are not local properties

Sequential Consistency

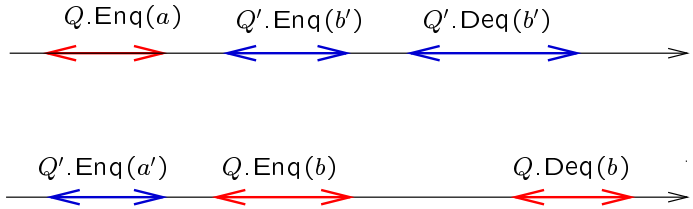
- An execution (or "history") H is sequentially consistent if the operations issued by the processes appear as if they have been executed in some sequential order that respects the seq specification of the object (and process order)



A "witness" seq history:

$Q.\mathrm{Enq}(b)$ $Q.\mathrm{Enq}(a)$ $Q.\mathrm{Deq}(b)$

Sequential Consistency is not Local



- Q and Q' are sequentially consistent
- the whole history H is not sequentially consistent

  Impossibility to produce a "witness" sequential history

- See distributed caches

Linearizability is a Non-Blocking Property (1)

- An operation is total if it always defined

  Examples:

  Read and Write a file are total operations

  $Q.Enq(a)$ is total as soon as the capacity of Q is infinite

- Given an operation (e.g., (Write(x, v)) let:
  - ⋆ Inv (Write(x, v)): start event
  - ⋆ Res (Write(x, v)): terminating event
- Non-Blocking means that a pending invocation of a total operation is never required to wait for another pending invocation to complete

Linearizability is a Non-Blocking Property (2)

- Theorem: Linearizability is a Non-Blocking Property
- This means that linearizability per se never forces a process with a pending invocation to block
- More specifically
  - ⋆ If blocking (or deadlock) occurs, it is due to the particular implementation. It is not inherent to the correctness property itself
- This suggests that linearizability is an appropriate correctness criterion for systems where concurrency and realtime responses are important
- Remark: Serializability is not a non-blocking property

Part III

PROGRESS PROPERTIES

OBSTRUCTION-FREEDOM

NON-BLOCKING

WAIT-FREEDOM

Classical Lock-based Implementations

- Use a lock-based solution: one process at a time can access a given object (semaphores or monitors to implement critical sections)
- Make processes depend one from the others
  - ⋆ Deadlock
  - ⋆ Asynchrony and Failures
  - ⋆ Process scheduling (swap-out/interrupts/...)
- Hence, lock-based solutions:
  - ⋆ Are deadlock-prone, Starvation-prone
  - ⋆ Can entail inefficient executions (waiting chains)

Object liveness

- Obstruction-freedom: if I am alone during a long enough period, I'll terminate my operation (termination when no concurrency)

  Herlihy M.P., Luchangco V. and Moir M., Obstruction-free synchronization: double-ended queues as an example. *Proc. 23th IEEE ICDCS*, pp. 522-529, 2003

- Non-blocking: deadlock freedom despite asynchrony and process crashes

Object liveness cont'd

- Wait-freedom: starvation-freedom despite asynchrony and process crashes

  Basic papers:

  - Lamport L., Concurrent Reading and Writing. *Communications of the ACM*, 20(11):806-811, 1977

  - Herlihy M.P., Wait-Free Synchronization. *ACM Toplas*, 13(1):124-149, 1991

  The progress of a process depends only on it. This means that, even if all but one process crash, the remaining process is not prevented from terminating

  An introductive paper for the sophomore:

  - Raynal M., Wait-Free Computing: an Introductory Lecture. *Future Generation Computer Systems*, 21(5):655-663, 2005

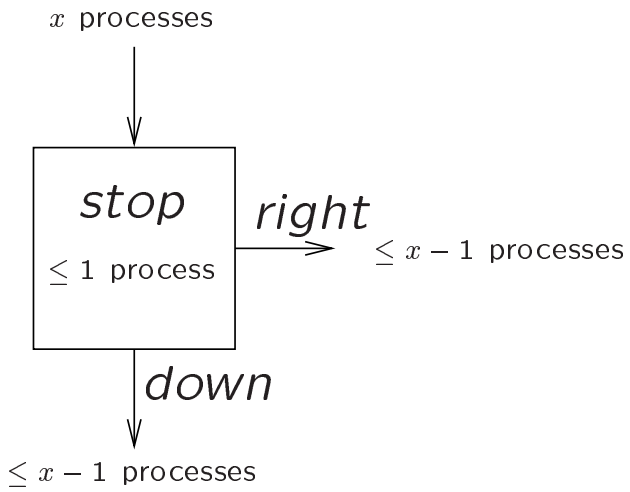
Obstruction-free vs Wait-free

- k-concurrency: when concur is limited to k processes
- k-obstruction-free: safety always, liveness when no more than k concurrent accesses (on an object)
- k-set agreement: at most k values are decided
- Equivalences (Gafni):
  - $\star$ k-concur = k-obst-free = k-set agreement
  - $\star$ System of n processes: n-obst-free = wait-free

Wait-free Implementation

- A wait-free implementation of a concurrent data object is one that guarantees that any process can complete any operation in a finite number of steps, whatever the behavior of the other processes (i.e., regardless of their execution speed)
- A wait-free implementation of an object
  - $\star$ Is automatically deadlock-free and
  - $\star$ tolerates up to $n-1$ process crashes
  - $\star$ Can address both real-time and fault-tolerance

Introductory example: the SPLITTER



Splitter: Specification

- At most one process exits with $stop$
- At most $(x-1)$ processes exit with $right$
- At most $(x-1)$ processes exit with $down$
- Solo execution: If a single process invokes the splitter, it exits with $stop$ (if it does not crashes)

-Lamport L., A Fast Mutual Exclusion Algorithm. *ACM Transactions on Computer Systems*, 5(1):1-11, 1987

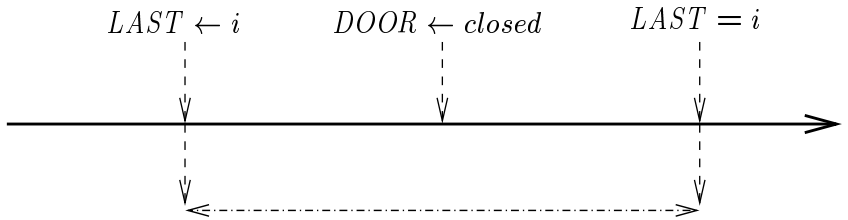
Splitter: a WF Implementation

Two shared registers: $LAST$ (init $\forall$) and $DOOR$ (init $open$)

operation direction() % issued by p_i %

$LAST \leftarrow id_i$;
if $(DOOR = closed)$ **then** $move_i \leftarrow right$
else $DOOR \leftarrow closed$
if $(LAST = id_i)$ **then** $move_i \leftarrow stop$
else $move_i \leftarrow down$
end if
end if;
return $(move_i)$

Splitter: Proof (very easy)



No process p_j has modified the atomic register $LAST$

The (Static WF) Renaming Problem

- $id_1, \ldots, id_n$ are the initial identities of the processes
- $id_1, \ldots, id_n \in [0..N-1]$, where $N >>>>> n$
- Acquire a new name in the set $[0..M-1]$
- M: "as small as possible"
- No two processes can get the same name
- Type of renaming:
  - ⋆ Static renaming: a name is acquired once for all
  - ⋆ Dynamic renaming: names are (acquired; released)*
- Lower bound: $M \geq 2n-1$ (Herlihy-Shavit)

Basic papers

- Introduction of the problem:

  Attiya H., Bar-Noy A., Dolev D., Peleg D. and Reischuk R., Renaming in an Asynchronous Environment. *Journal of the ACM*, 37(3):524-548, 1990.

- Lower bound:

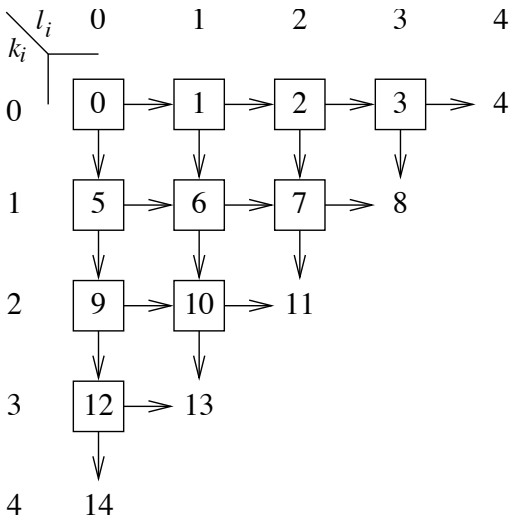
  Herlihy M.P. and Shavit N., The topological Structure of Asynchronous Computabiilty. *Journal of the ACM*, 46(6):858-923,, 1999

Moir-Anderson's Protocol

- Considers $M = n(n+1)/2$
- Basic idea: a grid of splitters

- Moir M. and Anderson J.H., Wait-Free Algorithms for Fast, Long-Lived Renaming. *Science of Computer Programming*, 25:1-39, 1995

Grid of Splitters



The Protocol

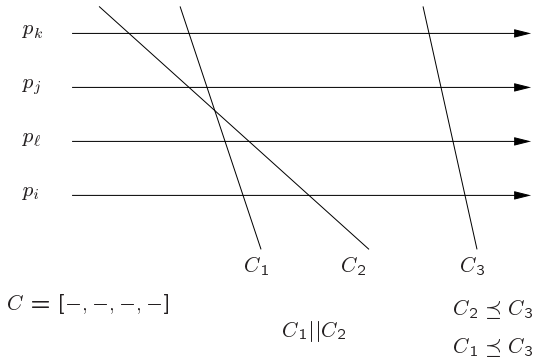
operation get _name(id_i)

$d_i \leftarrow 0$; $r_i \leftarrow 0$; $move_i \leftarrow down$;
while $(move_i \neq stop)$ **do**
  $move_i \leftarrow$ splitter$[d_i, r_i].direction()$;
  case $(move_i = right)$ **then** $r_i \leftarrow r_i + 1$
    $(move_i = down)$ **then** $d_i \leftarrow d_i + 1$
    $(move_i = stop)$ **then** **exit_loop**
  end case
end while
return $(n \times d_i + r_i - (d_i(d_i - 1)/2))$
% the new name is the position $[d_i, r_i]$ in the grid %

store/collect Object: specification

- View = set of values obtained by a collect() operation
- No new-old inversion when no-concurrency Property:
  Let C_1 and C_2 be two executions of collect(). If C_1 has terminated before C_2 starts, (denoted $C1 \preceq C2$) then for each j, the value returned by C_2 is n ot older than the one obtained by C_1
- Uptodateness property:
  If store(v) is the last store operation p_i has executed before collect(), it obtains v for its last deposited value

An Example

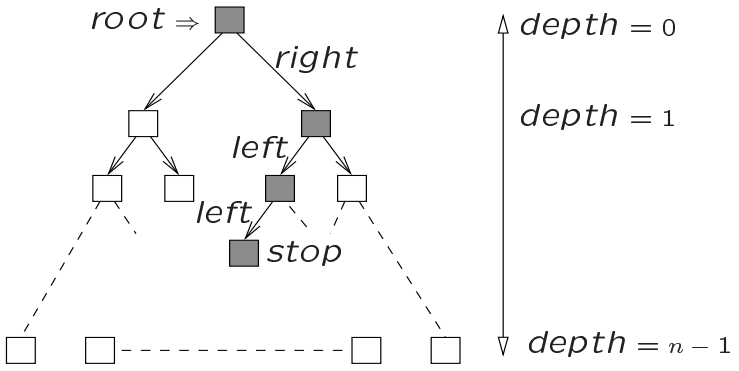


- Theorem: Partial order
  The set of views obtained by the collect operations is partially ordered

A Simple Implementation

- An Array $V[1..n]$
- store(v) by p_i: $V[i] \leftarrow v$
- collect by a process: reads the whole array
- Costs:
  - ⋆ store: $O(1)$
  - ⋆ collect: $O(n)$
- Drawback: if at most $k << n$ processes deposit values, we would like to have an $O(k)$ cost

A Bin Tree of Splitters: Attiya-Fouren-Gafni



- Attiya H., Fouren A. and Gafni E., An Adaptive Collect Algorithm with Applications. *Distributed Computing*, 15(2):87-96, 2002

Attiya-Fouren-Gafni's Protocol (1)

operation store(v)
% aux_i: local var pointing to the vtx currently visited %
if ($my\_vertex_i = \perp$) **then**
$aux_i \leftarrow root$;
repeat $aux_i.marked \leftarrow true$;
aux_i.direction($move_i$);
case ($move_i = left$) **then** $aux_i \leftarrow aux_i.left$
($move_i = right$) **then** $aux_i \leftarrow aux_i.right$
($move_i = stop$) **then exit_loop**
end case
end repeat;
$my\_vertex_i \leftarrow aux_i$; $my\_vertex_i.pid \leftarrow id_i$;
end if;
$my\_vertex_i.value \leftarrow v$

Attiya-Fouren-Gafni's Protocol (2)

function collect returns a $view$
return(DF_Search($root$))

---

function DF_Search(vtx) **returns** a $view$
Let V_i a view **init to** $\emptyset$;
if ($vtx.marked$) **then**
if ($vertex.value \neq \perp$)
then $V_i \leftarrow \{\langle vtx.pid, vtx.value\rangle\}$ **endif**;
$V_i \leftarrow V_i \cup$ DF_Search($vtx.left$) $\cup$ DF_Search($vtx.left$)
endif;
return(V_i)

A snapshot Object: specification

- Keeps data provided by processes
- When p_i invokes store(v) it defines v as its last deposited value
- A process invokes snapshot to get the values deposited by the processes
- Atomicity: Everything has to appear as if the operations were executed instantaneously (at some time between their invocation and their termination)
- It is store/collect with the additional constraint that there is never new/old inversion (even in presence of concurrency)

Basic papers

- Problem statement:

  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

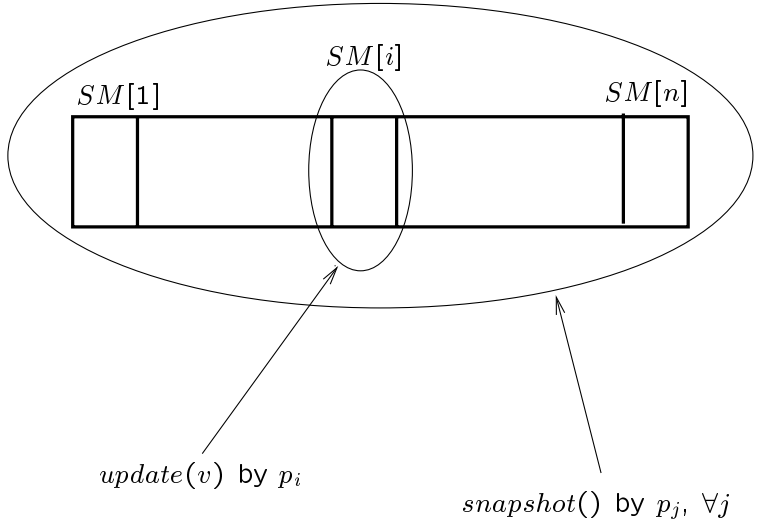
- Best known algorithm (optimal?):

  Attiya H. and Rachman O., Atomic Snapshots in O($n \log n$) Operations. *SIAM Journal of Computing*, 27(2):319-340, 1998

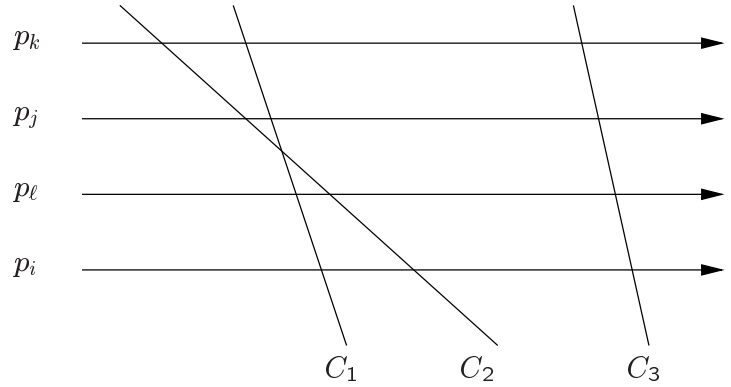
- A multiwriter algorithm:

  Inoue M, Chen W., Masuzawa T. and Tokura N., Linear Time Snapshot Using Multi-Reader Multi-Writer Registers. *Proc. 8th Int'l Workshop on Distributed Algorithms (WDAG'94)*, Springer-Verlag LNCS Vol. 857, pp. 130-140, 1994

Snapshot operations



An Example



$C = [-, -, -, -]$

$C_1 || C_2$

$C_2 \preceq C_3$

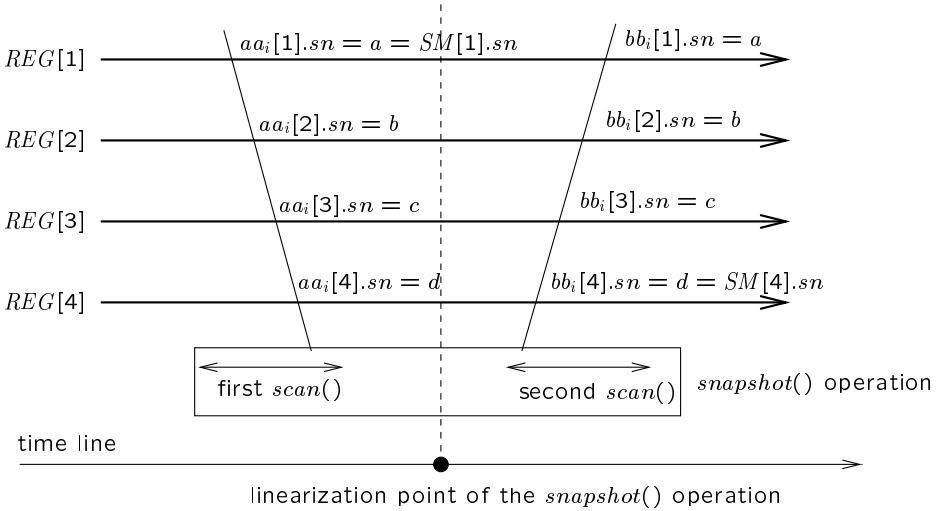
$C_1 \preceq C_3$

Underlying idea

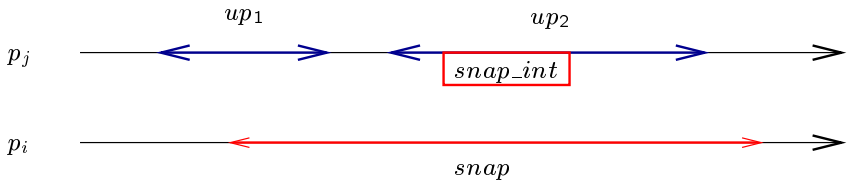
procedure update (v)
  $sn_i \leftarrow sn_i + 1$; % local seq number generator %
  $SM[i] \leftarrow (v, sn_i)$ % atomic write %

operation snapshot()
  while $true$ **do**
    $A_i \leftarrow$ *scan*; $B_i \leftarrow$ *scan*;
    % double "asynchronous" scan %
    if $(\forall j : A_i[j].sn = B_i[j].sn)$ **then** return $(A_i.val)$ **end_if**
      % $A_i.val = [A_i[1].val, \ldots, A_i[n].val]$ %
  end_while

Snapshot: Partial proof (easy)



How an update can help a snapshot



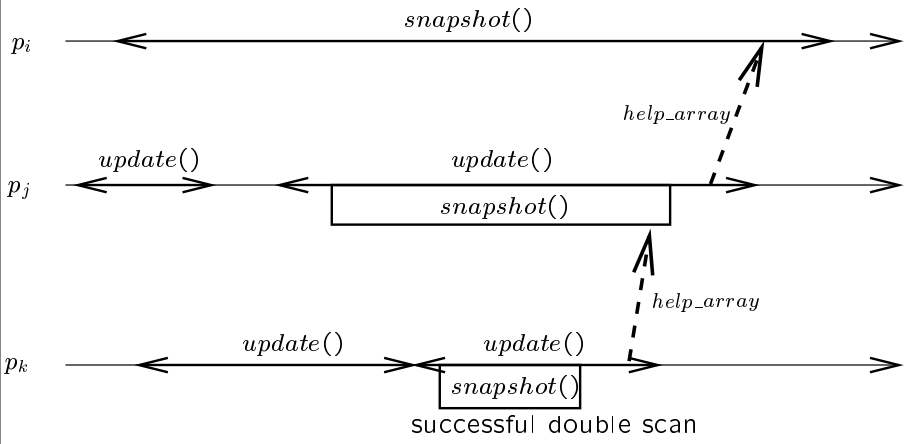
Afek et al.s algorithm (1)

operation update (v):
$help\_array_i \leftarrow$ snapshot();
$sn_i \leftarrow sn_i + 1$;
$SM[i] \leftarrow (v, sn_i, help\_array_i)$

Afek et al.s algorithm (2)

operation snapshot():
$could\_help_i \leftarrow \emptyset$;
while $true$ **do**
$A_i \leftarrow scan$; $B_i \leftarrow scan()$; % double "asynch" collect %
if $(\forall j : A_i[j].sn = B_i[j].sn)$
then return $(A_i.val)$
else for $j : 1 \leq j \leq n$ **do**
if $(A_i[j].sn \neq B_i[j].sn)$ **then**
if $(j \in could\_help_i)$
then return $(B_i[j].help\_array)$
else $could\_help_i \leftarrow could\_help_i \cup \{j\}$
end if end if
end for
end if
end while

Linearization point of a $snapshot()$ operation



From snapshot to optimal adaptive renaming

- p participating processes, $1 \leq p \leq n$
- Adaptive renaming: the size of the new name space M depends only on the number p of competing processes
- Size of the new name space $M = 2p-1$ (The inescapable price to pay due to asynchrony anf failures is $p-1$ unused new names)
- The initial name of p_i is id_i
- Anonymity property: the code executed by p_i with initial name id is the same as the code executed by p_j with name initial id
- Corollary: The indexes $1, \ldots, i, \ldots, n$ are used only for addressing purpose

- Attiya H. and Welch J., *Distributed Computing: Fundamentals, Simulations and Advanced Topics,* (2d Edition), Wiley-Interscience, 414 pages, 2004

From snapshot to (optimal) adaptive renaming

A single shared object: snapshot object $STATE[1..n]$

- $STATE[i]$ init to $< \bot, \bot >$
- $STATE[i].old$: to contain the old name of p_i

  $STATE[i].old \neq \bot$ means p_i is competing
- $STATE[i].new$: to contain the new prop/name of p_i

  Last value of $STATE[i].new$ is p_i new name

Idea of the algorithm: competition to acquire a free slot in the set $[1, \ldots, 2p-1]$ of slots

Optimal adaptive wait-free renaming

operation new_name(id_i):
  $prop_i \leftarrow 1$;
  while $true$ **do**
    $STATE[i] \leftarrow < id_i, prop_i >$;
    $my\_view_i \leftarrow STATE$.snapshot();
    if $\forall\ j \neq i :\ my\_view_i[j].new \neq\ prop_i$
      then return $(prop_i)$
      else **let** r = rank of id_i in $\{my\_view_i[j].old \neq \bot\}$
        $prop_i \leftarrow r$th integer $\notin X$ where
          $X = \{my\_view_i[j].new \neq \bot \mid j \neq i\}$
    end if
  end while

Properties

- $\forall\ j \neq i$: $my\_view_i[j].new \neq prop_i$ true means that no p_j is competing for the new name $prop_i$
- $X = \{my\_view_i[j].new \neq \perp \mid j \neq i\}$ is the set of processes competing with p_i for a new name
- $[1..2p-1] \setminus X$ contians at least p empty slots
- r defines a (arbitrary but unique) rank for p_i among the competitors for a new name: always such that $1 \leq r \leq p$
- $prop_i$ is a pointer to an "empty slot": it is always an integer such that $1 \leq prop \leq 2p-1$
- Remark: the proof of the property "no two new names are eqaul" does not depend on the way the new are chosen. It only depends on the snapshot operation

Example of an obstruction-free object (1)

- A timestamp (integer) object with one operation get_ts()
- Specification
  - ⋆ Consistency: If get_ts$_2$() starts after get_ts$_1$() has terminated, it obtains a greater value
  - ⋆ Unicity: No two invocations return the same value
  - ⋆ Liveness (wrt Concurrency):
    - ∗ No concurrency: get_ts() always returns
    - ∗ Concur.: it might happen that no get_ts() returns

Example of an obstruction-free object (2)

- Based on a splitter!
- ME (Me): unbounded array of atomic registers (integers), init $[\perp, \perp, \ldots]$
- TT (Try to Take): unbounded array of atomic boolean registers, init $[false, false, \ldots]$
- $TT[k] = true$: a process tried to obtain the timestamp k
- $NEXT$: next timestamp to be captured, init 1

Example of an obstruction-free object (3)

operation TS.get_ts(i) (**invoked by** p_i):
  $k \leftarrow NEXT$;
  while ($true$) **do**
    $\boxed{ME[k] \leftarrow i}$;
    if ($\neg TT[k]$) **then** (try to capture k)
      $TT[k] \leftarrow true$;
      if $\boxed{(ME[k] = i)}$ **then** $NEXT \leftarrow k+1$;
        return (k)
      end if;
    end if;
    $k \leftarrow k+1$ (try the next integer)
  end while

Concurrent set object: definition

Sequential specification of a set S

- Operation add(x):
  adds x to S; returns $true$ iff x was not in S
- Operation remove(x):
  suppress x from S; returns $true$ iff x was in S
- Operation contain(x): returns $true$ iff $x \in S$

- Heller S., Herlihy M., Luchangco V., Moir M., Scherer W. and Shavit N., A lazy conucurrent list-based set algorithm, *Parallel Processing Letters*, 17(4):411-424, 2007

Concurrent set: context of use

- No failures (hence locks are meaningful)
- Aim: efficiency
- The elements of the set S belong to a well-founded set
- Assumption: lots of contain(x) wrt add(x)/remove(x)
  - ⋆ Hence, the operation contain(x) has to be always very efficient despite the existence of concurrent invocations of remove(x) and add(x) (if any)

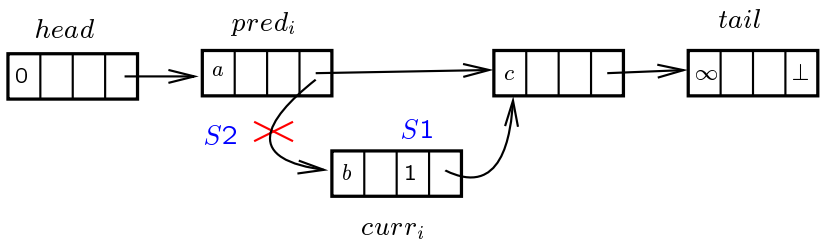
Concurrent set: Fail-free list-based impl.

- A list of cells
- Sorted in increasing order according to the value of its elements (e.g., positive integers)
- A cell is made up of 4 fields:
  - ⋆ The value of the element (key)
  - ⋆ A pointer $next$ to the next cell of the list
  - ⋆ A boolean mk (marked) initialized to $false$
  - ⋆ A lock $lock$ to protect from some concurrent acesses
- Two special cells: $head$ and $tail$ ($head.next \neq tail$ means the list is not empty) $head.key = 0$ and $tail.key = +\infty$

A few design pinciples

- No atomically markable reference
- Wait-free list traversal
- Optimistic remove() and add() operations (i.e., delay locking as long as possible)
- Dissociate logical removal and physical removal
  - ⋆ Logical removal: set the $cell.mk$ bit to $true$
  - ⋆ Physical removal: update the $cell.pred$ pointer
- Only the remove() operation execute physical removals
- add() and contain() are not delayed by physical removals

Operation remove(b) (1)



- $S1$ (Logical removal): the $cell.mk$ bit is set to $true$
- $S2$ (Physical removal): update the $pred.next$ pointer

Operation remove(x) (2)

operation S.remove(x):
  $pred \leftarrow$ @$head$; $curr \leftarrow head.next$; % pointers %
  while $(curr.key < x)$ **do** $pred \leftarrow curr$;
    $curr \leftarrow curr.next$ **end while**;
  lock($pred.lock$); lock($curr.lock$);
  if validate($pred, curr$) **then**
    if $(curr.key \neq x)$
      then unlock($pred.lock$); unlock($curr.lock$); return $(false)$
      else $(S1)$ $curr.mk \leftarrow true$; $(S2)$ $pred.next \leftarrow curr.next$;
        unlock($pred.lock$); unlock($curr.lock$); return $(true)$
    end if
  else restart the operation
  end if

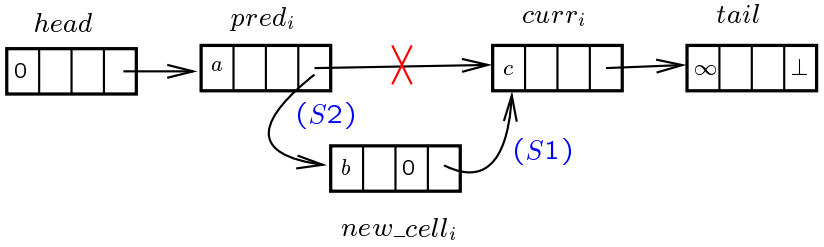
Operation validate($pred, curr$)

Because there is a gap between the unsynchronized traversal and the lock acquisition, it is necessary to validate that the operation has locked the correct cells. This is due to the following observations:

- The $curr$ cell could have been removed
- The $pred$ cell could have been removed
- A cell could have been inserted between $pred$ and $curr$

operation validate($pred, curr$):
  let $res = (\neg\ pred.mk) \ \wedge\ (\neg\ curr.mk)$
    $\wedge\ (pred.next = curr)$;
  return (res)

Operation add(b) (1)

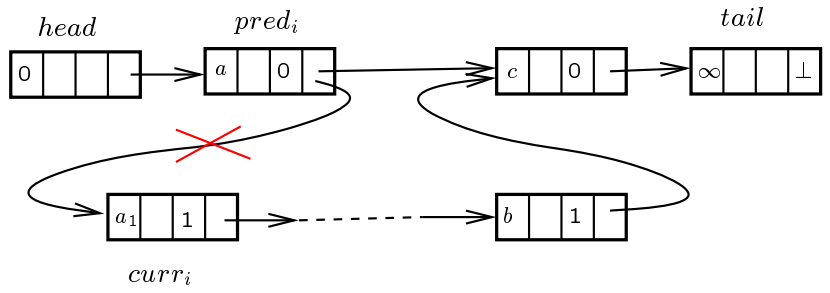


- $S1$: set the $new\_cell.next$ pointer
- $S2$: update the $pred.next$ pointer

Operation add(x)

```
operation S.add(x):
  pred ← @head; curr ← head.next;   % pointers %
  while (curr.key < x) do pred ← curr;
                          curr ← curr.next end while;
  lock(pred.lock);
  if validate(pred, curr) then
    if (curr.key = x)
      then unlock(pred.lock); return (false)
      else new_cell ← new cell;
           new_cell.key ← x; (S1) new_cell.next ← curr;
           (S2) pred.next ← @new_cell;
           unlock(pred.lock); return (true)
    end if
  else restart the operation
  end if
```


Operation contain(b) (1)



- While all the elements between a_1 and b are logically and physically removed, it is possible that b is (concurrently) added again
- Subtleties in defining the linearization point

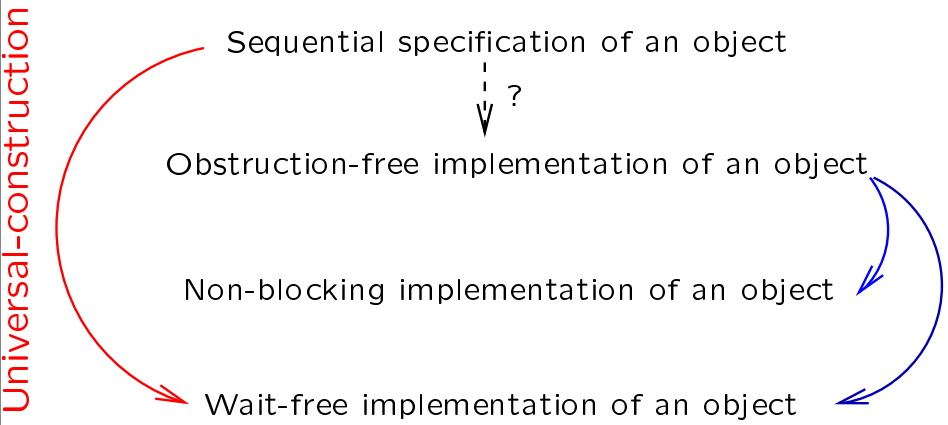
Operation contain(x)

```
operation S.contain(x):
  curr ← @head;
  while (curr.key < x) do curr ← curr.next end while;
  let res= (curr.key = x) ∧ (¬ curr.mk);
  return (res)
```


Properties

- Wait-free list traversal in all operations
- Locks used on at most two cells (add and remove)
- Calls of add() and remove() on non-adjacent list entries never interfere
- The operation contain() is wait-free
- No use of atomically markable reference

The big picture



Part IV

A FUNDAMENTAL PROBLEM

Why is Wait-Freedom Fundamental?

- Aim: wait-free implement linearizable concurrent objects in presence of process crashes

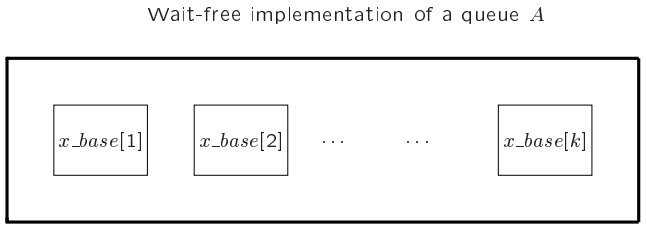
  Compose base objects to get "more powerful" objects

Problem: Given an object A (defined by a seq spec) and objects X, is there a wait-free implementation of A from atomic registers and objects X in a system of n processes (prone to crashes)?

- Wait-free = liveness despite any number of crashes

Example

Example: Build a wait-free queue A in an asynchronous SM system made up of read/write atomic registers



Is it possible??

Herlihy's Result

- An object is universal if it can be used to wait-free implement any object (that has a seq specification)
- Main result (Herlihy 1991):

  Consensus is a universal object

- Any concurrent object (with seq spec) can be built from a consensus object (and read/write registers)
- Notion of Consensus number
- A Universal construction is an algorithm that, given the sequential specification of an object, constructs a corresponding concurrent object (from consensus objects and registers)

The CONSENSUS Problem: Definition

Each process proposes a value and has to decide a value in such a way that:

- Termination: Every correct process eventually decides
- Validity: A decided value is a proposed value
- Uniform Agreement: No two $\neq$ values are decided

A consensus object allows the processes to agree on the same "view", whatever their current states

CONSENSUS in Action! (Concept)

Let X be a consensus object (a single operation Propose(v))

- Consensus is a Synchronization tool:
  Consensus makes a decision irreversible (only one action/value succeeds)

  $x_1 \leftarrow X$.Propose(a) at t_1 and $x_2 \leftarrow X$.Propose(b) at t_2 we always have $x_1 = x_2$ (either a or b) whatever the invocation times t_1 and t_2
- Consensus solves Non-determinism:
  Consensus makes a result UNIQUE

  Let f be a non-deterministic function

  If $x_1 \leftarrow X$.Propose($f(a)$) and $x_2 \leftarrow X$.Propose($f(a)$) we always have $x_1 = x_2$

... But there is a bad news: The Main Result

Fischer-Lynch-Paterson's Impossibility result (1985)

There is no protocol that solves the consensus problem in asynchronous systems (shared memory or message passing) that is subject to even a single process crash failure

- Fischer M.J., Lynch N.A. and Paterson M.S., Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM*, 32(2):374-382, 1985

- Loui M.C. and Abu-Amara H.H., Memory Requirements for Agreement Among Unreliable Asynchronous Processes. *Advances on Computer Research*, JAI Press, 4:163-183, 1987

On impossibilty results

- There are a lot of impossibility results in DC (and CS)
- Example:
  - ⋆ Build a reliable link on top of an unreliable link
  - ⋆ Additional assumption is required: link fairness
- Impossibilty results are "good":
  - ⋆ For the theoretician: state the minimal additional assumptions required to solve a problem
  - ⋆ For the engineer: investigate and state assumption coverage

Herlihy's Results

- Which objects allow implementing a consensus object?
- Idea: investigate the synchro power of base objects

  Each object has a consensus number

  The consensus number of X is the largest n for which X solves consensus among n processes
- FLP $\Rightarrow$ CN(Read/Write atomic variables)=1
  Which means that read/write operations are not powerful enough to solve consensus in presence of even a single crash when $n > 1$
- Which is the synchronization power needed to solve consensus in a SM asynchronous system?

Herlihy's Results

- As the synchronization power of read/write atomic variables is too weak to implement consensus, are they synchronization primitives powerful enough to solve consensus in a SM asynchronous system?
- A few synchronization primitives (synchr objects):
  - ⋆ Test&Set $(shared)$=
    $[prev \leftarrow shared; shared \leftarrow 1;$ return $(prev)]$
  - ⋆ Swap $(local, shared)$= $[local \leftrightarrow shared]$
  - ⋆ Move $(shared_1, shared_2)$= $[shared_1 \leftrightarrow shared_2]$
  - ⋆ Compare&Swap $(shared, old, new)$=
    $[prev \leftarrow shared;$ if $prev = old$ then $shared \leftarrow new$ fi; return $(prev)]$

Herlihy's Results

- Consensus numbers define a hierarchy on the power of synch primitives
  - ⋆ CN(Test&Set) = CN(Swap) = CN(Stack)=
    CN(Fetch&Add) = CN(Fifo Queue) = ... = 2
  - ⋆ CN(Move) = CN(Compare&Swap)= $+\infty$

Consensus with Test&Set (2 processes)

- Let $shared$ shared variable init to 0
- Let $prefer[0], prefer[1]$ be two shared variables init to $\perp$

Propose (v) = % issued by p_i $(i = 0$ or $1)$ %

$prefer[i] \leftarrow v$;
$val \leftarrow$ Test&Set $(shared)$;
case $val = 0$ **then** return (v)
$val = 1$ **then** return $(prefer[1 - i])$
endcase

The "winner" is the first that executes Test&Set($shared$)

Consensus with a FIFO Queue (2 processes)

- Let $queue$ be a shared queue init to $< 0, 1 >$ (return from an empty queue returns $\perp$)
- Let $prefer[0], prefer[1]$ be two shared variables init to $\perp$

Propose (v) = % issued by p_i $(i = 0$ or $1)$ %

$prefer[i] \leftarrow v$;
$val \leftarrow$Dequeue $(queue)$;
case $val = 0$ **then** return (v)
$val = 1$ **then** return $(prefer[1 - i])$
endcase

The "winner" is the first process that dequeues 0

Consensus with Compare&Swap (n processes)

- Let $shared$ shared variable init to $\perp$

Propose (v) = % issued by p_i $(i = 1, 2, \ldots, n)$ %

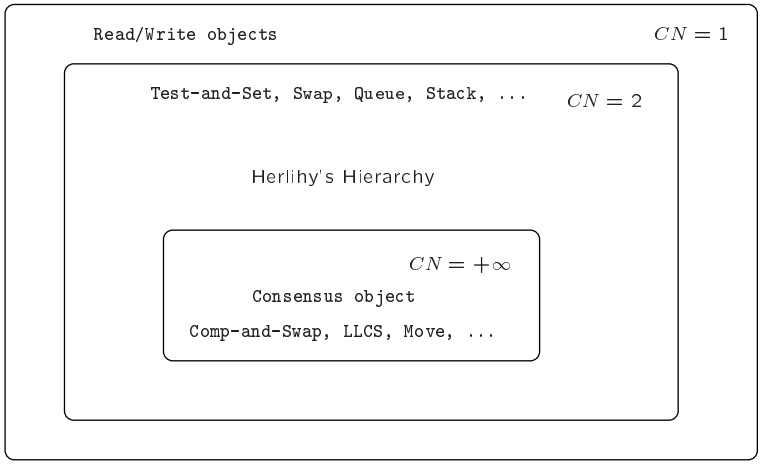
Let $first$ be a local variable;
$first \leftarrow$Compare&Swap $(shared, \perp, v)$
if $first = \perp$
then return (v)
else return $(first)$
endif

The "winner" is the first that deposits its v in $shared$

An Observation

- With Test&Set or FIFO Queue:
  the shared register used to synchronize (determine a winner) and the shared register used to store the decided value are distinct registers
- With Compare&Swap:
  the shared register used to synchronize (determine a winner) and the shared register used to store the decided value are the same register

Herlihy's Hierarchy



What has been learnt from Herlihy's Hierarchy

- Test&Set, Swap, Fetch&Add, etc. are synchronization primitives that are too weak to implement reliable objects in presence of process crashes

  A hierarchy on the power of synchr primitives when addressing fault-tolerance

- Additional synchr power is required to design objects tolerant to process crashes.

  Object combination does not always work! (e.g., atomic read/write objects do not allow the construction of more sophisticated objects)

  FLP means (here) that fault-masking can be impossible to achieve when solving non-trivial problems if we do not rely on powerful enough synch primitives

Part V

UNIVERSAL CONSTRUCTION

- Herlihy M.P., Wait-free synchronization. *ACM Toplas*, 11(1):124-149, 1991.

- Chandra T. and Toueg S., Unreliable Failure Detectors for Reliable Distributed Systems. *Journal of the ACM*, 43(2):225-267, 1996.

- Guerraoui R. and Raynal M., A Universal construction for wait-free objects. *Proc. ARES 2007 Int'l Workshop on Foundations of Fault-tolerant Distributed Computing (FOFDC 2007)*, IEEE Press, pp. 959-966, 2007

A (WF) universal construction

- Based on the state machine replication paradigm
- Each process manages a local copy of the object
- Only control variables in the shared memory
- Consensus objects are used to order the operation invocations
- Shared registers are used as helping mechanism to ensure every invocation is eventually applied to the copies

The Specification of the Objects

- $\mathcal{X}$ an object of type T
- Operations: $\mathcal{X}$.op($param$) (returns always a response)
- The type of $\mathcal{X}$ is defined by a transition function $\delta()$
  - ⋆ $\delta(sx,\text{op}(param))$ is a non-empty set of (sx', res) pairs defining all the possible "results" we can obtain when the object $\mathcal{X}$ is in the state sx
  - ⋆ Each pair (sx', res) is such that sx' is a possible new state of $\mathcal{X}$, and then res is the corresponding result returned by op($param$)
  - ⋆ If the set has only one pair, the type T is deterministic
    Otherwise, it is non-deterministic
    Here, we consider deterministic types

Local data structures

- sx_i is a local state of $\mathcal{X}$ as known by p_i
- $next\_sn_i[1..n]$ is a local array local of sequence numbers

  $next\_sn_i[j]$ (initialized to 1) is the next sequence number that, to p_i's knowledge, p_j will associate with its next operation on $\mathcal{X}$
- $prop_i$ (a list), $exec_i$ (a list), $result_i$ (an invocation result) and k_i (an integer) are auxiliary variables

  Notation:

  $exec_i[r]$ = rth element of the list $exec_i$

  $|exec_i|$ = size of the list $exec_i$

Shared Base Objects

- $LAST\_OP[1..n]$: shared array of 1WnR atomic registers

  Only p_i can write $LAST\_OP[i]$

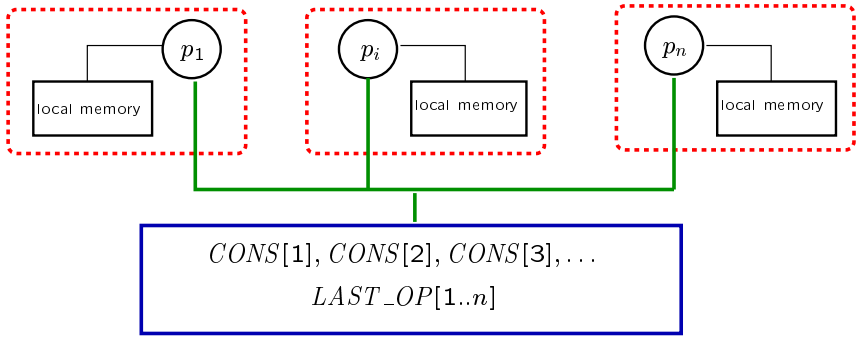
  Each register $LAST\_OP[j]$ has two fields:

  - ⋆ $LAST\_OP[j].op$: last operation invoked by p_j
  - ⋆ $LAST\_OP[j].sn$: associated sequence number
  - ⋆ Each entry of the array is initialized to $(\perp, 0)$
- A list of consensus objects $CONS[k]$ for $k = 1, 2, \ldots$

  (used sequentially by each process)

  A process invokes $CONS[k]$.propose (v)

Software architecture



Universal construction: User interface

when operation $\mathcal{X}$.op($param$) **is invoked by** p_i:
$result_i \leftarrow \perp$;
$LAST\_OP[i] \leftarrow$ (op($param$),$next\_sn_i[i]$);
wait until ($result_i \neq \perp$);
return ($result_i$)

Universal construction: Background task

while ($true$) **do**
Step 1: Build its own proposal
$prop_i \leftarrow$ build a proposal from
$LAST\_OP[1..n]$ and $next\_sn_i[1..n]$);
Step 2: Commit one of the proposals
if ($prop_i \neq \epsilon$) **then** $k_i \leftarrow k_i + 1$;
Decide a proprosal $prop_j$ with the help of $CONS[k_i]$;
Apply the sequence of operations to sx_i
end_if
end_while

Universal construction (2): Background task

while ($true$) **do**
Step 1: Build a proposal
$prop_i \leftarrow \epsilon$; % empty list %
for $1 \leq j \leq n$ **do**
if ($LAST\_OP[j].sn \geq next\_sn_i[j]$) **then**
add ($LAST\_OP[j].op, j$) to $prop_i$ **end_if**;
end_for;
Step 2: Try to commit the proposal
if ($prop_i \neq \epsilon$) **then** $k_i \leftarrow k_i + 1$;
.....
end_if
end_while

Universal construction (3): Background task

Assume first that the type T of the object is deterministic

$exec_i \leftarrow CONS[k_i]$.propose ($prop_i$);
let $\ell = |exec_i|$;
for r from 1 **to** ℓ **do**
$(sx_i, res) \leftarrow \delta(sx_i, exec[r].op)$;
let $j = exec_i[r].proc$;
$next\_sn_i[j] \leftarrow next\_sn_i[j] + 1$;
if ($i = j$) **then** $result_i \leftarrow res$ **end_if**
end_for

The case of non-deterministic types

- Brute force strategy:

  Consider a deterministic reduction $\delta'()$ of $\delta()$

- A nicer solution:

  Use the consensus invocation to solve ordering + single result of each operation + single resulting state

Correctness proof: The object is live and safe

- Wait-free property:

  Show that each operation invoked by a correct process terminates whatever the behavior of the other processes

  This is the liveness property stating that the implementation ensures the operations do terminate

- Linearizability (semantics of the objet):

  From an external observer point of view, the operations on the concurrent object occur as if the object was accessed sequentially by the processes

  This is the safety property stating that the implementation of the object is linearizable

A Variant

"Simplified" version of the construction where the shared array $LAST\_OP[1:n]$ and the seq numbers are suppressed, $prop_i$ is a simple variable (init. to $\perp$) and $\delta(sx_i, \perp) = sx_i$

when operation $\mathcal{X}$.op($param$) **is invoked by** p_i:
  $result_i \leftarrow \perp$; $prop_i \leftarrow$ op($param$);
  wait until ($result_i \neq \perp$); *return* ($result_i$)

while ($true$) **do** % Background Task %
  $k_i \leftarrow k_i + 1$;
  $exec_i \leftarrow CONS[k_i]$.propose ($prop_i$);
  $(sx_i, res) \leftarrow \delta(sx_i, exec_i.op)$;
  let $j = exec_i[r].proc$;
  if $(i = j)$ **then** $result_i \leftarrow res$; $prop_i \leftarrow \perp$ **end_if**
end_while

Wait-free vs Non-Blocking

- The previous construction:
  - ⋆ Satisfies the non-blocking property: if processes propose operations at least one process progress
  - ⋆ Does not satisfy the wait-free property: the progress of a correct process cannot be ensured
- The universal construction algorithm ensures the wait-free property thanks to a helping mechanism

  The shared array $LAST\_OP[1:n]$ (and the associated seq numbers) allows a process to propose to the consensus instances not only its own operations but all the pending operations

  Helping mechanisms: feature of wait-free computing

Part VI

IMPLEMENTING CONSENSUS from TIMED REGISTERS

Raynal M. and Taubenfeld G., The notion of a timed register and its application to indulgent synchronization. *Proc. 19th ACM Symposium on Parallel Algorithms and Architectures (SPAA'07)*, pp. 200-209, 2007

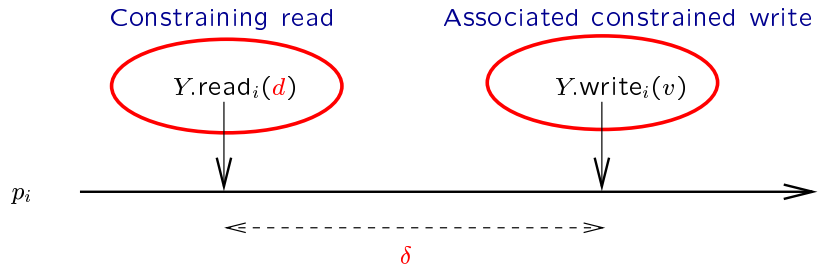
Timed register (1)

- Generalize the notion of atomic register by imposing time constraints on some write operations
- A write operation returns $true$ or $false$
- Context:
  - ⋆ There are both time-free and timed registers
  - ⋆ Let Y be a timed register and p_i a process
  - ⋆ When p_i reads Y, it makes a promise d that (if any) its next write on Y will occur by at most d time units
  - ⋆ Operation $Y.\text{read}_i(d)$ constraining read
  - ⋆ Between these two operations:
    - p_i can issue other operations on other registers
    - Other processes can access Y

Timed register (2)

- $Y.\text{write}_i(v)$ succeeds if $Y.\text{write}_i(v)$ and $Y.\text{read}_i(d)$ are separated by at most d time units
- 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



$d \leq \delta$: $Y.\text{write}_i(v)$ is successful $\quad$ $d > \delta$: $Y.\text{write}_i(v)$ fails

During the period δ:

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

Underlying timing-based system

- The system has to help in order writes can be successful!

  Otherwise, promises are only "best effort promises"
- Assumption Δ: There is an upper bound Δ 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 Δ is LOCAL
- Δ can be known or unknown

Failures and indulgence

- Transient failures:
  when the bound Δ 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 Δ assumption

Indulgent consensus with known bound

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

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 $[false, \ldots, false]$

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

operation consensus(v_i):
  $X[v_i] \leftarrow true$;
  while(Y.read(Δ) $= \perp$) **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(∞))

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 Δ

operation consensus(v_i):
$X[v_i] \leftarrow true$;
while (Y.read(d_i) $= \perp$) **do**
if $\neg(Y.\text{write}(v_i))$
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(∞))

Part VI (Cont'd)

IMPLEMENTING CONSENSUS from STIcKY BITS

Plotkin S.A., Sticky Bits and Universality of Consensus. *8th ACM Symposium on Principles of Distributed Computing, (PODC'89)*, ACM Press, pp. 159-175, 1989

Atomic sticky bit: definition

- Initialized to $\perp$, can then contain 0 or 1
- A $write()$ invocation returns $false$ if the value it is trying to write disagree with the already written value; otherwise it returns $true$
- Sticky bits and timed registers are universal objects
- Sticky bits and timed registers have different types
  - ⋆ Sticky bits: write-once objects, not time-constrained
  - ⋆ Timed registers: not write-once but time-constrained

From a sticky bit to consensus

- From a sticky bit SB to binary consensus

operation TS.consensus(v_i) (**invoked by** p_i):
if $(SB.write(v_i))$ **then** return (v_i)
else return $(\neg v_i)$ **end if**

- From Binary consensus to multivalued consensus

- Turpin R. and Coan B.A., Extending Binary Byzantine Agreement to Multivalued Byzantine Agreement. *Information Processing Letters*, 18:73-76, 1984

- Mostefaoui A., Raynal M. and Tronel F., From Binary Consensus to Multivalued Consensus in Asynchronous Message-Passing Systems. *Information Processing Letters*, 73:207-213, 2000

Part VII

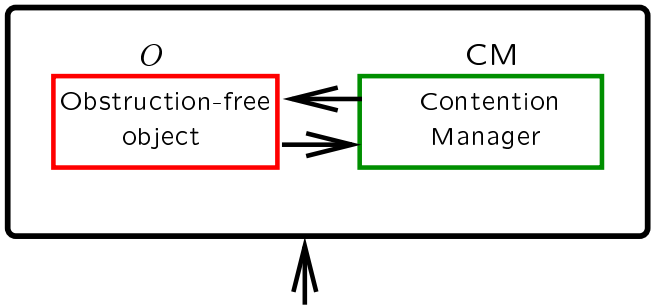
From OBSTRUCTION-FREEDOM to NON-BLOCKING and to WAIT-FREEDOM

Reminder: an obstruction-free implementation

operation TS.get_ts(i) (**invoked by** p_i):
  $k \leftarrow NEXT$;
  while ($true$) **do**
    $\boxed{ME[k] \leftarrow i}$;
    if ($\neg TT[k]$) **then** (try to capture k)
      $TT[k] \leftarrow true$;
      if $\boxed{(ME[k] = i)}$ **then** $NEXT \leftarrow k + 1$;
        return (k)
      end if;
    end if;
    $k \leftarrow k + 1$ (try the next integer)
  end while

Software architecture: Contention Manager

From obstruction-freedom to non-blocking/wait-freedom



Additional power is required

- Approach based on synchrony assumptions on the system behavior plus synchro operations (Compare&Swap, or Fetch&Add)

  - Fich E.F., Luchangco V. and Moir M. and Shavit N., Obstruction-free algorithms can be practically wait-free. *Proc. 19th Int'l Symp. on Distr. Computing (DISC'05)*, Springer-Verlag LNCS #3724, pp. 78-92, 2005

  - Taubenfeld G., Efficient transformations of obstruction-free algorithms into non-blocking algorithms. *Proc. 21th Int'l Symposium on Distributed Computing (DISC'07)*, Springer-Verlag LNCS #4731, pp. 450-464, 2007

- Approach based on enriching the system with failure detectors

  - Guerraoui R., Kapalka M. and Kouznetsov P., The weakest failure detector to boost obstruction freedom. *Distributed Computing*, 20(1), 2008

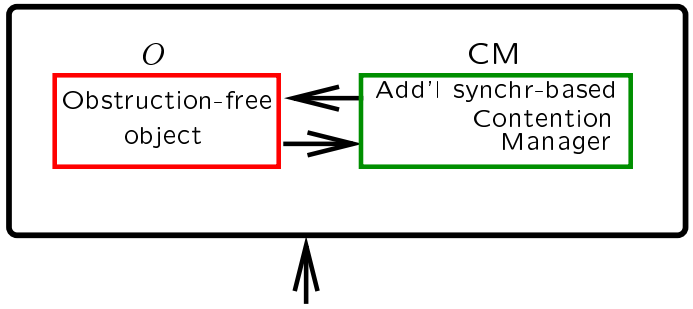
The "additional synchrony" approach

- Fich E.F., Luchangco V. and Moir M. and Shavit N., Obstruction-free algorithms can be practically wait-free. *Proc. 19th Int'l Symp. on Distr. Computing (DISC'05)*, Springer-Verlag LNCS #3724, pp. 78-92, 2005

- Max (min) step time: longest (shortest) time between the completion times of two consecutive steps by any process
- Semi-synchronous with unknown bound model: there is a finite R (that can never be explicitly known) such that (in all executions) the ratio of the maximum and minimun step times is bounded by R
- A contention can require additional synchronization operations (such as Compare&Swap, Fetch&Add, etc.)

From an obst-free to a wait-free implementation

From obstruction-freedom to wait-freedom



Using additional synchrony: the FLMS'05 approach

- Detect competition
  - ⋆ The processes process share a boolean $COMPET$
  - ⋆ A process sets it to $true$ to indicate it suspects concurrency
- Solve competition
  - ⋆ When a process p_i suspects competing processes, it acquires a timestamp $TS[i]$ from a shared integer register C
  - ⋆ Reminder: timestamps are totally ordered
  - ⋆ The idea is then to direct the competing processes to execute their operations in their timestamp order

The difficult issue to solve

Reminder: a process can crash while accessing the object!

- When it suspects competition, a process p_k uses an activity counter $AC[k]$ (hearbeat-like mechanism) to indicate it is alive
- A process p_i suspects and demotes p_k if it observes no changes of $AC[k]$ during a "long enough period"
- Due to the semi-synchrony unknown bound assumption, "Long enough period" is meaningful!

FLMS'05 contention manager

```
when the operation op(param) is invoked by p_i:
  if (¬COMPET) then
     execute B steps of op(param);
     if op(param) completed then return(results) end if;
     COMPET ← true
  end if;
  t ← Fetch&Add(C);      % Acquire a timestamp %
  AC[i] ← 1;      % Initialize the activity counter (hello!) %
  repeat
     TS[i] ← t; let m = min(TS[1], ··· , TS[n]);
     let p_y proc. with smallest timestamp previously read;
     ∀ x ≠ y do TS[x] ← +∞ end do;
              % Demote all processes but p_y %
     if y = i then ... else .... end if
  until op(param) completed end repeat
```


FLMS'05 contention manager cont'd

```
% The proc. with the "smallest timestamp" is the winner%
% Due to asynchrony, several proc. can simult. be winner! %

if y = i then % p_i considers it as a winner %
  repeat execute b steps;
         if op() completed then TS[i] ← +∞;
                                COMPET ← false;
                                return(results) end if;
         AC[i] ← AC[i] + 1; COMPET ← true
  until TS[i] = +∞ % p_i has been demoted % end repeat
else y ≠ i ...
end if
```


FLMS'05 contention manager cont'd

```
else % case y ≠ i % p_i considers it as a looser %
     repeat
          a ← AC[y];
          wait a steps; % To not prevent p_y to progress %
          s ← TS[y]
     until (a = AC[y]) ∨ (s = +∞) end repeat;
     if (s ≠ +∞) then % Demote p_y %
                     TS[y] ← +∞ end if
end if
```


FD-based approach: basic principle

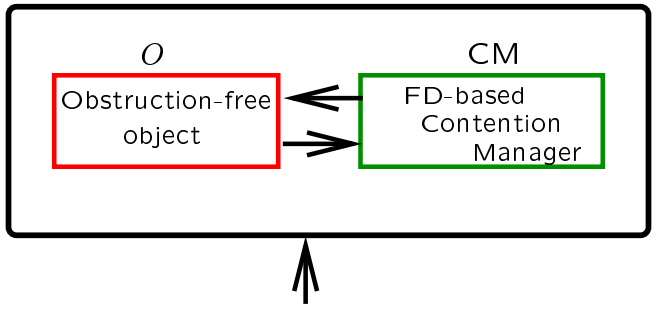
- Guerraoui R., Kapalka M. and Kouznetsov P., The weakest failure detector to boost obstruction freedom. *Distributed Computing*, 20(1), 2008

Designing a FD-based contention manager consists in designing two FD-based operations contender(i) and finished(i) used as follows:

- A process p_i invokes the primitive contender(i) to signal possible contention (competition)
- When it terminates its operation on the object a process p_i invokes the primitive finished(i) to indicate it is no longer accessing the internal representation of the object

From an obst-free to a wait-free implementation

From obstruction-freedom to wait-freedom



Example of "enriched" obstruction-free object

operation TS.get_ts(i) (**invoked by** p_i):
  $k \leftarrow NEXT$;
  while ($true$) **do**
    $\boxed{ME[k] \leftarrow i}$;
    if ($\neg TT[k]$) **then** (try to capture k)
      $TT[k] \leftarrow true$;
      if $\boxed{(ME[k] = i)}$ **then** $NEXT \leftarrow k + 1$;
        CM.finished(i);
        return (k)
      end if;
    end if;
    CM.contender(i);
    $k \leftarrow k + 1$ (try the next integer)
  end while

From obst-freedom to non-blocking: Leadership

- A process can invoke the leader(X) operation where X is the set of processes it perceives as competing)
- The class Ω_* of failure detectors:
  - ⋆ Eventual leadership for each set X: there is a time after which all the invocations leader(X) return the same process identity that is a correct process of X
- When X = {all the processes}, Ω_* boils down to Ω

The class $\boxed{\Omega_* \text{ is the weakest class of FDs}}$
to boost from obstruction-freedom to non-blocking

CM: from obstruction-freedom to non-blocking

init $PART[1..n] \leftarrow [false, \ldots, false]$

operation contender(i):
  $PART[i] \leftarrow true$;
  repeat $X \leftarrow \{j \mid PART[j]\}$ **until** (leader(X) $= i$) **end repeat**

operation finished(i): $PART[i] \leftarrow false$

From obstruction-freedom to wait-freedom

- Leadership is not sufficient too weak!)

- The failure detector class $\boxed{\diamond\mathcal{P} \text{ is the weakest}}$
to boost obstruction-freedom to wait-freedom

- This is the set of eventually perfect failure detectors. Each process p_i is provided with a set $suspected_i$ such that:

- Completeness: Eventually, the set $suspected_i$ of each correct process p_i contains every crashed process
- Eventual strong accuracy: Eventually, the set $suspected_i$ of each correct process does not contain correct processes

CM: from obstruction-freedom to wait-freedom

init $TS[1..n] \leftarrow [0, \ldots, 0]$

operation contender(i):
if $(TS[i] = 0)$ **then** $TS[i] \leftarrow$ get_timestamp() **end if**;
repeat $compet_i \leftarrow \{j \mid (TS[j] \neq 0 \ \wedge \ j \notin \text{suspected}_i)\}$;
let (ts, j) **be** the smallest pair in the set
$\in \{(TS[x], x) \mid x \in compet_i\}$
until $(j = i)$ **end repeat**

operation finished(i): $TS[i] \leftarrow 0$

Part VIII

OBJECT t-RESILIENCE
and
GRACEFUL DEGRADATION

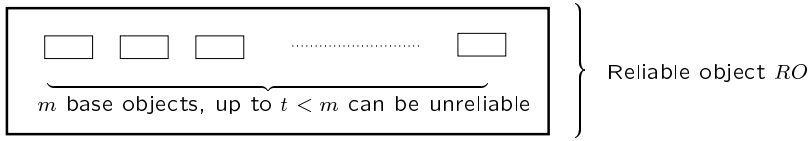
- Jayanti P., Chandra T.D. and Toueg S., Fault-Tolerant Wait-Free Shared Objects. *Journal of the ACM*, 45(3):451-500, 1998

- Guerraoui R. and Raynal M., From unreliable objects to reliable objects: the case of atomic registers and consensus. *9th Int'l Conference on Parallel Computing Technologies (PaCT'07)*, Springer Verlag LNCS LNCS #4671, pp. 47-61, 2007

Universal constructions are based on reliable objects

- A (wait-free) universal construction assumes that the base atomic registers and consensus objects it relies on are reliable
- Here we considers that these base objects can fail, and investigate the construction of reliable registers and consensus objects from unreliable ones
- We want algorithms that are both:
  - ⋆ Wait-free:
    they can cope with any number of faulty processes
  - ⋆ t-resilient:
    they can cope with up to t faulty base objects

Building a t-Resilient object



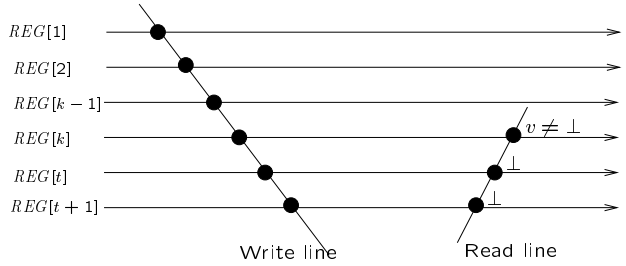
Crash failure model for base objects

- Responsive crash failures (Fail-stop): The object behaves correctly until some pont after which it always returns the default value $\perp$ (once $\perp$, forever $\perp$)
- Non-responsive crash failures (Fail-silent): after it has crashed, an object stops answering forever (and the invocations that have not returned remain pending forever)

Building a 1W1R register when failures are responsive

A bounded construction: principles

Use $REG[1..t+1]$: unreliable base register objects



Building a register when failures are responsive

Bounded construction: algorithm

operation write(v): % invoked by the writer %
  for j **from** 1 **to** $t+1$ **do** $REG[j] \leftarrow v$ **end do**;
  return ()

operation read(): % invoked by the reader %
  for j **from** $t+1$ **to** 1 **do**
    $aux \leftarrow REG[j]$;
    if $(aux \neq \perp)$ **then** return (aux) **end if**
  end do

Properties

- Theorem:
  The previous algorithm wait-free implements a t-resilient *1W1R* atomic register from $(t+1)$ *1W1R* base atomic registers that can suffer responsive crash failures
- The crash-prone base registers are accessed in a predefined order
- The algorithm is space optimal

An improvement

Bounded construction: an improved algorithm

$shortcut$ **init** $t+1$

operation read(): % invoked by the reader %
  for j **from** $shortcut$ **to** 1 **do**
    $aux \leftarrow REG[j]$;
    if $(aux \neq \perp)$ **then** $shortcut \leftarrow j$; return (aux) **end if**
  end do

Building consensus when failures are responsive

Consensus object from base unreliable consensus objects

- The "parallel invocations" approach does not work
- Sequential traversal of base consensus objects does work!
- The algorithm is space optimal (everything is bounded)

Building consensus when failures are responsive

bounded construction

$CONS[1..t+1]$: unreliable base consensus objects

operation propose(v):
  $est \leftarrow v$;
  for k **from** 1 **to** $t+1$ **do**
    $aux \leftarrow CONS[k].propose(est)$;
    if $(aux \neq \perp)$ **then** $est \leftarrow aux$ **end if**
  end do;
  return (est)

State Machine Replication

- Usually: Replication management using the State Machine approach (Thomas, Lamport-Schneider's approach): concurrent RPC-like + votes
- Here: Replication + Sequential iteration on replicas
  The type of control structure applied to replicas becomes decisive (concurrent=independent vs sequential=dependent)

Object failure modes

- Crash. An object experiences a crash failure if after some time all its operations return $\perp$ (default value)
- Omission. An object experiences an omission failure with respect to a process p_i, it has the crash behavior with respect to p_i.
- Byzantine. An object experiences a Byzantine failure if its operations answer values that are not in agreement with the object specification

Failure Mode Hierarchy

- An implementation of a concurrent object is *t-fault tolerant with respect to the failure mode $\mathcal{F}$* (crash, omission, arbitrary) if the object remains correct and its implementation remains wait-free despite the occurrence of up to t base objects that fail according to $\mathcal{F}$
- Severity. A failure mode $\mathcal{F}$ is *less severe* than the failure mode $\mathcal{G}$ denoted $\mathcal{F} \prec \mathcal{G}$ if any implementation that is t-fault tolerant with respect to $\mathcal{G}$ is also t-fault tolerant with respect to $\mathcal{F}$
- crashes $\prec$ omissions $\prec$ arbitrary failures

Graceful Degradation

- A wait-free implementation of a shared object is *gracefully degrading* if it never fails more severely than the base objects it is derived from, whatever the number of base objects that fail
- Remark: the "severity" relation on failure modes ($\prec$) involves only the existence of a fault-tolerant protocol (it does not involve the notion of graceful degradation)

- Jayanti P., Chandra T.D. and Toueg S., The Cost of Graceful Degradation for Omission Failures. *Inf. Proc. Letters*, 71:167-172, 1999

Graceful Degradation: Example

- Let us assume that base objects can fail by crashing. If the implementation remains wait-free and correct despite the crash of any number of processes and the crash of up to t base objects, then this implementation is t-fault tolerant with respect to the crash failure mode
- If the implementation is wait-free and fails only by crash (if it fails) despite the crash of any number of processes and the crash of any number of base objects, then it is gracefully degrading

A Gracefully Degrading t-Omission Tolerant Impl.

$2t+1$ copies of the base object: $base\_cons[1..(2t+1)]$

operation PROPOSE_v

% $V[1..2t+1], est, k$: local variables of the invoking process %
$est \leftarrow v$;
for k **from** 1 **to** $(2t+1)$ **do**
$V[k] \leftarrow base\_cons[k]$.propose_$est$;
if $(V[k] \neq \bot) \wedge (V[k] \neq est)$ **then**
$est \leftarrow V[k]$;
$V[1..(k-1)] \leftarrow [\bot, \ldots, \bot]$ **end if**
end for;
if $(\#_\bot(V) > t)$ **then** return($\bot$) **else** return(est) **endif**

Impossibility for the Crash Failure Mode

Jayanti-Chandra-Toueg 1998

- There are objects (e.g., consensus) for which it is impossible to design gracefully degrading t-fault tolerant wait-free implementations for the crash failure mode
- It is possible to design gracefully degrading t-fault tolerant wait-free implementations for the omission (or arbitrary) failure mode
- Hence: Combining fault-tolerance and graceful degradation is not possible for all failure modes

Part IX

CONCLUSION

Conclusion

- Safety property: Linearizability
- Progress properties: Obstruction-free, Wait-free
- Consensus number, Herlihy Hierarchy
- Universal construction
- Object failure modes
- Process failure vs object failures
- t-Resilient wait-free objects
- Gracefully degrading objects
- Sub-consensus objects ... (next lecture!)

More on subconsensus taks

-Afek Y., Gafni E., Rajsbaum S., Raynal M. and Travers C., Simultaneous Consensus Tasks: a Tighter Characterization of Set Consensus. *Proc. 8th Int'l Conference on Distributed Computing and Networking (ICDCN'06)*, Springer Verlag LNCS #4308, pp. 331-341, 2006

- 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

- Gafni E., Rajsbaum S., Raynal M. and Travers C., The Committee Decisin Problem. *Proc. Latin American Theoretical Informatics Symposium (LATIN'06)*. Springer Verlag LNCS #3887, pp. 502-514, 2006

- Gafni E., Raynal M. and Travers C., Test&set, Adaptive Renaming and Set Agreement: a Guided Visit to Asynchronous Computability. *Proc. 26th IEEE Symposium on Reliable Distributed Systems (SRDS'07)*, IEEE Computer Society Press, pp. 93-102, 2007

The only slide to remember

Failures do modify

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