Renaming Algorithms

Corentin Travers

LaBRI, Bordeaux

CoA Workshop , April 2019

Origin: Mutual Exclusion

```
a := 2;
// enter critical section
{ // critical section
        b:= 2 +a;
        c:= 2*b - 4;
}
// exit critical section
```

[Dijsktra 1965]

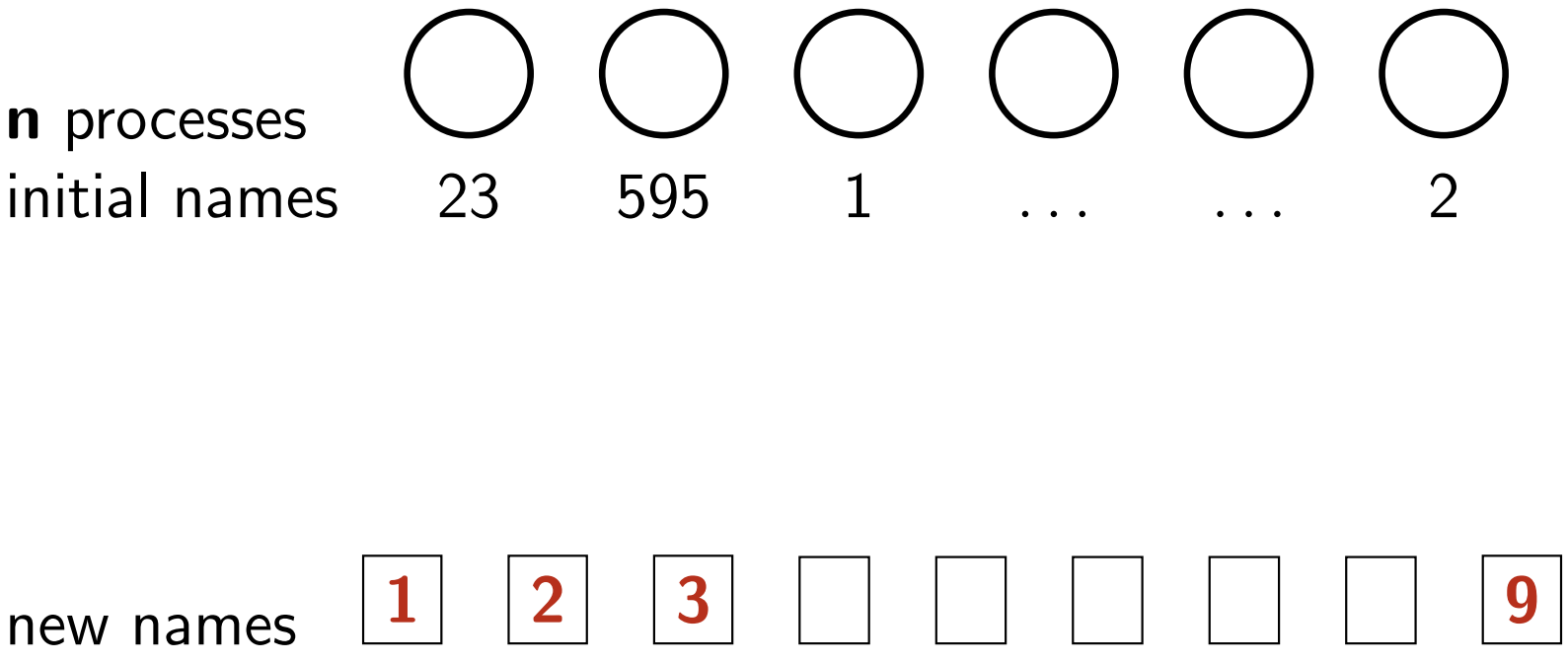
Exclusive Access

Resource can be accessed by at most one process at a time

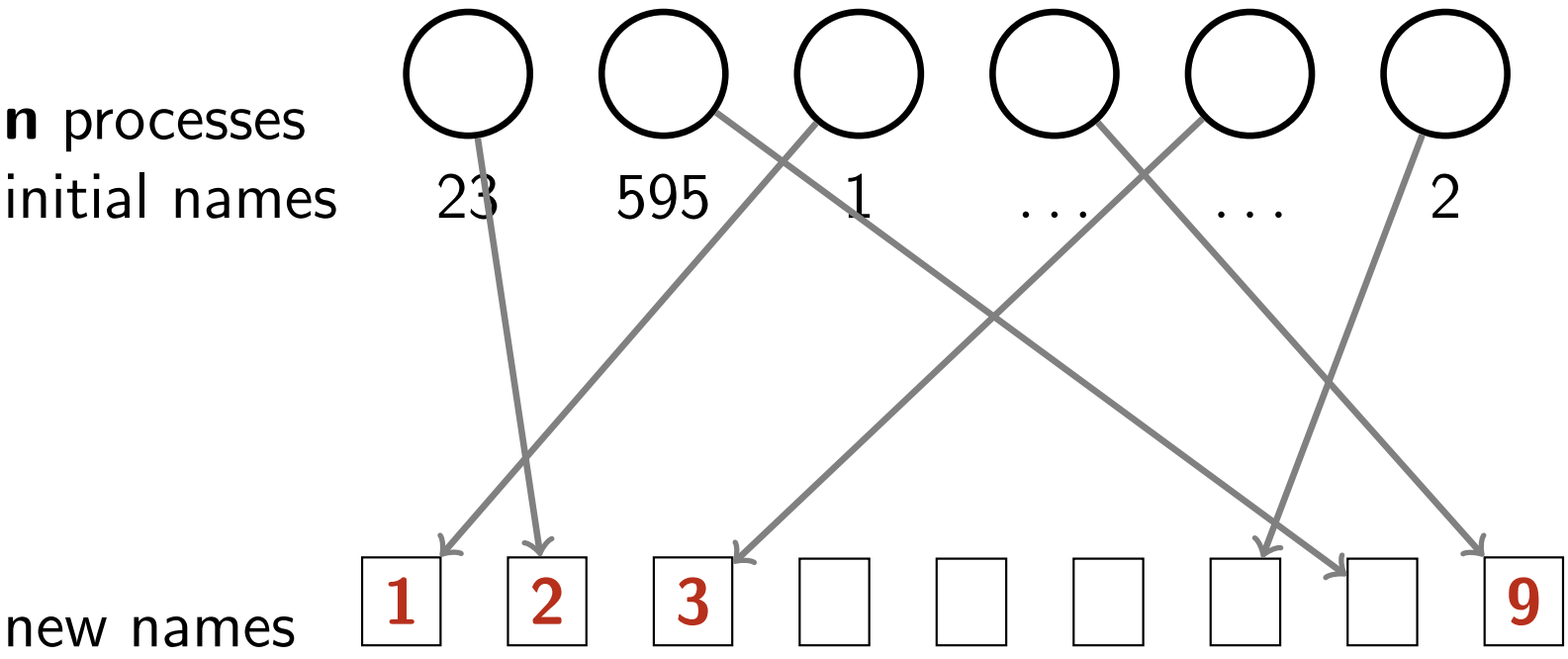
Exclusive Access

Resource can be accessed by at most one process at a time

Renaming



Renaming



One-shot Renaming

Parameters

- total **n** number of processes
- **[1..N]** range of initial names
- **[1..M]** range of new names **M** $<<$ **N**

One-shot Renaming

Parameters

- total **n** number of processes
- **[1..N]** range of initial names
- **[1..M]** range of new names **M** $<<$ **N**

getname()

- Returns a unique name in $[1..M]$
- At most one invocation per process

Adaptive Renaming

- Range of allocated names is a function of # invocations of getname()

if k procs invoke getname, each new name $\in [1..f(k)]$

Renaming in Shared Memory

Communication: Registers

Operations :

- **write**(v)
- **read**()

Communication: Registers

register Operations :

- **write**(v)
- **read**()

Size :

- unbounded

Communication: Registers

register Operations :

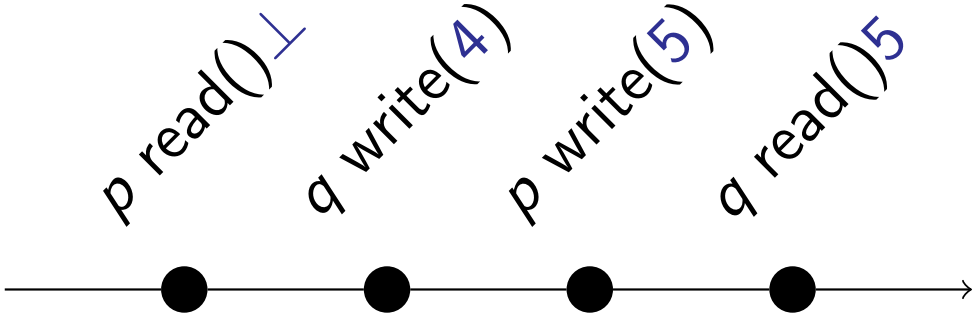
- **write**(v)
- **read**()

Size :

- unbounded
- (large enough to store any initial name)

Atomic Register

- Each operation is instantaneous
- **Read** returns last value **written**



Asynchronous Process

- Each Proc. is arbitrarily slow

Asynchronous Process

- Each Proc. is arbitrarily slow

p read()
q write()
p write()
q read()

Asynchronous Process

- Each Proc. is arbitrarily slow

p read()
q write()
p write()
q read()

p read()
p write()
q write()
q read()

p read()
q write()
q read()
p write()

Asynchronous Process

- Each Proc. is arbitrarily slow
- Each proc. may not participate

p read()
q write()
p write()
q read()

p read()
p write()
q write()
q read()

p read()
q write()
q read()
p write()

Wait-free Algorithms

n processes, initial names 1..**N**

getname()
    perform at most **B**(**n**, **N**) read/write ops
    return *new name*

Wait-free Algorithms

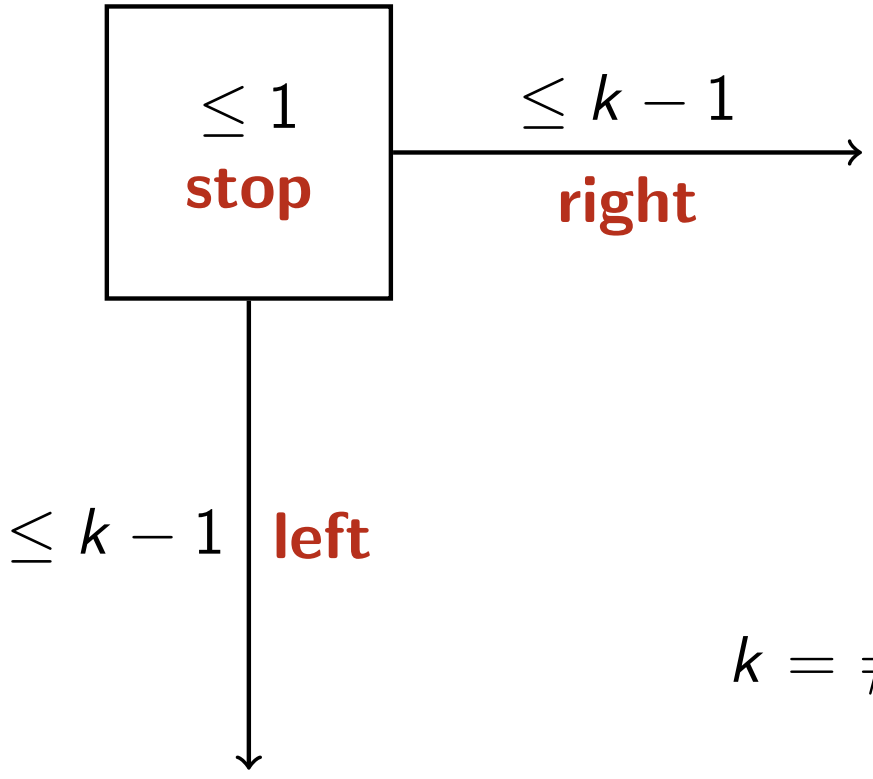
n processes, initial names 1..**N**

```
getname()
    perform at most B(n, N) read/write ops
    return new name
```

- Mutual exclusion prohibited
- Provide crash-fault tolerance

Moir and Anderson Renaming Algorithm

Splitter



$k = \#$ procs invoking the object

Splitter Object

- Each invocation returns **stop**, **left** or **right**

If only one process invokes the object:

- it obtains **stop**

If $k \geq 2$ processes invoke the object:

- At *least one proc.* obtains **stop** or **right**
- At *least one proc.* obtains **stop** or **left**

Splitter implementation

register C, $Door$
Init $Door \leftarrow open$

Splitter implementation

register C, $Door$
Init $Door \leftarrow open$

split():
    C.write(my_id)
    $d \leftarrow Door$.read()

Splitter implementation

register C, $Door$
Init $Door \leftarrow open$

split():
    C.write(my_id)
    $d \leftarrow Door$.read()
    if $d = closed$ **then** return **right**

Splitter implementation

```
register C, Door
Init Door ← open

split():
    C.write(my_id)
    d ← Door.read()
    if d = closed then return right
    else
        Door.write(closed)
        c ← C.read()
```

Splitter implementation

```
register C, Door
Init Door ← open

split():
    C.write(my_id)
    d ← Door.read()
    if d = closed then return right
    else
        Door.write(closed)
        c ← C.read()
        if c = my_id then return stop
                     else  return left
```

Solo Executions

```
register C, Door
Init Door ← open

split():
    C.write(my_id)
    d ← Door.read()
    if d = closed then return right
    else
        Door.write(closed)
        c ← C.read()
        if c = my_id then return stop
                     else  return left
```

Solo execution:
object returns **stop**

right $\leq k - 1$

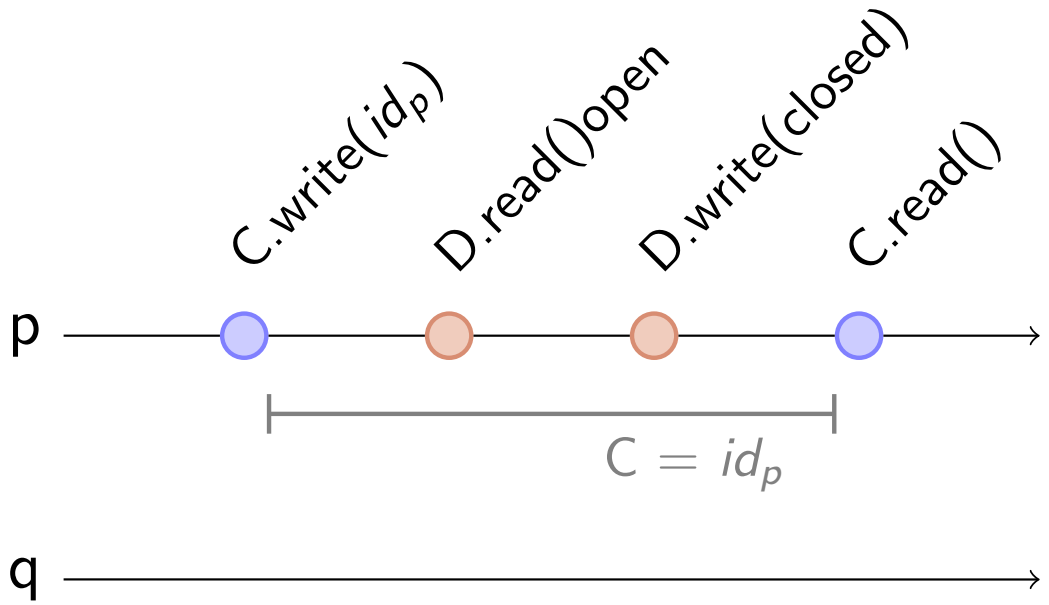
```
register C, Door
Init Door ← open

split():
    C.write(my_id)
    d ← Door.read()
    if d = closed then return right
    else
        Door.write(closed)
        c ← C.read()
        if c = my_id then return stop
                     else return left
```

Some proc has to close the door :
at least one proc. *does not* obtain **right**

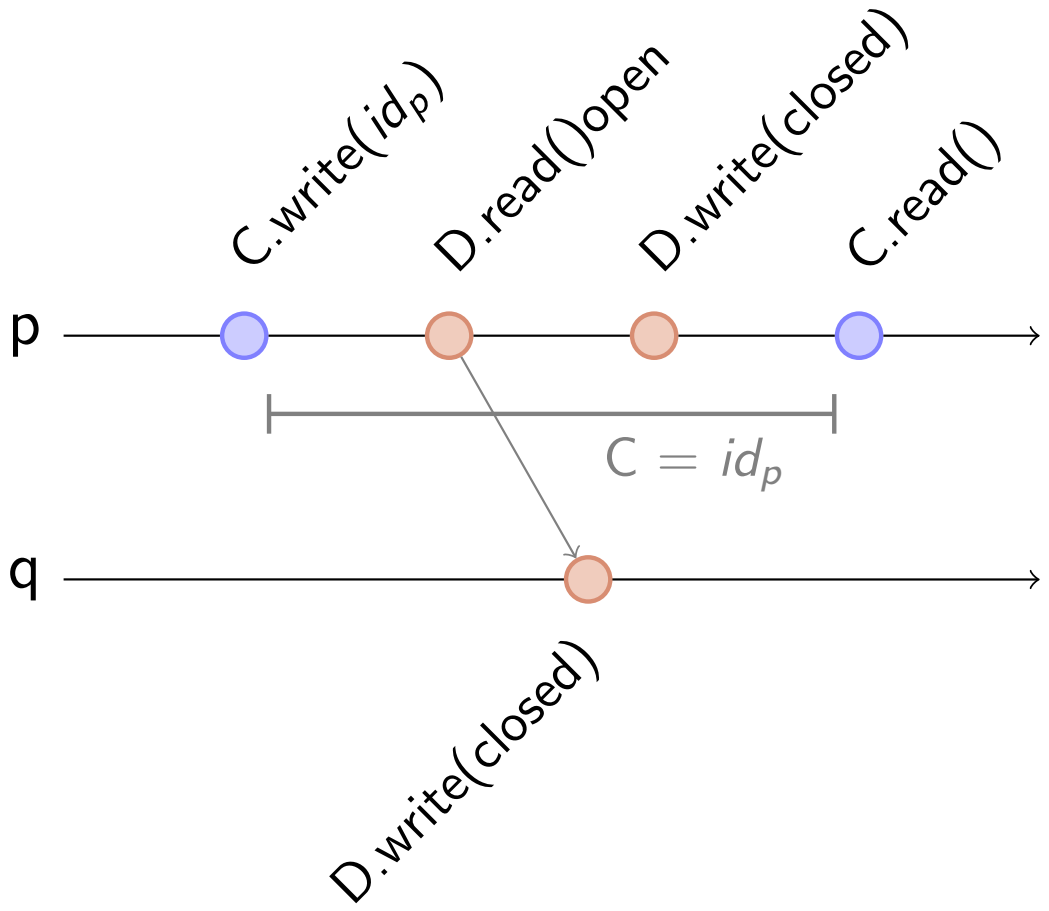
stop ≤ 1

By contradiction: assume p and q obtain **stop**



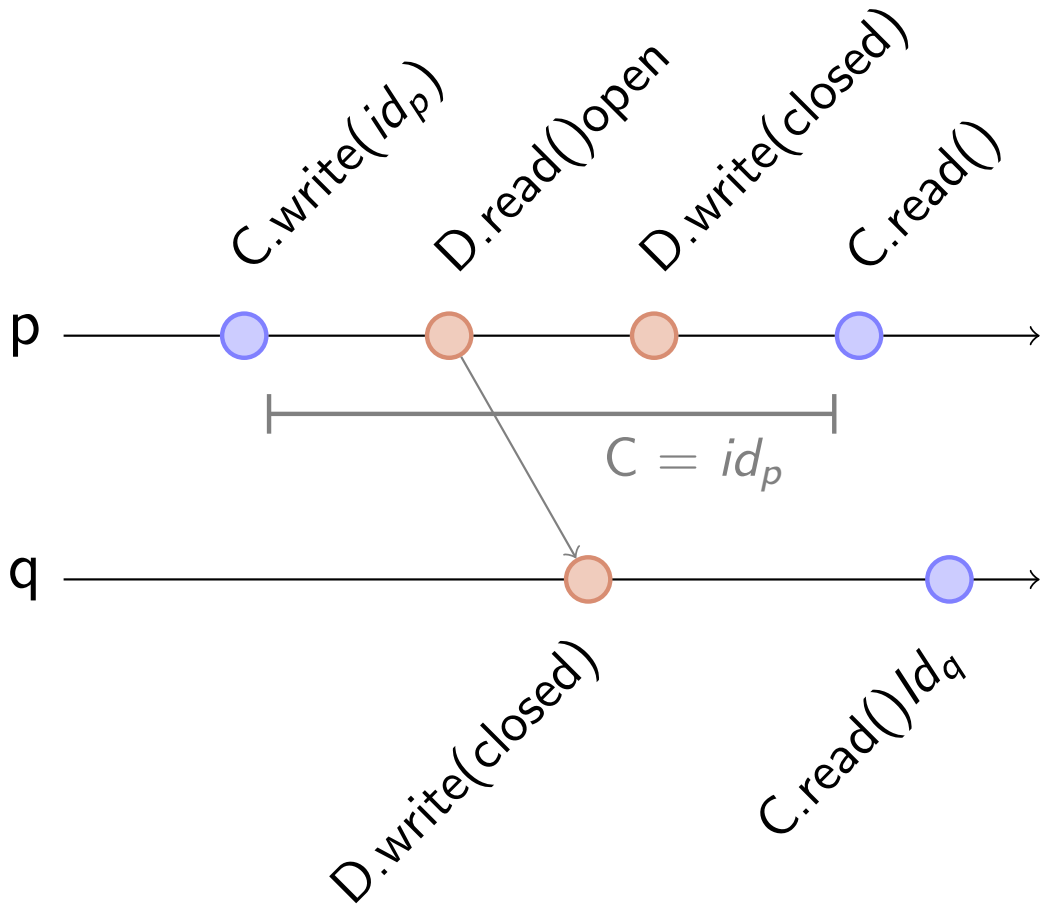
stop ≤ 1

By contradiction: assume p and q obtain **stop**



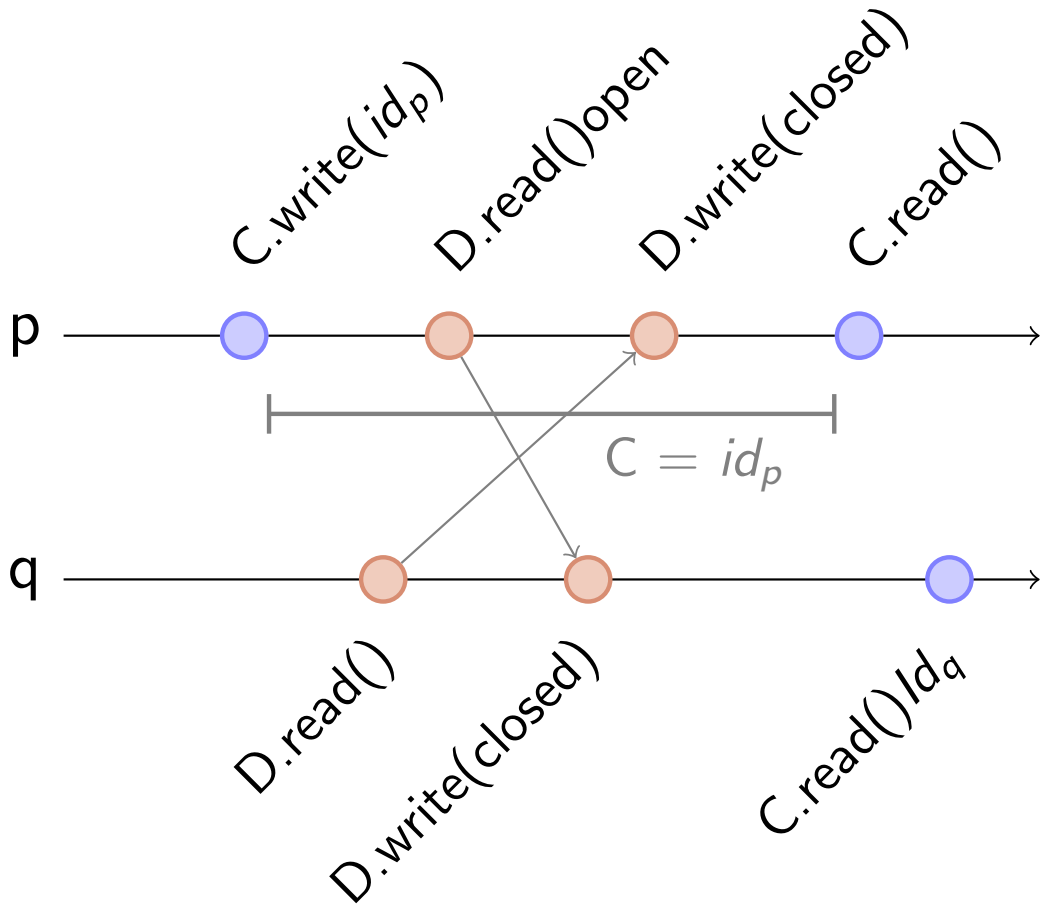
stop ≤ 1

By contradiction: assume p and q obtain **stop**



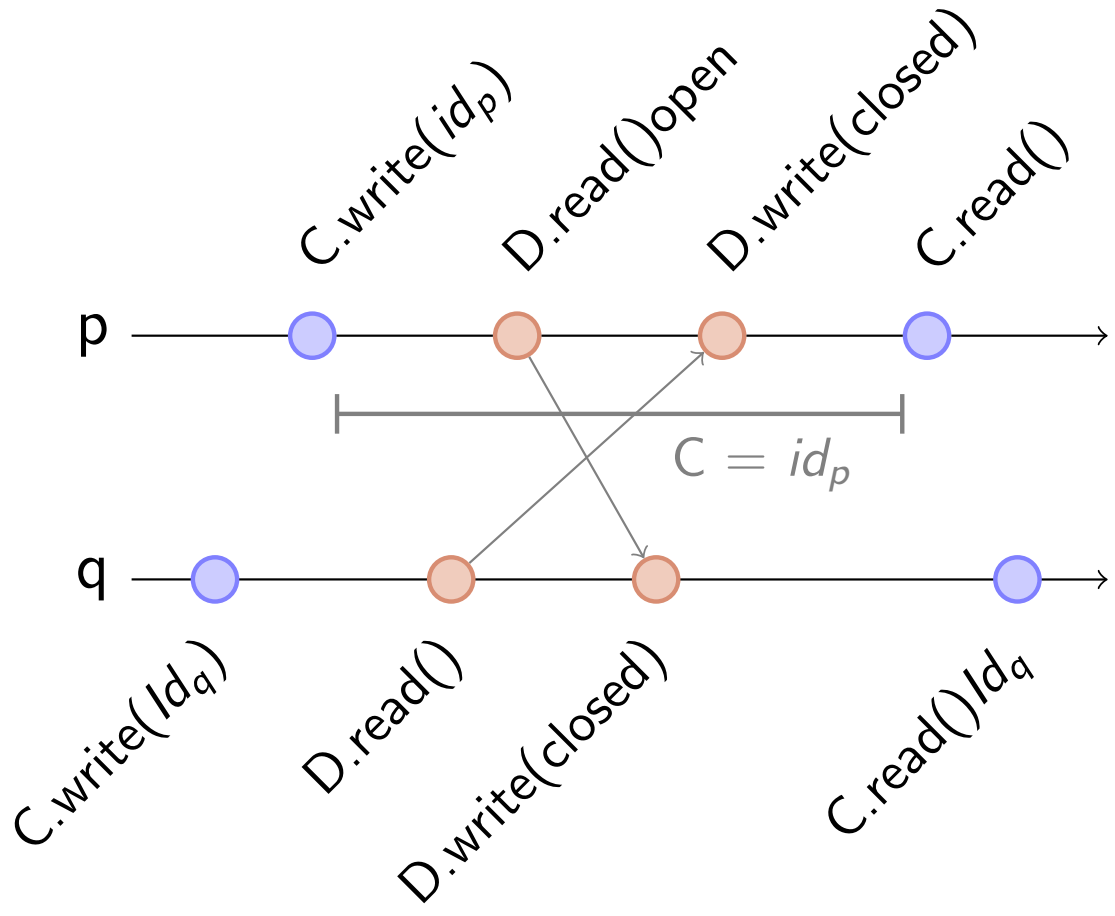
$\#\ \text{stop} \leq 1$

By contradiction: assume p and q obtain **stop**



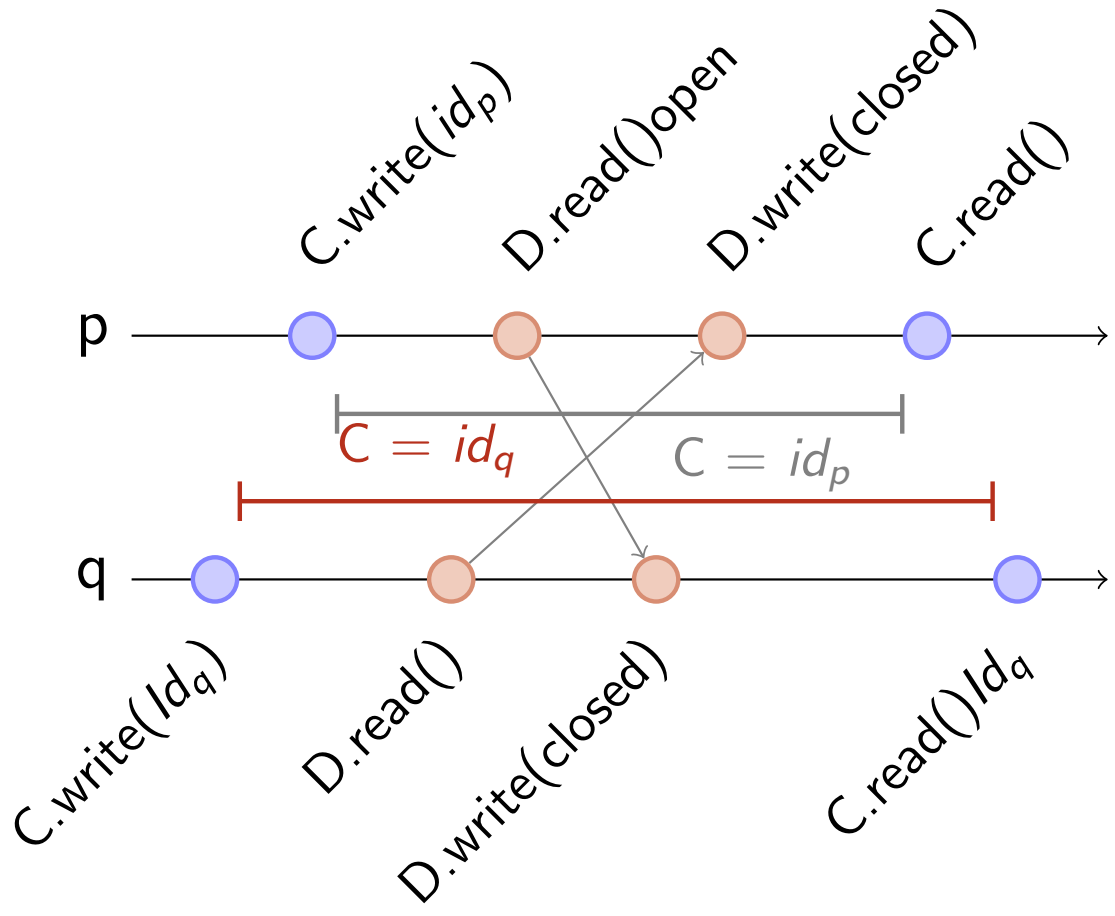
stop ≤ 1

By contradiction: assume p and q obtain **stop**



stop ≤ 1

By contradiction: assume p and q obtain **stop**



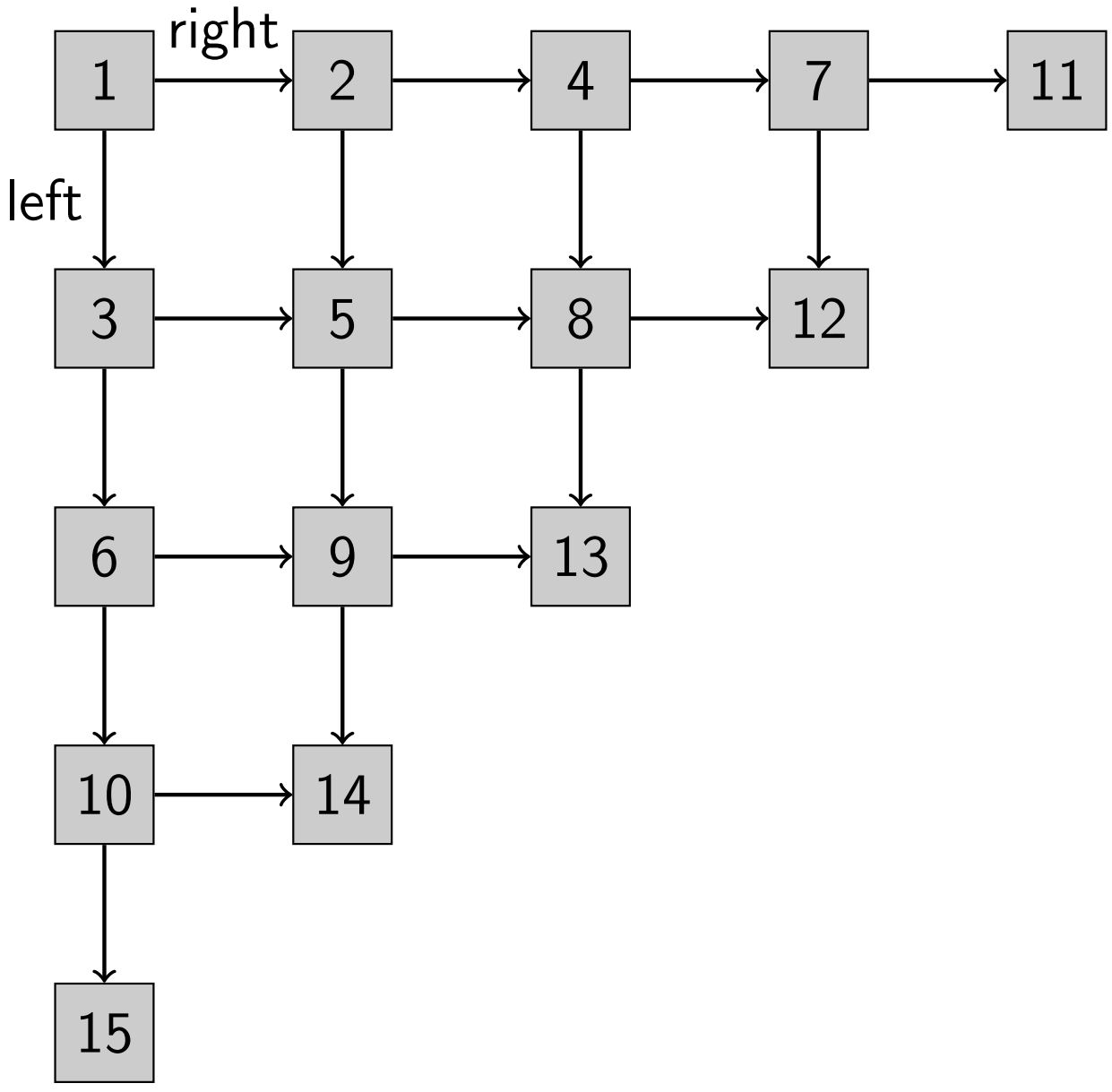
left $\leq k - 1$

```
register C, Door
Init Door ← open

split():
    C.write(my_id)
    d ← Door.read()
    if d = closed then return right
    else
        Door.write(closed)
(*)     c ← C.read()
        if c = my_id then return stop
                     else  return left
```

If every proc gets **left** :
last proc. that writes C
sees its id at (*)
and return **stop**

A Network of Splitters



Moir-Anderson Algorithm

A network of splitters

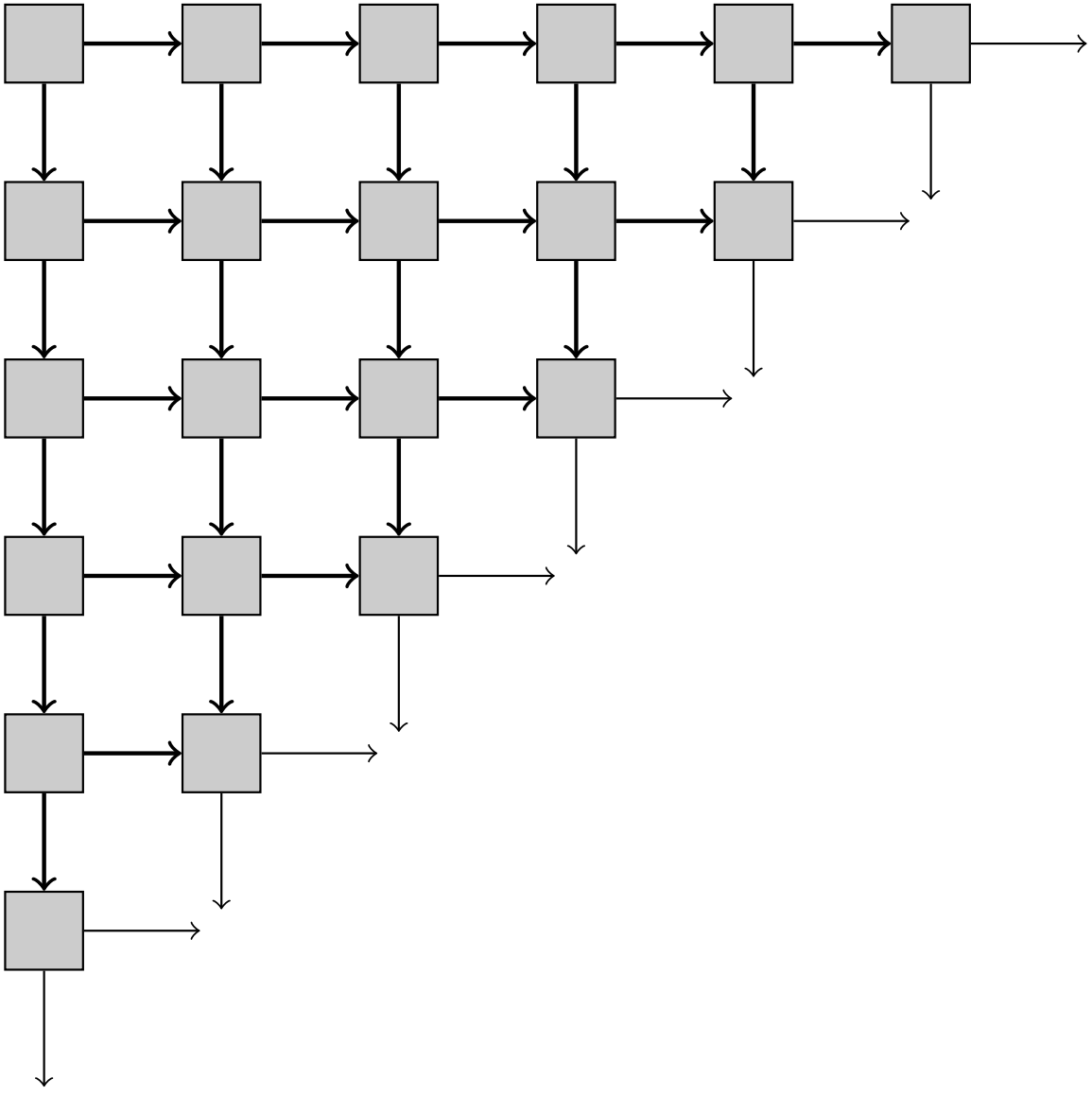
- Each splitter numbered from 1 to $k(k+1)/2$
- if p gets **stop** from splitter i :
  $\rightarrow$ returns i as its new name

If k procs. invoke getname()

- new names $\in [1..\theta(\mathbf{k^2})]$
- $\mathbf{O(k)}$ read/write operations per process

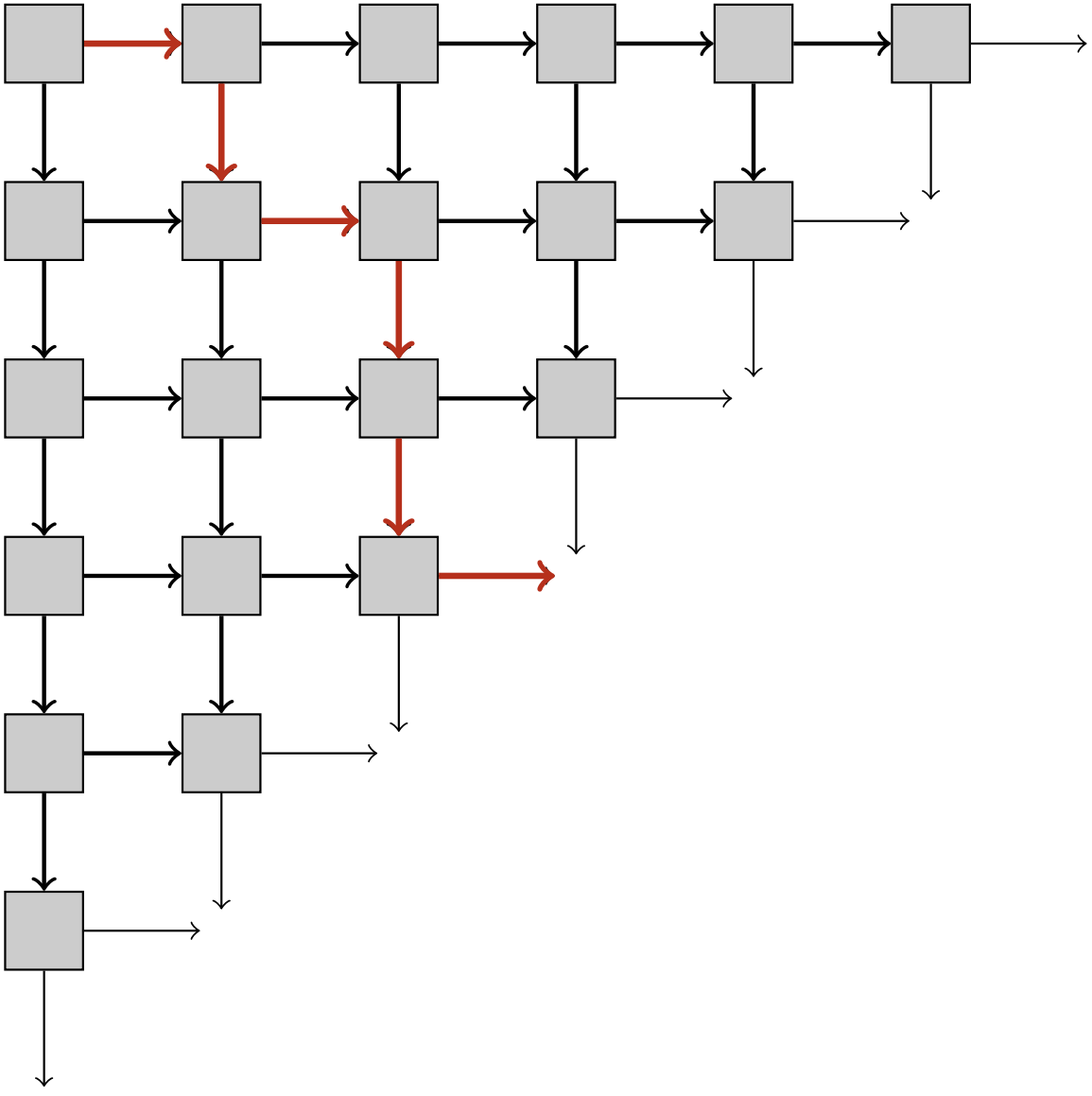
Termination

Every participating process gets a new name



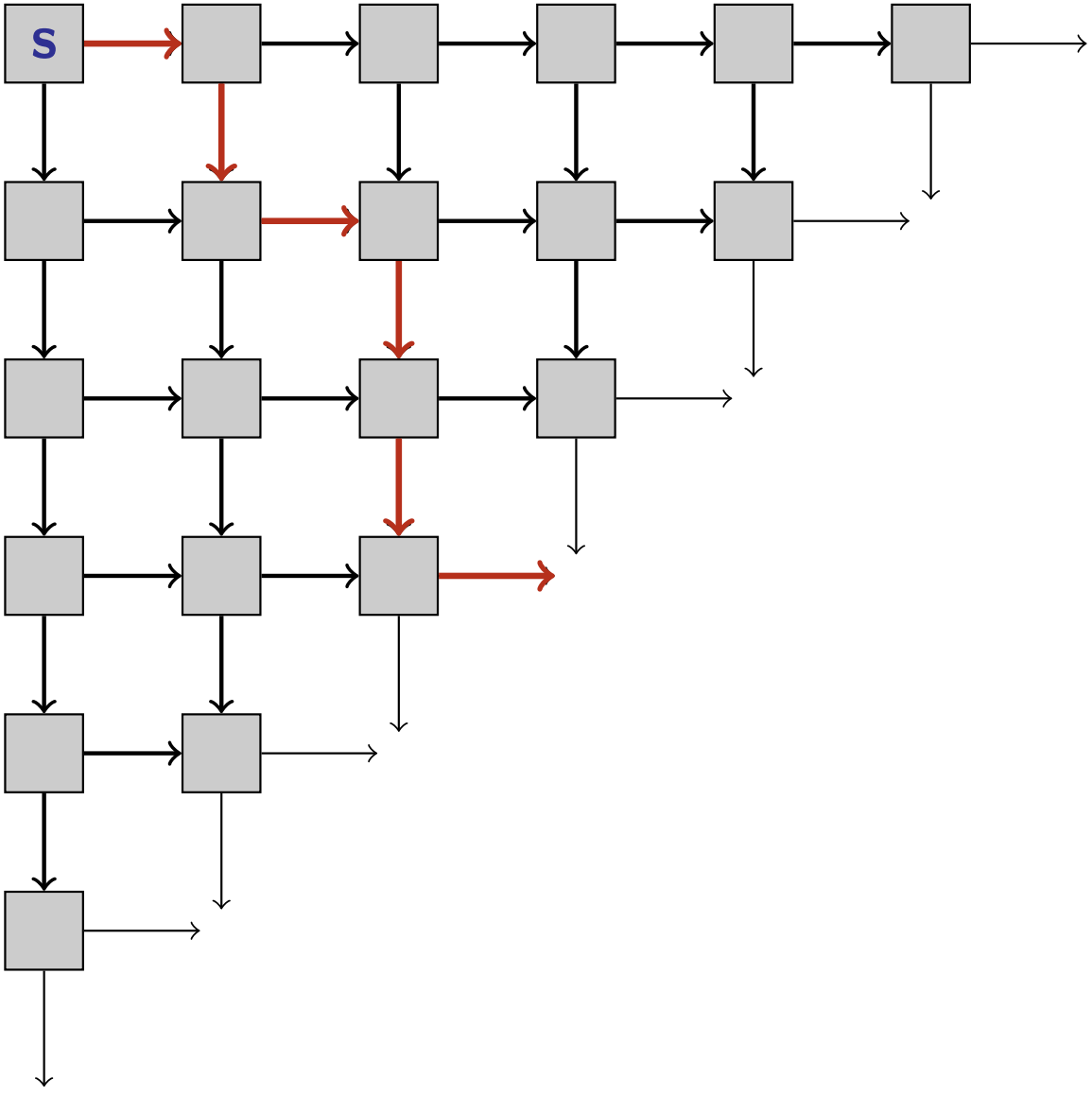
Termination

Every participating process gets a new name



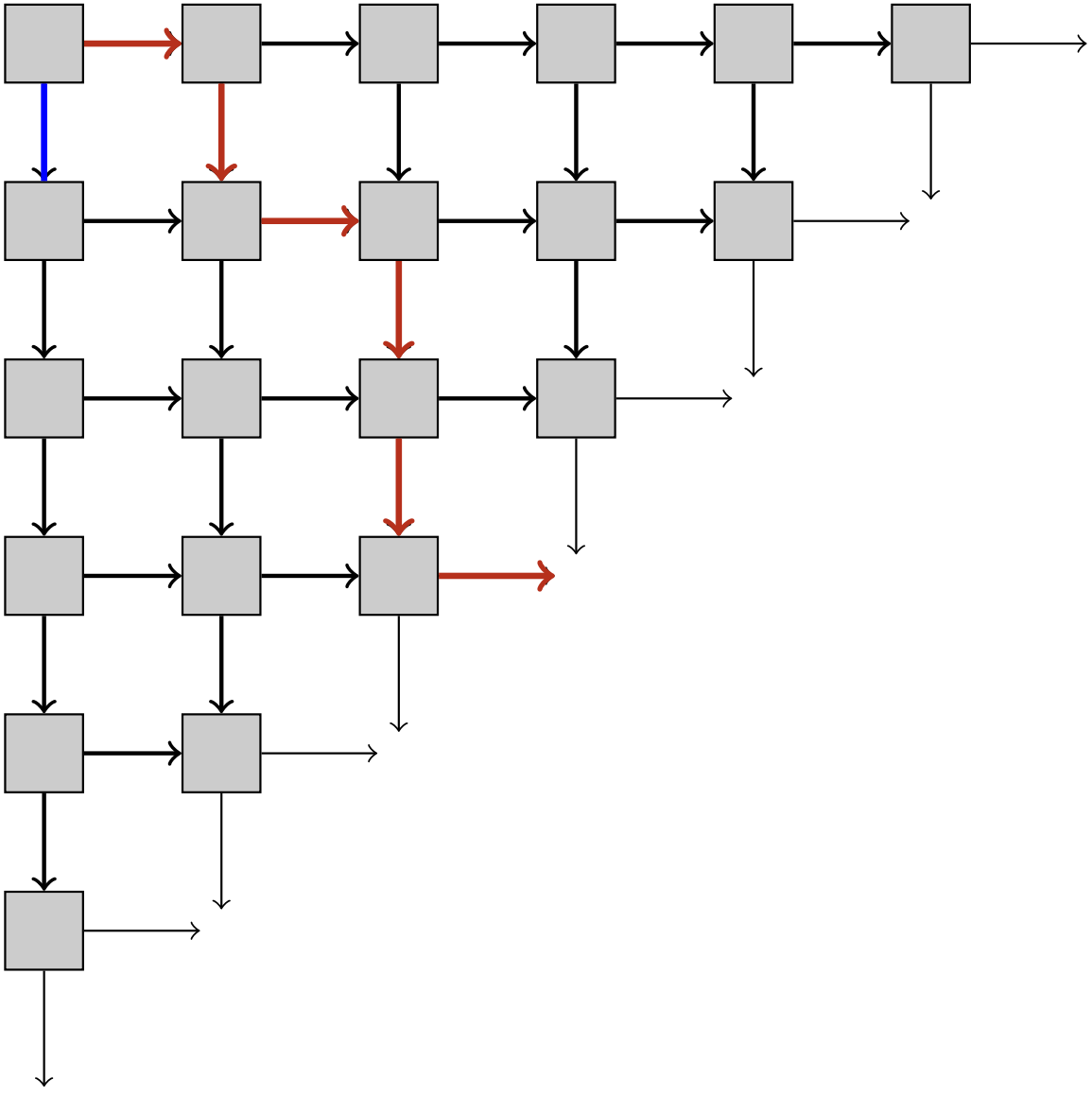
Termination

Every participating process gets a new name



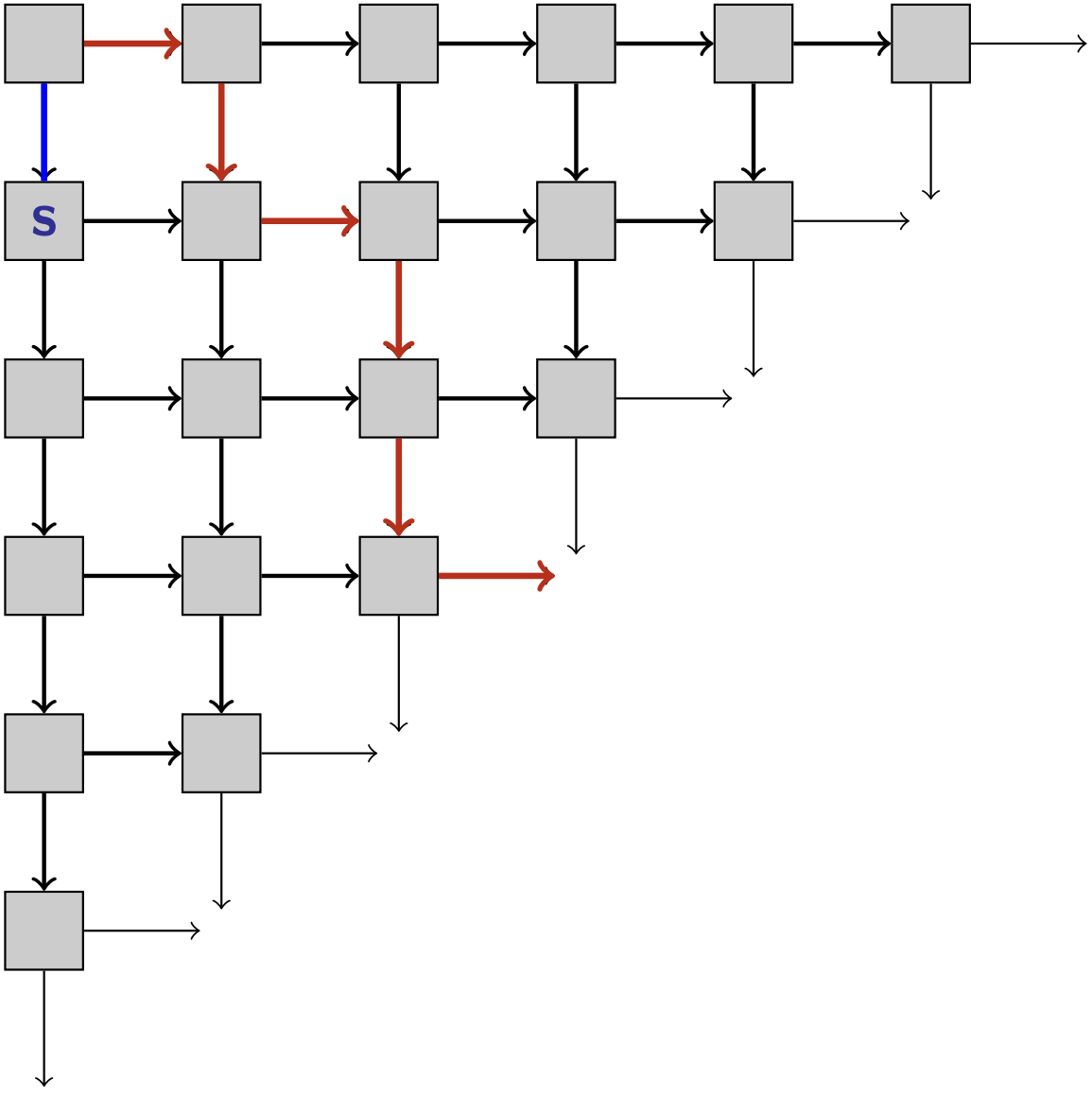
Termination

Every participating process gets a new name



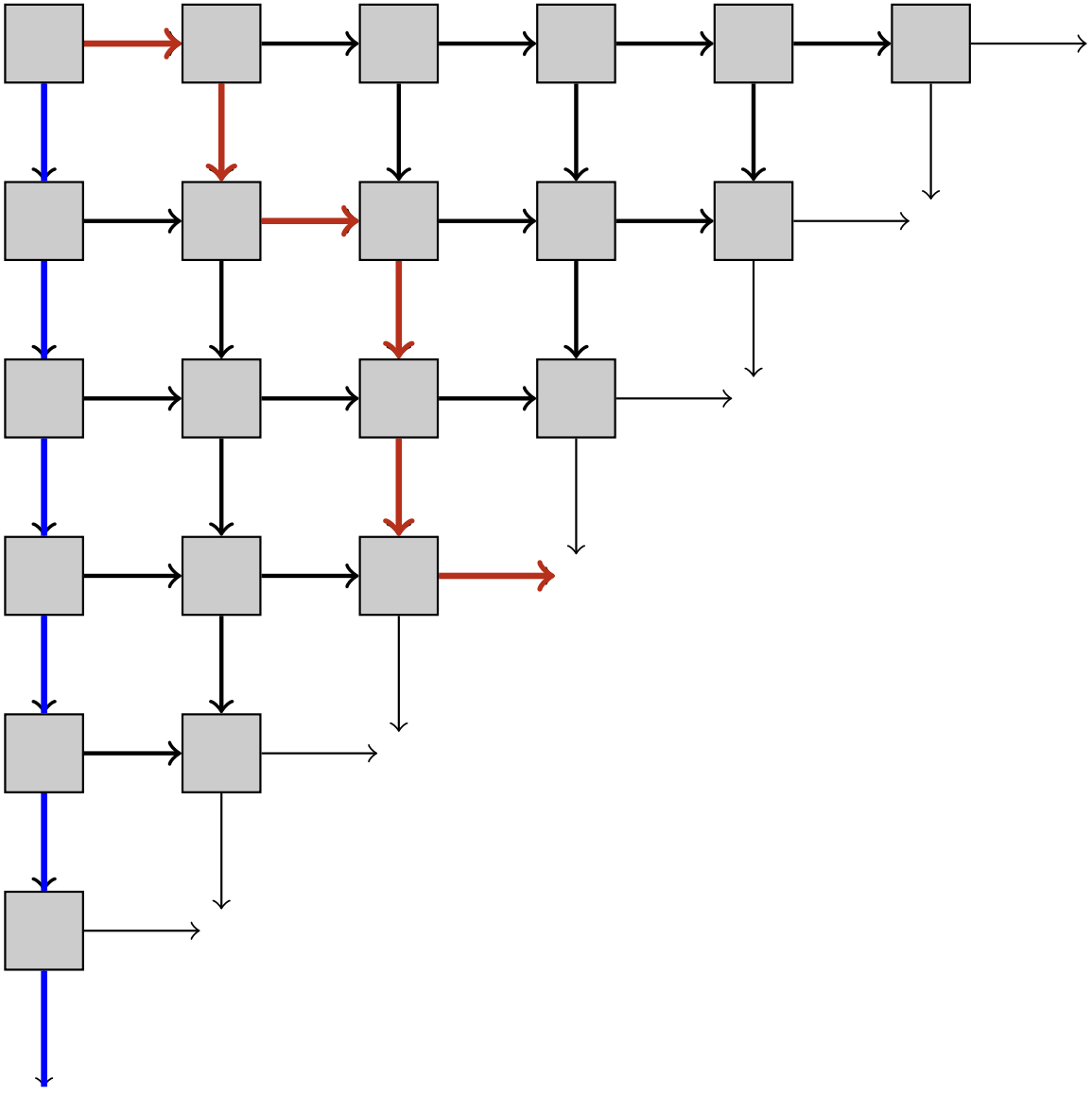
Termination

Every participating process gets a new name



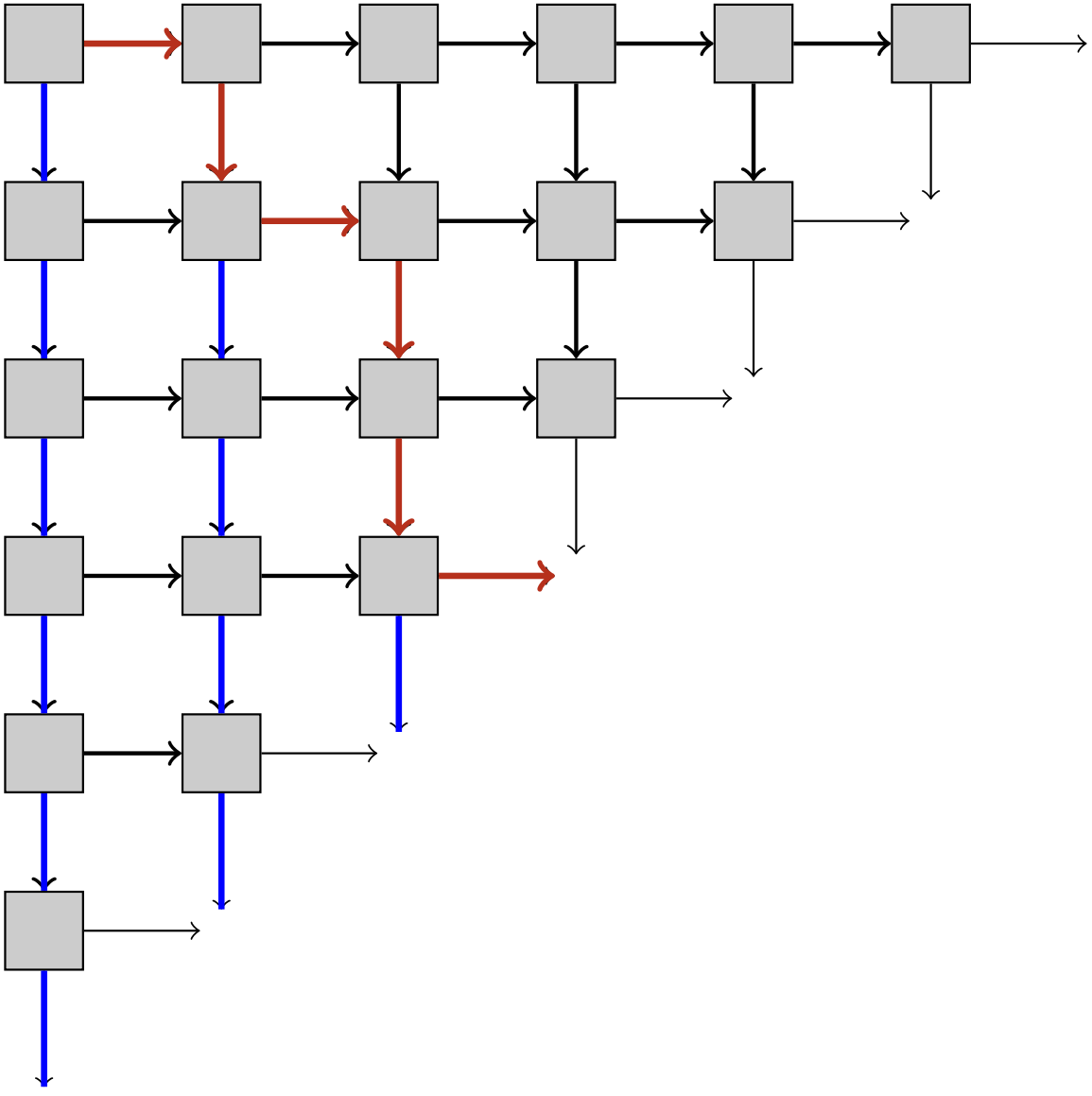
Termination

Every participating process gets a new name



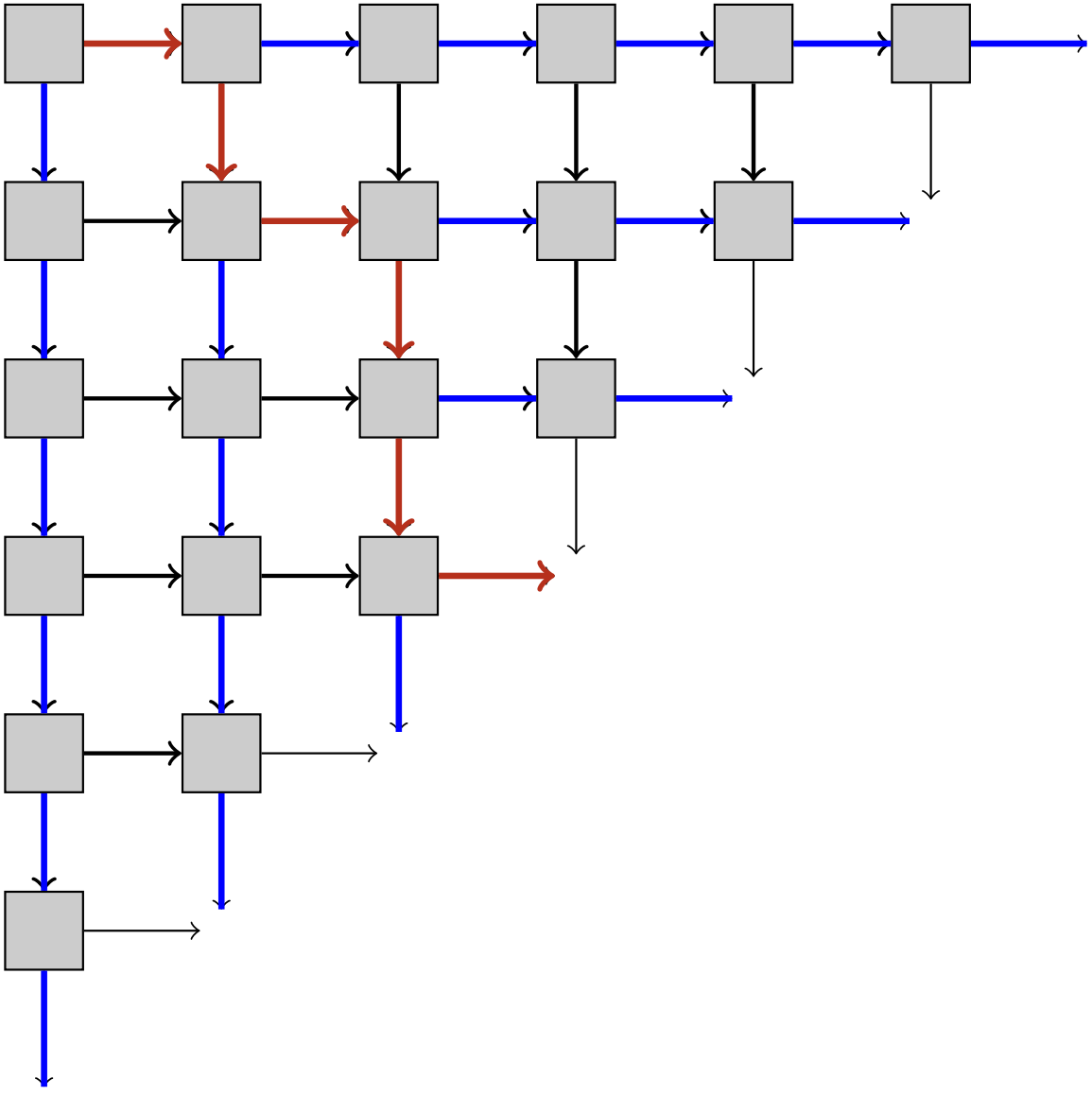
Termination

Every participating process gets a new name



Termination

Every participating process gets a new name

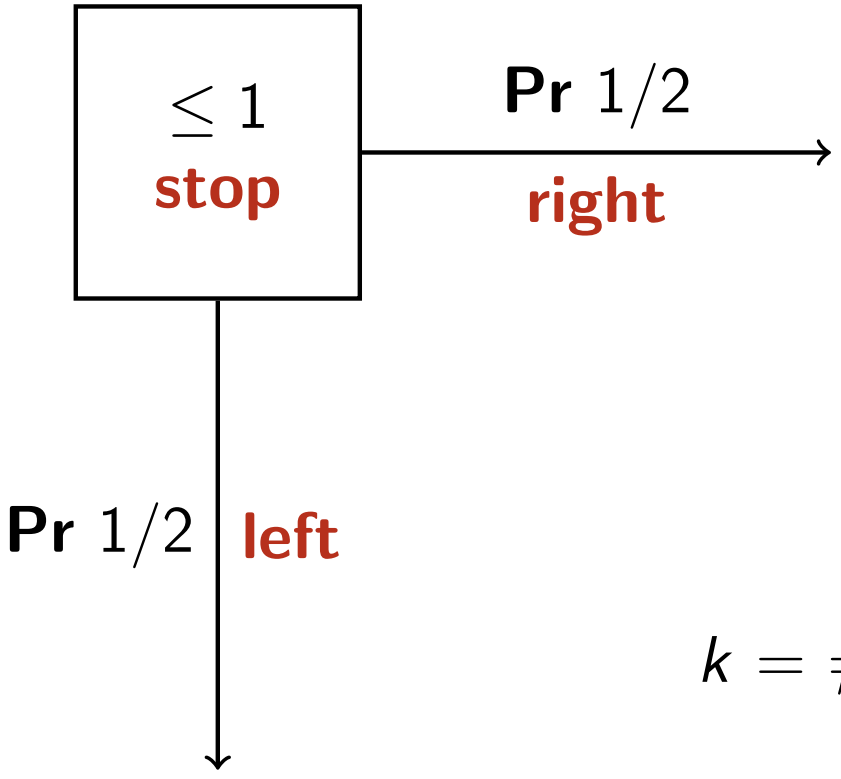


Splitter-based renaming

For k participating processes

- namespace: $[1..\Theta(k^2)]$
- work: $O(k)$ per process/ $O(k^2)$ total

Randomized splitter



$k = \#$ procs invoking the object

Randomized Splitter Object

- Each invocation returns **stop**, **left** or **right**

If only one process invokes the object:

- It obtains **stop**

If $k \geq 2$ processes invoke the object:

- At most one process obtains **stop**
- Each process that does not **stop**
  - obtains **right** with proba. $1/2$
  - obtains **left** with proba. $1/2$.

Randomized Splitter Implementation

shared: splitter S `/* non randomized */`

randsplit():

$dir \leftarrow S$.split()

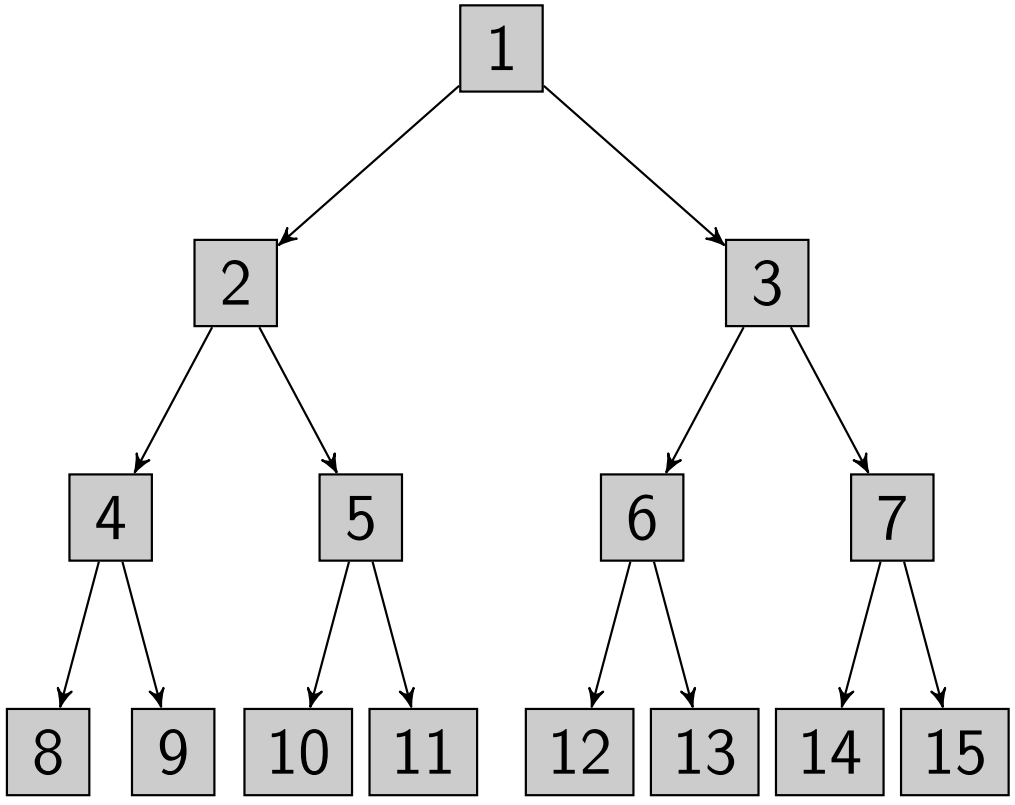
if $dir = stop$ **then** return **stop**

$x \leftarrow$ flip_coin() `/* dir` $\in \{left, right\}$ `*/`

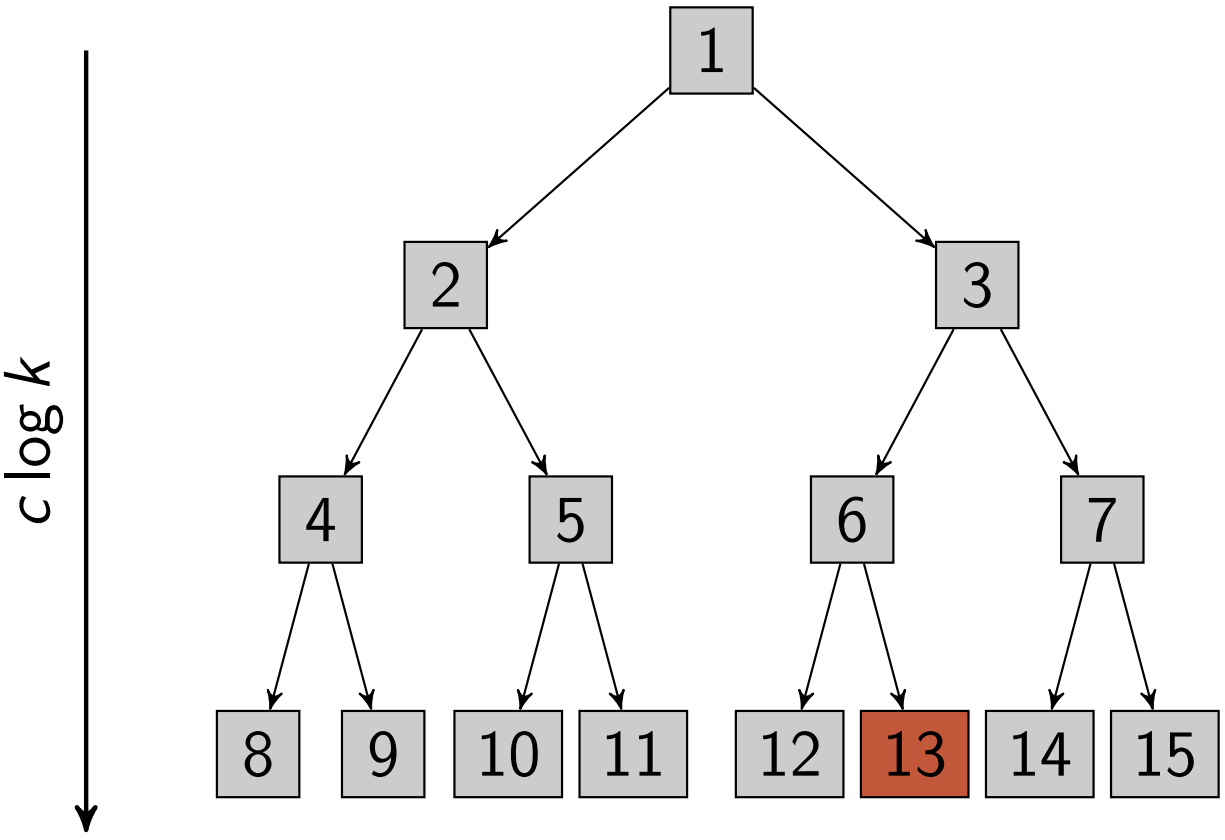
if $x = head$ **then** return **right else** return **left**

Tree of Randomized Splitters

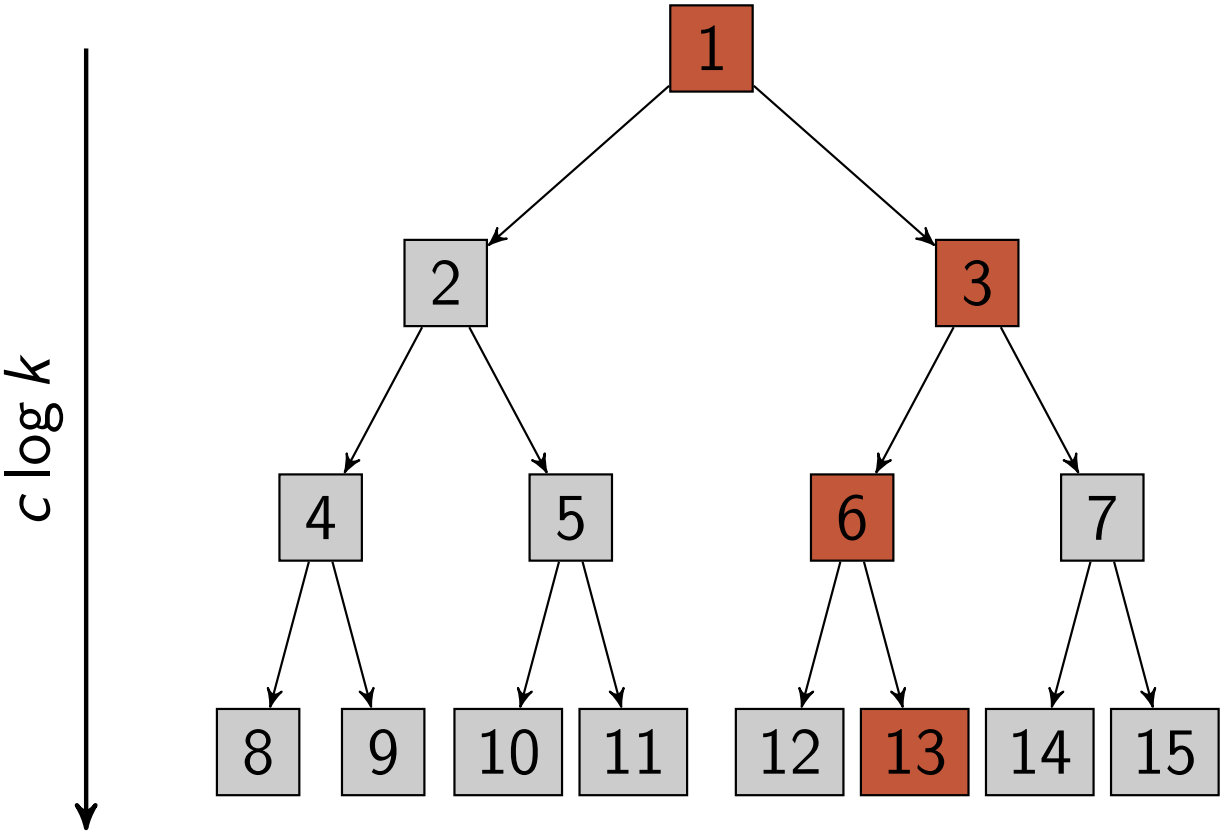
Attiya et. al 2006



Tree of Randomized Splitters

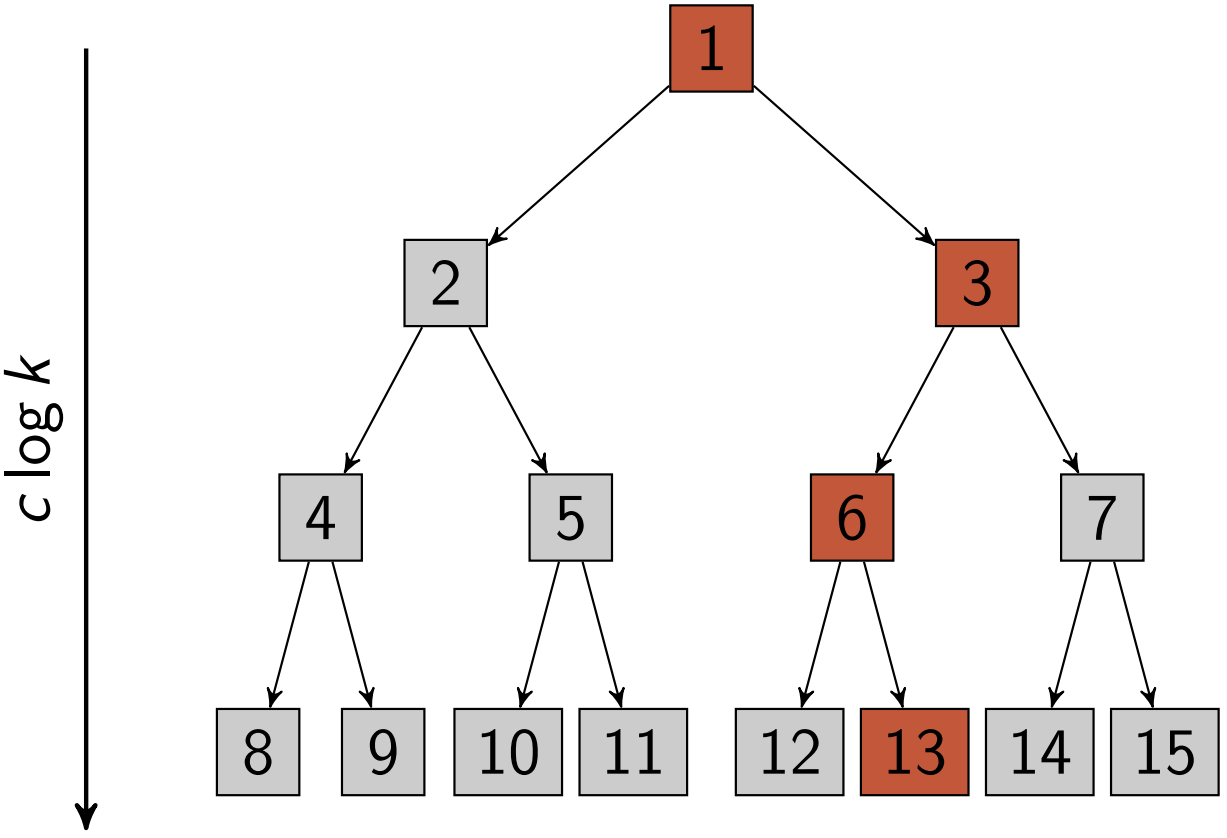


Tree of Randomized Splitters



- $Pr[q \text{ same rand. choices as } p] = \frac{1}{2}^{c \log k}$

Tree of Randomized Splitters



- $Pr[q \text{ same rand. choices as } p] = \frac{1}{2}^{c \log k}$
- $Pr[\text{some proc does not decide}] \leq \binom{k}{2} \frac{1}{k^c} \leq \frac{1}{k^{c-2}}$

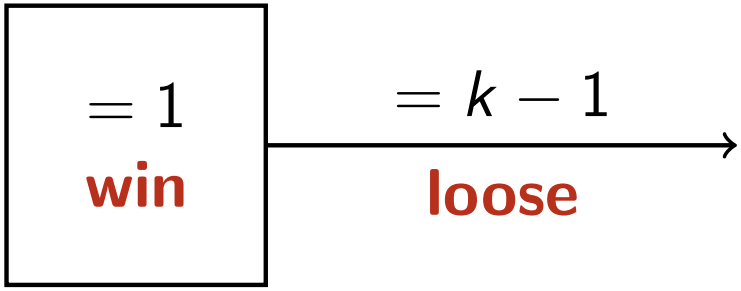
So far

	namespace	work per proc.	
grid of splitter	$O(k^2)$	$O(k)$	
tree of randomized splitters	$O(k^c)$	$O(c \log k)$	w.h.p.

So far

	namespace	work per proc.	
grid of splitter	$O(k^2)$	$O(k)$	
tree of randomized splitters	$O(k^c)$	$O(c \log k)$	w.h.p.

Linear namespace ?



$k = \#$ procs invoking the object

Test&set Object

Each invocation returns **win** or **loose**

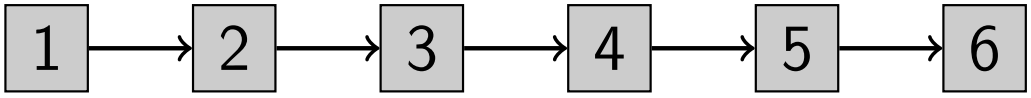
- *Exactly one proc.* obtains **win**

Test&set Object

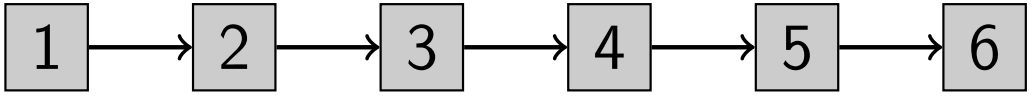
Each invocation returns **win** or **loose**

- *Exactly one proc.* obtains **win**
- no deterministic implementation with registers

Simple Renaming from Test&set



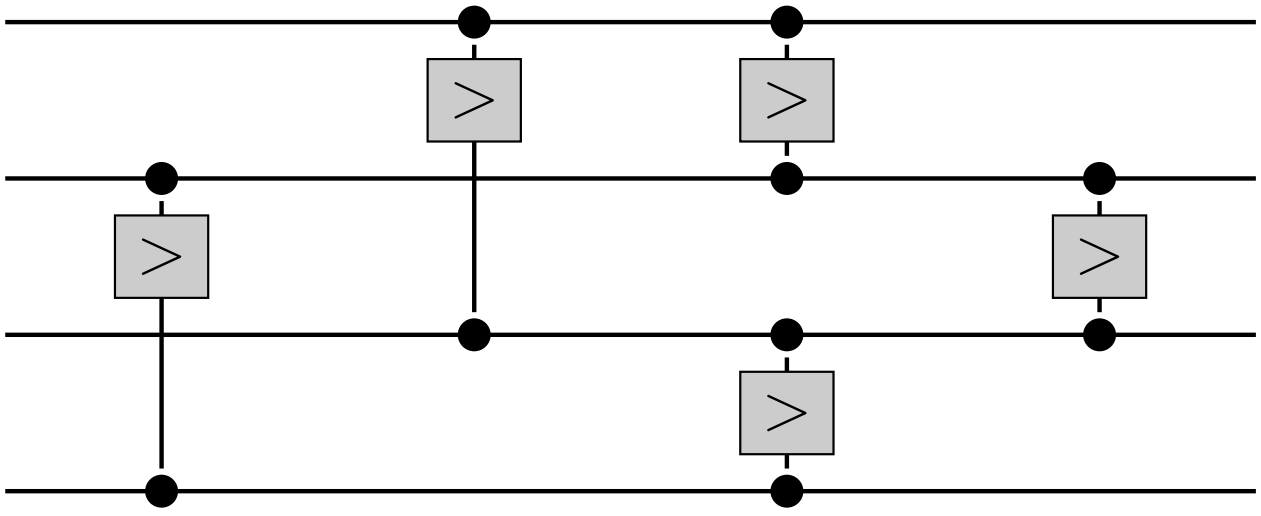
Simple Renaming from Test&set



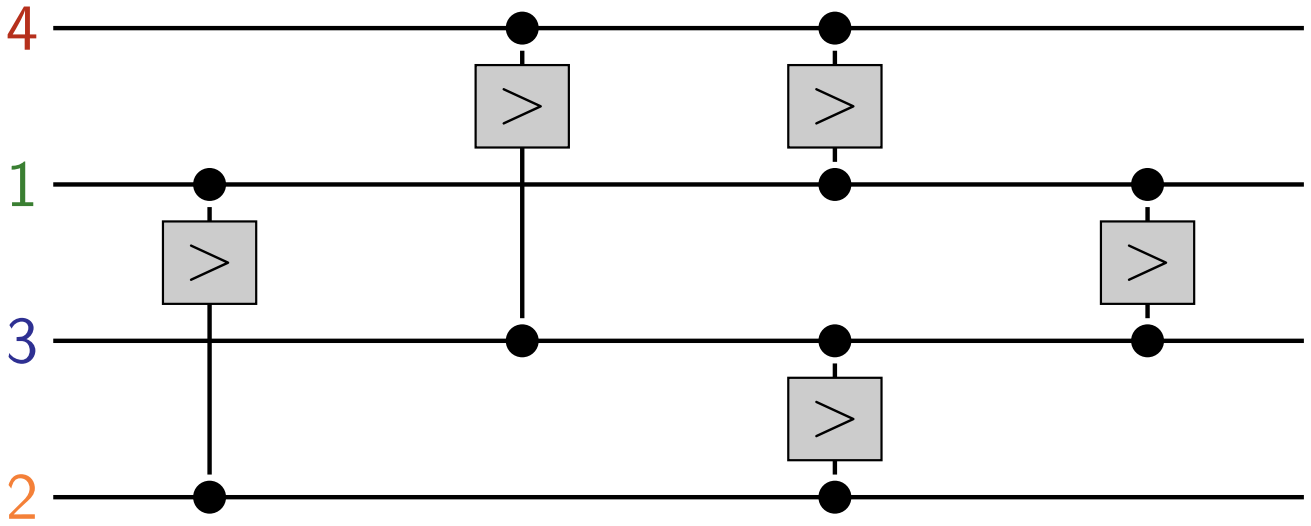
If k processes participate:

namespace	work per proc.	total work
k	$O(k)$	$O(k^2)$

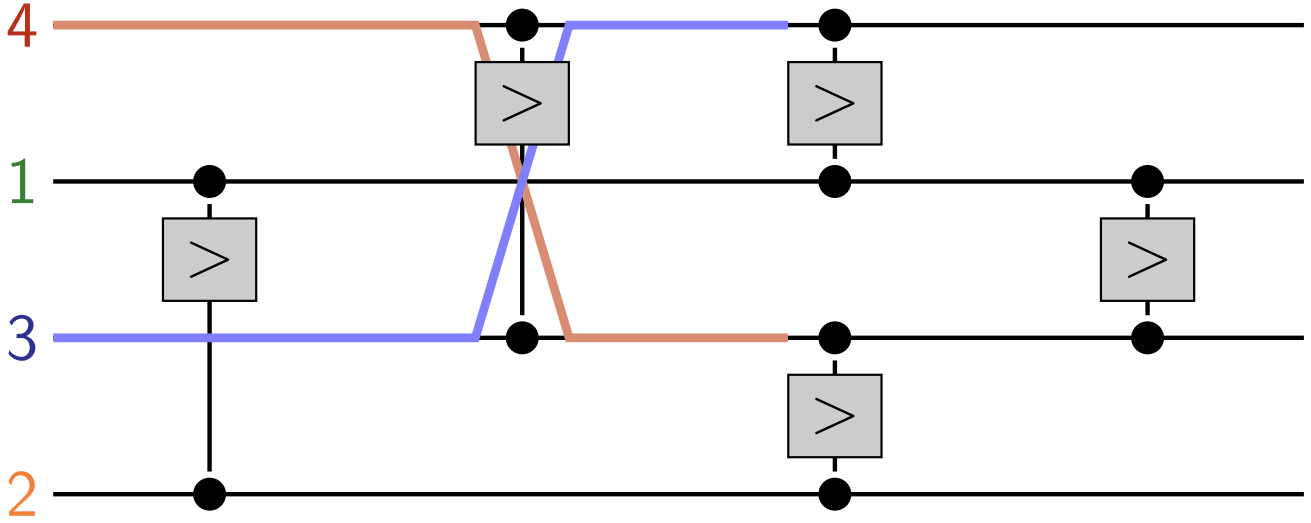
Sorting Networks



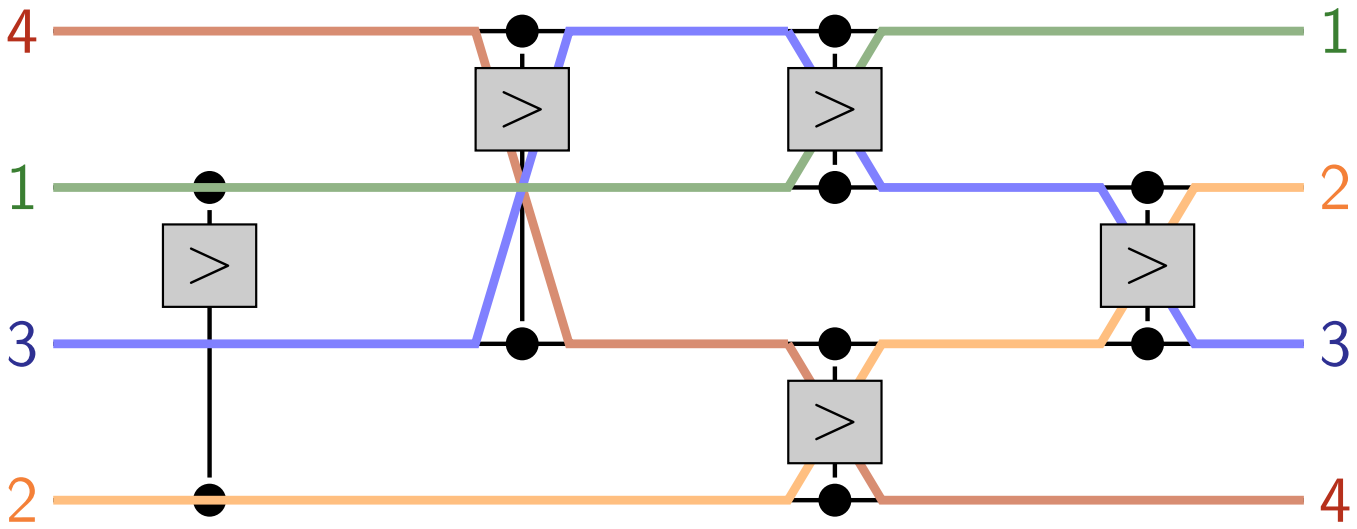
Sorting Networks



Sorting Networks

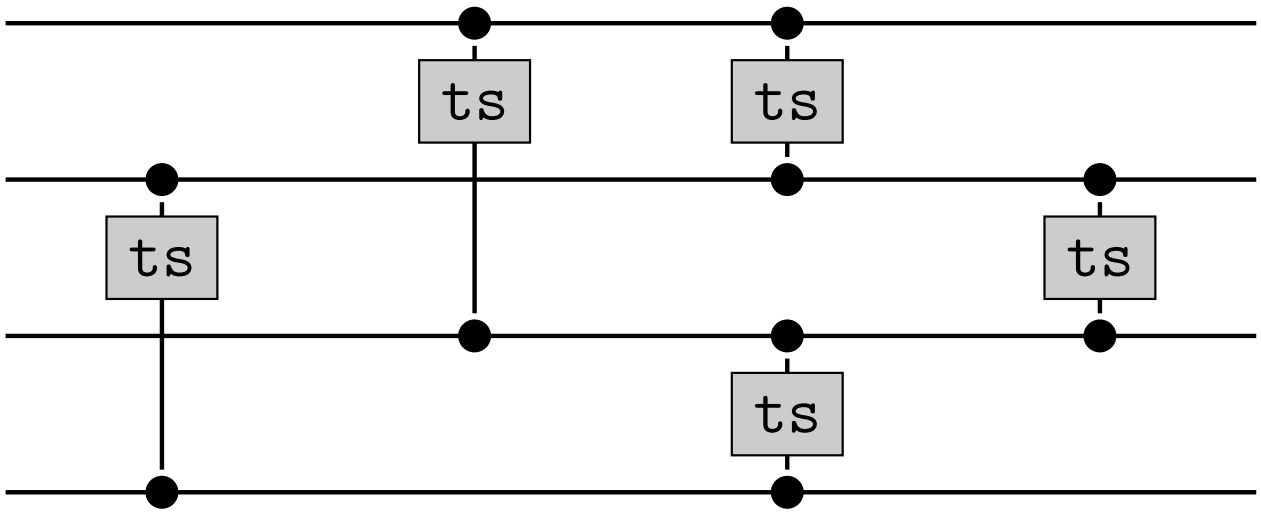


Sorting Networks



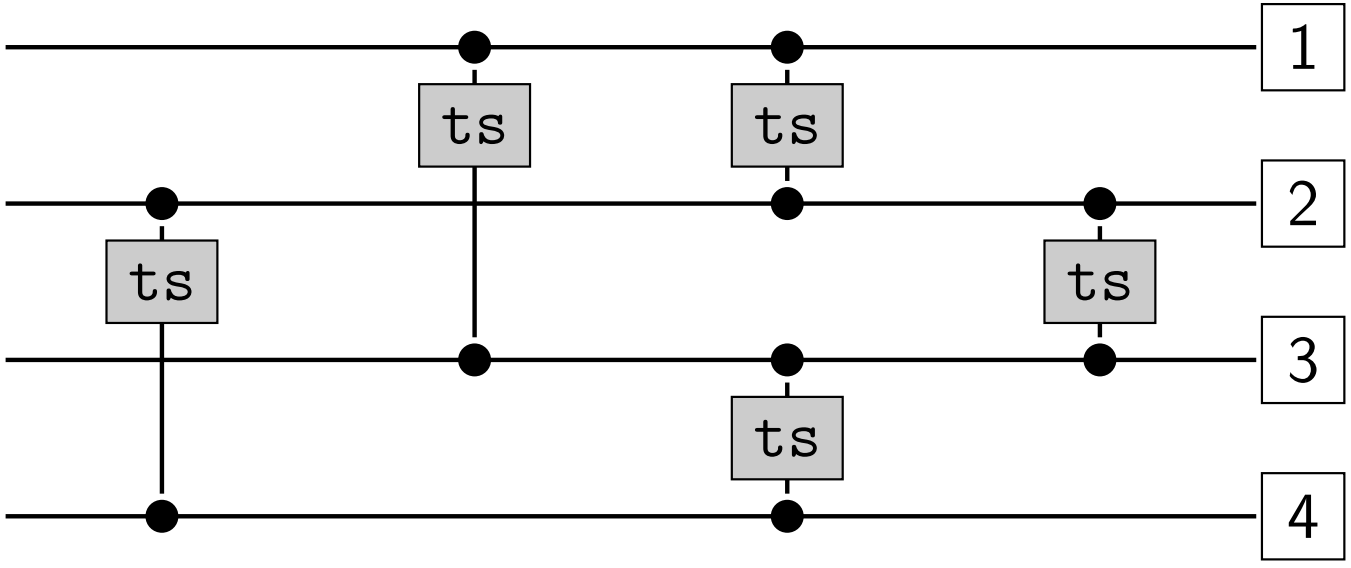
From Sorting to Renaming

[Alistarh et. al AACH+11]



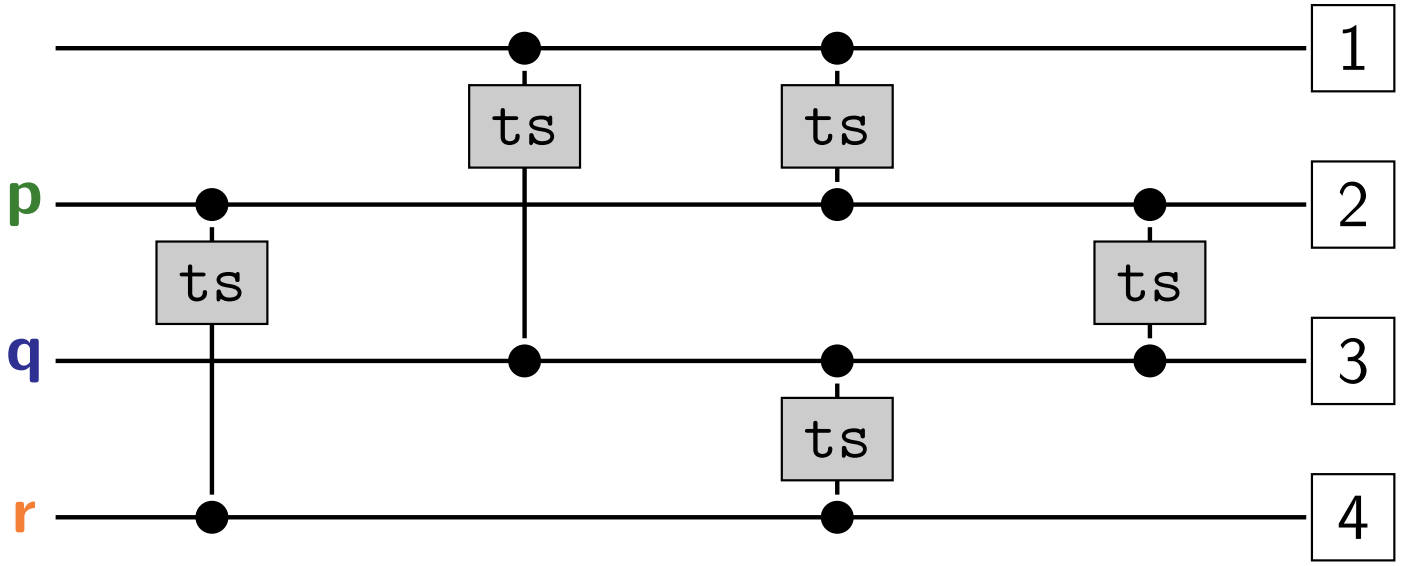
From Sorting to Renaming

[Alistarh et. al AACH+11]



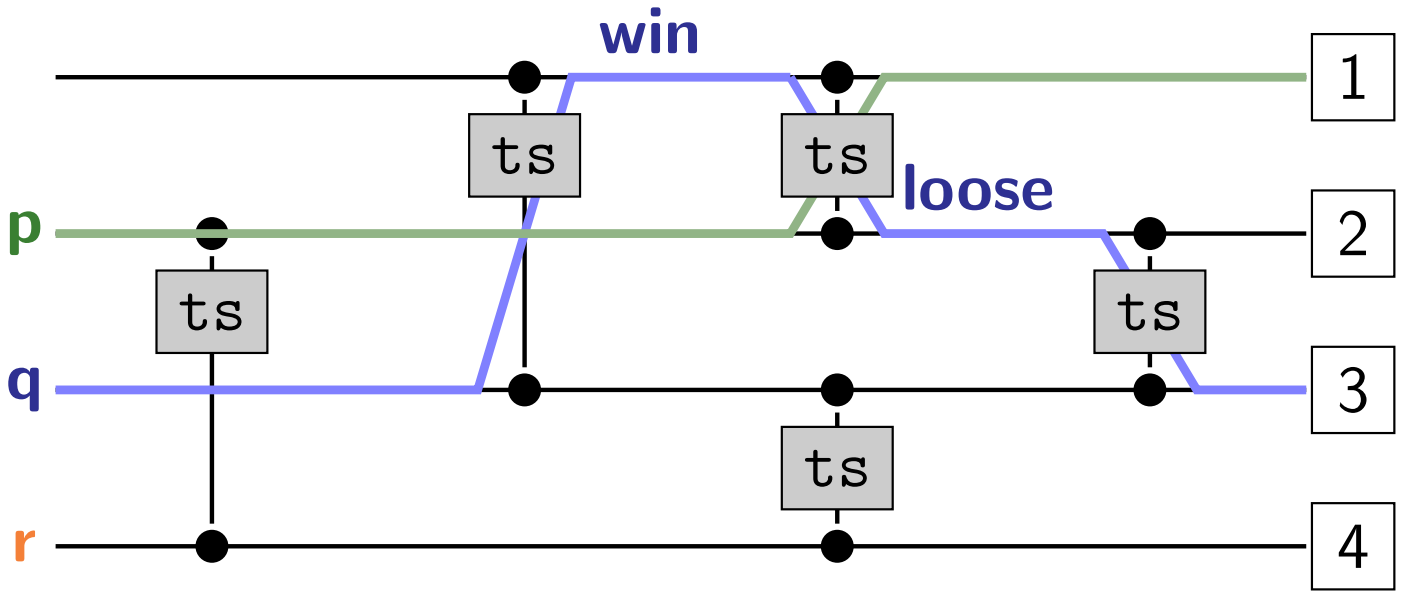
From Sorting to Renaming

[Alistarh et. al AACH+11]



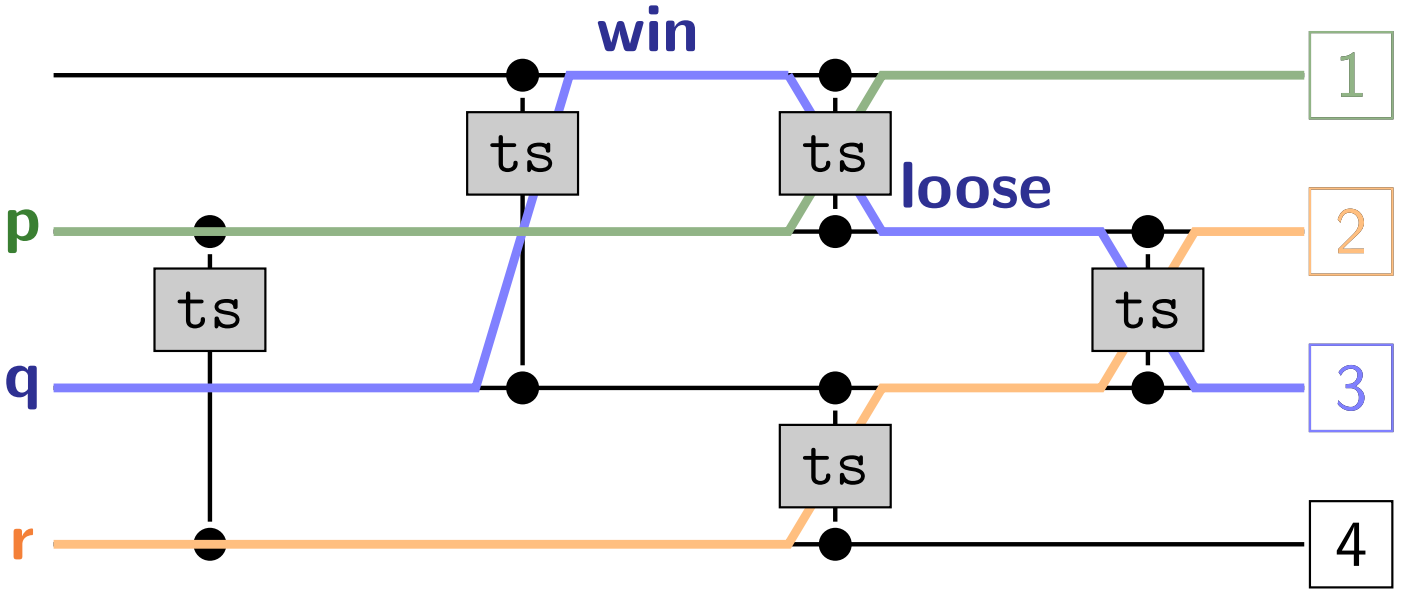
From Sorting to Renaming

[Alistarh et. al AACH+11]

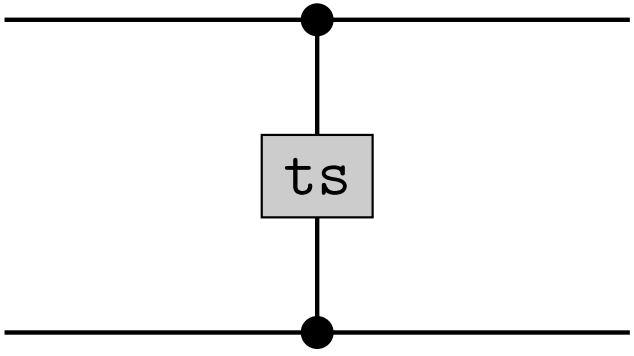


From Sorting to Renaming

[Alistarh et. al AACH+11]

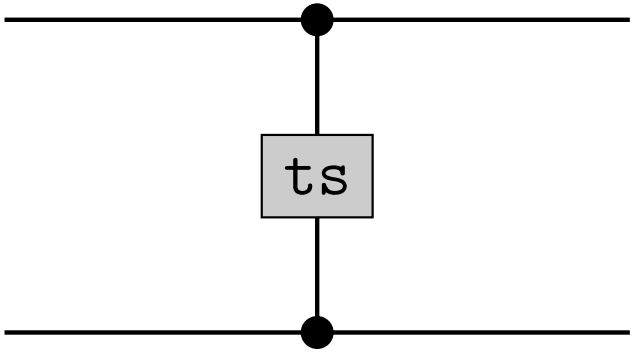


From Sorting to Renaming



At most 2 processes access each Test&Set

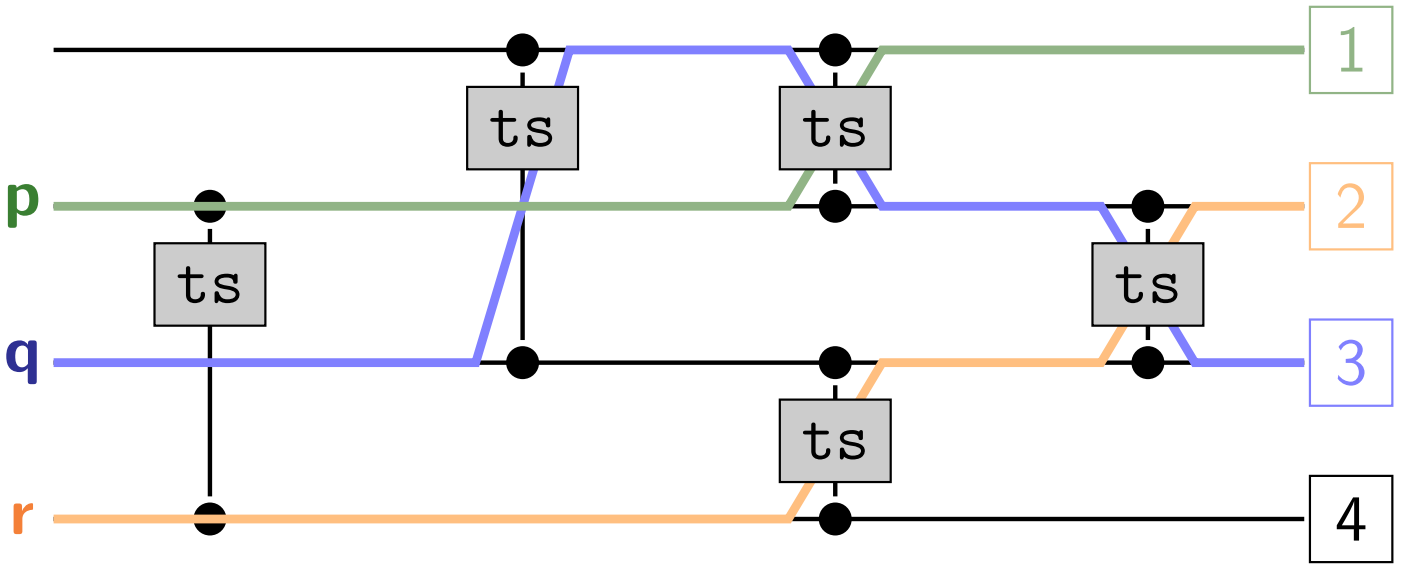
From Sorting to Renaming



At most 2 processes access each Test&Set

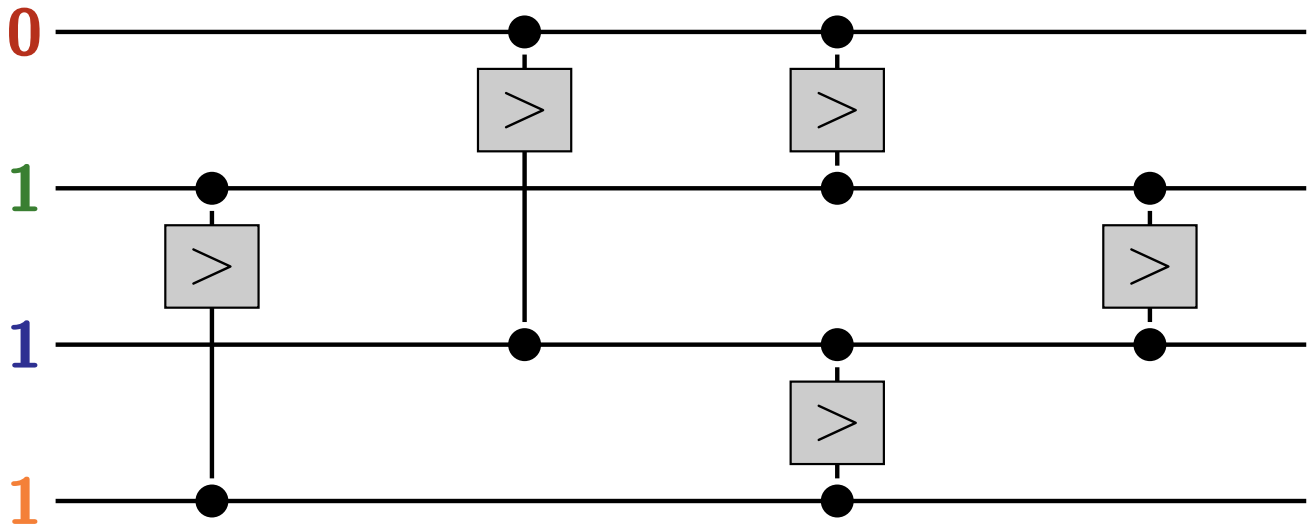
2 proc. Test&Set randomized implementation [Tromp Vitanyi]
$O(1)$ expected work

Namespace



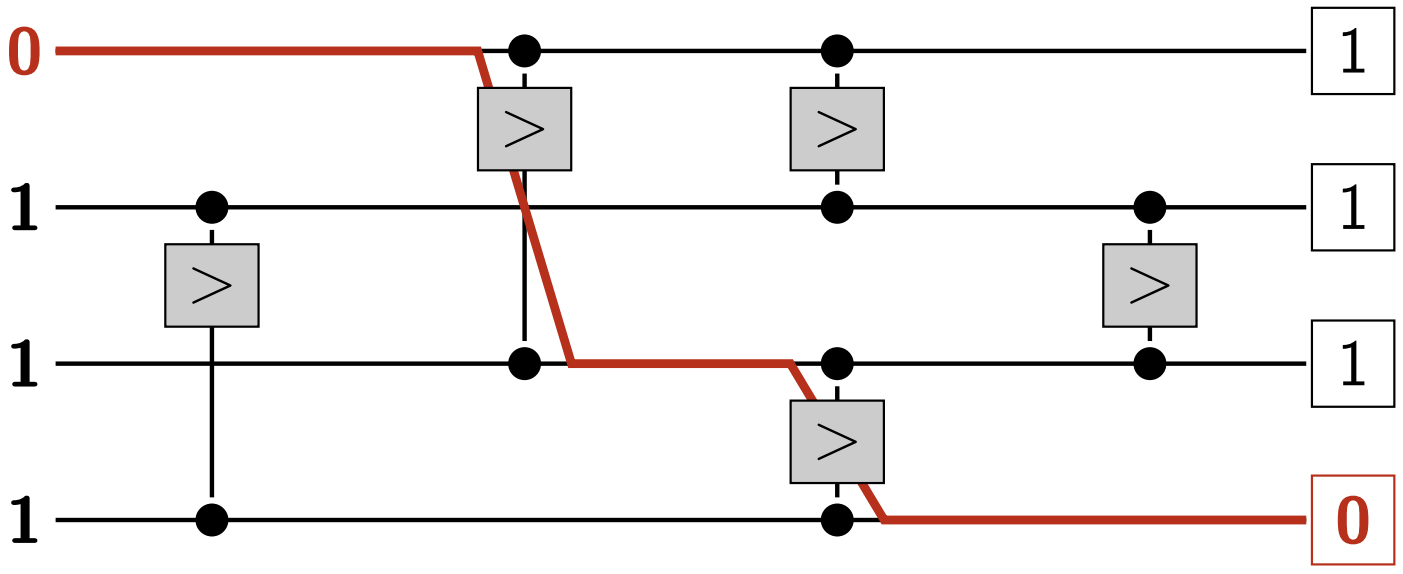
- k procs participate:
  exit by the *first* k output wires
- namespace $= [1..k]$

Namespace



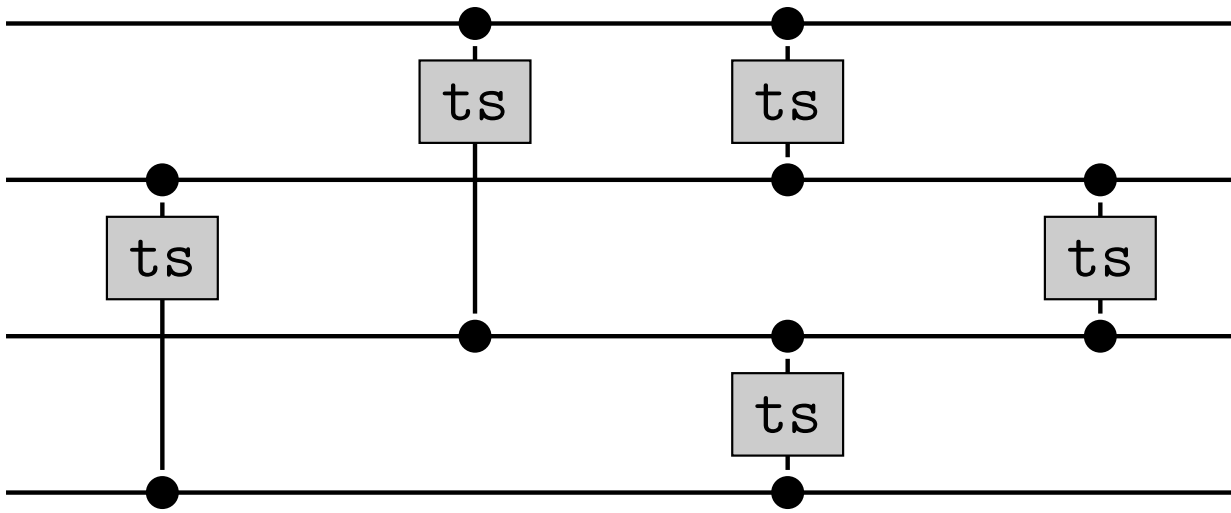
- k procs participate:
  exit by the *first* k output wires
- namespace $= [1..k]$

Namespace

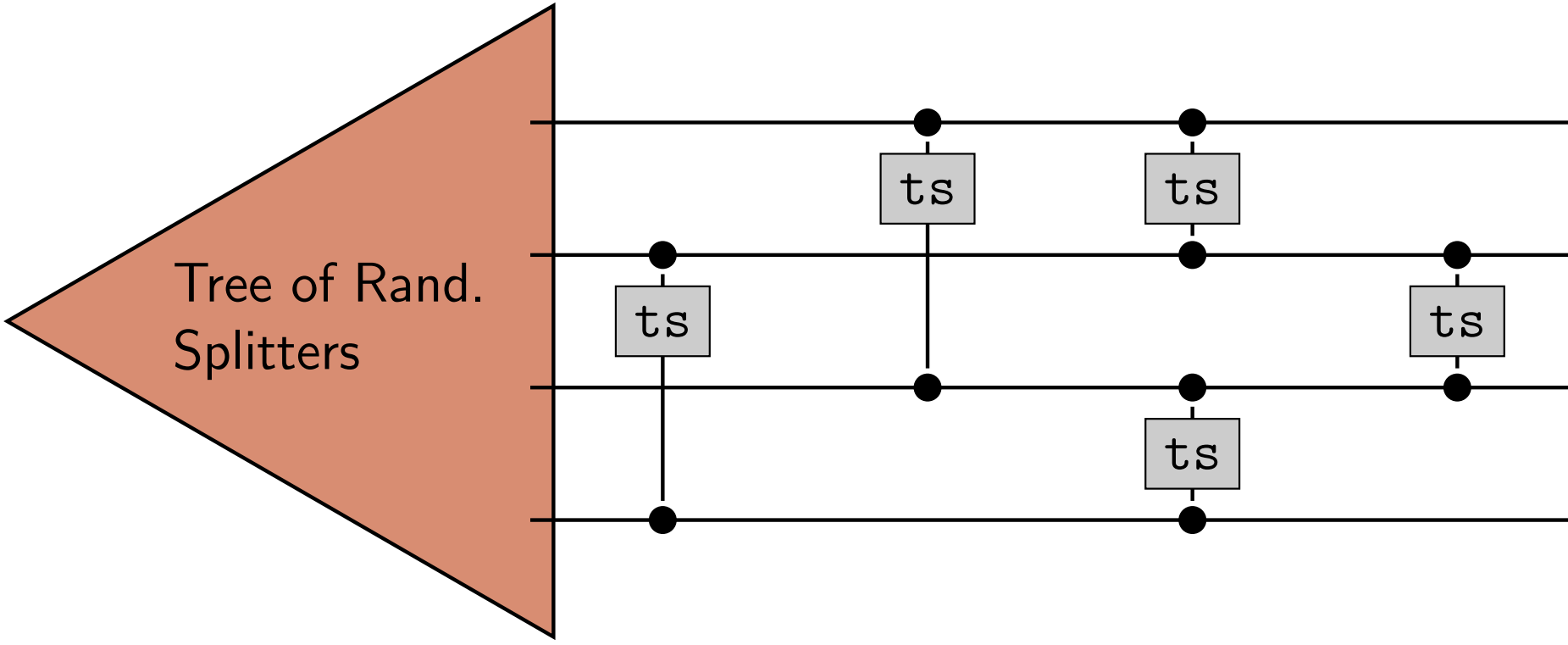


- k procs participate:
  exit by the *first* k output wires
- namespace $= [1..k]$

Assigning Input Wires



Assigning Input Wires



Complexity

- Namespace $= [1..k]$
- (Expected) individual work $=$ O(depth of the sorting network)
  - $O(\log n)$ [AKS] (implicit)
  - $O(\log^2 n)$ Bitonic sort [Batcher] (explicit)

Complexity can be made adaptive to # participating procs.

Deterministic Wait-free R/W Algorithms

Namespace

- Adaptive renaming in $[1..2k-2]$ impossible
- Non-adaptive renaming in $[1..2n-2]$ impossible
  iff n is a prime power [Castañeda Rajsbaum, Attiya Paz]

Deterministic Wait-free R/W Algorithms

Namespace

- Adaptive renaming in $[1..2k-2]$ impossible
- Non-adaptive renaming in $[1..2n-2]$ impossible
  iff n is a prime power [Castañeda Rajsbaum, Attiya Paz]

Namespace	Complexity	
$O(k^2)$	$O(k)$	Moir-Anderson splitter network
$2k-1$	$O(k^2)$	Afek Merrit
$6k-1$	$O(k \log k)$	Attiya Fouren
$8k - \log k - 1$	$O(k)$	Chlebus Kowalski (implicit)

Deterministic Wait-free R/W Algorithms

Namespace

- Adaptive renaming in $[1..2k-2]$ impossible
- Non-adaptive renaming in $[1..2n-2]$ impossible
  iff n is a prime power [Castañeda Rajsbaum, Attiya Paz]

Namespace	Complexity	
$O(k^2)$	$O(k)$	Moir-Anderson splitter network
$2k-1$	$O(k^2)$	Afek Merrit
$6k-1$	$O(k \log k)$	Attiya Fouren
$8k - \log k - 1$	$O(k)$	Chlebus Kowalski (implicit)

Lower bound $\Omega(k)$ for adaptive algorithm

Randomized Renaming

- Expected individual/total work
- And/or name uniqueness w.h.p

Randomized Renaming

- Expected individual/total work
- And/or name uniqueness w.h.p

Namespace	Complexity	
k	$O(\log k)$	Alistarh et. al (implicit [AKS])
k	$O(\log^2 k)$	Alistarh et. al (Bitonic sort)
$(1+\epsilon)k$	$O(\log \log k)$	Alistarh et. al (relax name uniqueness)

Randomized Renaming

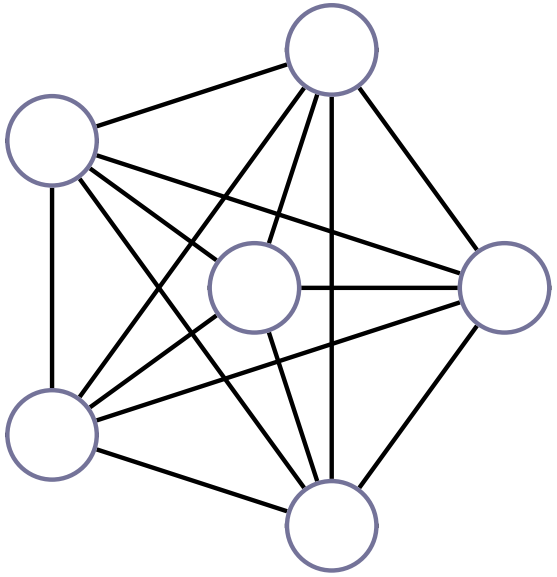
- Expected individual/total work
- And/or name uniqueness w.h.p

Namespace	Complexity	
k	$O(\log k)$	Alistarh et. al (implicit [AKS])
k	$O(\log^2 k)$	Alistarh et. al (Bitonic sort)
$(1+\epsilon)k$	$O(\log \log k)$	Alistarh et. al (relax name uniqueness)

weak Lower bound $\Omega(\log \log k)$ work for at least one processes

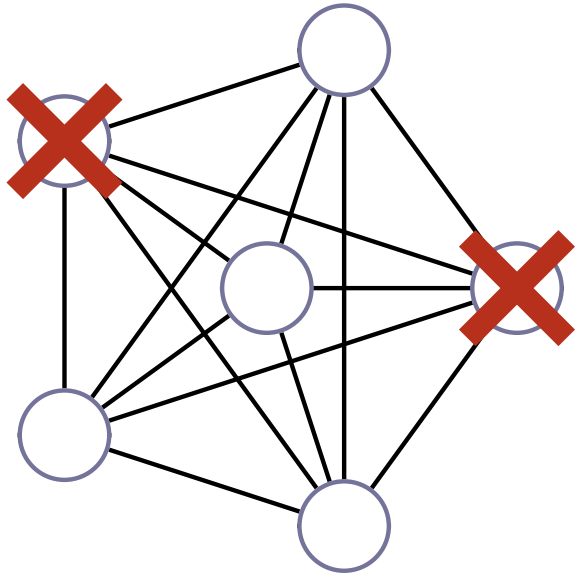
Message Passing

- Complete Network
- **Failures** : Crash
- Asynchronous



Message Passing

- Complete Network
- **Failures** : Crash
- Asynchronous



Renaming in Message Passing

Synchronous		Partially Synchronous	Asynchronous
$f < \sqrt{n}$	$f < n$		$f < n/2$
cst	$\log f$		$exp(n)$
[AAGT]			[ABDPR]

Renaming in Message Passing

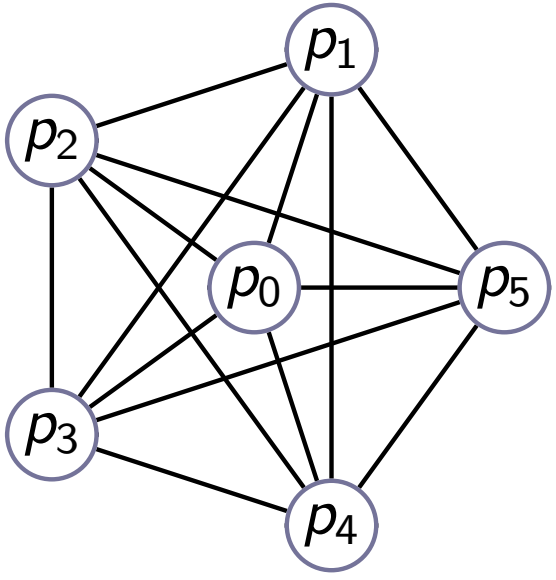
Synchronous		Partially Synchronous	Asynchronous
$f < \sqrt{n}$	$f < n$		$f < n/2$
cst	$\log f$	???	$exp(n)$
[AAGT]			[ABDPR]

Renaming in Message Passing

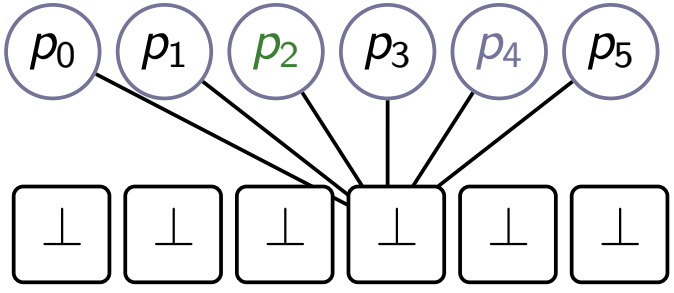
Synchronous		Partially Synchronous	Asynchronous
$f < \sqrt{n}$	$f < n$		$f < n/2$
cst	$\log f$	???	$exp(n)$
[AAGT]			[ABDPR]

- Byzantine failures?
- Messages adversaries?
- Dynamic networks?

Equivalence



$\approx$



registers can be simulated in message passing iff #crash $< \frac{n}{2}$
[ABD]

Filling the Gaps

- Descartes Abstraction Layers For Distributed Computing
  - Non-generalist simulations
  - Inherent cost of simulations
- FREDDA FoRmal mEthods for the Design of Distributed Algorithms
  - Robust algorithms: from synchronous to partially synchronous algorithms
    from benign to more severe failures

Thanks