How to Elect a Leader Faster than a Tournament

Dan Alistarh, Rati Gelashvili, Adrian Vladu

PODC 2015

Presenter: Lily Li

CSC 2221: Introduction to Distributed Algorithms

30 November 2017

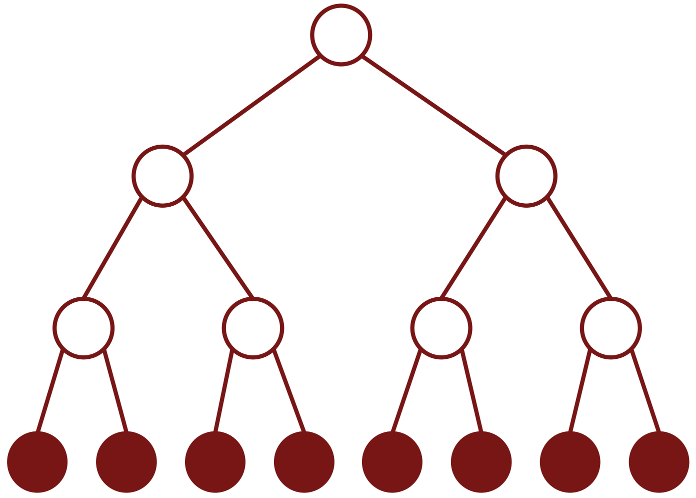
Alistarh, D., Gelashvili, R., & Vladu, A. (2015, July). How to elect a leader faster than a tournament. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing* (pp. 365-374). ACM.

Results New and Old

Tournament Tree

Step Complexity: $O(\log n)$

Message Complexity: $O(n^2 \log n)$

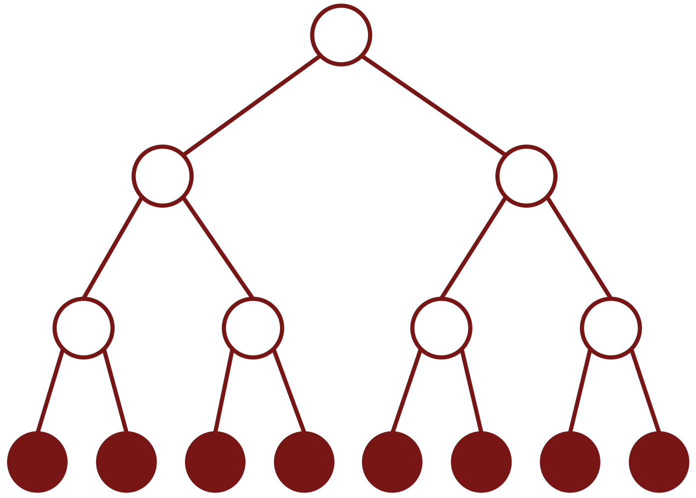


Results New and Old

Tournament Tree

Step Complexity: $O(\log n)$

Message Complexity: $O(n^2 \log n)$



Poison Pill

Step Complexity: $O(\log^* n)$

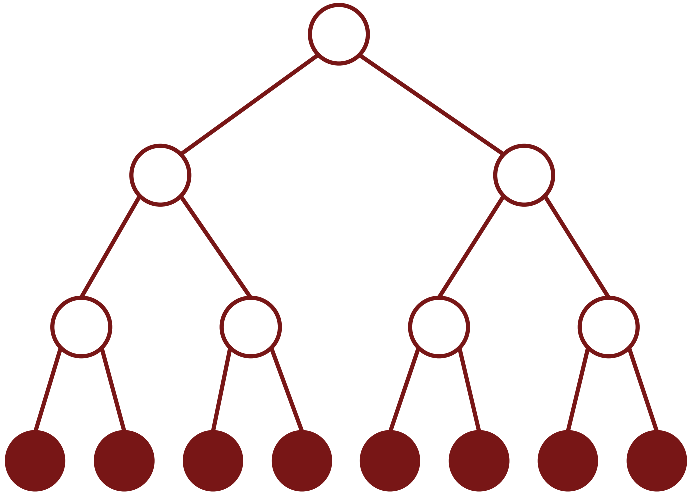
Message Complexity: $O(n^2)$

Results New and Old

Tournament Tree

Step Complexity: $O(\log n)$

Message Complexity: $O(n^2 \log n)$



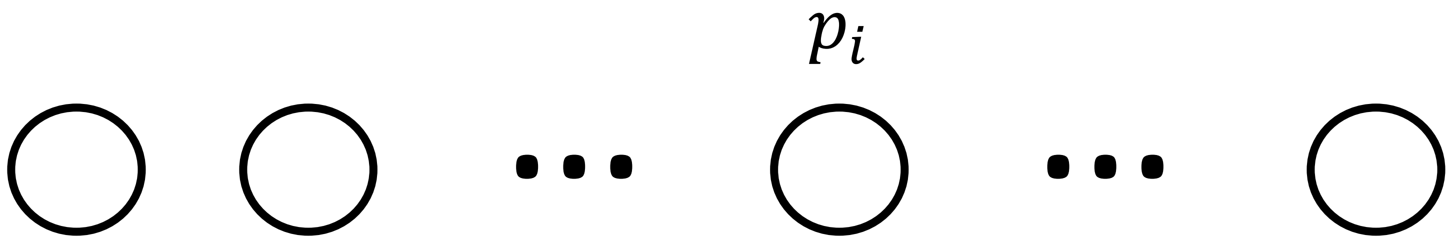
Poison Pill

Step Complexity: $O(\log^* n)$

Message Complexity: $O(n^2)$

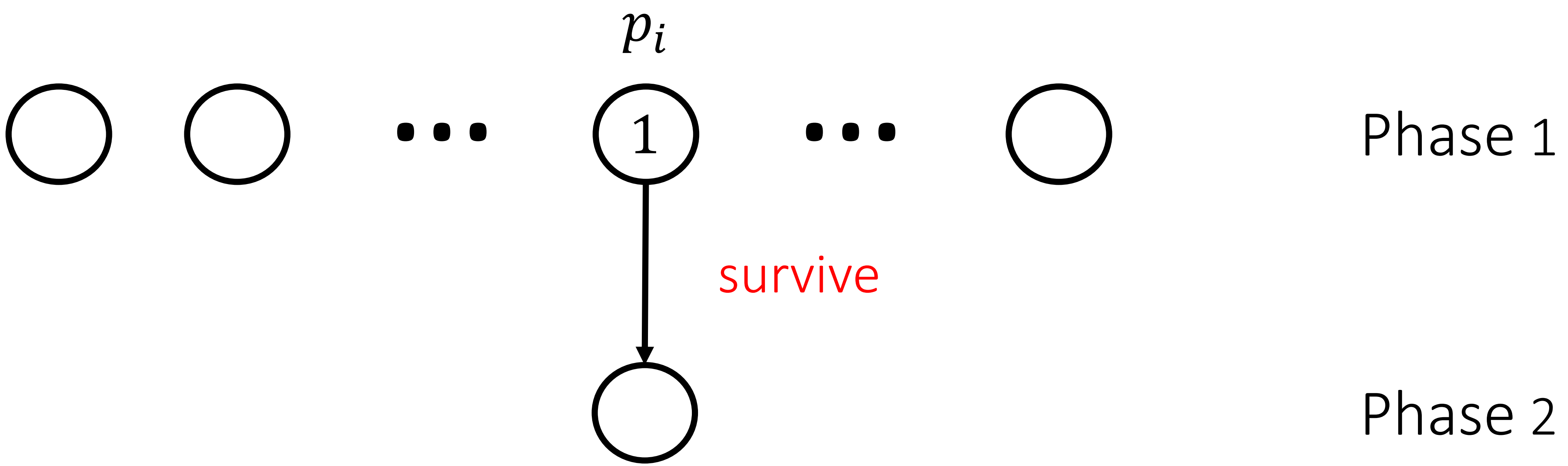
Lower bound: $\Omega(n^2)$

Algorithm Overview

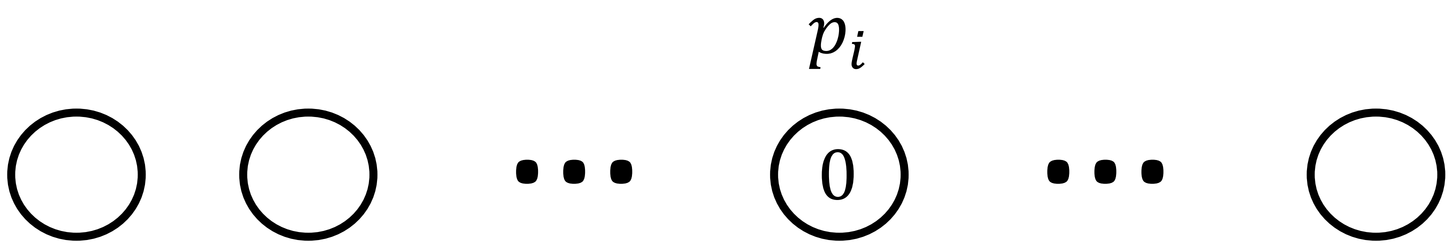


Phase 1

Algorithm Overview

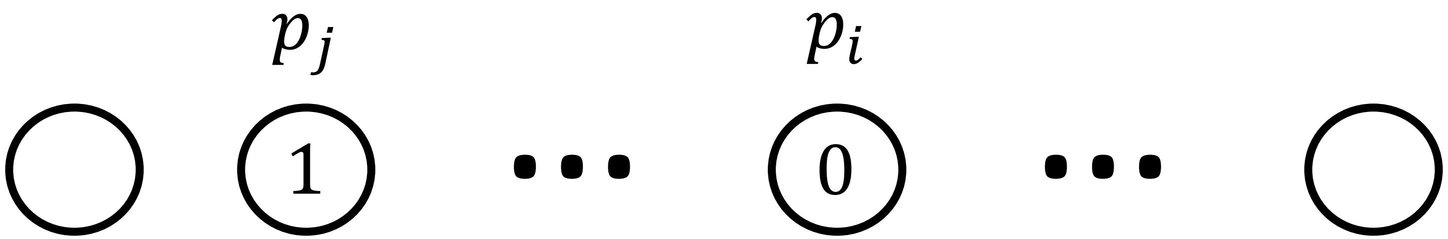


Algorithm Overview



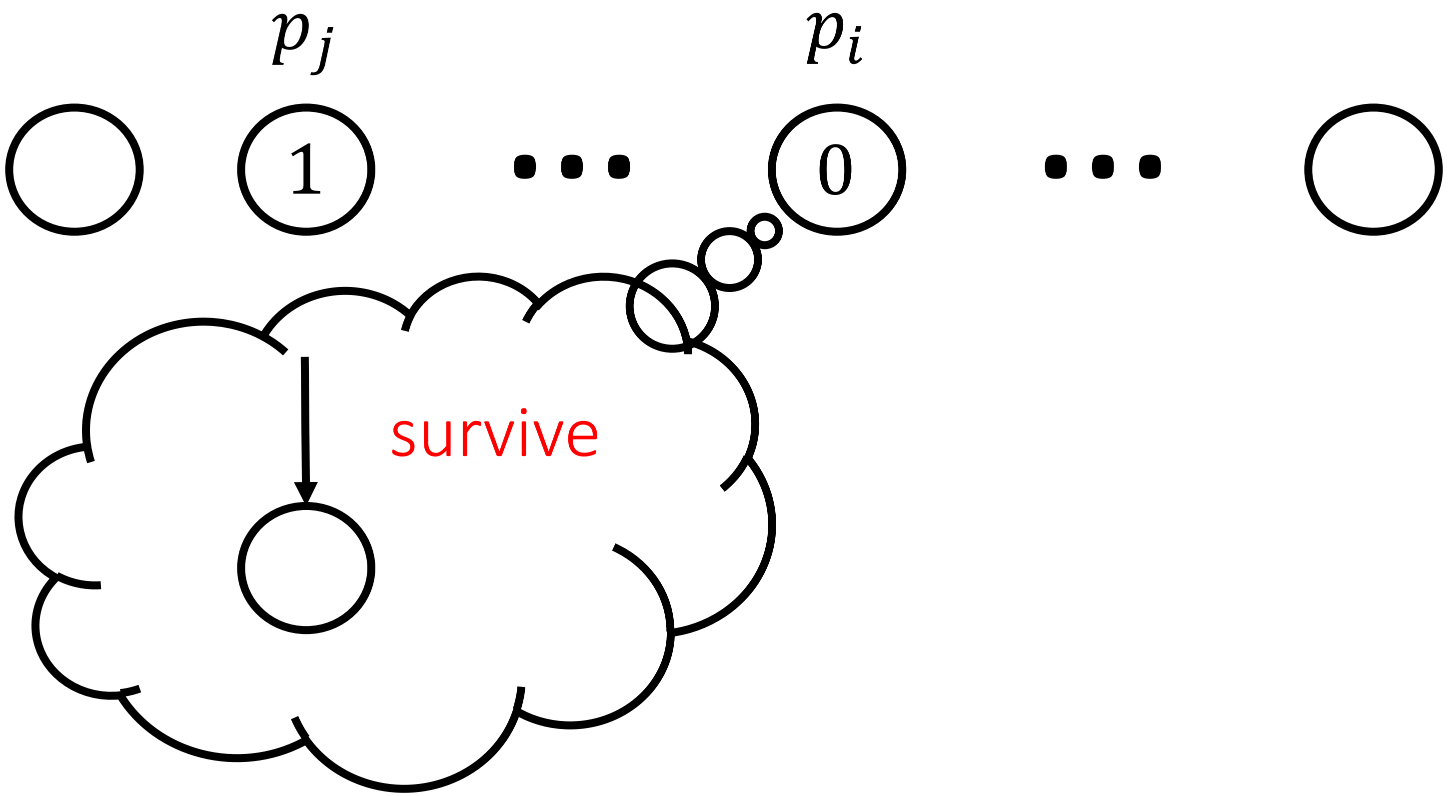
Phase 1

Algorithm Overview



Phase 1

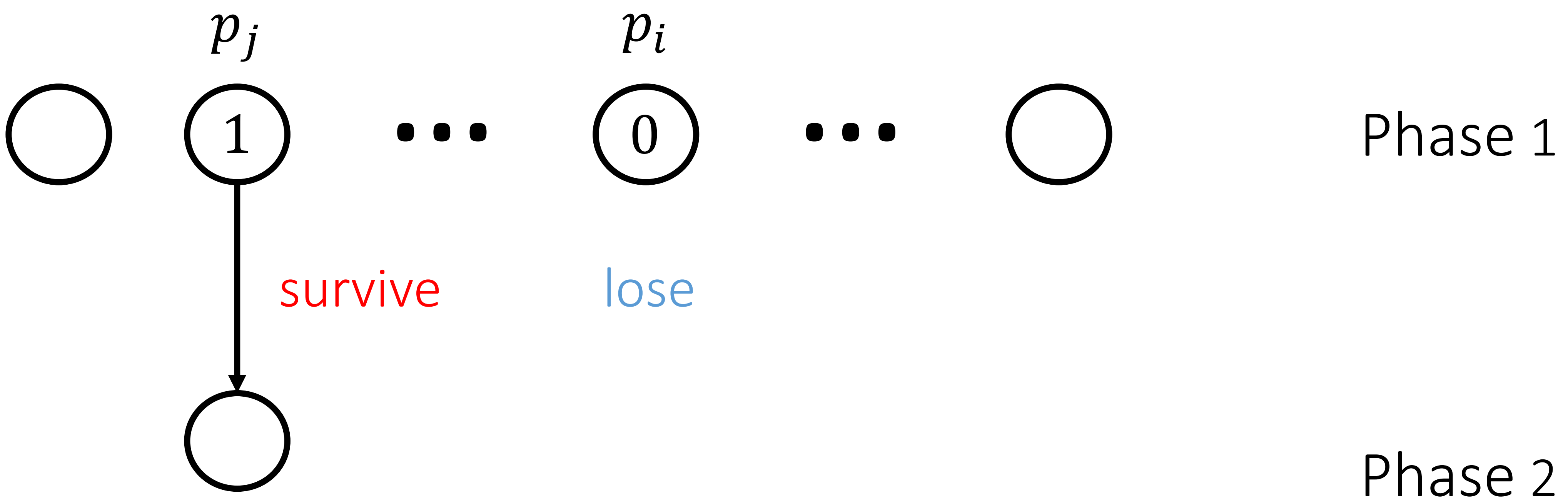
Algorithm Overview



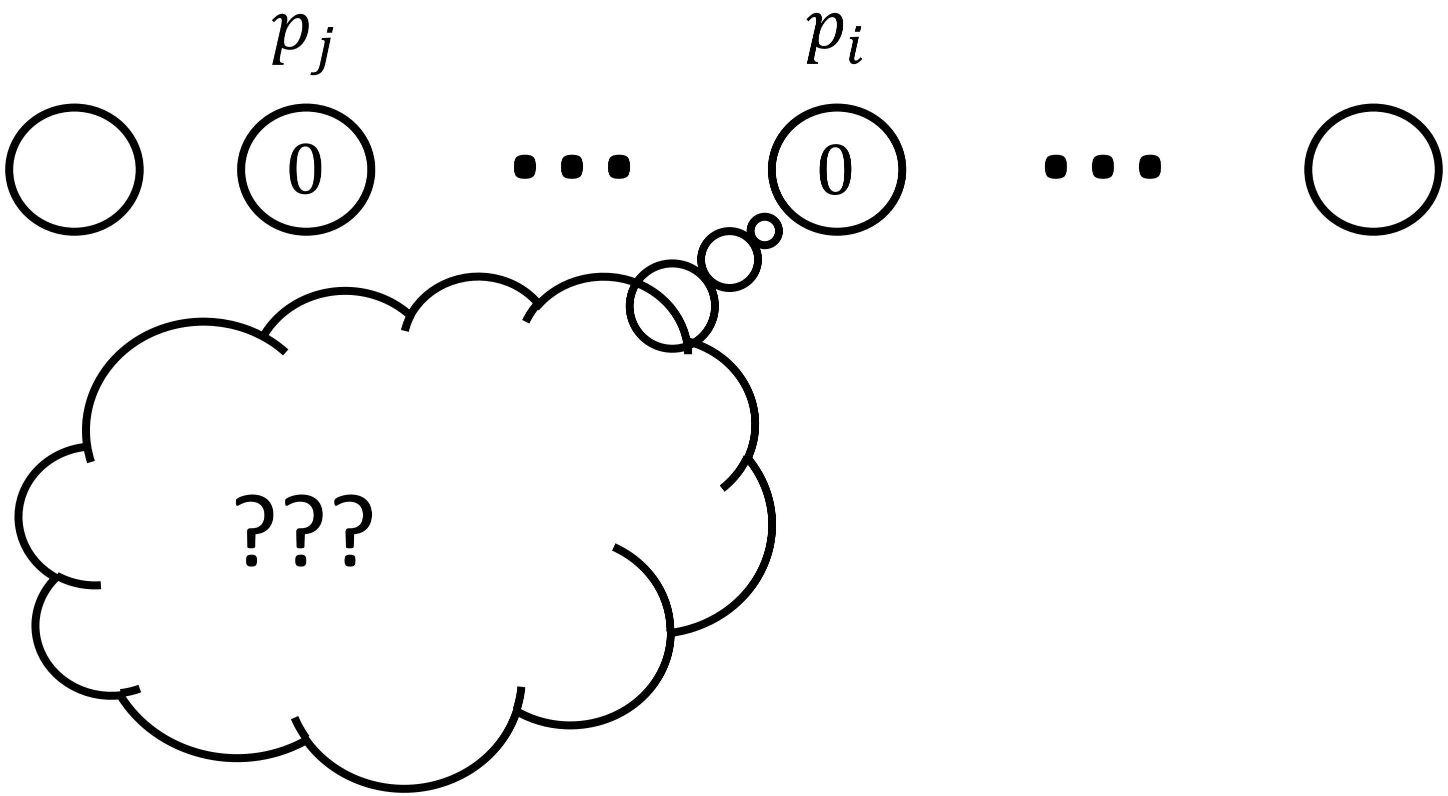
Phase 1

Phase 2

Algorithm Overview



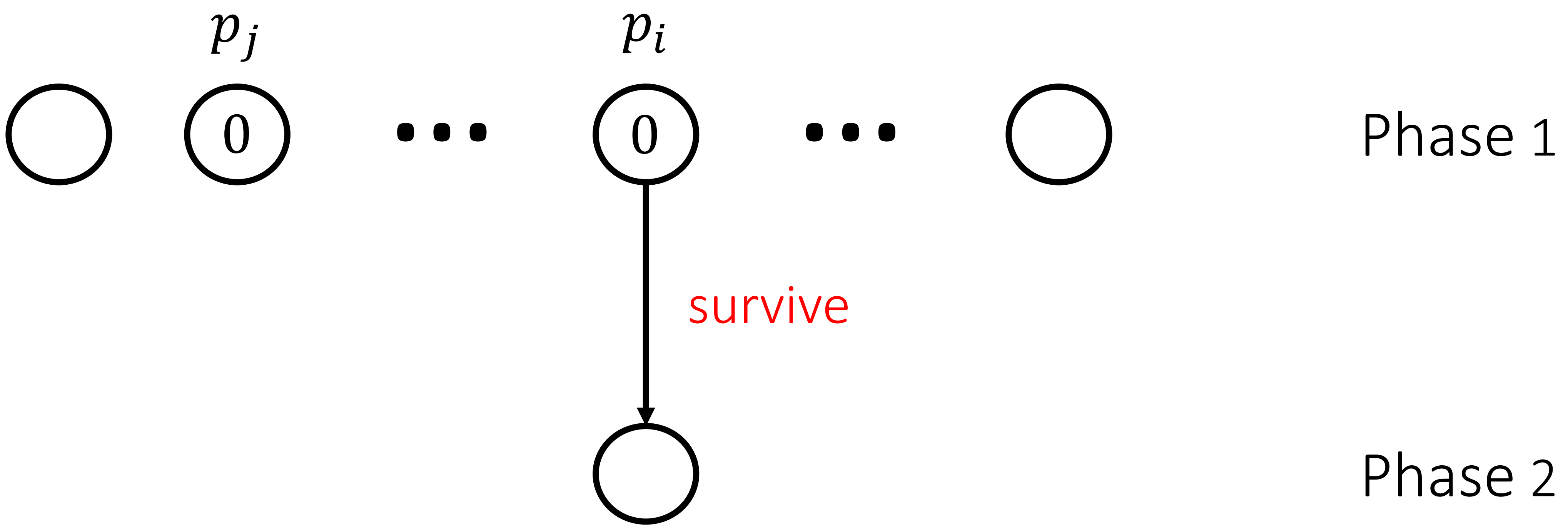
Algorithm Overview



Phase 1

Phase 2

Algorithm Overview



Local Variables and Communication Primitives

Local variables of process p_j:

- S_i: records the states of all observed processes
- V_i: view stores the state vectors received from other processes

Local Variables and Communication Primitives

Local variables of process p_j:

- S_i: records the states of all observed processes
- V_i: view stores the state vectors received from other processes

High-Level Protocol **`COMMUNICATE<`**`p`**`>`** (where `p` is a procedure which broadcasts a message and wait for a response):

Executes `p` and waits for $\left\lceil \frac{n}{2} \right\rceil + 1$ processes to respond before proceeding.

`p` can be `broadcast(v)` or `collect( )` where

1. If p_j receives `broadcast(v)` from p_i, store $S_j[i] \leftarrow v$ and send p_i $\langle ack \rangle$.
2. If p_j receives `collect( )` from p_i, send p_i $\langle S_j \rangle$.

Local Variables and Communication Primitives

Local variables of process p_j:

- S_i: records the states of all observed processes
- V_i: view stores the state vectors received from other processes

High-Level Protocol **`COMMUNICATE<p>`** (where `p` is a procedure which broadcasts a message and wait for a response):

Executes `p` and waits for $\left\lceil \frac{n}{2} \right\rceil + 1$ processes to respond before proceeding.

`p` can be `broadcast(v)` or `collect( )` where

1. If p_j receives `broadcast(v)` from p_i, store $S_j[i] \leftarrow v$ and send p_i $\langle ack \rangle$.
2. If p_j receives `collect( )` from p_i, send p_i $\langle S_j \rangle$.

`random(r):` outputs 1 with probability `r` and 0 with probability `1-r`.

(Simple) Poison Pill

```
Algorithm 1. Code for process p_i during one phase.
Initialize: S_i = [w,...,w], V_i = EMPTY
    S_i[i] = c
    COMMUNICATE<broadcast(S_i[i])>
    S_i[i] = random(1/sqrt(n))
    COMMUNICATE<broadcast(S_i[i])>
    V_i = COMMUNICATE<collect()>
    if(S_i[i] = 0 and exists k: p_k is active)
        return 0
    return 1
```

(Simple) Poison Pill

```
Algorithm 1. Code for process p_i during one phase.
Initialize: S_i = [w,...,w], V_i = EMPTY
    S_i[i] = c
    COMMUNICATE<broadcast(S_i[i])>
    S_i[i] = random(1/sqrt(n))
    COMMUNICATE<broadcast(S_i[i])>
    V_i = COMMUNICATE<collect()>
    if(S_i[i] = 0 and exists k: p_k is active)
        return 0
    return 1
```

Claim 1. If all processes return then at least one will output 1.

(Simple) Poison Pill

```
Algorithm 1. Code for process p_i during one phase.
Initialize: S_i = [w,...,w], V_i = EMPTY
    S_i[i] = c
    COMMUNICATE<broadcast(S_i[i])>
    S_i[i] = random(1/sqrt(n))
    COMMUNICATE<broadcast(S_i[i])>
    V_i = COMMUNICATE<collect()>
    if(S_i[i] = 0 and exists k: p_k is active)
        return 0
    return 1
```

Claim 1. If all processes return then at least one will output 1.

Claim 2. The expected number of processes which output 1 is $O(\sqrt{n})$.

(Simple) Poison Pill

```
Algorithm 1. Code for process p_i during one phase.
Initialize: S_i = [w,...,w], V_i = EMPTY
    S_i[i] = c
    COMMUNICATE<broadcast(S_i[i])>
    S_i[i] = random(1/sqrt(n))
    COMMUNICATE<broadcast(S_i[i])>
    V_i = COMMUNICATE<collect()>
    if(S_i[i] = 0 and exists k: p_k is active)
        return 0
    return 1
```

Claim 1. If all processes return then at least one will output 1.

Claim 2. The expected number of processes which output 1 is $O(\sqrt{n})$.

Lemma 1. If a process receives 1 from `random` at time t, then all processes which received 0 from `random` at any time $\geq t$ will return 0.

…to Recap

- processes commit to an action before performing the action
- the probability of surviving each round changes dynamically

…to Recap

- processes commit to an action before performing the action
- the probability of surviving each round changes dynamically

…the Result

$\mathrm{O}(\log^* \mathrm{n})$ step complexity and
$\mathrm{O}(\mathrm{n}^2)$ message complexity.

…to Recap

- processes commit to an action before performing the action
- the probability of surviving each round changes dynamically

…the Result

$\mathrm{O}(\log^* \mathrm{n})$ step complexity and
$\mathrm{O}(\mathrm{n}^2)$ message complexity.

This randomized algorithm can be used to solve the tight renaming problem in $\mathrm{O}(\log^2 \mathrm{n})$ steps using $\mathrm{O}(\mathrm{n}^2)$ messages.