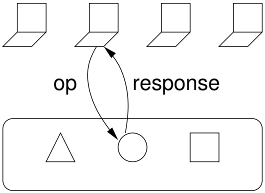
When is Recoverable Consensus Harder Than Consensus?

Carole Delporte-Gallet	*IRIF, Université Paris Cité*	*France*
Panagiota Fatourou	*LIPADE, Université Paris Cité*	*France*
	& FORTH ICS & University of Crete	*Greece*
Hugues Fauconnier	*IRIF, Université Paris Cité*	*France*
Eric Ruppert	*York University*	*Canada*

July 27, 2022

Context

Classical shared memory

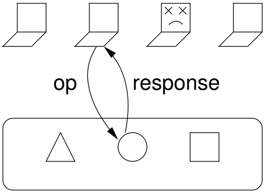


processes

shared memory
with objects of
various types

Context

Classical shared memory
Wait-free algorithms



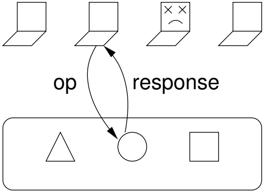
asynchronous processes

shared memory with objects of various types

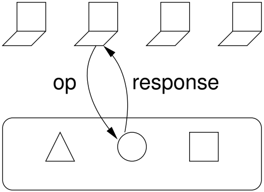
Permanent crash failures

Context

Classical shared memory
Wait-free algorithms



Non-volatile shared memory
Recoverable algorithms



asynchronous processes

shared memory with objects of various types

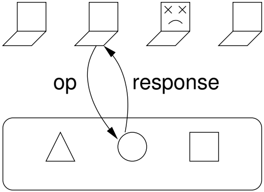
Permanent crash failures

Crash-recovery failures

-erase *local* memory of process (including programme counter)

Context

Classical shared memory
Wait-free algorithms

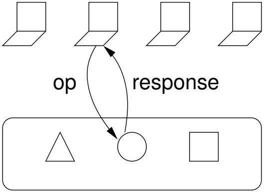


asynchronous processes

shared memory with objects of various types

Permanent crash failures

Non-volatile shared memory
Recoverable algorithms



Crash-recovery failures
- erase *local* memory of process (including programme counter)

Algorithm $A \xrightarrow{?}$ Algorithm A'

Consensus

Consensus Problem

Each process has an input value and must output a value.

- Each output is the input of some process
- No 2 outputs differ
- If a process takes enough steps without crashing, it outputs a value

Recoverable Consensus

Consensus in context of crash-recovery failures

Recoverable Consensus Problem (RC) [Golab SPAA 2020]

Each process has an input value and must output a value.

- Each output is the input of some process
- No 2 outputs differ (including 2 outputs of 1 process)
- If a process takes enough steps between crashes, it outputs a value

Consensus Hierarchy

cons(T)

maximum number of processes that can solve wait-free consensus using objects of type T and registers
tolerating permanent crashes

rcons(T)

maximum number of processes that can solve recoverable consensus using objects of type T and registers
tolerating crash-recovery failures

Recoverable Consensus Hierarchy

cons(T)

maximum number of processes that can solve wait-free consensus using objects of type T and registers
tolerating permanent crashes

rcons(T)

maximum number of processes that can solve recoverable consensus using objects of type T and registers
tolerating crash-recovery failures

Significance of Consensus

Consensus numbers tell us about wait-free implementations [Herlihy 1991]

Universality

$cons(T) \geq n \Rightarrow T$ implements *every* object for n processes

Non-implementability

$cons(T) < cons(T') = n \Rightarrow T$ cannot implement T' for n processes.

Analogous results for $rcons(T)$.
[Berryhill, Golab, Tripunitara OPODIS 2015; this work]

YORK UNIVERSITÉ UNIVERSITY

Significance of Recoverable Consensus

Consensus numbers tell us about wait-free implementations [Herlihy 1991]

Universality

$cons(T) \geq n \Rightarrow T$ implements *every* object for n processes

Non-implementability

$cons(T) < cons(T') = n \Rightarrow T$ cannot implement T' for n processes.

Analogous results for $rcons(T)$.
[Berryhill, Golab, Tripunitara OPODIS 2015; this work]

Key Question

$rcons(T) \leq cons(T)$

Any RC algorithm also solves consensus.
So RC is at least as hard as consensus.

Question

Is RC (much) harder than consensus?
Can $rcons(T)$ be (much) smaller than $cons(T)$?

Key Question

$rcons(T) \leq cons(T)$

Any RC algorithm also solves consensus.
So RC is at least as hard as consensus.

Question

Is RC (much) harder than consensus?
Can $rcons(T)$ be (much) smaller than $cons(T)$?

Key Question

$rcons(T) \leq cons(T)$

Any RC algorithm also solves consensus.
So RC is at least as hard as consensus.

Question

Is RC (much) harder than consensus?
Can $rcons(T)$ be (much) smaller than $cons(T)$?

Previous Results

System-wide crash-recovery failures

$rcons(T) = 2 \Leftrightarrow cons(T) = 2.$ [Golab 2020]

Previous Results

System-wide crash-recovery failures

$rcons(T) = 2 \Leftrightarrow cons(T) = 2.$ [Golab 2020]

Independent crash-recovery failures:

- With *known bound* on number of failures: $rcons(T) = cons(T)$. [Golab 2020]
- Necessary condition for $rcons(T) \geq 2$. [Golab 2020]

Previous Results

System-wide crash-recovery failures

$rcons(T) = 2 \Leftrightarrow cons(T) = 2.$ [Golab 2020]

Independent crash-recovery failures:

- With *known bound* on number of failures:
  $rcons(T) = cons(T)$. [Golab 2020]
- Necessary condition for $rcons(T) \geq 2$. [Golab 2020]

Previous and New Results

System-wide crash-recovery failures

$rcons(T) = 2 \Leftrightarrow cons(T) = 2.$ [Golab 2020]

$rcons(T) = cons(T)$

Independent crash-recovery failures:

- With *known bound* on number of failures:
  $rcons(T) = cons(T).$ [Golab 2020]
- Necessary condition for $rcons(T) \geq 2.$ [Golab 2020]

Previous and New Results

System-wide crash-recovery failures

$rcons(T) = 2 \Leftrightarrow cons(T) = 2.$ [Golab 2020]

$rcons(T) = cons(T)$

Independent crash-recovery failures:

- With *known bound* on number of failures:
  $rcons(T) = cons(T)$. [Golab 2020]
- Necessary condition for $rcons(T) \geq 2$. [Golab 2020]
  We (partially) characterize when $rcons(T) = n$ for all n.

Main Results

Focus on readable objects, independent failure model

We define n-recording property of shared object types.

n-recording
$\Downarrow$
n-proc RC solvable
$\Downarrow$
$(n-1)$-recording

Main Results

Focus on readable objects, independent failure model

We define *n*-recording property of shared object types.

n-recording

$\Downarrow$

n-proc RC solvable

$\Downarrow$

$(n-1)$-recording

$\Downarrow$

$(n-1)$-proc RC solvable

$\Downarrow$

$(n-2)$-recording

$\Downarrow$

$(n-2)$-proc RC solvable

Main Results

Focus on readable objects, independent failure model

We define *n*-recording property of shared object types.

n-recording

$\Downarrow$

n-proc RC solvable $\Rightarrow$ n-process consensus solvable

$\Downarrow$ $\Updownarrow$ [Ruppert PODC 1997]

$(n-1)$-recording $\not\Leftarrow$ n-discerning

$\Downarrow$

$(n-1)$-proc RC solvable

$\Downarrow$

$(n-2)$-recording $\Leftarrow$

$\Downarrow$

$(n-2)$-proc RC solvable

Main Results

Focus on readable objects, independent failure model

We define *n*-recording property of shared object types.

n-recording

$\Downarrow$

n-proc RC solvable $\Rightarrow$ n-process consensus solvable

$\Updownarrow$ [Ruppert PODC 1997]

$(n-1)$-recording $\not\Leftarrow$ n-discerning

$\Downarrow$

$(n-1)$-proc RC solvable

$\Downarrow$

$(n-2)$-recording $\Longleftarrow$ (n-discerning)

$\Downarrow$

$(n-2)$-proc RC solvable

Corollary

$cons(T) - 2 \leq rcons(T) \leq cons(T)$

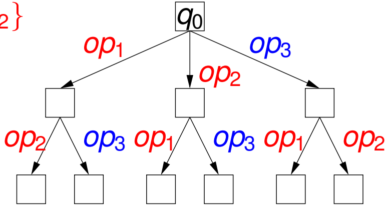
n-recording Property: First Attempt

- Pick a starting state q_0.
- Divide n processes into two teams *Red* and *Blue*.
- Assign an operation op_i to each process p_i.

Look at states reached from q_0 by permutations of $op_1, \ldots, op_n$.

Example: 3 processes p_1, p_2, p_3.

$Red = \{p_1, p_2\}$
$Blue = \{p_3\}$



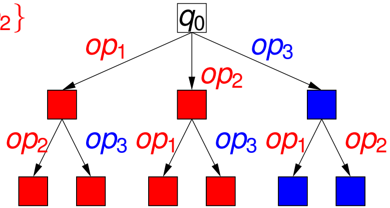
n-recording Property: First Attempt

- Pick a starting state q_0.
- Divide n processes into two teams *Red* and *Blue*.
- Assign an operation op_i to each process p_i.

Look at states reached from q_0 by permutations of $op_1, \ldots, op_n$.

Example: 3 processes p_1, p_2, p_3.

$Red = \{p_1, p_2\}$
$Blue = \{p_3\}$



State should *record* which team did the *first* operation after q_0.

- Red states are disjoint from blue states
- q_0 is neither red nor blue

Sufficiency of *n*-recording Property

Team RC problem

Same as RC with constraint: each team gets a common input

Theorem

An n-recording type T can solve n-process team RC.

Proof.

Use object O of type T (initially q_0) and one register per team

Decide(v)
    write v into my team's register
    if O's state is q_0 then perform op_i on O
    read O and determine which team accessed O first
    output value from that team's register

If red process accesses O first, state stays red forever.

If blue process accesses O first, state stays blue forever. □

Sufficiency of *n*-recording Property

Team RC problem

Same as RC with constraint: each team gets a common input

Theorem

An n-recording type T can solve n-process team RC.

Proof.

Use object O of type T (initially q_0) and one register per team

Decide(v)
   write v into my team's register
   if O's state is q_0 then perform op_i on O
   read O and determine which team accessed O first
   output value from that team's register

If red process accesses O first, state stays red forever.

If blue process accesses O first, state stays blue forever. □

Sufficiency of n-recording Property

Team RC problem

Same as RC with constraint: each team gets a common input

Theorem

An n-recording type T can solve n-process team RC.

Proof.

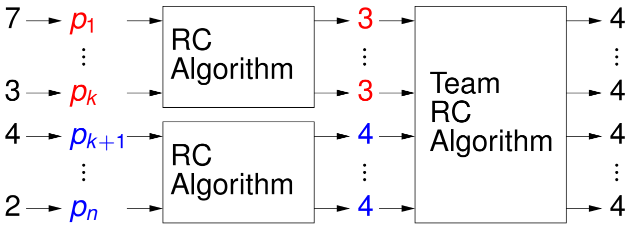
Use object O of type T (initially q_0) and one register per team

Decide(v)
  write v into my team's register
  if O's state is q_0 then perform op_i on O
  read O and determine which team accessed O first
  output value from that team's register

If red process accesses O first, state stays red forever.

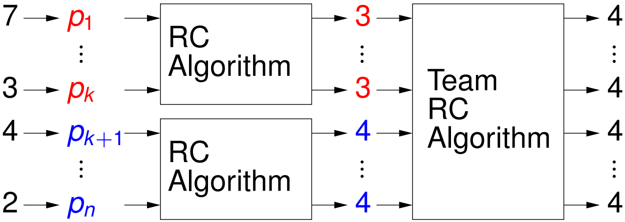
If blue process accesses O first, state stays blue forever. □

Sufficiency: Solving RC using team RC



[Neiger 1995, Ruppert 1997]

Sufficiency: Solving RC using team RC



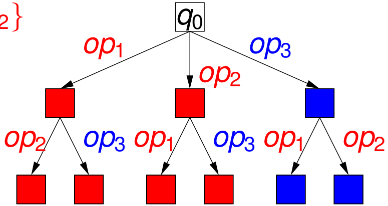
Solve smaller RC instances recursively.
→ Yields a tournament algorithm

[Neiger 1995, Ruppert 1997]

Refining the Condition

Example: 3 processes p_1, p_2, p_3.

$Red = \{p_1, p_2\}$
$Blue = \{p_3\}$

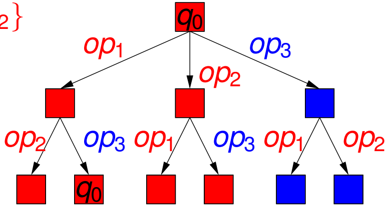


- Red states are disjoint from blue states
- q_0 is neither red nor blue
- q_0 *can* be red *if* there is only one blue process
- q_0 *can* be blue *if* there is only one red process

Refining the Condition

Example: 3 processes p_1, p_2, p_3.

$Red = \{p_1, p_2\}$
$Blue = \{p_3\}$

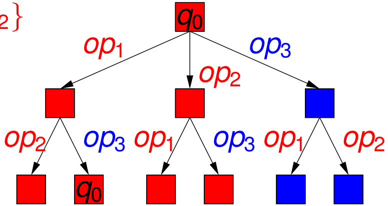


- Red states are disjoint from blue states
- ~~q_0 is neither red nor blue~~
- q_0 *can* be red *if* there is only one blue process
- q_0 *can* be blue *if* there is only one red process

Refining the Condition

Example: 3 processes p_1, p_2, p_3.

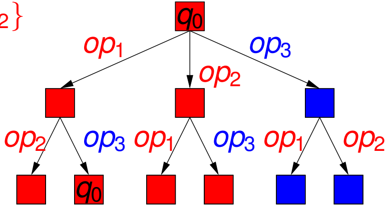
$Red = \{p_1, p_2\}$
$Blue = \{p_3\}$



- Red states are disjoint from blue states
- ~~q_0 is neither red nor blue~~
- q_0 *can* be red *if* there is only one blue process
- q_0 *can* be blue *if* there is only one red process

Modified Definition Still Sufficient for Team RC

$Red = \{p_1, p_2\}$
$Blue = \{p_3\}$



Key idea to modify team RC algorithm if q_0 is red:
p_3 performs op_3 on O only if
p_3 sees state is q_0 *and* no red process has woken up.

$\Rightarrow$ Ensures that if state of O returns to q_0, it remains red forever.

n-recording Property

Definition

A readable type T is *n-recording* if there exist

- an initial state q_0
- partition of n processes into red and blue team,
- operations $op_1, \ldots, op_n$

such that

- Red states are disjoint from blue states
- either q_0 is not red or there is only 1 blue process
- either q_0 is not blue or there is only 1 red process.

Red state: reachable from q_0 by sequence of operations $op_{i_1}, \ldots, op_{i_k}$ with distinct indices starting with red op_{i_1}
Blue state defined symmetrically.

Sufficiency

Theorem (Sufficient Condition)

T *is n-recording* $\Rightarrow$ *rcons*$(T) \geq n$

Proof Sketch

Build team RC algorithm using *n*-recording object.
Use team RC in tournament to solve RC.

Sufficiency

Theorem (Sufficient Condition)

T *is n-recording* $\Rightarrow rcons(T) \geq n$

Proof Sketch

Build team RC algorithm using n-recording object.
Use team RC in tournament to solve RC.

Necessity

Theorem (Necessary Condition)

T is $(n-1)$*-recording* $\Leftarrow$ *rcons*$(T) \geq n$

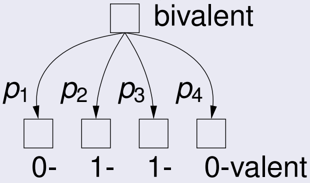
Necessity

Theorem (Necessary Condition)

T is $(n-1)$-recording $\Leftarrow$ $rcons(T) \geq n$

Ideas for proof

- Valency argument
- Critical configuration used to define q_0, $op_1, \ldots, op_n$, teams



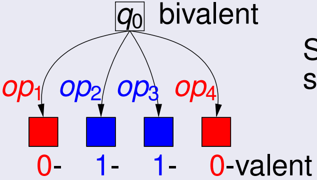
Necessity

Theorem (Necessary Condition)

T *is* $(n-1)$*-recording* $\Leftarrow rcons(T) \geq n$

Ideas for proof

- Valency argument
- Critical configuration used to define q_0, $op_1, \ldots, op_n$, teams



Show that these choices satisfy definition

Necessity

Theorem (Necessary Condition)

T *is* $(n-1)$*-recording* $\Leftarrow rcons(T) \geq n$

Ideas for proof

- Valency argument
- Critical configuration used to define q_0, $op_1, \ldots, op_n$, teams
- Challenge: Not all executions produce output.

  Solution: Use restricted set of runs:
  - Only p_1 can crash.
  - # crashes by $p_1 \leq$ # total steps by $p_2, \ldots, p_n$.

  Ensures every run produces output.

Necessity

Theorem (Necessary Condition)

T is $(n-1)$*-recording* $\Leftarrow$ *rcons*$(T) \geq n$

Ideas for proof

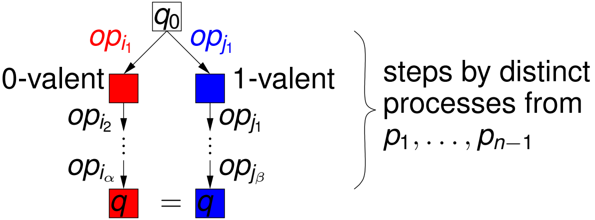
- Valency argument
- Critical configuration used to define q_0, $op_1, \ldots, op_n$, teams
- Challenge: Not all executions produce output.
  Solution: Use restricted set of runs:
  - Only p_1 can crash.
  - # crashes by $p_1 \leq$ # total steps by $p_2, \ldots, p_n$.

  Ensures every run produces output.
- Challenge: Must construct runs that belong to this set.
  Solution: “Extra process” takes steps to enable crashes.

Example of Valency Argument

Prove red and blue states are disjoint in definition of $(n-1)$-recording.

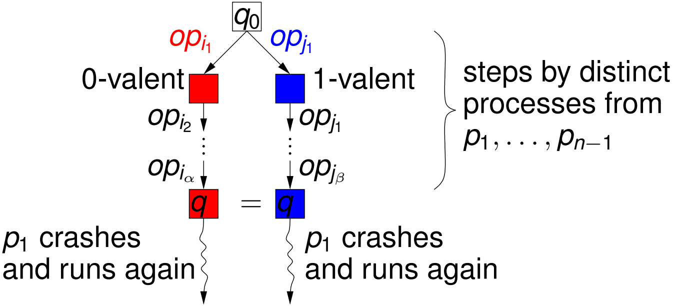
Suppose not.



Example of Valency Argument

Prove red and blue states are disjoint in definition of $(n-1)$-recording.

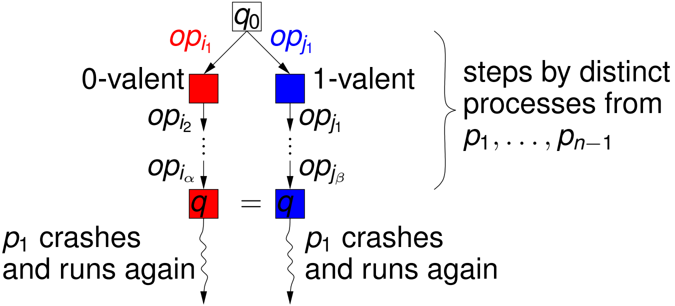
Suppose not.



Example of Valency Argument

Prove red and blue states are disjoint in definition of $(n-1)$-recording.

Suppose not.

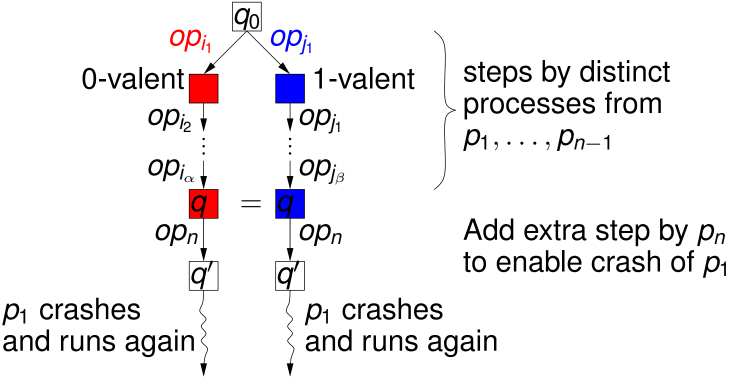


But crashing p_1 might not be allowed if one sequence is just op_1.

Example of Valency Argument

Prove red and blue states are disjoint in definition of $(n-1)$-recording.

Suppose not.



Main Results (Readable Types, Indep. Failures)

n-recording

$\Downarrow$

n-proc RC solvable $\Rightarrow$ *n*-process consensus solvable

$\Updownarrow$ [Ruppert PODC 1997]

$(n-1)$-recording $\not\Leftarrow$ *n*-discerning

$\Downarrow$

$(n-1)$-proc RC solvable

$\Downarrow$

$(n-2)$-recording $\Longleftarrow$ (from *n*-discerning)

$\Downarrow$

$(n-2)$-proc RC solvable

Corollary

$cons(T) - 2 \leq rcons(T) \leq cons(T)$

Examples

Sometimes $rcons(T) = cons(T)$ and
sometimes $rcons(T) < cons(T)$.

Bonus Result: Robustness

Theorem

If RC is solvable using several readable types together, then RC is solvable using one of those types.

$$rcons(T_1, \ldots, T_k) = \max(rcons(T_1), \ldots, rcons(T_k))$$

Bonus Result: Robustness

Theorem

If RC is solvable using several readable types together, then RC is solvable using one of those types.

$$rcons(T_1, \ldots, T_k) = \max(rcons(T_1), \ldots, rcons(T_k))$$

Research Directions

- Is $rcons(T) = cons(T) - 2$ for some readable type T?
- Is $rcons(T) << cons(T)$ for some non-readable type T?
- Close gap between necessary and sufficient condition.
  First step: Is 2-recording necessary for solving 2-process RC?
- Efficient algorithms for RC and recoverable implementations of data structures