Set-Constrained Delivery Broadcast: Definition, Abstraction Power, and Computability Limits

D. IMBS[1] A. MOSTEFAOUI[2] M. PERRIN[2] M. RAYNAL[3]

[1]LIF, Université d'Aix-Marseille, France

[2]LINA, Université de Nantes, France

[3]IUF & IRISA, Université de Rennes, France &

Dpt of Comp., Polytechnic University, Hong Kong

Alternate title

A Broadcast Abstraction Suited to the the Family of Read/Write Implementable Objects

Remark:

Not all DC objects are Implementable from Read/Write in asynchronous failure-prone dsistrbiuted systems

Table of contents

Imbs Damien, Mostéfaoui Achour, Perrin Matthieu, and Raynal Michel,
Set-Constrained Delivery Broadcast:
Definition, Abstraction Power, and Computability Limits
ArXiv-1706-05267.pdf

Fundamental issues in DC

- Herlihy M. and Shavit N., The topological structure of asynchronous computability. *Journal of the ACM*, 46(6):858-923 (1999)

- Herlihy M.P., Rajsbaum S., and Raynal M., Power and limits of distributed computing shared memory models, *Theoretical Computer Science*, 509:3-24 (2013)

- Raynal M., What can be computed in a distributed system? *From Programs to Systems*, Springer LNCS 8415, pp. 209-224 (2014)

- Raynal M., A look at basics of distributed computing. *Proc. 36th IEEE Int'l Conference on Distributed Computing (ICDCS'16)*, IEEE Press, pp. 1-11 (2016)

- Informatics is a science of abstractions

  Informatics is a science of abstractions, and a main difficulty consists in providing users with a "desired level of abstraction and generality: one that is broad enough to encompass interesting new situations, yet specific enough to address the crucial issues", M. Fischer and M. Merritt, Appraising two decades of distributed computing theory research. *Distributed Computing*, 16(2-3):239-247 (2003)

- Classical examples
  - ⋆ Sequential programming languages
  - ⋆ Synchronization side: STM
  - ⋆ Distributed computing
    - ∗ Communication abstractions
    - ∗ Agreement objects
    - ∗ Symmetry-breaking objects
    - ∗ etc.

    in the presence of adversaries such as asynchrony, failures, mobility, etc.

On communication abstractions in DC

send/receive, basic broadcast: network machine language

Associating agreement objects and communication abstractions

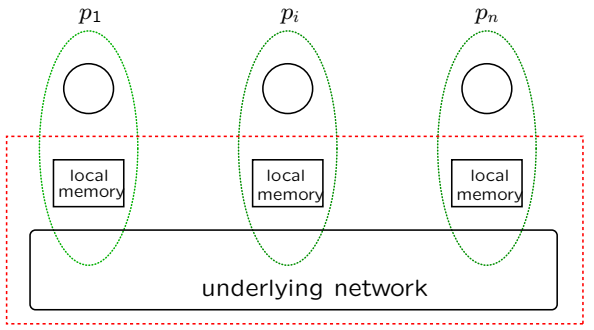
Concurrent object	Communication abstraction

Consensus	Total order broadcast
Causal memory	Causal order broadcast
k-set agreement object	k-BO-broadcast (DISC'17)

Process and low level communication model

- Process model:
  - ⋆ n sequential processes p_1, ..., p_n
  - ⋆ asynchrony: unknown arbitrary speed
- Failure model:
  - ⋆ Up to t processes may crash (unexpected halting)
- Communication model: network/machine level
  - ⋆ complete point-to-point network
  - ⋆ no bound on transfer delays (but finite)
  - ⋆ reliable (no loss, creation, duplication, alteration)
- Notation: $\mathcal{CAMP}_{n,t}[\emptyset]$

Implementing objects in a MP system



Peer-to-peer system model
Each p_i is both a client and a server

What is distributed computing?

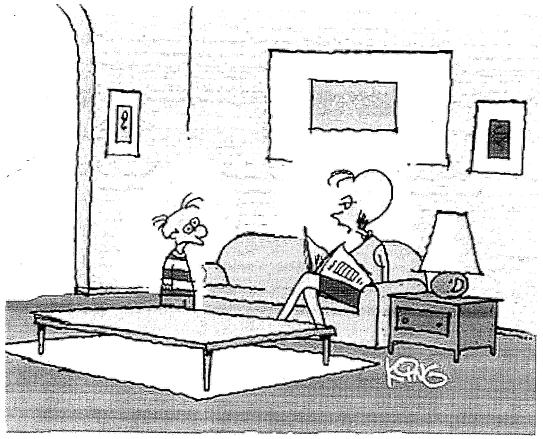
DC arises when one has to solve a problem in terms of entities (processes, agents, sensors, peers, actors, nodes, processors, ...) such that each entity has only a partial knowledge of the many parameters involved in the problem that has to be solved

"In sequential systems, computability is understood through the Church-Turing Thesis: anything that can be computed, can be computed by a Turing Machine.

In distributed systems, where computations require coordination among multiple participants, computability questions have a different flavor. Here, too, there are many problems which are not computable, but these limits to computability reflect the difficulty of making decisions in the face of ambiguity, and have little to do with the inherent computational power of individual participants."

- Herlihy M., Rajsbaum S., and Raynal M., Power and limits of distributed computing shared memory models. *Theoretical Computer Science*, 509:3-24 (2013)

Never forget ....



"No, you weren't downloaded. Your were born."

SCD-broadcast: definition

The SCD-Broadcast abstraction: definition (1)

SCD = Set-Constrained Delivery

- Two operations:
  - ⋆ scd_broadcast(m): broadcasts a message m
  - ⋆ scd_deliver(): returns a non-$\emptyset$ set of messages
- Five properties:
  - ⋆ Validity:
    If a process scd-delivers a set containing a message m, then m was scd-broadcast by some process
  - ⋆ Integrity:
    A msg is scd-delivered at most once by each process

The SCD-Broadcast abstraction: definition (2)

- MS-Ordering:
  A process p_i scd-delivers a message set ms_i containing a message m and later a message set ms'_i containing a message m'
  $\Rightarrow$
  no process scd-delivers first a message set ms'_j containing m' and later a message ms_j containing m
- Termination-1:
  If a non-faulty process scd-broadcasts a message m, it terminates its scd-broadcast invocation and scd-delivers a message set containing m
- Termination-2:
  If a process scd-delivers a message m, every non-faulty process scd-delivers a message set containing m

A simple example

- Messages SCD-broadcast by processes:
  m_1, m_2, m_3, m_4, m_5, m_6, m_7 and m_8
- SCD-deliveries:
  - ⋆ at p_1: $\{m_1, m_2\}$, $\{m_3, m_4, m_5\}$, $\{m_6\}$, $\{m_7, m_8\}$
  - ⋆ at p_2: $\{m_1\}$, $\{m_3, m_2\}$, $\{m_6, m_4, m_5\}$, $\{m_7\}$, $\{m_8\}$
  - ⋆ at p_3: $\{m_3, m_1, m_2\}$, $\{m_6, m_4, m_5\}$, $\{m_7\}$, $\{m_8\}$

A simple example: incorrect SCD-deliveries

- Messages SCD-broadcast by processes:
  m_1, m_2, m_3, m_4, m_5, m_6, m_7 and m_8
- SCD-deliveries:
  - ⋆ at p_1: $\{m_1, m_2\}$, $\{m_3, m_4, m_5\}$, ...
  - ⋆ at p_2: $\{m_1, m_3\}$, $\{m_2\}$, ...

The SCD-Broadcast abstraction

Graph interpretation

- Local scd-delivery order: $m \mapsto_i m'$
  - ⋆ p_i scd-delivers a set containing m
  - ⋆ before a set containing m'
- Global scd-delivery order: $\mapsto \; = \cup_{1 \leq i \leq n} \mapsto_i$

$\mapsto$ is partial order (no cycle)
(useful to understand and proofs)

The SCD-Broadcast abstraction: PROPERTIES (1)

If each message set contains a single message

- Validity + Integrity + Termination-1 + Termination-2

= Uniform Reliable Broadcast

The SCD-Broadcast abstraction: PROPERTIES (2)

A containment property

- let ms_i^ℓ = ℓ-th message set scd-delivered by p_i
- at some time: p_i scd-delivered the sequence of message sets $ms_i^1, \cdots, ms_i^x$
- let $MS_i^x = ms_i^1 \cup \cdots \cup ms_i^x$
- let $MS_j^y = ms_i^1 \cup \cdots \cup ms_j^y$
- $\forall\ i, j, x, y$: $(MS_i^x \subseteq MS_j^y) \vee (MS_j^y \subseteq MS_i^x)$

SCD-broadcast: a quick look at an implementation in $\mathcal{CAMP}_{n,t}[t < n/2]$

To simplify (wlog):
assume $\mathcal{CAMP}_{n,t}[t < n/2]$ enriched with FIFO-broadcast

Local variables at process p_i

- $buffer_i$: buffer (init. empty) where are stored quadruplets containing messages that have been fifo-delivered but not yet scd-delivered in a message set
- $to\_deliver_i$: set of quadruplets containing messages to be scd-delivered
- sn_i: local logical clock (init. 0), which increases by step 1 and measures the local progress of p_i

  Each application message scd-broadcast by p_i is identified by a pair $\langle i, sn\rangle$, where sn is the current value of sn_i
- $clock_i[1..n]$: array of logical dates

  $clock_i[j]$ is the greatest date x such that the application message m identified $\langle x, j\rangle$ has been scd-delivered by p_i

Basic data structure: Quadruplets

$$qdplt = \langle qdplt.msg, qdplt.sd, qdplt.sn, qdplt.cl[1..n] \rangle$$

- $qdplt.msg$: application message m
- $qdplt.sd$: id of the sender of this application message
- $qdplt.sn$:local date (seq. number) associated with m by its sender. Hence, $\langle qdplt.sd, qdplt.sn \rangle$ is the identity of m
- $qdplt.cl$: array of size n, initialized to $[+\infty, \ldots, +\infty]$

  $qdplt.cl[x]$: sequence number associated with m by p_x when it broadcast the message forward$(m, -, -, -, -)$; This last field is crucial in the scd-delivery by the process p_i of a message set containing m

  $qdplt.cl$ current knowledge on m dissemination

Implementing SCD in $\mathcal{CAMP}\{t < n/2\}$

operation scd_broadcast(m) **is**
  forward(m, i, sn_i, i, sn_i);
  wait$(\nexists\ qdplt \in buffer_i : qdplt.sd = i)$.

when the message forward$(m, sd, sn_{sd}, f, sn_f)$ **is fifo-delivered do** % from p_f
  forward$(m, sd, sn_{sd}, f, sn_f)$;
  try_deliver().

procedure forward$(m, sd, sn_{sd}, f, sn_f)$ **is**
  if $(sn_{sd} > clock_i[sd])$
    then if $(\exists\ qdplt \in buffer_i : qdplt.sd = sd \wedge qdplt.sn = sn_{sd})$
      then $qdplt.cl[f] \leftarrow sn_f$
      else $threshold[1..n] \leftarrow [\infty, \ldots, \infty]$; $threshold[f] \leftarrow sn_f$;
        let $qdplt \leftarrow \langle m, sd, sn_{sd}, threshold[1..n] \rangle$;
        $buffer_i \leftarrow buffer_i \cup \{qdplt\}$;
        fifo_broadcast forward$(m, sd, sn_{sd}, i, sn_i)$;
        $sn_i \leftarrow sn_i + 1$
    end if
  end if.

Broadcast of the messages forward$(m, sd, -, -)$ implement Uniform Reliable Broadcast of the application msg m

Implementing SCD in $\mathcal{CAMP}\{t < n/2\}$ (Cont'd)

procedure try_deliver() **is**
let $to\_deliver_i \leftarrow \{qdplt \in buffer_i : |\{f : qdplt.cl[f] < \infty\}| > \frac{n}{2}\}$;
while $(\exists qdplt \in to\_deliver_i, qdplt' \in buffer_i \setminus to\_deliver_i : |\{f : qdlpt.cl[f] < qdplt'.cl[f]\}| \leq \frac{n}{2})$
  do $to\_deliver_i \leftarrow to\_deliver_i \setminus \{qdplt\}$ **end while**;
if $(to\_deliver_i \neq \emptyset)$
  then for each $(qdplt \in to\_deliver_i$ such that $clock_i[qdplt.sd] < qdplt.sn)$
    do $clock_i[qdplt.sd] \leftarrow qdplt.sn$ **end for**;
  $buffer_i \leftarrow buffer_i \setminus to\_deliver_i$;
  $ms \leftarrow \{qdlpt.msg : \exists\ qdplt \in to\_deliver_i\}$; scd_deliver$(ms)$
end if.

From p_i 's point of view (uncertainty created by asynchrony and failures)

- Line **let**: $to\_deliver_i$: contains the messages that have been seen by a majority of processes
- **while loop**: no majority of processes has yet seen the message in $qdplt$ before the message in $qdplt'$

Implementing SCD in $\mathcal{CAMP}\{t < n/2\}$ (Cont'd)

- Reminder: $t < n/2$ is N & S to build read/write registers in $\mathcal{CAMP}_{n,t}[\emptyset]$, we will see it is also N & S to build SCD-broadcast in $\mathcal{CAMP}_{n,t}[\emptyset]$
- Distributed software engineering: All "technical details" are hidden in this algorithm which is designed and proved once for all!

Cost of SCD-broadcast implementation

- Assumption:
  - ⋆ Let Δ = message delay
  - ⋆ Local computation: zero cost
- Cost:
  - ⋆ Time: 2Δ ($2 \times$ network latency)
  - ⋆ Messages: n^2

On the computability power side

SCD-broadcast in Action

SCD-broadcast in Action

- Any RW-implementable object defined by a sequential specification
  - ⋆ Consistency conditions:
    - ∗ Atomicity (Linearizability)
    - ∗ Sequentially consistency
  - ⋆ Example: Conflict-free replicated data type
    - ∗ Data type with commutative operations (e.g. counter)
    - ∗ Data type with overwriting operations (e.g. read/write)
- Any read/write solvable distributed task

A few references

- Lamport L., On interprocess communication, Part I: basic formalism. *Distributed Computing*, 1(2):77-85 (1986)

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

-Lamport L., How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C28(9):690–691 (1979)

- Moran S. and Wolfstahl Y., Extended impossibility results for asynchronous complete networks. *Information Processing Letters*, 26(3):145-151 (1987)

- Biran O., Moran S., and Zaks S., A combinatorial characterization of the distributed tasks which are solvable in the presence of one faulty processor. *Proc. 7th ACM Symposium on Principles of Distributed Computing (PODC'88)*, ACM Press, pp. 263-275 (1988)

- Castañeda A., Rajsbaum S., and Raynal M., Specifying concurrent problems: beyond linearizability and up to tasks. *Proc. 29th Symposium on Distributed Computing (DISC'15)*, Springer LNCS 9363, pp. 420-435 (2015)

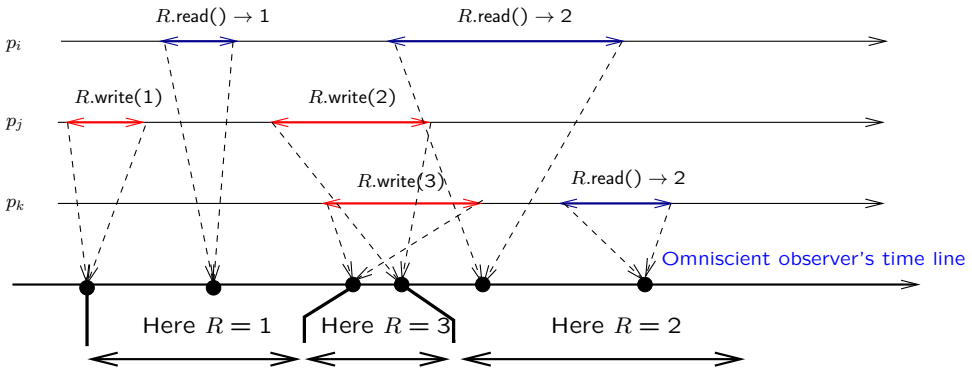
- Perrin M., Mostéfaoui A., Pétrolia M., and Jard Cl., On composition and implementation of sequential consistency. *Proc. 30th Int'l Symposium on Distributed Computing (DISC'16)*, Springer LNCS 9888, pp. 284-297 (2017)

Atomicity

- Operations appear as
  - ⋆ if they have been executed sequentially,
  - ⋆ and this sequence
    - ∗ complies with real-time order
      $\Rightarrow$ respects process order
    - ∗ satisfies the seq spec of a register
- Non-deterministic behavior when concurrency
- Why atomicity is fundamental:

  Atomic objects compose for free!

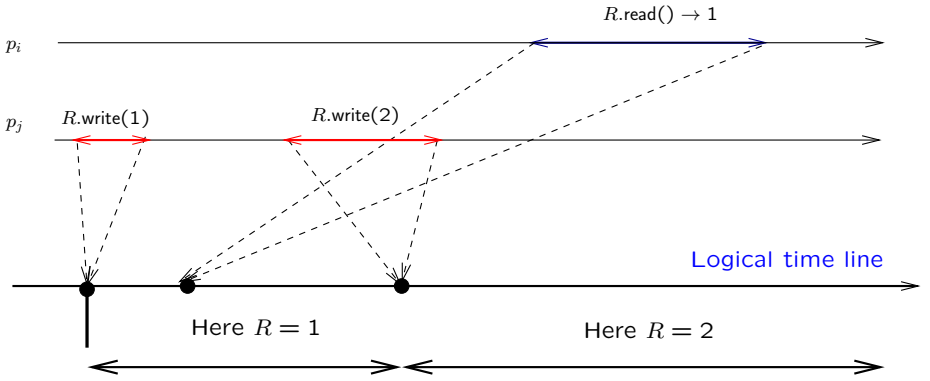
Example: Atomic read/write register



Sequential consistency

- Operations appear as
  - ⋆ if they have been executed sequentially,
  - ⋆ and this sequence
    - ∗ not required to comply with real-time order
      but respects process order
    - ∗ satisfies the seq spec of a register
- Non-deterministic behavior when concurrency
- Sequentially consistent objects do not compose for free!

Example: Sequentially consistent read/write register



SCD-broadcast in action: Snapshot object

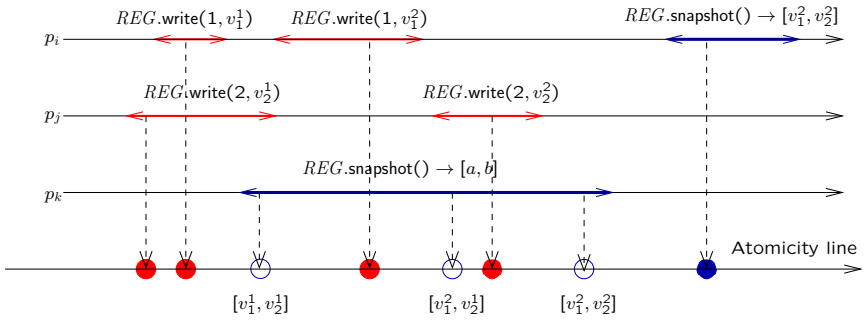
Reminder: read/write registers are universal in sequential computing

- Turing A. M., On computable numbers with an application to the Entscheidungsproblem. *Proc. of the London Mathematical Society*, 42:230-265 (1936)

Snapshot object

- Atomic Snapshot objects can be built in $\mathcal{CARW}_{n,t}[\emptyset]$
  - ⋆ array $REG[1..m]$ of atomic read/write registers with two operations, write() and snapshot()
  - ⋆ MWMR snapshot
    - ∗ write(r, v) assigns v to $REG[r]$
    - ∗ snapshot() returns the value of the full array as if the operation had been executed instantaneously (atomicity/sequential consistency)
  - ⋆ SWMR snapshot:
    - ∗ $m = n$ and
    - ∗ $r = i$ for write(r, v) by p_i

Example of an MWMR atomic snapshot object



$[v_1^1, v_2^2]$ cannot be output

Snapshot from SCD-broadcast

Local representation of the snapshot object REG

- $reg_i[1..m]$: current value of $REG[1..m]$, as known by p_i
- $done_i$: Boolean variable
- $tsa_i[1..m]$: array of timestamps associated with the values stored in $reg_i[1..m]$
  - ⋆ $tsa_i[j].date$ and $tsa_i[j].proc$ (timestamp of $reg_i[j]$)
- Lexicographical total order $<_{ts}$:
  - ⋆ $ts1 = \langle h1, i1\rangle$ and $ts2 = \langle h2, i2\rangle$
  - ⋆ $ts1 <_{ts} ts2 \stackrel{def}{=} (h1 < h2) \vee ((h1 = h2) \wedge (i1 < i2))$

Algorithm: snapshot operation

operation snapshot() **by** p_i **is**
$done_i \leftarrow$ `false`;
scd_broadcast SYNC (i);
wait($done_i$); % end of synchronization
return($reg_i[1..m]$).

- SYNC (i) synchronization message
- allows p_i to obtain an atomic value of $REG[1..m]$

Algorithm: write operation

operation write(r, v) **by** p_i **is**
$done_i \leftarrow$ `false`;
scd_broadcast SYNC (i);
wait($done_i$); % end of synchronization 1
$done_i \leftarrow$ `false`;
scd_broadcast WRITE $(r, v, \langle tsa_i[r].date + 1, i\rangle)$;
wait($done_i$). % end of synchronization 2

Algorithm: snapshot operation

when the message set
{WRITE$(r_{j_1}, v_{j_1}, \langle date_{j_1}, j_1\rangle)$, $\cdots$, WRITE$(r_{j_x}, v_{j_x}, \langle date_{j_x}, j_x\rangle)$,
SYNC(j_{x+1}), $\cdots$, SYNC(j_y) } **is scd-delivered do**

for each r such that WRITE$(r, -, -) \in$ the message set **do**
let $\langle date, writer\rangle$ = greatest timestamp in WRITE$(r, -, -)$;
if $(tsa_i[r] <_{ts} \langle date, writer\rangle)$
then let v the value in WRITE$(r, -, \langle date, writer\rangle)$;
$reg_i[r] \leftarrow v$; $tsa_i[r] \leftarrow \langle date, writer\rangle$
end if
end for;
if $\exists\ \ell : j_\ell = i$ **then** $done_i \leftarrow$ `true` **end if**.

Observation: no quorum at this abstraction level!

Time cost of MWMR snapshot

$O(n^2)$ messages in both cases

	RW-stacking	SCD-broadcast
snapshot	$n^2\Delta$	2Δ
write	$n\Delta$	4Δ

The case of a sequentially consistent snapshot object

Suppress the messages SYNC!

These messages ensure compliance wrt real-time

operation snapshot() **by** p_i **is**
return($reg_i[1..m]$).

operation write(r, v) **by** p_i **is**
$done_i \leftarrow$ `false`;
scd_broadcast WRITE $(r, v, \langle tsa_i[r].date + 1, i\rangle)$;
wait($done_i$).

when the message set
$\{\text{WRITE}(r_{j_1}, v_{j_1}, \langle date_{j_1}, j_1\rangle), \ \cdots, \ \text{WRITE}(r_{j_x}, v_{j_x}, \langle date_{j_x}, j_x\rangle)\}$
is scd-delivered do .....

SCD-broadcast in action: counter object

- Aspnes J. and Herlihy M., Wait-free data structures in the asynchronous PRAM model. *Proc. 2nd ACM Symposium on Parallel algorithms and architectures (SPAA'00)*, ACM Press, pp. 340-349 (1990)

-Shapiro M., Preguiça N., Baquero C., and Zawirski M., Conflict-free replicated data types. *Proc. 13th Int'l Symp. on Stabilization, Safety, and Security of Distr. Systems (SSS'11)*, Springer LNCS 6976, pp. 386-400 (2011)

Reminder: a counter has commutative operations (CRDT)
Conflict-free replicated data type

Counter operations

- C.increase() adds 1 to C
- C.decrease() subtracts 1 to C
- C.read() returns the value of C

Counter algorithm (1)

operation increase() **is**
$done_i \leftarrow$ `false`; scd_broadcast PLUS (i); wait($done_i$);
return().

operation decrease() **is**
same as increase() where PLUS(i) is replaced by MINUS(i).

operation read() **is**
$done_i \leftarrow$ `false`; scd_broadcast SYNC (i); wait($done_i$);
return($counter_i$).

Possibility to add an approximate read operation returning the local value of the counter ($counter_i$)

Counter algorithm (2)

when the message set
{ PLUS(j_1), ...,MINUS(j_x), ..., SYNC(j_y), ... }
is scd-delivered do
let p = number of messages PLUS() in the message set;
let m = number of messages MINUS() in the message set;
$counter_i \leftarrow counter_i + p - m$;
if $\exists \ell$: $j_\ell = i$ **then** $done_i \leftarrow$ `true` **end if**.

From atomicity to sequential consistency (1)

lsc_i: local synchronization operation counter (init 0)

operation increase() **is**
$lsc_i \leftarrow lsc_i + 1$;
scd_broadcast PLUS (i);
return().

operation decrease() **is**
same as increase() where PLUS(i) is replaced by MINUS(i)

operation read() **is**
wait($lsc_i = 0$);
return($counter_i$).

From atomicity to sequential consistency (2)

when the message set
{ PLUS(j_1), ...,MINUS(j_x), ..., SYNC(j_y), ... }
is scd-delivered do
let p = number of messages PLUS() in the message set;
let m = number of messages MINUS() in the message set;
$counter_i \leftarrow counter_i + p - m$;
$lsc_i \leftarrow lsc_i -$ # msgs PLUS() and MINUS() in the msg set

SCD-broadcast in Action: the lattice agreement task

On lattices

- S: partially ordered set
- $\leq$: its partial order
- Given $S' \subseteq S$, an upper bound of S' is an element x of S such that $\forall\ y \in S' :\ y \leq x$
- The least upper bound of S' is an upper bound z of S' such that, for all upper bounds y of S', $z \leq y$
- S is called a *semilattice* if all its finite subsets have a least upper bound. Let lub(S') denotes the least upper bound of S'

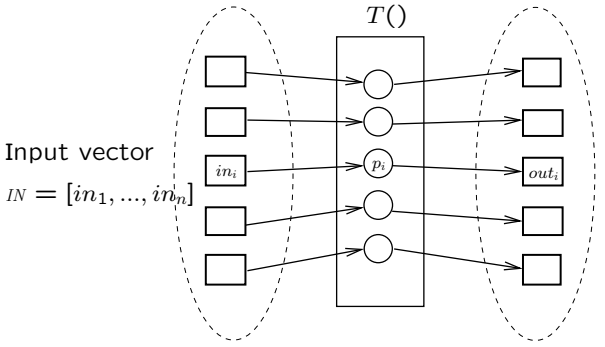
Lattice agreement task

- p_i has an input value $in_i \in$ a semilattice S
- propose(in_i) returns an element $out_i \in S$
- Properties:
  - ⋆ Validity: $in_i \leq out_i \leq \text{lub}(\{in_1, \ldots, in_n\})$
  - ⋆ Containment: If p_i decides out_i and p_j decides out_j, we have $out_i \leq out_j$ or $out_j \leq out_j$
  - ⋆ Termination: If a non-faulty proposes a value, it decides a value

Lattice agreement is task

- Input:
  - ⋆ Each p_i has its own input in_i (known only by it)
  - ⋆ $I = [in_1, \cdots, in_n]$: a distributed input vector
  - ⋆ $\mathcal{I}$: set of all allowed input vectors
- Output:
  - ⋆ $O = [out_1, \ldots, out_n]$: distributed output vector
  - ⋆ out_i: output of process p_i
  - ⋆ $\mathcal{O}$: set of all allowed output vectors
- Task: mapping T s.t. $\forall I \in \mathcal{I}$: $T(I) \subseteq \mathcal{O}$
- Process crashes

Distributed task



Objects vs tasks: see Castañeda A., Rajsbaum S., and Raynal M.,

* Specifying concurrent problems: beyond linearizability and up to tasks. *Proc. 29th Symposium on Distributed Computing (DISC'15)*, Springer LNCS 9363, pp. 420-435 (2015)

* Long-lived tasks. *Proc. 5th Int'l Conference on Networked Systems (NETYS'17)*, Springer LNCS 10299, pp. 439-454 (2017)

Lattice agreement algorithm

operation propose(in_i) **is**
$done_i \leftarrow$ `false`; scd_broadcast VAL (i, in_i); wait($done_i$);
return(lub(rec_i)).

when the message set
{ VAL(j_1, v_{j_1}), ..., VAL(j_x, v_{j_x})} **is scd-delivered do**
$rec_i \leftarrow rec_i \cup \{v_{j_1}, \ldots, v_{j_x}\}$;
if $\exists \ell$: $j_\ell = i$ **then** $done_i \leftarrow$ `true` **end if**.

First read/write lattice agreement algorithm!

On the computability limit side

From MWMR Snapshot to SCD-Broadcast

Building SCD-Broadcast
in $\mathcal{CARW}_{n,t}$[Snapshot] ($\mathcal{CARW}_{n,t}[\emptyset]$)

Shared objects

ϵ: empty sequence
$\oplus$: concatenation

- $SENT[1..n]$: SWMR snapshot object, initialized to $[\emptyset, \cdots, \emptyset]$
  $SENT[i]$ = messages scd-broadcast by p_i
- $SETS\_SEQ[1..n]$: SWMR snapshot object, initialized to $[\epsilon, \cdots, \epsilon]$
  $SETS\_SEQ[i]$ = seq. of msg sets scd-delivered by p_i

Local objects

- $sent_i$: local copy of the snapshot object $SENT$
- $sets\_seq_i$: local copy of the snapshot object $SETS\_SEQ$.
- $to\_deliver_i$: set whose aim is to contain the next message set that p_i has to scd-deliver
- members(set_seq) returns the set of messages in set_seq

Algorithm (1)

operation scd_broadcast(m) **by** p_i **is**
$sent_i[i] \leftarrow sent_i[i] \cup \{m\}$; $SENT$.write($sent_i[i]$); progress().

background task T **is**
repeat forever progress() **end repeat**.

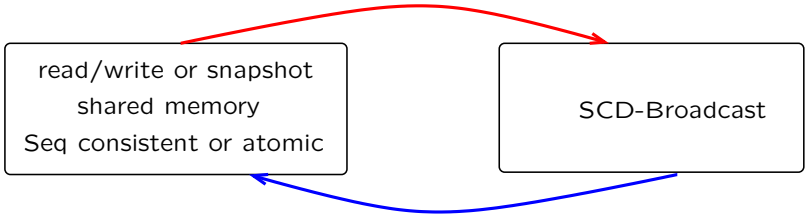
Algorithm (2)

procedure progress() **by** p_i **is**
enter_mutex();
catch_up();
$sent_i \leftarrow SENT$.snapshot();
$to\_deliver_i \leftarrow (\cup_{1 \leq j \leq n}\ sent_i[j]) \setminus$ members($sets\_seq_i[i]$);
if $(to\_deliver_i \neq \emptyset)$
then $sets\_seq_i[i] \leftarrow sets\_seq_i[i] \oplus to\_deliver_i$;
$SETS\_SEQ$.write($i, sets\_seq_i[i]$);
scd_deliver($to\_deliver_i$)
end if;
exit_mutex().

Algorithm (3)

procedure catch_up() **by** p_i **is**
$sets\_seq_i \leftarrow SETS\_SEQ$.snapshot();
while
$(\exists j, set : set$ first set in $sets\_seq_i[j] \wedge set \not\subseteq$ members($sets\_seq_i[i]$)
do $to\_deliver_i \leftarrow set \setminus$ members($sets\_seq_i[i]$);
$sets\_seq_i[i] \leftarrow sets\_seq_i[i] \oplus to\_deliver_i$;
$SETS\_SEQ$.write($i, sets\_seq_i[i]$);
scd_deliver($to\_deliver_i$)
end while.

The computability limit of SCD-broadcast



Hence the implementation of SCD-broadcast previously given in the system model $\mathcal{CAMP}_{n,t}[t < n/2]$ is t-resilient optimal

From sequentially consistency to atomicity

From non-composable to composable snapshot objects

The power of the messages SYNC() (real-time compliance)

- Start from a sequentially consistent snapshot object ($\mathcal{CARW}_{n,t}$[Snapshot])
- Build SCD-Broadcast on top of it
  we obtain $\mathcal{CAMP}_{n,t}$[SCD-broadcast]
- Build atomic snapshot on top of $\mathcal{CAMP}_{n,t}$[SCD-broadcast]

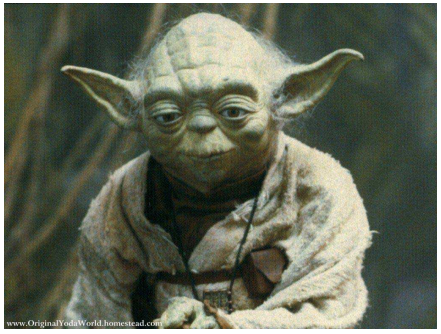
First (?) systematic construction from SC to Atomicity

Conclusion

Conceptual issues

- Better understanding of basic mechanisms needed to implement a read/write shared memory
- SCD-broadcast captures the "right" abstraction level
- Simplicity of the proposed (register/snapshot) algo.
- Genericity of the proposed algorithms wrt
  - ⋆ read/write vs snapshot objects (same algorithms)
  - ⋆ atomicity vs sequential consistency (SYNC msgs)
- SCD-broadcast generic communication pattern
- Distributed software engineering
- On progr. languages for DC (reminder DC ≠ PP)
- Distributed computing vs Parallel computing

More important: He Told me



"Algorithms are at the core of Informatics"