BG-simulation and Renaming

Julien Stainer

Concurrent Algorithms
Distributed Programming Laboratory
julien.stainer@epfl.ch

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Laboratoire de Programmation Distribuée
Distributed Programming Laboratory

Outline

1. BG-Simulation
    - Problem Statement
    - Simulation Algorithm
    - Safe-Agreement
    - Putting Pieces Together
    - Computability Consequences
2. Renaming
    - Problem Statement
    - Renaming and Crashes
    - A Splitter-based Algorithm
    - A Snapshot-based Algorithm

1 BG-Simulation
Problem Statement
Simulation Algorithm
Safe-Agreement
Putting Pieces Together
Computability Consequences

2 Renaming
Problem Statement
Renaming and Crashes
A Splitter-based Algorithm
A Snapshot-based Algorithm

BG-Simulation

The BG-Simulation algorithm allows to wait-free simulate a t-resilient system of n asynchronous processes sharing memory with $t+1$ asynchronous simulators sharing memory.

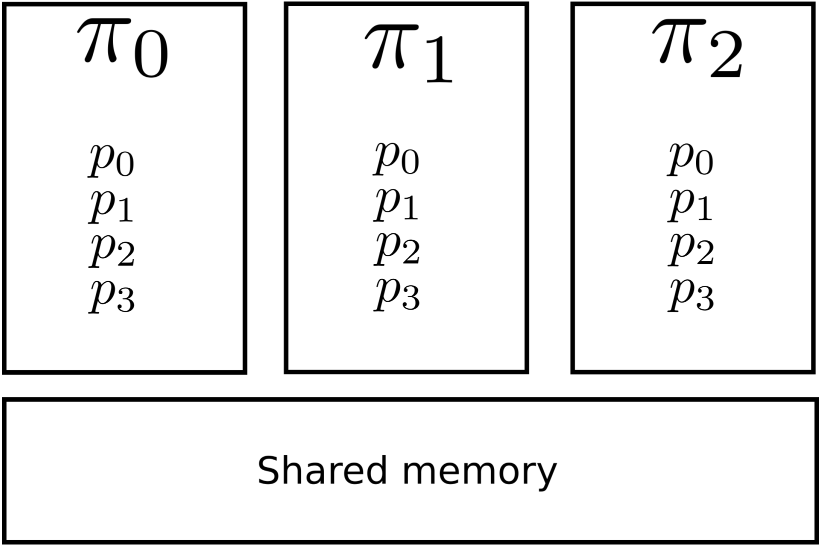
- $t + 1$ asynchronous simulators $\pi_0, \ldots, \pi_t$;

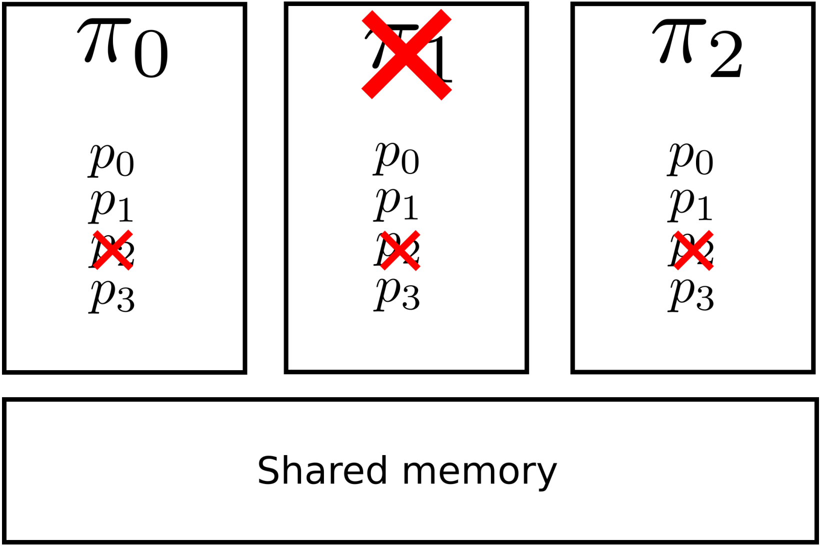
- $t+1$ asynchronous simulators $\pi_0, \ldots, \pi_t$;
- in any execution, up to t simulators may crash;

- $t + 1$ asynchronous simulators $\pi_0, \ldots, \pi_t$;
- in any execution, up to t simulators may crash;
- n simulated processes $p_0, \ldots, p_{n-1}$.

- $t + 1$ asynchronous simulators $\pi_0, \ldots, \pi_t$;
- in any execution, up to t simulators may crash;
- n simulated processes $p_0, \ldots, p_{n-1}$.

We must ensure that, in any simulated execution, at most t simulated processes crash.


π_0
p_0
p_1
p_2
p_3
π_1
p_0
p_1
p_2
p_3
π_2
p_0
p_1
p_2
p_3
Shared memory



π_0
p_0
p_1
p_2
p_3
π_1
p_0
p_1
p_2
p_3
π_2
p_0
p_1
p_2
p_3
Shared memory

- Simulators are provided with a program $prog_i$ and an input $input_i$ for each simulated process p_i;

- Simulators are provided with a program $prog_i$ and an input $input_i$ for each simulated process p_i;
- each simulator π_s simulates all the processes $p_0, \ldots, p_{n-1}$;

- Simulators are provided with a program $prog_i$ and an input $input_i$ for each simulated process p_i;
- each simulator π_s simulates all the processes $p_0, \ldots, p_{n-1}$;
- it computes an output $output_i$ for each process that do not crash during the simulation.

We consider deterministic programs $prog_i$ supposed to be of the following form:

$state_i \leftarrow init_i$
while not_decided($state_i$) **do**
  $val \leftarrow$ next_write($state_i$)
  write(val, $\mathrm{MEM}[i]$)
  $snap_i \leftarrow$ snapshot(MEM)
  $state_i \leftarrow$ update_state($snap_i$, $state_i$)
end while
return compute_output($state_i$)

For each process p_i, simulator π_s maintains:

- p_i's state: *state*[*i*]

  (values of variables, instruction pointer, etc.);

For each process p_i, simulator π_s maintains:

- p_i's state: *state*[*i*]
  (values of variables, instruction pointer, etc.);
- a sequence number for p_i's last write:
  write_sn[*i*];

For each process p_i, simulator π_s maintains:

- p_i's state: $state[i]$
  (values of variables, instruction pointer, etc.);
- a sequence number for p_i's last write: $write\_sn[i]$;
- a sequence number for p_i's last snapshot: $snap\_sn[i]$.

- Each simulator runs in parallel the n simulations.

- Each simulator runs in parallel the n simulations.
- The n simulation threads are scheduled such that they all always eventually take steps, even if some of them never terminate.

- Each simulator runs in parallel the n simulations.
- The n simulation threads are scheduled such that they all always eventually take steps, even if some of them never terminate.
- The crash of the simulator prevents future progress of any thread.

- Each simulator runs in parallel the n simulations.
- The n simulation threads are scheduled such that they all always eventually take steps, even if some of them never terminate.
- The crash of the simulator prevents future progress of any thread.
- Threads do not crash individually.

- Each simulator π_s keeps a local copy $smem_s$ of the simulated shared memory;

- Each simulator π_s keeps a local copy $smem_s$ of the simulated shared memory;
- $smem_s$ is an array of n elements, one for each simulated process;

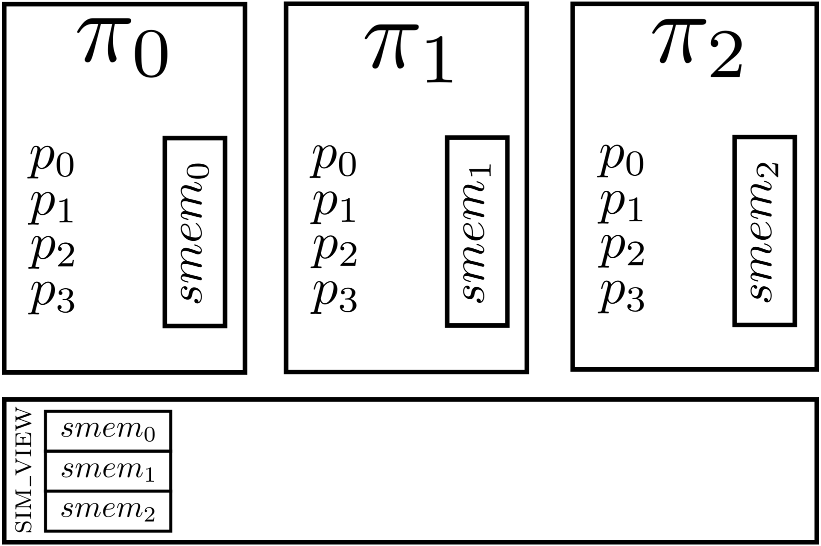
- Each simulator π_s keeps a local copy $smem_s$ of the simulated shared memory;
- $smem_s$ is an array of n elements, one for each simulated process;
- $smem_s[i]$ is the last value written by p_i in its simulation by π_s;

- SIM_VIEW is an array of shared atomic SWMR registers, one for each simulator;

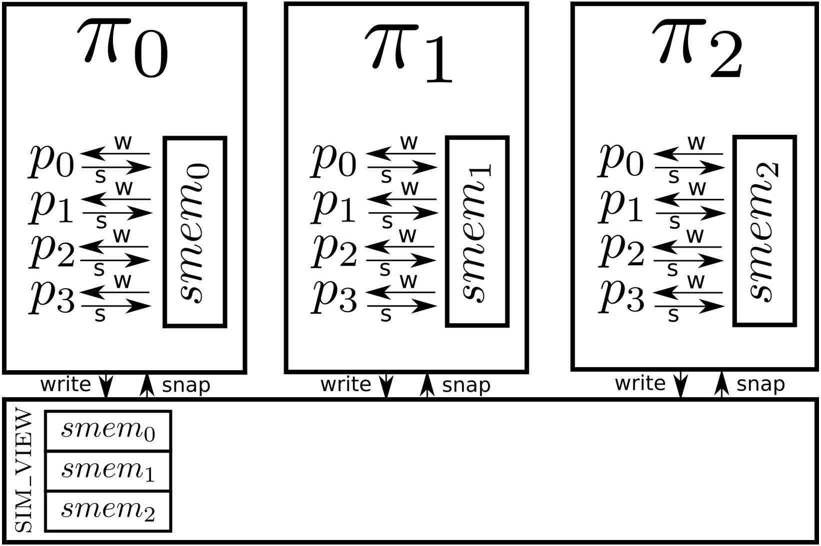
- SIM_VIEW is an array of shared atomic SWMR registers, one for each simulator;
- π_s writes it's view $smem_s$ of the simulated shared memory in its assigned shared register SIM_VIEW$[s]$;

- SIM_VIEW is an array of shared atomic SWMR registers, one for each simulator;
- π_s writes it's view $smem_s$ of the simulated shared memory in its assigned shared register $\text{SIM\_VIEW}[s]$;
- it can take an atomic snapshot of the views of other simulators $\text{SIM\_VIEW}[0, \ldots, t]$ on the simulated shared memory.

- SIM_VIEW is an array of shared atomic SWMR registers, one for each simulator;
- π_s writes it's view $smem_s$ of the simulated shared memory in its assigned shared register $\text{SIM\_VIEW}[s]$;
- it can take an atomic snapshot of the views of other simulators $\text{SIM\_VIEW}[0, \ldots, t]$ on the simulated shared memory.
- $\text{SIM\_VIEW}[s'][i]$ is a pair containing the last value written by p_i according to simulator $\pi_{s'}$, and the corresponding sequence number.



π_0
p_0
p_1
p_2
p_3
$smem_0$
π_1
p_0
p_1
p_2
p_3
$smem_1$
π_2
p_0
p_1
p_2
p_3
$smem_2$
SIM_VIEW
$smem_0$
$smem_1$
$smem_2$


π_0
π_1
π_2
p_0
p_1
p_2
p_3
w
s
$smem_0$
$smem_1$
$smem_2$
write
snap
SIM_VIEW

We need the simulation to be coherent across simulators.

We need the simulation to be coherent across simulators.

We need to ensure the atomicity of the simulated shared memory.

A safe-agreement object offers two operations: PROPOSE(v) and DECIDE().

Termination Any invocation of PROPOSE by a correct process terminates. If no process crashes while executing PROPOSE, then any correct process invoking DECIDE() terminates.

A safe-agreement object offers two operations: PROPOSE(v) and DECIDE().

Termination: Any invocation of PROPOSE by a correct process terminates. If no process crashes while executing PROPOSE, then any correct process invoking DECIDE() terminates.

Agreement: At most one value is decided.

A safe-agreement object offers two operations: PROPOSE(v) and DECIDE().

Termination Any invocation of PROPOSE by a correct process terminates. If no process crashes while executing PROPOSE, then any correct process invoking DECIDE() terminates.

Agreement At most one value is decided.

Validity A decided value is a proposed value.

A safe-agreement object offers two operations: PROPOSE(v) and DECIDE().

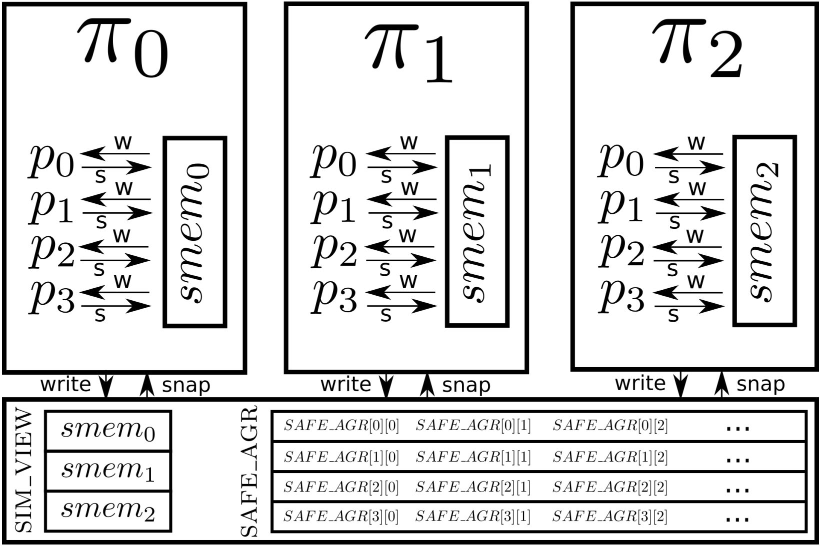
Termination
: Any invocation of PROPOSE by a correct process terminates. If no process crashes while executing PROPOSE, then any correct process invoking DECIDE() terminates.

Agreement
: At most one value is decided.

Validity
: A decided value is a proposed value.

In a crash-free system, safe-agreement objects implement consensus.

init $REG[0, \ldots, n-1] \leftarrow [\langle \bot, 0 \rangle]$
operation PROPOSE(v)
    $REG[s] \leftarrow \langle v, 1 \rangle$
    $snap_s \leftarrow REG.snapshot()$
    if $\exists x \ : \ snap_s[x].level = 2$ **then**
        $REG[s] \leftarrow \langle v, 0 \rangle$
    else
        $REG[s] \leftarrow \langle v, 2 \rangle$
    end if
end operation
operation DECIDE( )
    repeat
        $snap_s \leftarrow REG.snapshot()$
    until $\forall x \ : \ snap_s[x].level \neq 1$
    $x \leftarrow \min \{ y \mid snap_s[y] = 2 \}$
    return $snap_s[x].value$
end operation



π_0
π_1
π_2
p_0
p_1
p_2
p_3
w
s
$smem_0$
$smem_1$
$smem_2$
write
snap
SIM_VIEW
SAFE_AGR
$SAFE\_AGR[0][0]$ $SAFE\_AGR[0][1]$ $SAFE\_AGR[0][2]$ ...
$SAFE\_AGR[1][0]$ $SAFE\_AGR[1][1]$ $SAFE\_AGR[1][2]$ ...
$SAFE\_AGR[2][0]$ $SAFE\_AGR[2][1]$ $SAFE\_AGR[2][2]$ ...
$SAFE\_AGR[3][0]$ $SAFE\_AGR[3][1]$ $SAFE\_AGR[3][2]$ ...

$state_i \leftarrow init_i$
while not_decided($state_i$) **do**
    $val \leftarrow$ next_write($state_i$)
    write(val, MEM[i])
    $snap_i \leftarrow$ snapshot(MEM)
    $state_i \leftarrow$ update_state($snap_i$, $state_i$)
end while
return compute_output($state_i$)

$state_i \leftarrow init_i$; $write\_sn[i] \leftarrow 0$; $snap\_sn[i] \leftarrow 0$
while not_decided($state_i$) **do**
    $val \leftarrow$ next_write($state_i$)
    simulate_write($i, val, \mathrm{MEM}[i]$)
    $snap_i \leftarrow$ simulate_snapshot(i, MEM)
    $state_i \leftarrow$ update_state($snap_i, state_i$)
end while
return compute_output($state_i$)

operation $\text{SIMULATE\_WRITE}(i, val, \text{MEM}[i])$

$write\_sn[i] \leftarrow write\_sn[i] + 1$

$smem_s[i] \leftarrow \langle val, write\_sn[i] \rangle$

$\text{SIM\_VIEW}[s] \leftarrow smem_s$

end operation

operation SIMULATE_SNAPSHOT(i,MEM)
    $snap \leftarrow$ SIM_VIEW.$snapshot()$
    for $x \in \{1, \ldots, n\}$ **do**
        let z **be s.t.** $\forall y, snap[z][x].sn \geq snap[y][x].sn$
        $sim\_snap[x] \leftarrow snap[z][x]$
    end for
    $snap\_sn[i] \leftarrow snap\_sn[i] + 1$
    SAFE_AGR[i][$snap\_sn[i]$].PROPOSE(sim_snap)
    return SAFE_AGR[i][$snap\_sn[i]$].DECIDE()
end operation

- Simulators now all agree on the snapshots of simulated processes;

- Simulators now all agree on the snapshots of simulated processes;
- $prog_i$ being deterministic, they also agree on the remaining of their simulations (including writes).

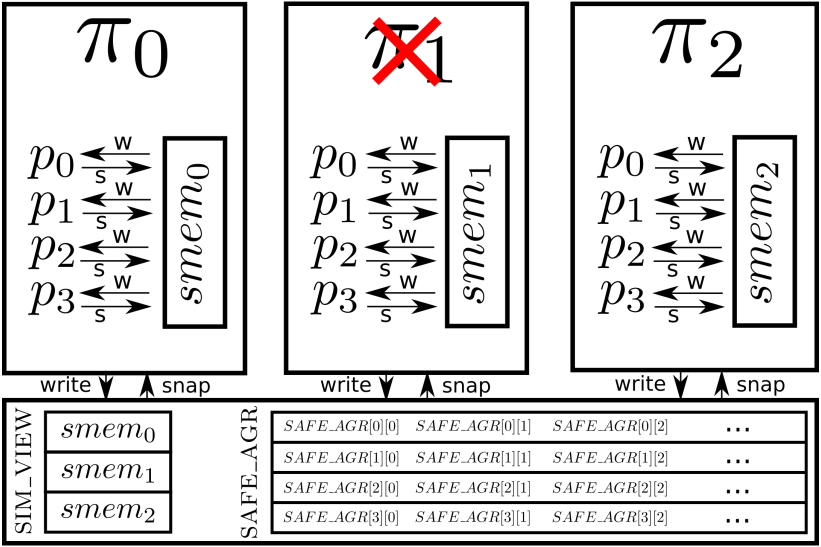
- Simulators now all agree on the snapshots of simulated processes;
- $prog_i$ being deterministic, they also agree on the remaining of their simulations (including writes).
- A simulated write is linearized at the first moment a simulator π_s writes a $smem_s$ containing this write into $\text{SIM\_VIEW}[s]$.

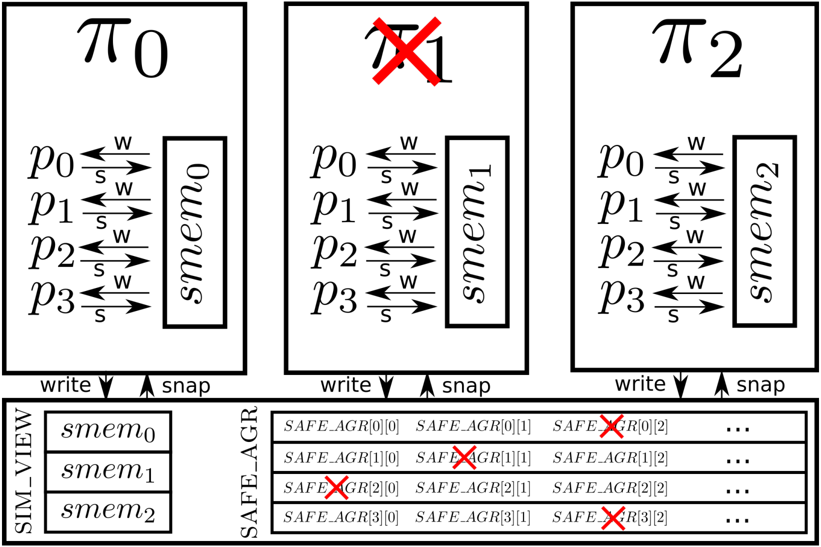
- Simulators now all agree on the snapshots of simulated processes;
- $prog_i$ being deterministic, they also agree on the remaining of their simulations (including writes).
- A simulated write is linearized at the first moment a simulator π_s writes a $smem_s$ containing this write into SIM_VIEW[s].
- A simulated snapshot is linearized at the moment the simulator that proposes it to the safe-agreement took its corresponding snapshot of SIM_VIEW.

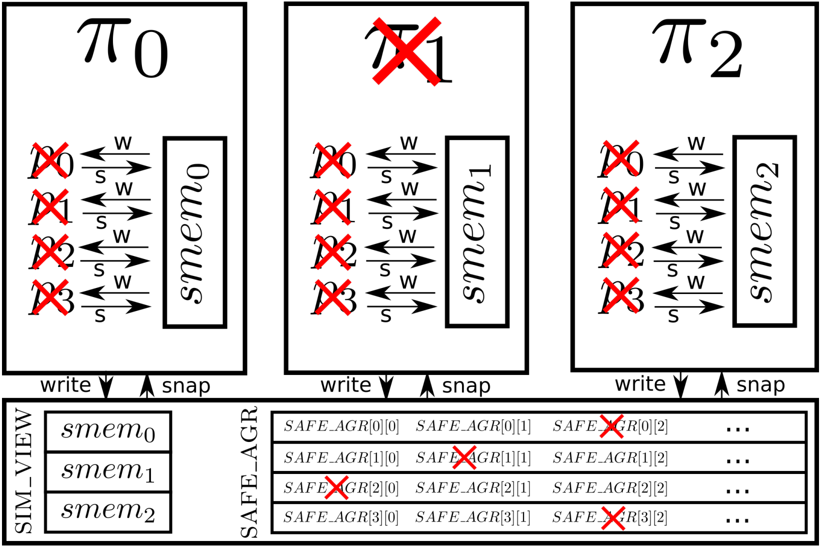
- A simulator can be executing PROPOSE operations on several safe-agreement objects at the same time (at most one per simulation thread).

- A simulator can be executing PROPOSE operations on several safe-agreement objects at the same time (at most one per simulation thread).
- If it crashes at that point, DECIDE operations of these objects may block forever for all simulators;

- A simulator can be executing PROPOSE operations on several safe-agreement objects at the same time (at most one per simulation thread).
- If it crashes at that point, DECIDE operations of these objects may block forever for all simulators;
- the corresponding simulated processes then stop making progress.


π0
π1
π2
p0 p1 p2 p3
w
s
smem0
smem1
smem2
write
snap
SIM_VIEW
smem0
smem1
smem2
SAFE_AGR
SAFE_AGR[0][0] SAFE_AGR[0][1] SAFE_AGR[0][2] ...
SAFE_AGR[1][0] SAFE_AGR[1][1] SAFE_AGR[1][2] ...
SAFE_AGR[2][0] SAFE_AGR[2][1] SAFE_AGR[2][2] ...
SAFE_AGR[3][0] SAFE_AGR[3][1] SAFE_AGR[3][2] ...


π_0
π_1
π_2
p_0 p_1 p_2 p_3
w
s
$smem_0$
$smem_1$
$smem_2$
write
snap
SIM_VIEW
SAFE_AGR
SAFE_AGR[0][0] SAFE_AGR[0][1] SAFE_AGR[0][2] ...
SAFE_AGR[1][0] SAFE_AGR[1][1] SAFE_AGR[1][2] ...
SAFE_AGR[2][0] SAFE_AGR[2][1] SAFE_AGR[2][2] ...
SAFE_AGR[3][0] SAFE_AGR[3][1] SAFE_AGR[3][2] ...



π_0
π_1
π_2
p_0
p_1
p_2
p_3
w
s
$smem_0$
$smem_1$
$smem_2$
write
snap
SIM_VIEW
SAFE_AGR
$SAFE\_AGR[0][0]$ $SAFE\_AGR[0][1]$ $SAFE\_AGR[0][2]$ ...
$SAFE\_AGR[1][0]$ $SAFE\_AGR[1][1]$ $SAFE\_AGR[1][2]$ ...
$SAFE\_AGR[2][0]$ $SAFE\_AGR[2][1]$ $SAFE\_AGR[2][2]$ ...
$SAFE\_AGR[3][0]$ $SAFE\_AGR[3][1]$ $SAFE\_AGR[3][2]$ ...

All we need is local synchronization!

All we need is local synchronization!

We can easily force a thread never to be in more than one PROPOSE operation.

operation SIMULATE_SNAPSHOT(i,MEM)
    $snap \leftarrow$ SIM_VIEW.$snapshot()$
    for $x \in \{1, \ldots, n\}$ **do**
        let z **be s.t.** $\forall y, snap[z][x].sn \geq snap[y][x].sn$
        $sim\_snap[x] \leftarrow snap[z][x]$
    end for
    $snap\_sn[i] \leftarrow snap\_sn[i] + 1$
    enter mutex
    SAFE_AGR[i][$snap\_sn[i]$].PROPOSE(sim_snap)
    leave mutex
    return SAFE_AGR[i][$snap\_sn[i]$].DECIDE()
end operation

A simulator is never in more than one PROPOSE.

A simulator is never in more than one PROPOSE.

The crash of a simulator can prevent the progress of at most one simulated process.

- We now have a protocol for $t+1$ asynchronous simulators sharing memory.

- We now have a protocol for $t+1$ asynchronous simulators sharing memory.
- t of them may crash.

- We now have a protocol for $t + 1$ asynchronous simulators sharing memory.
- t of them may crash.
- They can simulate the execution of a protocol between $n > t$ asynchronous processes sharing a memory with at most t crashes.

- We now have a protocol for $t + 1$ asynchronous simulators sharing memory.
- t of them may crash.
- They can simulate the execution of a protocol between $n > t$ asynchronous processes sharing a memory with at most t crashes.
- They need to be provided with:

- We now have a protocol for $t+1$ asynchronous simulators sharing memory.
- t of them may crash.
- They can simulate the execution of a protocol between $n > t$ asynchronous processes sharing a memory with at most t crashes.
- They need to be provided with:
  - a deterministic program for each simulated process;

- We now have a protocol for $t + 1$ asynchronous simulators sharing memory.
- t of them may crash.
- They can simulate the execution of a protocol between $n > t$ asynchronous processes sharing a memory with at most t crashes.
- They need to be provided with:
    - a deterministic program for each simulated process;
    - an input for each simulated process.

BG-Simulation

- We now have a protocol for $t+1$ asynchronous simulators sharing memory.
- t of them may crash.
- They can simulate the execution of a protocol between $n > t$ asynchronous processes sharing a memory with at most t crashes.
- They need to be provided with:
  - a deterministic program for each simulated process;
  - an input for each simulated process.
- The correct simulators can then compute an output for each simulated process that do not block in the simulation.

Tasks are distributed functions defined by a triple $(\mathcal{I}, \mathcal{O}, \delta)$.

- $\mathcal{I}$ denotes the set of input configurations,

Tasks

Tasks are distributed functions defined by a triple $(\mathcal{I}, \mathcal{O}, \delta)$.

- $\mathcal{I}$ denotes the set of input configurations,
- Any $I \in \mathcal{I}$ is a vector of inputs, one for each process.

Tasks are distributed functions defined by a triple $(\mathcal{I}, \mathcal{O}, \delta)$.

- $\mathcal{I}$ denotes the set of input configurations,
- Any $I \in \mathcal{I}$ is a vector of inputs, one for each process.
- $\mathcal{O}$ is the set of possible output configurations.

Tasks

Tasks are distributed functions defined by a triple $(\mathcal{I}, \mathcal{O}, \delta)$.

- $\mathcal{I}$ denotes the set of input configurations,
- Any $I \in \mathcal{I}$ is a vector of inputs, one for each process.
- $\mathcal{O}$ is the set of possible output configurations.
- $O \in \mathcal{O}$ is a vector of outputs, one for each process, possibly with some missing values (due to crashes).

Tasks

Tasks are distributed functions defined by a triple $(\mathcal{I}, \mathcal{O}, \delta)$.

- $\mathcal{I}$ denotes the set of input configurations,
- Any $I \in \mathcal{I}$ is a vector of inputs, one for each process.
- $\mathcal{O}$ is the set of possible output configurations.
- $O \in \mathcal{O}$ is a vector of outputs, one for each process, possibly with some missing values (due to crashes).
- $\delta : \mathcal{I} \rightarrow 2^{\mathcal{O}}$ is a function that, to any input configuration $I \in \mathcal{I}$, associates the set $\delta(I) \subseteq \mathcal{O}$ of the output configurations that are allowed when starting from I.

- Decision tasks do not depend on inputs and outputs assignment.

- Decision tasks do not depend on inputs and outputs assignment.
- $\delta(I)$ only depends on the set of values in I.

- Decision tasks do not depend on inputs and outputs assignment.
- $\delta(I)$ only depends on the set of values in I.
- If I' only contains values of I then $\delta(I') \subseteq \delta I$.

- Decision tasks do not depend on inputs and outputs assignment.
- $\delta(I)$ only depends on the set of values in I.
- If I' only contains values of I then $\delta(I') \subseteq \delta I$.
- If $O \in \delta(I)$, then any vector O' containing only values appearing in O also belongs to $\delta(I)$.

- Decision tasks do not depend on inputs and outputs assignment.
- $\delta(I)$ only depends on the set of values in I.
- If I' only contains values of I then $\delta(I') \subseteq \delta I$.
- If $O \in \delta(I)$, then any vector O' containing only values appearing in O also belongs to $\delta(I)$.

- Consensus, k-set agreement are decision tasks, renaming isn't.

Any decision task that we can solve with n processes and t crashes, we can solve it with $t+1$ processes and t crashes.

The study of decision tasks computability can be reduced to the $n-1$-resilient case.

- t-set agreement is impossible to solve among $t+1$ processes with t crashes.

- t-set agreement is impossible to solve among $t + 1$ processes with t crashes.
- For any $n > t$, suppose we have a t-resilient algorithm for t-set agreement.

- t-set agreement is impossible to solve among $t + 1$ processes with t crashes.
- For any $n > t$, suppose we have a t-resilient algorithm for t-set agreement.
- We can then build a t-resilient algorithm for $t + 1$ processes/simulators.

- t-set agreement is impossible to solve among $t+1$ processes with t crashes.
- For any $n > t$, suppose we have a t-resilient algorithm for t-set agreement.
- We can then build a t-resilient algorithm for $t+1$ processes/simulators.
  - Use n safe-agreements to decide on how to map simulators inputs on processes.

- t-set agreement is impossible to solve among $t + 1$ processes with t crashes.
- For any $n > t$, suppose we have a t-resilient algorithm for t-set agreement.
- We can then build a t-resilient algorithm for $t + 1$ processes/simulators.
    - Use n safe-agreements to decide on how to map simulators inputs on processes.
    - Simulate the protocol.

- t-set agreement is impossible to solve among $t + 1$ processes with t crashes.
- For any $n > t$, suppose we have a t-resilient algorithm for t-set agreement.
- We can then build a t-resilient algorithm for $t + 1$ processes/simulators.
    - Use n safe-agreements to decide on how to map simulators inputs on processes.
    - Simulate the protocol.
    - Decide any value decided in the simulation.

- t-set agreement is impossible to solve among $t + 1$ processes with t crashes.
- For any $n > t$, suppose we have a t-resilient algorithm for t-set agreement.
- We can then build a t-resilient algorithm for $t + 1$ processes/simulators.
  - Use n safe-agreements to decide on how to map simulators inputs on processes.
  - Simulate the protocol.
  - Decide any value decided in the simulation.
- This solves t-set agreement between our $t + 1$ simulators.

Example

- t-set agreement is impossible to solve among $t + 1$ processes with t crashes.
- For any $n > t$, suppose we have a t-resilient algorithm for t-set agreement.
- We can then build a t-resilient algorithm for $t + 1$ processes/simulators.
  - Use n safe-agreements to decide on how to map simulators inputs on processes.
  - Simulate the protocol.
  - Decide any value decided in the simulation.
- This solves t-set agreement between our $t + 1$ simulators.
- Contradiction, so there is no such algorithm.

What matters is the number of crashes, not the number of processes.

- Borowsky, E., Gafni, E., Lynch, N., Rajsbaum, S.: *The BG distributed simulation algorithm*. Distributed Computing 14(3), 127146 (2001).

References

- Borowsky, E., Gafni, E., Lynch, N., Rajsbaum, S.: *The BG distributed simulation algorithm*. Distributed Computing 14(3), 127146 (2001).
- Gafni, E.: *The Extended BG Simulation and the Characterization of t-Resiliency*. STOC 2009.

- Borowsky, E., Gafni, E., Lynch, N., Rajsbaum, S.: *The BG distributed simulation algorithm*. Distributed Computing 14(3), 127146 (2001).
- Gafni, E.: *The Extended BG Simulation and the Characterization of t-Resiliency*. STOC 2009.
- Damien Imbs, Michel Raynal: *Visiting Gafni's Reduction Land: From the BG Simulation to the Extended BG Simulation*. SSS 2009.

Table of Contents

1 BG-Simulation
Problem Statement
Simulation Algorithm
Safe-Agreement
Putting Pieces Together
Computability Consequences

2 Renaming
Problem Statement
Renaming and Crashes
A Splitter-based Algorithm
A Snapshot-based Algorithm

- n asynchronous processes sharing atomic registers;

- n asynchronous processes sharing atomic registers;
- up to $n - 1$ of them may crash;

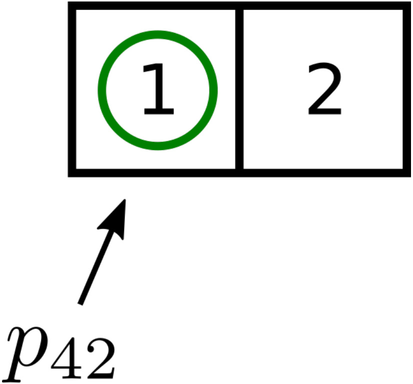
- n asynchronous processes sharing atomic registers;
- up to $n - 1$ of them may crash;
- they are given names in a large namespace $\{-N, \ldots, N\}, N >> n$;

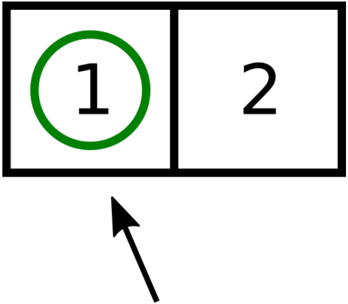
- n asynchronous processes sharing atomic registers;
- up to $n-1$ of them may crash;
- they are given names in a large namespace $\{-N, \ldots, N\}, N >> n$;
- k-renaming provides them with a GET_NAME operation that returns a new unique name in a smaller namespace $\{1, \ldots, k\}$.

- Using shorter names spares bandwidth, memory, storage, etc.

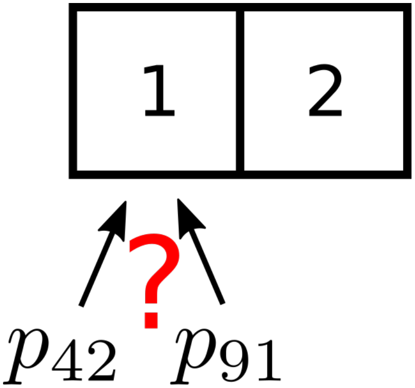
- Using shorter names spares bandwidth, memory, storage, etc.
- The "big" names are just a way to break symmetry, renaming protocols allow to dynamically compute unique identifiers.

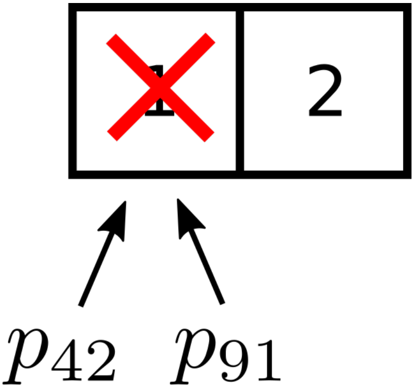
- Using shorter names spares bandwidth, memory, storage, etc.
- The "big" names are just a way to break symmetry, renaming protocols allow to dynamically compute unique identifiers.
- Several problems can be reduced to renaming (e.g. picking a unique transmitting frequency).

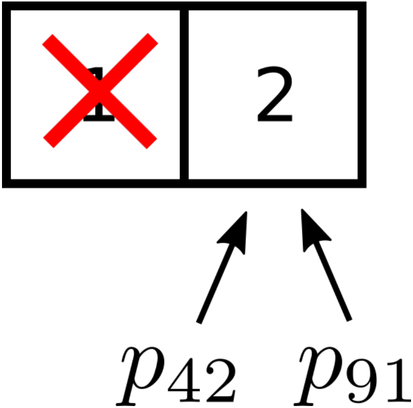

1
2
p_{42}

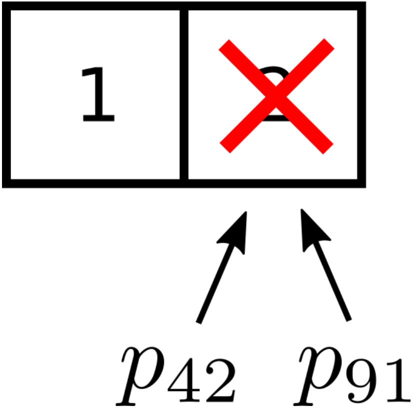


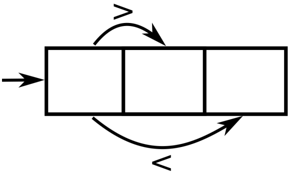
p_{91}

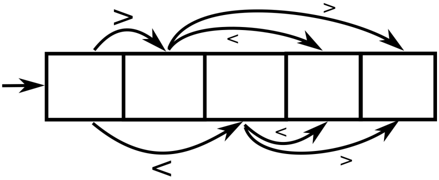

1
2
?
p_{42}
p_{91}

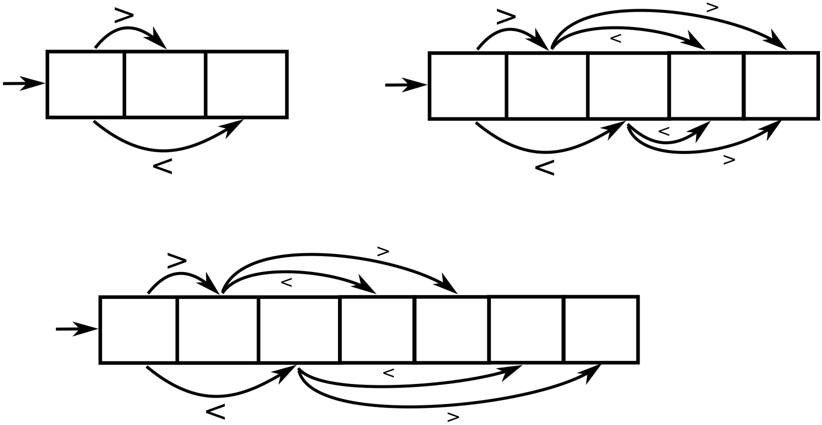

1
2
p_{42}
p_{91}

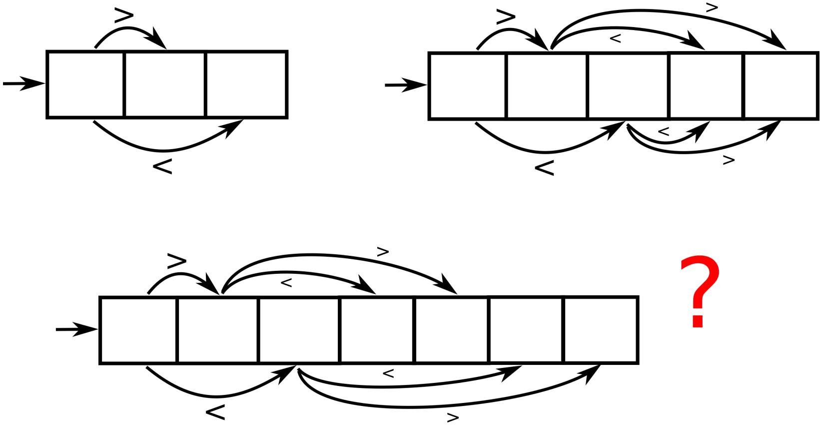

1
2
p_{42}
p_{91}


1
2
p_{42}
p_{91}









- Because of asynchrony and crashes, processes have to change name after a conflict.

- Because of asynchrony and crashes, processes have to change name after a conflict.
- We need to break symmetry between conflicting processes.

- Because of asynchrony and crashes, processes have to change name after a conflict.
- We need to break symmetry between conflicting processes.

Adaptive Renaming

Ideally the protocol should be adaptive: the largest name obtained should depend on the actual number of participating processes, not on the total number of processes.

The Splitter Object

- The splitter object offers a single operation DIRECTION.

The Splitter Object

- The splitter object offers a single operation DIRECTION.
- DIRECTION returns *right down* or *stop*.

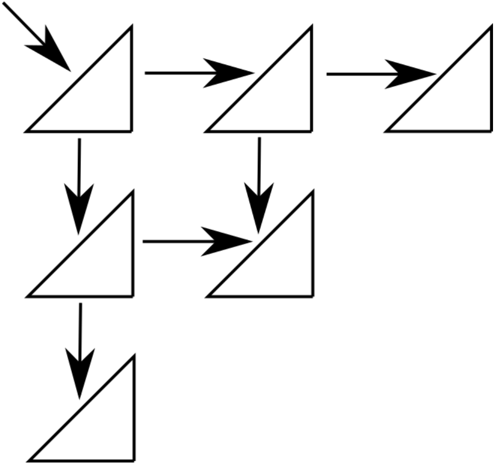
- The splitter object offers a single operation DIRECTION.
- DIRECTION returns *right down* or *stop*.
- If x processes invoke DIRECTION:

- The splitter object offers a single operation DIRECTION.
- DIRECTION returns *right down* or *stop*.
- If x processes invoke DIRECTION:
  - at most $x - 1$ obtain *right*;

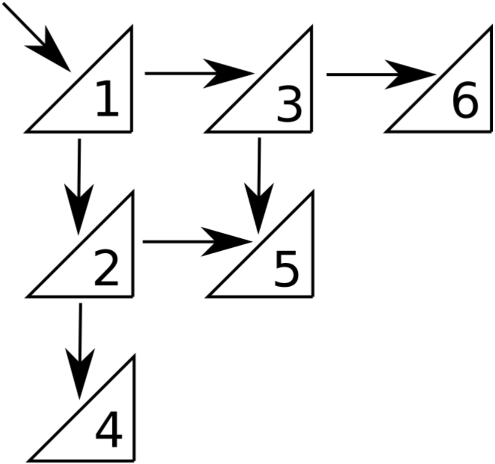
- The splitter object offers a single operation DIRECTION.
- DIRECTION returns *right* *down* or *stop*.
- If x processes invoke DIRECTION:
  - at most $x - 1$ obtain *right*;
  - at most $x - 1$ obtain *down*;

- The splitter object offers a single operation DIRECTION.
- DIRECTION returns *right down* or *stop*.
- If x processes invoke DIRECTION:
  - at most $x - 1$ obtain *right*;
  - at most $x - 1$ obtain *down*;
  - at most one obtains *stop*.

- The splitter object offers a single operation DIRECTION.
- DIRECTION returns *right down* or *stop*.
- If x processes invoke DIRECTION:
  - at most $x - 1$ obtain *right*;
  - at most $x - 1$ obtain *down*;
  - at most one obtains *stop*.
- Any invocation to DIRECTION by a correct process terminates.



```
operation NEW_NAME(id)
    d ← 1; r ← 1; move ← down
    while move ≠ stop do move ← S[d, r].DIRECTION(id);
        if move = right then
            r ← r + 1
        else if move = down then
            d ← d + 1
        end if
    end while
    return (d + r − 1)(d + r − 2)/2 + r
end operation
```


1
3
6
2
5
4

Great! But we can do even better.

```
operation NEW_NAME(id)
    name ← 1
    while true do
        MEM[i] ← ⟨id, name⟩
        snap ← MEM.snapshot()
        if ∀j ≠ i : snap[j].name ≠ name then
            return name
        else
            free ← names of {1, . . . , ∞} that do not appear in snap
            rank ← rank of id in the set of identifiers appearing in snap
            name ← name at position rank in free
        end if
    end while
end operation
```

Unicity After deciding their name, a process lets it in its register.

Unicity After deciding their name, a process lets it in its register.

No snapshot occurring after its last write can lead to another process deciding the same name.

Unicity
: After deciding their name, a process lets it in its register.

  No snapshot occurring after its last write can lead to another process deciding the same name.

Termination
: If a set of processes never decides, their *rank* variables will stabilize on distinct values.

Unicity After deciding their name, a process lets it in its register.

No snapshot occurring after its last write can lead to another process deciding the same name.

Termination If a set of processes never decides, their *rank* variables will stabilize on distinct values.

The one with the smallest *rank* is then eventually alone to propose its new name. Contradiction.

- BG-simulation allows to simulate larger systems while preserving the number of crashes.

- BG-simulation allows to simulate larger systems while preserving the number of crashes.

- Renaming algorithms distribute new unique names from a smaller namespace.