wait-free and t-resiliency

Borowsky-Gafni Simulation

Remember safe agreement…

relaxing liveness : safe agreement

Safety: Agreement + Validity

Liveness : If every participant takes enough steps then every correct process decides

a process *participates* if it takes at least one step

More precisely:

- propose(v) returns the decided value (or doesn't terminate)
  - every decided value was proposed (validity)
  - no two different values are decided (agreement)
  - Termination
    - every correct process eventually decides (terminates) if some process decides or if every participant takes enough steps

Safe agreement for n processes

Shared variables

$A[0, ..., n-1]$ atomic snapshot object init $\perp$
$B[0, ..., n-1]$ atomic snapshot object init $\perp$

Upon *propose*(v) **by process** p_i

A.update(v)
U:= A.snapshot()
B.update(U)
repeat
V:=B.snapshot()
until $\forall j : U[j] \neq \perp \Rightarrow V[j] \neq \perp$
Let X be the vector in V with the smallest number of non $\perp$ value
decide on the smallest non $\perp$ value in X

proof...

- Liveness : clear
- Validity: all non $\perp$ values in vector of B have been previoulsy proposed
- Agreement : Consider the process p_i that wrote the smallest snapshot S to B (B.update(S))
  - for all j that takes a snapshot V of B, we have:
  - p_j has taken previously a snapshot U of A (code), and $S \subseteq U$ (snapshot containment).
  - $S[i] \neq \perp$ (code), then $U[i] \neq \perp$. So p_j waits until p_i updates B by S.

Another version

with *propose* and *resolve*: *resolve* returns a value (decide) ou ⊥:

- *propose(v)* and *resolve()* are wait-free (*liveness*)
- every decided value was proposed (*validity*)
- no two different values (≠⊥) are decided (*agreement*)
- *Termination*: in repeated invocations of *resolve* eventually every correct process decides if (a) some process decides or (b) no process fails while executing propose operation.

Safe Agreement (V2)

Shared variables

$A[0, ..., n-1]$ atomic snapshot object init $\perp$
$B[0, ..., n-1]$ atomic snapshot object init $\perp$
D decision register init $\perp$

propose(v) by process p_i

A.update(v)
U:= A.snapshot()
B.update(U)

resolve() by process p_i

if ($x := D.read() \neq \perp$) then return x
$V := B.snapshot()$
Let S be the vector with the smallest number of non $\perp$ values
if $\forall j : S[j] \neq \perp \Rightarrow V[j] \neq \perp$ **then**
$x := \min(S)$
$D := x$
return s
else return $\perp$

To get the previous version of PROPOSE in safe-agreement:

upon PROPOSE(v):
    $SA.propose(v)$
    do $w \leftarrow SA.resolve()$ until $w \neq \perp$
    return w

Safe agreement for *n* processes

- Liveness (trivial)
- Validity: at least one non $\perp$ value in each vector written in B
- Agreement : Consider the process p_i that wrote the smallest snapshot S to B $(B.update(S))$ for every p_j that takes a snapshot V of B, we have p_j took previously a snapshot U of A and $S \subseteq U$ (snapshot containment) then if p_j decides it decides the $Min(S)$
- Termination clear

Relation between wait-freeness and t-resiliency

The Borowsky-Gafni (BG) simulation algorithm:

- a set of $t + 1$ asynchronous sequential processes may wait-free simulate an algorithm for n processes t-resilient ($n>t$)
- (the reverse is also true (and trivial))

Application

We know that it is impossible to achieve wait free consensus for two processes

Then we deduce by BG simulation that it is impossible to achieve 1-resilient consensus for n processes

Application

Proof by contradiction:

- Assume there exists a 1-resilient algorithm *A* that achieves consensus for n processes.
- By BG P_0 and P_1 simulate that 1-resilient algorithm *A* for n processes with their initial values (P_0 and P_1)
  - (In the first step of simulated processes, the simulator puts its initial value as initial value of the simulated process)
- When one of the simulated process decides, P_0 and P_1 adopt this decision.

Colorless tasks

Colorless task:

- each process may adopt the input of any other process
- each process may adopt the decision value of any other process

Examples: consensus, k-set …

BG : main ideas

Let $S_0, S_1, \ldots, S_n$ be the simulators

Let $p_0, p_1, \ldots, p_m$ the simulated processes. These processes have to execute a code C_i and use shared memory M .

Intuitively

- Simulation: each simulator maintains its view of the global sate of the simulation
- Each simulator $S_0, S_1, \ldots, S_n$ simulates steps for each p_i: they « agree » on the step
  - (WF: any simulator may stop then all simulator try to make each simulated step)

More precisely…

$MEM[i][j]$: pour S_i état mémoire de p_i:

For each i local variable $State[j]$ is the state of the simulated state for p_j.

Simulation of a step of p_j by S_i:

- To simulate a write v of p_j : S_i writes $MEM[i][j]$ with v and update $State[j]$
- To simulate a read j': S_i finds the « last » value v for j' (among $MEM[k][j']$) in and proposes it for the step of p_j and updates $State[j]$
  (here the new state of p_j depends on the value read, then all simulations of the read have to get the same read value)

Shared Memory

MEM, an n by m array of registers.

For each i,j, $MEM[i][j]$ contains a pair $(val, steps)$:

$MEM[i][j].val$, initially the initial value of j's register in $\mathcal{A}$

$MEM[i][j].steps \in N$, initially 0

For each i $MEM[i][-]$ can be written to by i

$CONSENSUS$, an m by infinite array of consensus objects

Process Local Variables

$input \in I_i$, initially the input of real process i

$state$, $steps$ arrays of size m, initially arbitrary

$decided$, a boolean, initially arbitrary

Thread Local Variables

$k', i', j, j', steps \in N$, initially arbitrary

val, v, w, o, variables, initially arbitrary

$mymem$, an array of n variables, initially arbitrary

Code of the simulator S_i

upon PROPOSE($input$)
    $decided \leftarrow false$
    In parallel for all $j = 1..m$
        $w \leftarrow CONSENSUS_n[j][0](input)$
        $state[j] \leftarrow init\_state_j(w)$
        $steps[j] \leftarrow 1$
        repeat forever
            if $nextop_j(state[j]) = (WRITE, v)$ then
                $MEM[i][j] \leftarrow (v, steps[j])$
                $state[j] \leftarrow trans_j(state[j], DONE)$
            else if $nextop_j(state[j]) = (READ, j')$ then
                for $i' = 1..n$
                    $mymem[i'] \leftarrow MEM[i'][j']$
                $(val, steps) \leftarrow VMAX(mymem)$
                $(w, k') \leftarrow CONSENSUS_n[j][steps[j]](val, steps)$
                $state[j] \leftarrow trans_j(state[j], w)$
            else if $nextop_j(state[j]) = (OUTPUT, o)$ then
                if $\neg decided$ **output** o; $decided \leftarrow true$
            $steps[j] \leftarrow steps[j] + 1$

With safe-agreement

upon PROPOSE($input$)
    $decided \leftarrow false$
    In parallel for all $j = 1..m$
        $SA_n[j][0].propose(input)$ // *in mutual exclusion*
        do $w \leftarrow SA_n[j][0].resolve()$ until $w \neq \perp$
        $state[j] \leftarrow init\_state_j(w)$
        $steps[j] \leftarrow 1$
        repeat forever
            if $nextop_j(state[j]) = (WRITE, v)$ then
                $MEM[i][j] \leftarrow (v, steps[j])$
                $state[j] \leftarrow trans_j(state[j], DONE)$
            else if $nextop_j(state[j]) = (READ, j')$ then
                for $i' = 1..n$
                    $mymem[i'] \leftarrow MEM[i'][j']$
                $(val, steps) \leftarrow VMAX(mymem)$
                $SA_n[j][steps[j]].propose(val, steps)$ // *in mutual exclusion*
                do $(w, k') \leftarrow SA_n[j][steps[j]].resolve()$ until $(w, k') \neq \perp$
                $state[j] \leftarrow trans_j(state[j], w)$
            else if $nextop_j(state[j]) = (OUTPUT, o)$ then
                if $\neg decided$ **output** o; $decided \leftarrow true$
            $steps[j] \leftarrow steps[j] + 1$

Assume a « true » CONSENSUS: we get a wait free simulation in which all p_i terminates and the (real) colorless task is solved.

With safe agreement when the simulator the operation of safe agreement are wait-free and give no decision only if some participant stops in propose(v) then a crashed simulator may block only the resolve safe agreement it currently runs (and by mutual exclusion only it may block only one safe agreement)

- Hence one real crash may stop at most one simulated crash:
  - $t+1$ processes wait-free simulate any t resilient (colorless) n-task

k-set agreement

n processes can solve k-set agreement (k-1)-resilient (easy):

- processes 1 to k write their input value in shared memory and decide their input value
- The other processes read the shared memory until they read some value and decide this value

K-set agreement

(Already know result): it is impossible to solve wait free set agreement (k+1 processes decide at most k values)

Result: n processus can not solve k-set agreement k-resilient

proof:

- by contradiction assume with n processes we can solve k-set agreement k-resilient
- with k+1 processes we can simulate WF n processes k-resilient that executes the k-set agreement algorithm and solve (wait-free) k-set agreement. Contradicting the known result.

Renaming

renaming

- $p_0,p_1,\ldots,p_n$: i is the index of pi
- each process has an initial name :old_name from some set [1..N] (with N big)
  - give a new_name in some set [1..M] with M <N

  new_name() such that:
  - termination
  - validity: each id returned by new_name() is in set [1..M]
  - agreement: no two processes obtain the same new name
  - index independence: the new name is independent of the index
  - one shot : each process invoke new_name at most once
  - (long-lived: a process can repeatedly invoke a new name and the release the name)

  (note that renaming is not colorless)

renaming with splitters..

(here renaming only with MWMR registers)

«splitters» [Moir, Anderson'95]:

validity: output Right, Down or Stop

solo: if a single process invokes the splitter only Stop is possible

concurrency:

when k processes invoke the splitter:

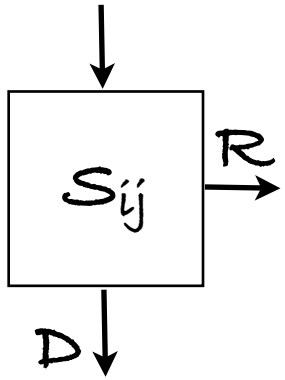
at *most k-1* processes obtain Right

at *most k-1* processes obtain Down

at *most one* process obtains Stop

termination

(when a process obtains Stop -> gets a new name associated to the splitter)

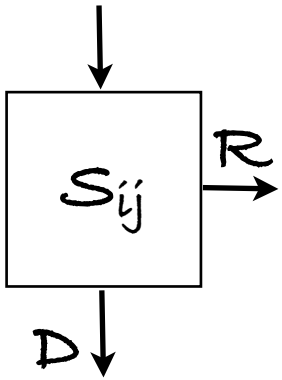


splitter

Code for the splitter with identity my_name:

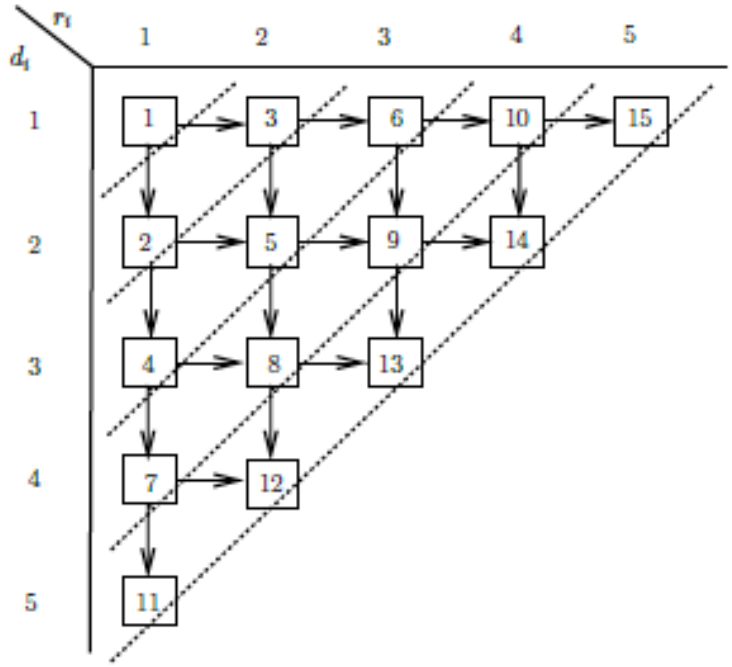
```
int closed=false

Last:=my_name
if(Closed) then return Right
else Closed:=True;
    if(Last=my_name)
    then return(Stop)
    else return(Down)
```



Renaming

- Grid of n(n+1)/2 splitters
  of the splitters on {1,..., n
- when a process obtains
  name the name of the sp
- No process stops on the
- Every process stops on some splitter



Complexity

- $\Theta(n^2)$ splitters
- then size is the set of new names: $\Theta(n^2)$
  - (n(n+1)/2)-renaming
- two registers for each splitter (only MWMR registers): $\Theta(n^2)$ registers
- (improvement [Aspnes'10] $\Theta(n^{3/2})$ splitters)

Wait free (2n-1) renaming

```
s:=1
forever do
    a.update(i,s)
    view:=a.snapshot()
    if view[j]=s for some j ≠i then
                r= |{j with view[j] ≠ undef and j <=i}|
                s= r-th positive integer not in
                        {view[j]| view[j] ≠undef }
    else return s
```

Proof

- r is the *rank*
- Because the rank is at most n and there are at most n-1 names used by the other processes, this always gives proposed names in the range [1..2n-1].
- **For uniqueness,** consider two process with original names i and j. Suppose that i and j both decide on s.
  - Then i sees a view in which a[i] = s and a[j] ≠ s, after it no longer updates a[i].
  - Similarly, j sees a view in which a[j] = s and a[i] ≠ s, after which it no longer updates a[j].
  - If i's view is obtained first, then j can't see a[i] ≠ s, but the same holds if j's view is obtained first. So in either case we get a contradiction, proving uniqueness.
-

T**ermination.** Assume some set S of processes run forever without picking a name, then p in S with minimal identity eventually picks a name.

- More precisely, say a process is *trying* if it runs for infinitely many steps without choosing a name.
- Then in any execution with at least one trying process, eventually we reach a configuration where all processes have either finished or are trying.
- In some subsequent configuration, all the processes have written to the *a* array at least once; after this point, any process reading the *a* array will see the same set of original names and compute the same rank r for itself.
  - Consider the trying process i with the smallest original name, and suppose it has rank r. (all the other trying processes have a rank greater than r)
  - Let F = { z1 < z2 ... } be the set of "free names" that are not proposed in *a* by any of the finished processes.
  - No trying process j ≠ i ever proposes a name in { z1 ... zr }, This leaves zr open for i to claim, provided the other names in { z1 ... zr } eventually become free.
    - j will the only process to propose zr

Better renaming?

Can we do better than 2n-1 (wait-free) renaming?

(Difficult) result:

no better than 2n-2 renaming if

$$\{\binom{n+1}{i+1} | 0 <= i <= \lfloor \frac{n-1}{2} \rfloor\}$$

no better than 2n-1 renaming in other cases