236755
Topic 5:
Sub-Consensus Problems

Winter 2019-20

Prof. Hagit Attiya

Problems Weaker than Consensus

- Take fewer rounds in the synchronous model
- Have wait-free algorithms in the asynchronous model, using only reads and writes
- But there are still limitations

Two examples:

- Set agreement
- Adaptive / non-adaptive renaming

k-Set Agreement

Each process starts with an input x_i and has to decide on an output y_i, such that

- k-agreement: at most k different values are decided
- Validity: every decided value is an input

There is a wait-free algorithm for k-Set Agreement **if and only if $k \geq n$**

Algorithm is trivial…

Impossibility of $(n-1)$-Set Agreement

Given a wait-free $(n-1)$-set agreement algorithm

C_m= all executions such that:

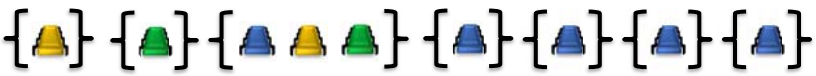
- Only processes $p_0, \ldots, p_{m-1}$ take steps
- Process p_i has input i
- All values $0, \ldots, m-1$ are decided

We show that $C_n \neq \emptyset$, for restricted executions

Immediate Snapshot (IS) Execution

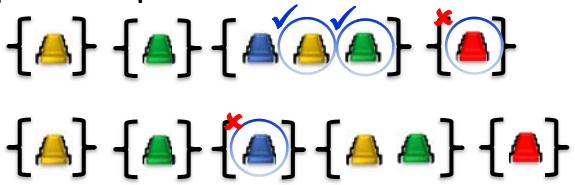
A sequence of **blocks**: sets of processes that

- Write together and then
- Scan (read everything) together



IS Executions: Indistinguishability

- **Indistinguishable** executions: $\alpha \sim_p \alpha'$,
  if process p has the same view in α and in α'



IS Executions: Seen & Unseen Processes

- **Indistinguishable** executions: $\alpha \sim_p \alpha'$, if process p has the same view in α and in α'
- A process is **seen** in α if it appears in some other process' view; otherwise, it is **unseen**

IS Executions: AR Lemma

- **Indistinguishable** executions: $\alpha \sim_p \alpha'$, if process p has the same view in α and in α'
- A process is **seen** in α if it appears in some other process' view; otherwise, it is **unseen**

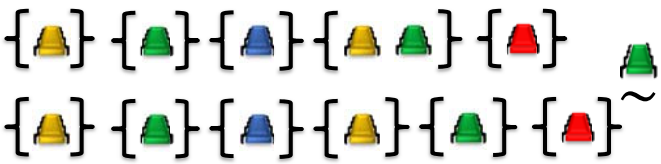
If p_i is seen in an IS execution α by processes P, then there is a **unique** IS execution $\alpha' \neq \alpha$ by P s.t.

- $\alpha \sim_{P-p_i} \alpha'$

Also, p_i is seen in α'

AR Lemma: Proof Sketch (Case 1)

p_i not alone in its last seen round (e.g., yellow)



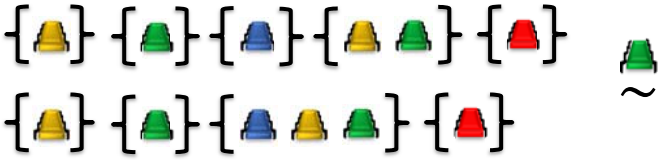
If p_i is seen in an IS execution α by processes P, then there is a **unique** IS execution $\alpha' \neq \alpha$ by P s.t.

- $\alpha \sim_{P-p_i} \alpha'$

Also, p_i is seen in α'

AR Lemma: Proof Sketch (Case 2)

p_i alone in its last seen round (e.g., blue)



If p_i is seen in an IS execution α by processes P, then there is a **unique** IS execution $\alpha' \neq \alpha$ by P s.t.

- $\alpha \sim_{P-p_i} \alpha'$

Also, p_i is seen in α'

Set Agreement Lower Bound

C_m = all IS executions such that:

- Only processes $p_0, \ldots, p_{m-1}$ take steps
- Process p_i has input i
- All values $0, \ldots, m-1$ are decided

Relies on C_m being finite

The size of C_m, $1 \leq m \leq n$, is odd

Proof by induction on m

C_1 contains one IS p_0-solo execution ⇨ decide 0

The induction step shows $|C_m| \equiv |C_{m+1}| (\mathrm{mod}\ 2)$

$|C_m| \equiv |C_{m+1}| (\mathrm{mod}\ 2)$

X_{m+1} = all tuples (α, p_i), IS execution α, in which processes $\neq p_i$ decide $\{0, \ldots, m-1\}$

$|X_{m+1}| \equiv |C_{m+1}| (\mathrm{mod}\ 2)$

$X'_{m+1} \subseteq X_{m+1}$ such that $\{0, \ldots, m\}$ are decided in α

Claim 1. $X'_{m+1} \overset{1-1}{\longleftrightarrow} C_{m+1}$

⇨ For $(\alpha, p_i) \in X'_{m+1}$, $\alpha \in C_{m+1}$, p_i is the unique process deciding m in α and there is no other $(\alpha, p_j) \in X'_{m+1}$

⇦ For $\alpha \in C_{m+1}$, a unique process p_i decides m in α, giving a unique $(\alpha, p_i) \in X'_{m+1}$

$$|C_m| \equiv |X_{m+1}| \equiv |C_{m+1}| (\text{mod } 2)$$

X_{m+1} = all tuples (α, p_i), IS execution α, in which processes $\neq p_i$ decide $\{0, \dots, m-1\}$

$$|X_{m+1}| \equiv |C_{m+1}| (\text{mod } 2)$$

$X'_{m+1} \subseteq X_{m+1}$ such that $\{0, \dots, m\}$ are decided in α

Claim 2. $|X_{m+1} \setminus X'_{m+1}|$ is even

$(\alpha, p_i) \in X_{m+1} \setminus X'_{m+1}$ ⇨ p_i decides $v \neq m$ in α

⇨ a unique process $p_j \neq p_i$ also decides v in α

⇨ a partitioning of $X_{m+1} \setminus X'_{m+1}$ into pairs

$$|C_m| \equiv |X_{m+1}| \equiv |C_{m+1}| (\text{mod } 2)$$

X_{m+1} = all tuples (α, p_i), IS execution α, in which processes $\neq p_i$ decide $\{0, \dots, m-1\}$

$$|X_{m+1}| \equiv |C_m| (\text{mod } 2)$$

1st subset: $(\alpha, p_m) \in X_{m+1}$ and p_m is unseen in α

In a prefix of α, processes $p_0, \dots, p_{m-1}$ run alone & decide $\{0, \dots, m-1\}$

⇨ 1st subset has 1:1 mapping with C_m and hence, the same size

$$|C_m| \equiv |X_{m+1}| \equiv |C_{m+1}| (\mathrm{mod}\ 2)$$

X_{m+1} = all tuples (α, p_i), IS execution α, in which processes $\neq p_i$ decide $\{0, \ldots, m-1\}$

$$|X_{m+1}| \equiv |C_m| (\mathrm{mod}\ 2)$$

2nd subset: $(\alpha, p_i) \in X_{m+1}$, p_i, $i \neq m$, is unseen in α

In α, processes $p_0, \ldots, p_{i-1}, p_{i+1}, \ldots, p_m$ run alone and cannot decide i (by Validity)

⇨contradiction

⇨ 2nd subset is empty

$$|C_m| \equiv |X_{m+1}| \equiv |C_{m+1}| (\mathrm{mod}\ 2)$$

X_{m+1} = all tuples (α, p_i), IS execution α, in which processes $\neq p_i$ decide $\{0, \ldots, m-1\}$

$$|X_{m+1}| \equiv |C_m| (\mathrm{mod}\ 2)$$

3rd subset: $(\alpha, p_i) \in X_{m+1}$ and p_i is seen in α

AR lemma ⇨ ∃ unique $\alpha' \neq \alpha$ in which p_i is seen, s.t., α' and α indistinguishable to processes $\neq p_i$

⇨ In α', they decide $\{0, \ldots, m-1\}$

⇨ $(\alpha', p_i) \in X_{m+1}$

⇨ The size of the 3rd subset is even

$$|C_m| \equiv |X_{m+1}| \equiv |C_{m+1}| (\text{mod } 2)$$

X_{m+1} = all tuples (α, p_i), IS execution α, in which processes $\neq p_i$ decide $\{0, \dots, m-1\}$

$$|X_{m+1}| \equiv |C_m| (\text{mod } 2)$$

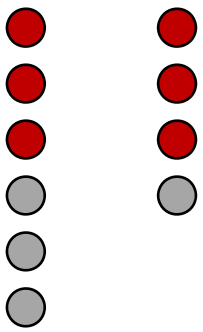
$$|C_{m+1}| \equiv |X_{m+1}| \equiv |C_m| (\text{mod } 2)$$

By induction on $|C_m|$, $|C_{m+1}|$ is odd
⇨ ∃ execution in which $\{0, \dots, m\}$ are decided
Contradiction to n-agreement , when $m = n$

Use More Processes?

Can $n > k$ processes solve k–set agreement with k crash failures?

Simulate an f-tolerant n-process algorithm by a wait-free (f+1)-process algorithm



contradiction

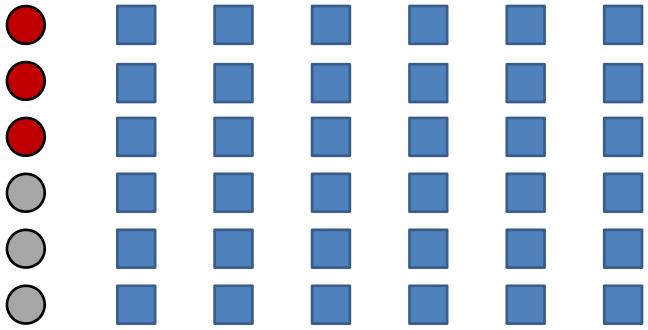
k-tolerant k–set agreement implies wait-free $(n-1)$–set agreement

BG Simulation

All f+1 processes simulate all steps of all n processes

Agree on each step???

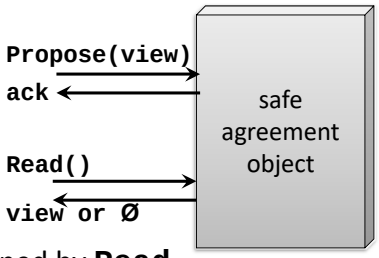
Only need safe agreement!



Safe Agreement: Specification

Separate the voting / negotiation on a decision
from figuring out the outcome

Two wait-free procedures:
`Propose` and **`Read`**



Validity of non-**Ø** views

Agreement on non-**Ø** views returned by **`Read`**

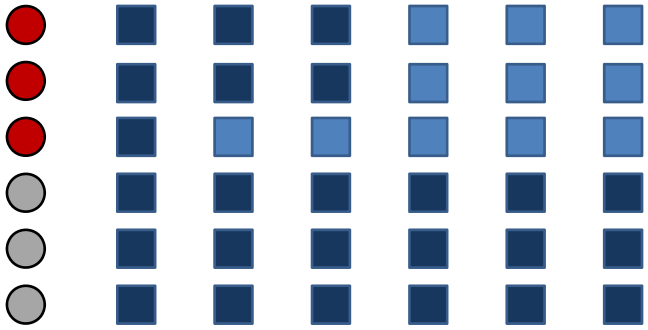
Termination:
If all processes that invoked **`Propose`** return,
then **`Read`** returns non-**Ø** view

BG Simulation: Outline

Try to simulate a step (calculate locally and propose)

If read **Ø**, step is "unresolved", move to next process

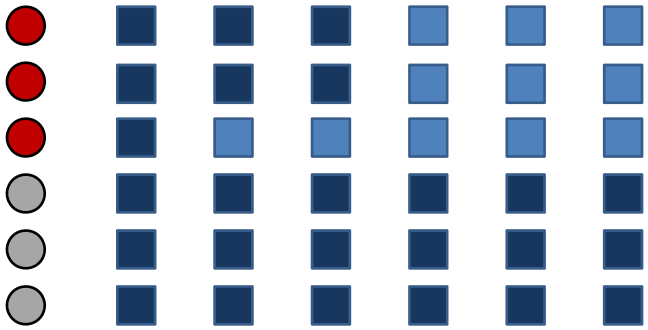
☝ Some faulty process accounts for unresolved step



BG Simulation: Termination

If some process decides, take its output

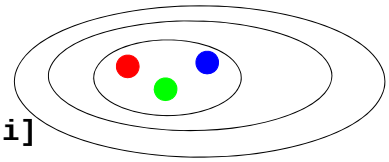
Good for some problems (e.g., consensus and set consensus) but not for others (e.g., renaming)



Safe Agreement: Implementation

Use an atomic snapshot object and an array **R**

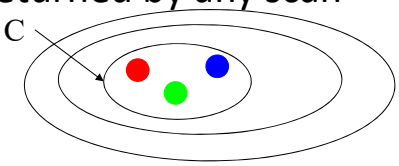
```
Propose( V )
   update( V )
   scan
   write returned view to R[i]
Read() returns view
   find minimal view C written in R
   if all processes in C wrote their view
      & it contains C
      return C // or min(C) if single value needed
   else return Ø
```



U U U S S U U S

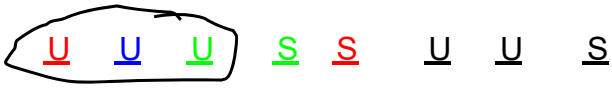
Safe Agreement: Safety

Let C be the minimal view returned by any scan



Can prove that all non-**Ø** views are equal to C

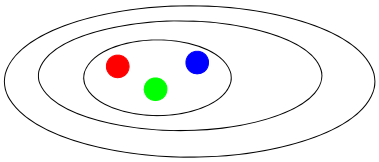
Must be a view of some process in C



Safe Agreement: Liveness

Clearly, both procedures are wait-free

But **Read** sometimes returns a meaningless value, **Ø**



If some process invokes **`Propose`**, then after all processes that invoke **`Propose`** return, a **Read** returns a non-**Ø** value

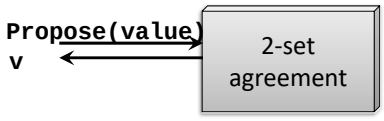
U U U S S U U S

Hierarchy? What Hierarchy?

For any integers $k_1 \geq k_2$, there are two objects X_1, with consensus number k_1, and X_2, with consensus number k_2, such that X_1 cannot wait-free implement X_2

We'll show for $k_1 = k$ and $k_2 = 1$

2-set agreement object: returns one of the first two values proposed

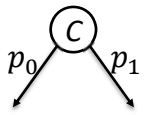


Consensus Number of 2-Set Object is 1

Two processes cannot solve consensus using 2-set agreement objects and read / write registers

Bivalence-style proof

Interesting case when p_0 and p_1 apply operations to the same 2-set agreement object X



- Case 1: No prior operations on X
  ⇨ Return self-arguments…
- Case 2: **`Propose(v)`** applied to X before C
  ⇨ Return **`v`** to both operations…

Consensus Objects Do not Help with Set Agreement

$2k + 1$ processes cannot solve 2-set agreement, with k-consensus objects and read / write registers

Otherwise, BG simulation (extended to consensus objects) allows 3 processes to simulate this algorithm using only read / write registers

Contradicting the impossibility of a wait-free 3-process algorithm for 2-set agreement

2-Set Agreement Object

- Has consensus number 1
- Gives a (trivial) wait-free algorithm for 2-set agreement, for any number of processes

⇨ $2k + 1$ processes cannot wait-free implement 2-set agreement objects with k-consensus objects and read / write registers

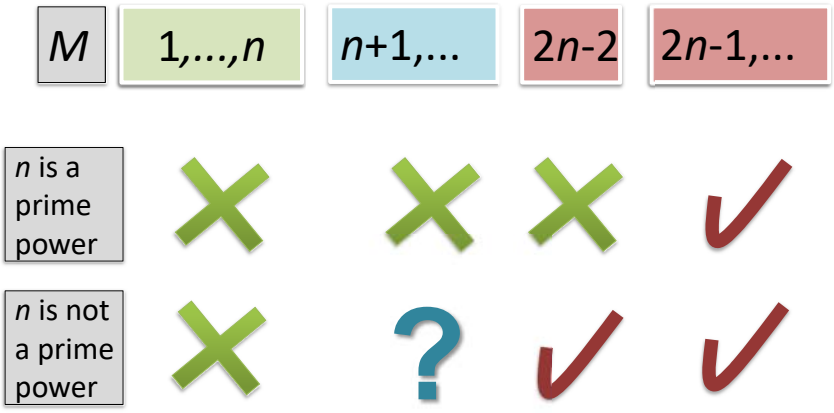
Interesting recent extensions to deterministic objects. Even number of processes?

M-Renaming

Each process starts with an unbounded name x_i and decides on a new name $y_i \in \{0, \ldots, M-1\}$, such that

- Uniqueness: no two processes get the same new name

Bounds for Non-Adaptive Renaming



Weak Symmetry Breaking (WSB)

Each process starts with an unbounded name x_i and decides on $y_i \in \{0,1\}$, such that

- not all processes decide 0
- not all processes decide 1

WSB ⇔ (2*n*-2)-renaming

If new name < *n* output 1, otherwise, output 0

WSB solvable ⇔ n is a prime power

Adaptive *M*-Renaming

Each process starts with an unbounded name x_i and decides on a new name $y_i \in \{0, \ldots, M-1\}$, such that

- Uniqueness: no two processes get the same new name
- Adaptiveness: if k processes participate, names in $\{0, \ldots, f(k)\}$

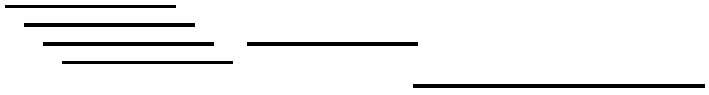
$(2k-2)$-adaptive renaming ⇨ k-set agreement

⇨ no wait-free adaptive $(2k-2)$-renaming

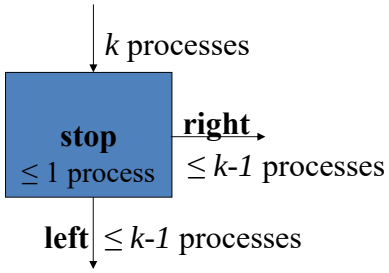
Adaptive Step Complexity

The step complexity of the algorithm depends only on the number of **participating** processes

Total contention: The number of processes that (ever) take a step during the execution.



Detecting Contention: Recall Splitter

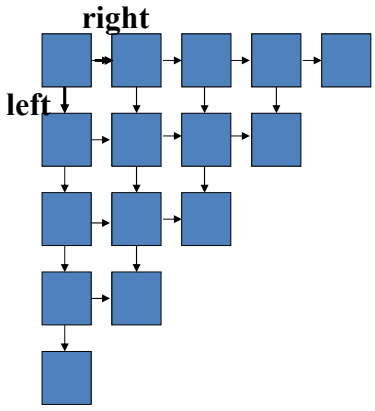


A process stops if it is alone in the splitter

O(1) steps

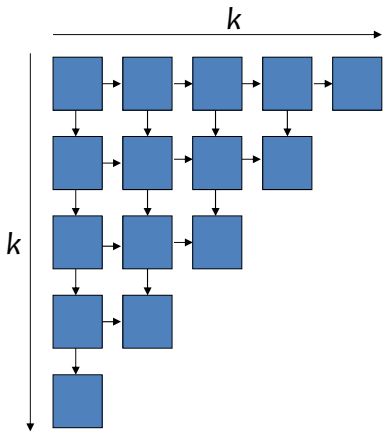
Putting Splitters Together

- A triangular matrix of splitters
- Traverse array, starting at the top left, according to the values returned by splitters
- Until stopping in some splitter



Putting Splitters Together

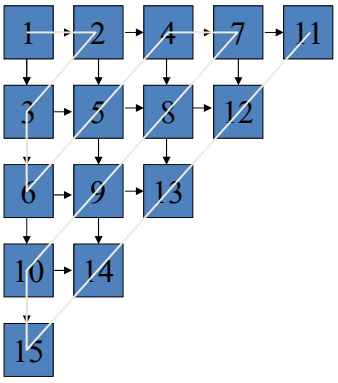
- ≥ one process does not go in each direction
- ⇨After ≤ *k* movements, a process is alone in a splitter
- ⇨A process stops at row, column ≤ *k*
- ⇨At most *O(k)* steps



Putting Splitters Together: k^2 -Renaming

Diagonal association of names with splitters

⇨Take a name ≤ k^2

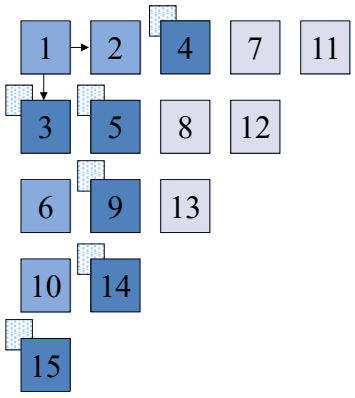


Store & Collect

- Each process has to **`store`** information (periodically)
- Processes **`collect`** the recent information stored by all processes (pairs of $\langle id, val \rangle$)

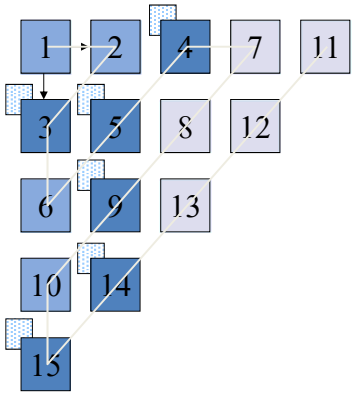
Better Things with a Splitter: Store

- Associate a register with each splitter
- A process writes its id + value in the splitter where it stops
- *Mark* a splitter if accessed by some process



Better Things with a Splitter: Collect

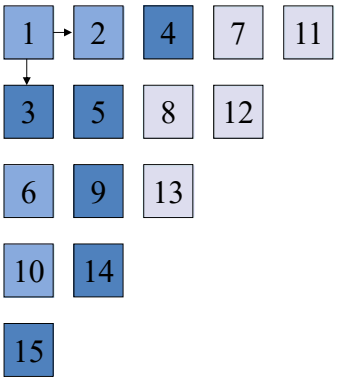
- Associate a register with each splitter
- The current values can be collected from the associated registers
- Going in diagonals, until reaching an unmarked diagonal



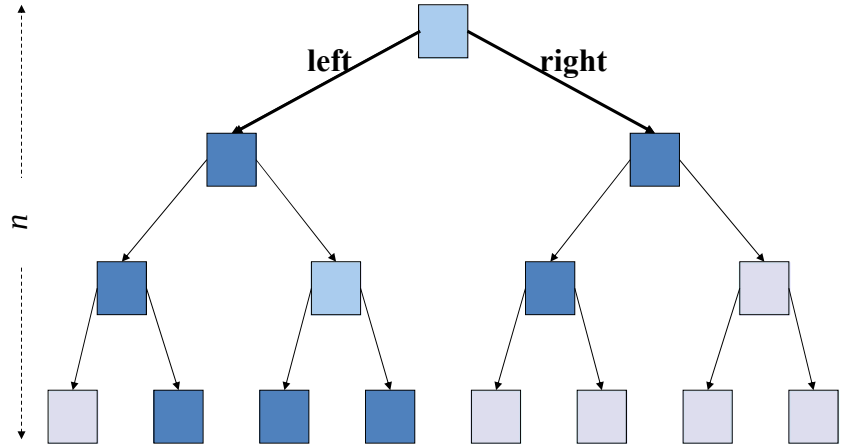
Even Better Things with a Splitter: Store and Collect

- The first store accesses $\leq k$ splitters
- A collect may need to access k^2 splitters…

Can we do better?

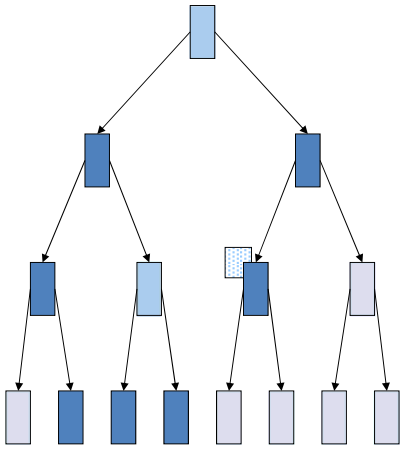


Binary Collect Tree



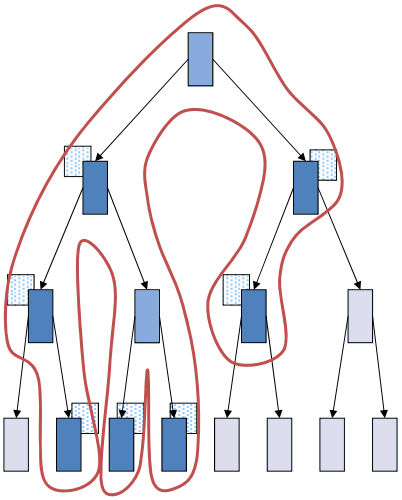
Binary Collect Tree

- To store: traverse the tree until stopping in some splitter
- Later, write in the register associated with this splitter



Binary Collect Tree

- To collect: DFS traverse the **marked tree**, and read the associated registers
- Marked tree contains $\leq 2k\text{-}1$ splitters

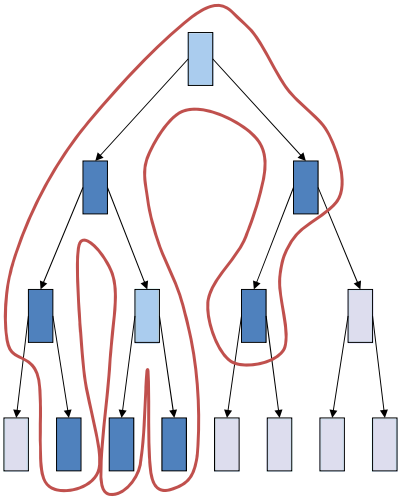


Size of Marked Sub-Tree

In the inorder sequence of the marked sub-tree…

There is an acquired node (where a process stops), between every pair of marked nodes

⇨ ≤ 2k+1 nodes



Simple Things to Do with a Linear Collect

- Every algorithm with *f(k)* iterations of collect and store operations can be made adaptive
- E.g., double-collect atomic snapshots

  O(k) iterations ⇨ $O(k^2)$ steps

Be More Adaptive?

- In a *long-lived* algorithm…

  …processes come and go.

- What if many processes start the execution, then stop participating?

  …then start again…

  …then stop again…

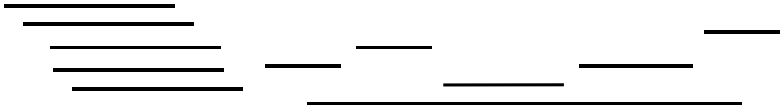
Long-Lived Adaptive M-Renaming

- A process has to **`acquire`** a **unique** new name in {0,…,M-1}; later, it may **`release`** it
- The range of new names should be **small**
  - Preferably **adaptive**: depending only on the number of active processes
  - Must have M ≥ *2k-1*

Renaming is a **building block** for adaptive algorithms

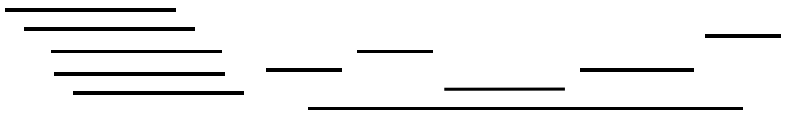
- First obtain names in an adaptive range
- Then apply an ordinary algorithm using these names

Who's Active Now?



Interval contention during an operation:
The number of processes (ever) taking a step during the operation

Who's Active Now?

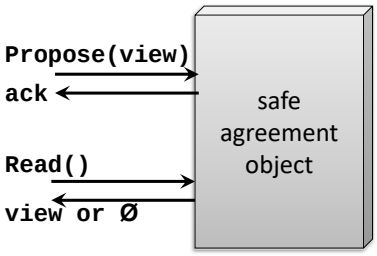


Point contention of an operation:
Max number of processes taking steps together during the operation

Clearly, point contention ≤ interval contention

Recall Safe Agreement

Wait-free **Propose** and **Read**

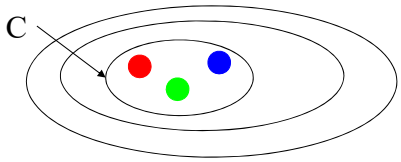


Use an atomic snapshot object and array **R**

```
Propose( view )
    update( view )
    scan
    write returned view to R[i]
Read() returns view
    find minimal view C written in R
    if all processes in C wrote their view
       & it contains C
       return C
    else return Ø
```

Validity and **Agreement** on non-**Ø** views returned by **Read**
Termination: If all processes that invoked **Propose** return, then **Read** returns non-**Ø** view

Safe Agreement: Even Better



A **Read** by some process in C returns a non-**Ø** value

E.g., the last process in C to write its view

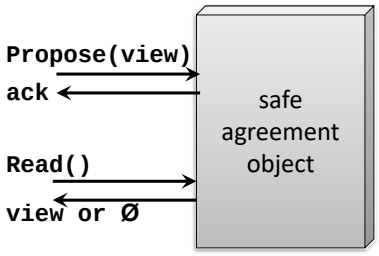
These processes are called **winners**

U U U S S U U S

Safe Agreement: Concurrency

All processes in C execute **`Propose`** concurrently

In particular, all **winners**

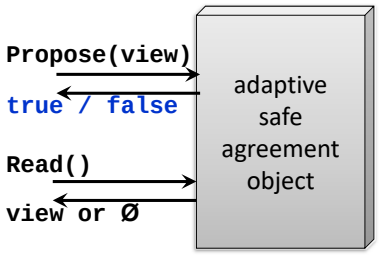


Use a doorway variable **`inside`** to avoid unnecessary update / scan (by non-winners)

Safe Agreement: Concurrency

All processes in C execute **`Propose`** concurrently

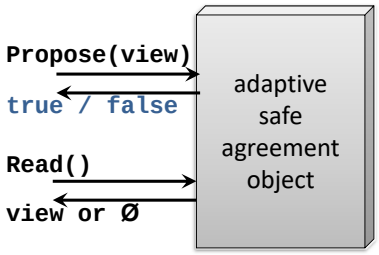
In particular, all **winners**



Use a doorway variable **`inside`** to avoid unnecessary update / scan (by non-winners)

Adaptive Safe Agreement

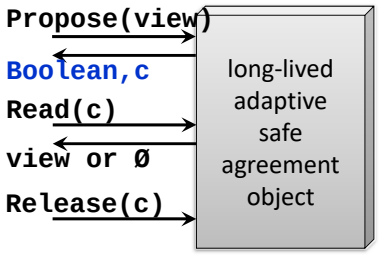
```
Propose( info )
if not inside then
   inside = true
   update( info )
   scan
   write returned view
   return( true )
else return( false )
```



Concurrency: If a process returns **`false`** then some "concurrent" process is accessing the object

Long-Lived Adaptive Safe Agreement

Enhance the interface with a **generation** number (nondecreasing counter)



Validity, agreement and termination as before but relative to a single generation

Concurrency: If a process returns **`false,c`** then some process is concurrently in generation **`c`** of the object

Long-Lived Adaptive Safe Agreement

Synchronization: processes are inside the same generation simultaneously

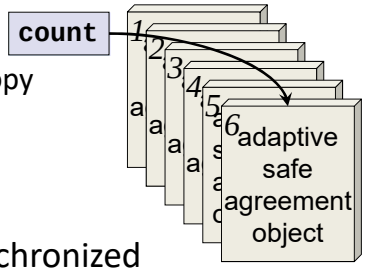
⇨Their number ≤ point contention

⇨Can employ algorithms adaptive to total contention within each generation

- e.g., collect, atomic snapshots

Long-Lived Adaptive Safe Agreement: Implementation

- Many copies of one-shot safe agreement
  - **count** points to the current copy



- Winners of each copy are synchronized
  - Increase **count** by 1.
  - Monotone…

When all processes release a generation, open the next generation by enabling the next copy

Catching Processes with Safe Agreement

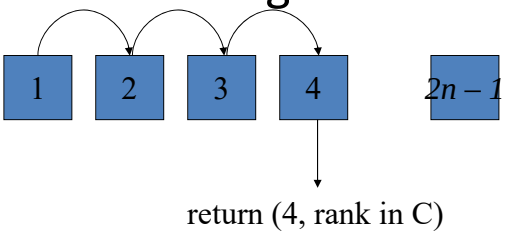
- When processes access an adaptive long-lived safe agreement object simultaneously, at least one wins

- If a process accesses an adaptive long-lived safe agreement object and does not win, some other process is accessing the object

Good for adaptivity…

Things to do with Long-Lived Safe Agreement: Renaming

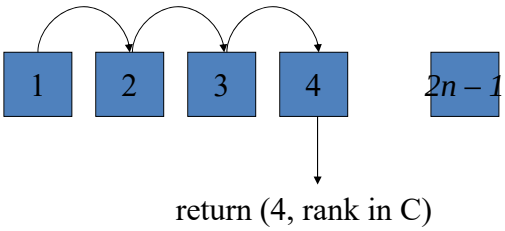
Place objects in a row



Agreement in each long-lived safe agreement object

⇨ **Uniqueness** of names.

Renaming: Size of Name Space



Concurrency for each long-lived safe agreement object

⇨ An object is skipped only due to a concurrent process

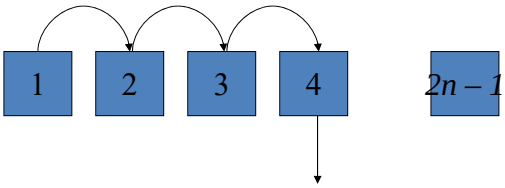
⇨ A process skips $\leq r$ objects

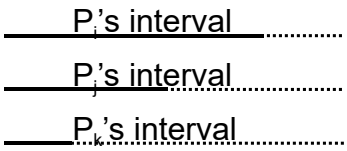
- r is the **interval contention**

- Range of names $\approx r^2$

Renaming: Size of Name Space

We promised point contention





P_i skips because of P_j
⇨ P_j skips because of P_k
⇨ P_k skips because of …
They all overlap

Renaming: Complexity & Size of Name Space

Proof is subtle since a process skips
either due to a concurrent winner
or due to a concurrent non-winner in C
(which it can meet again later in the row)

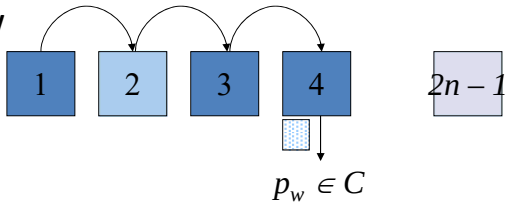
- Use a potential-function proof to show that a process skips ≤ *2k-1* objects
  - *k* is the point contention

⇨Name ≈ k^2

⇨*f(k)* step complexity

Store

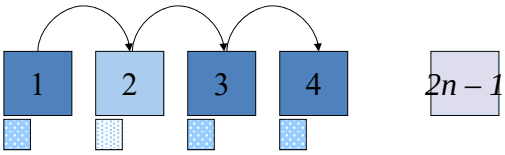
Place objects in a row



Agreement on set of candidates and uniqueness of copies

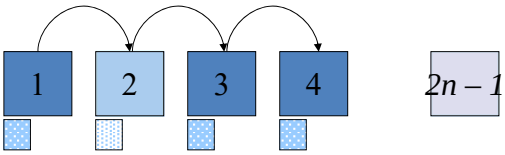
⇨p_w writes the values of all candidates in a register associated with the object

Collect



- Go over the associated registers and read…

Collect: A Problem



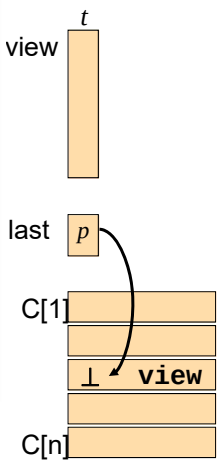
- p_w and all other operations complete.
- A collect still has to reach the object in which p_w has written its value!

Bubble-Up

- Before completing an operation, move information from far away objects to the top.

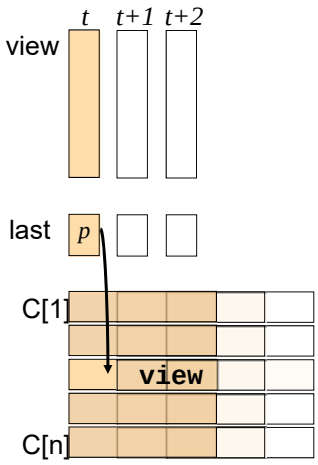
Bubble-Up

```
write ⊥ to C[p][t] // working on it
last[t] = p
C[p][t] = Gather(t)
```



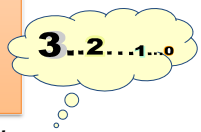
Bubble-Up

```
write ⊥ to C[p][t] // working on it
last[t] = p
C[p][t] = Gather(t)

Gather(t)
q = last[t]
tmp = C[q][t]
if tmp == ⊥ then
  tmp = Gather(t+1) ∩ view[t]
return tmp
```



Bubble-Up

```
write ⊥ to C[p][t] // working on it
last[t] = p
C[p][t] = Gather(t)

Gather(t)
q = last[t]
tmp = C[q][t]
if tmp == ⊥ then
   tmp = Gather(t+1) ∩ view[t]
return tmp
```



After storing in r, bubble-up from r to 1

To collect, **`Gather(1)`**