Silent Self-Stabilizing Scheme for Spanning-Tree-like Constructions

Stéphane Devismes[1] Colette Johnen[2] David Ilcinkas[2]

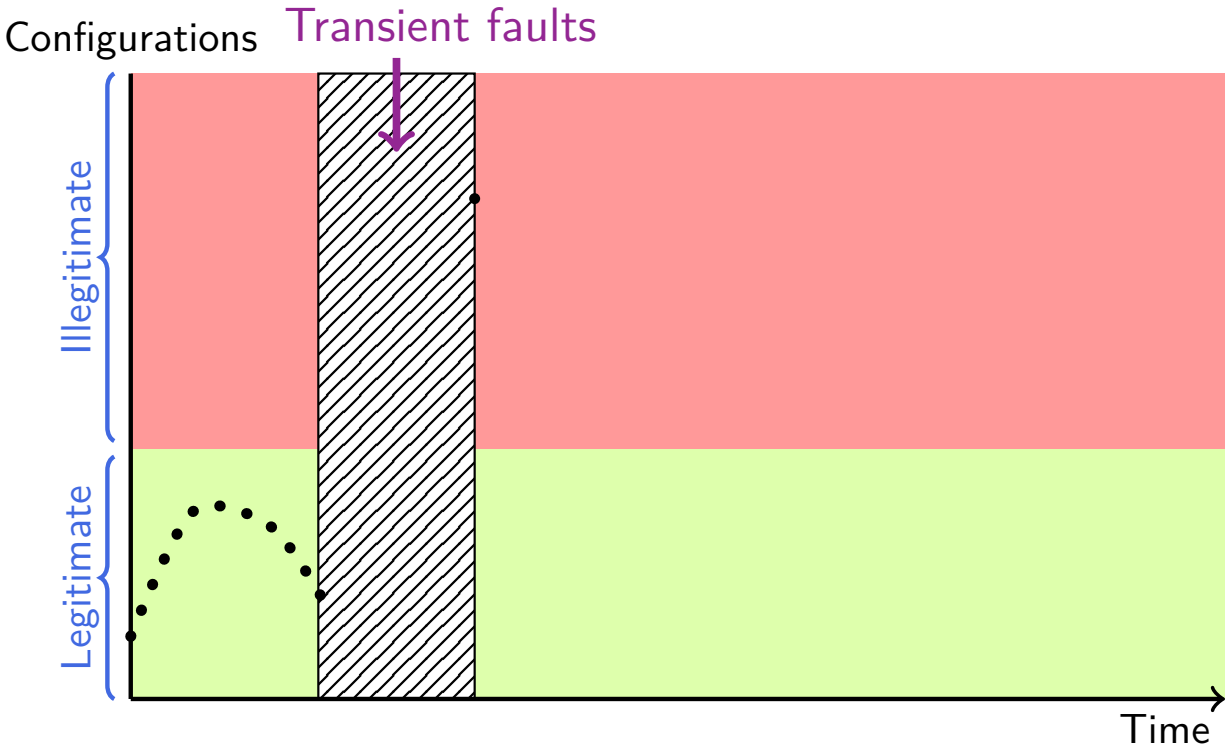
[1] Univ. Grenoble Alpes, VERIMAG, 38000 Grenoble, France
[2] CNRS & Univ. Bordeaux, LaBRI, UMR 5800, F-33400 Talence, France

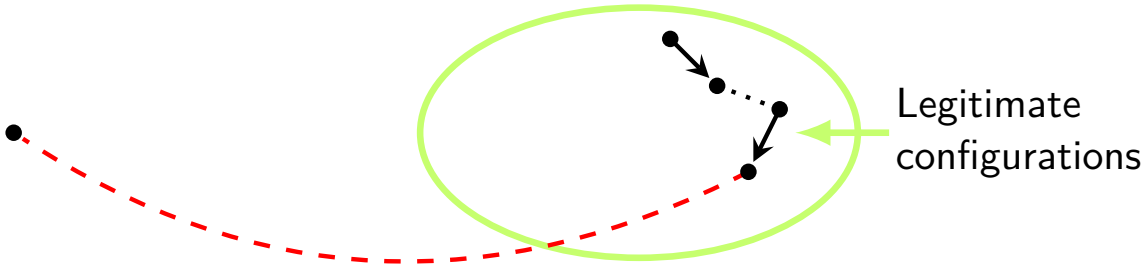
ICDCN, January 6th 2019, Bangalore (India)

Self-Stabilization

Self-stabilization

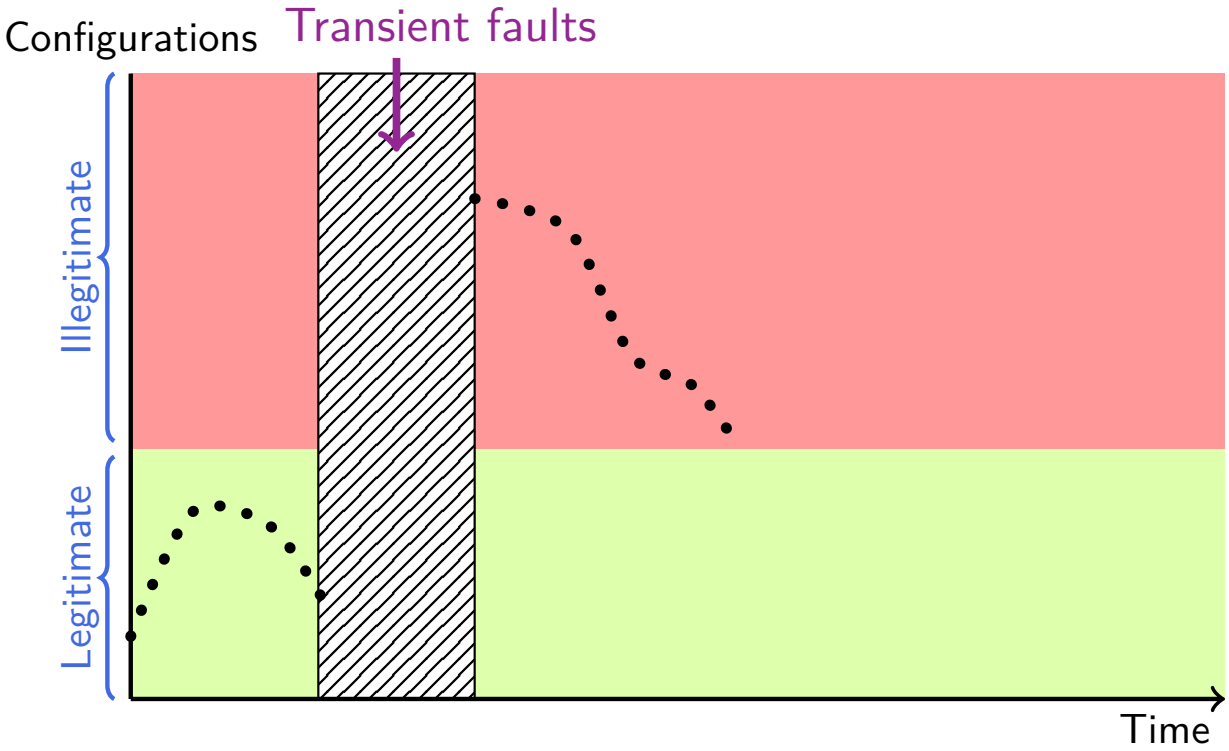
[Dijkstra, ACM Com., 74]

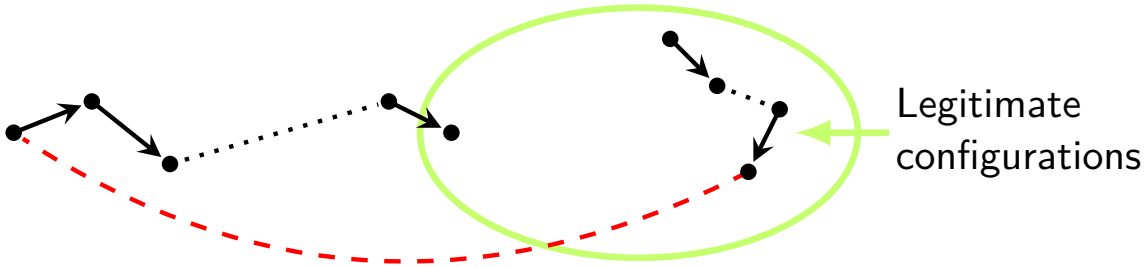




Self-stabilization

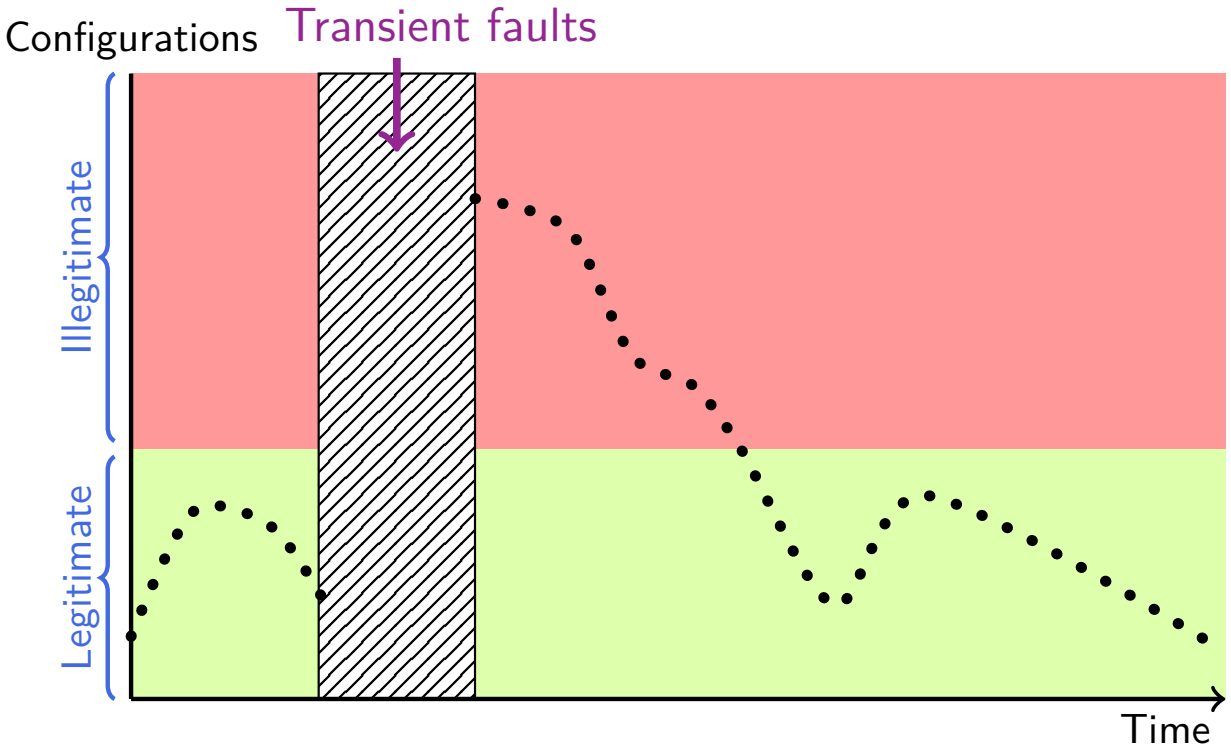
[Dijkstra, ACM Com., 74]

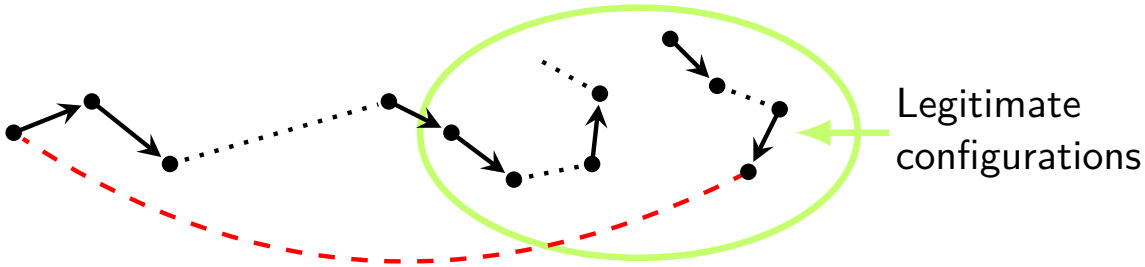




Self-stabilization

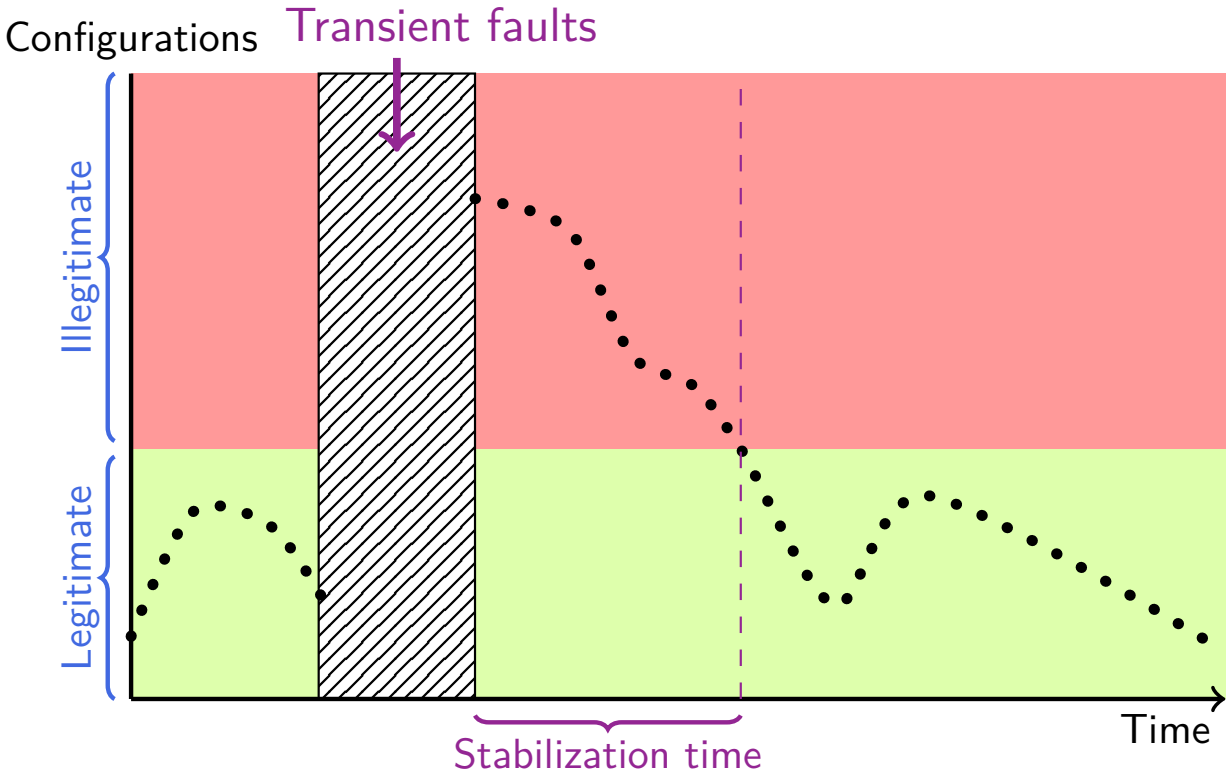
[Dijkstra, ACM Com., 74]

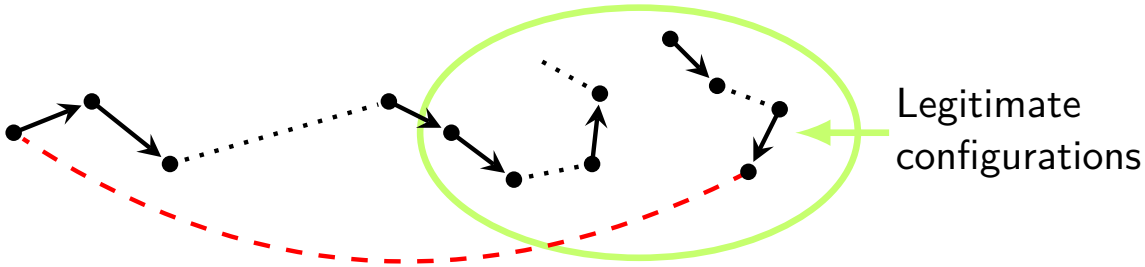




Self-stabilization

[Dijkstra, ACM Com., 74]

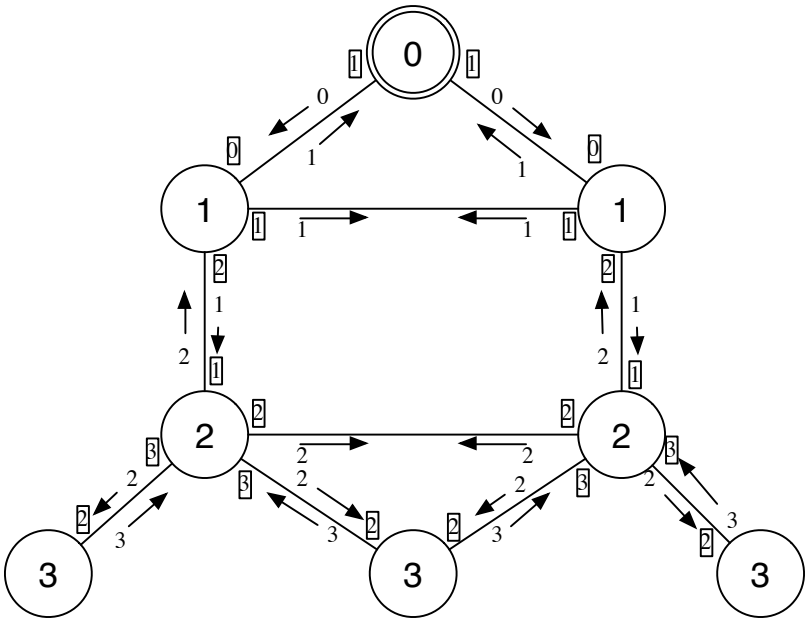




Silent Algorithm

[Dolev *et al.*, Acta Informatica, 96]

A silent self-stabilizing algorithm converges within finite time to a configuration from which the values of the registers used by the algorithm remain fixed.



Silent Algorithm

[Dolev *et al.*, Acta Informatica, 96]

A silent self-stabilizing algorithm converges within finite time to a configuration from which the values of the registers used by the algorithm remain fixed.

Advantages:

- Silence implies **more simplicity** in the algorithm design (classically used in compositions).
- A silent algorithm may utilize **less communication operations and communication bandwidth**.
- Well-suited to compute **distributed data structures** such as spanning trees.

Model

Locally Shared Memory Model with Composite Atomicity

Abstraction of the message-passing model

Locally Shared Memory Model with Composite Atomicity

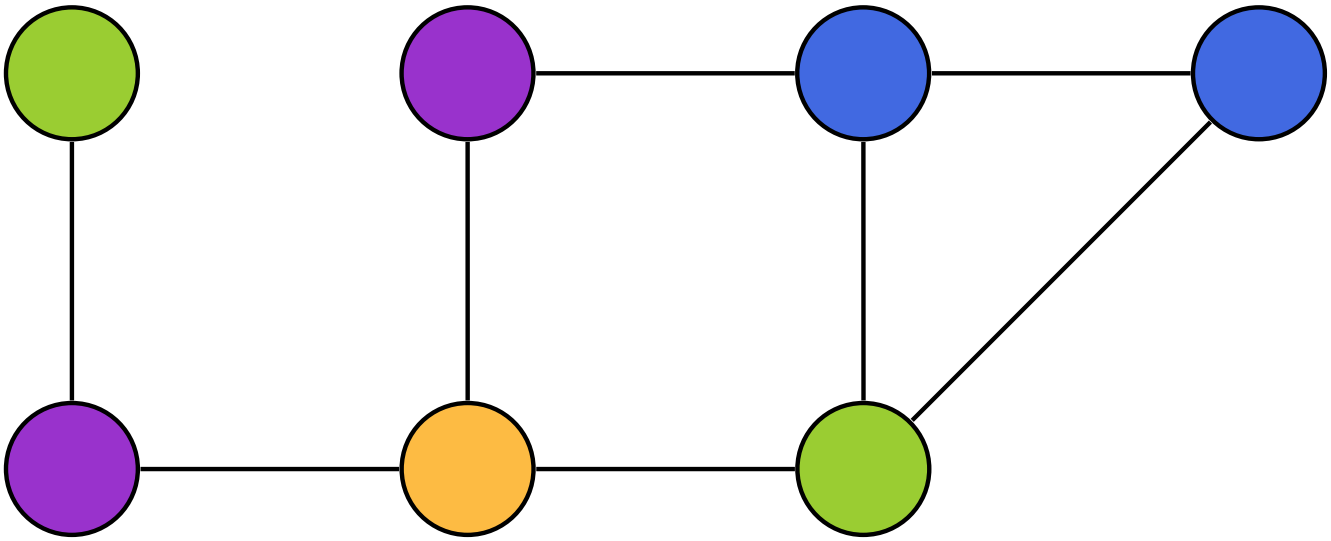
Abstraction of the message-passing model

Locally shared registers (variables) instead of communication links

A process can only read its variables and that of its neighbors.

Locally Shared Memory Model with Composite Atomicity

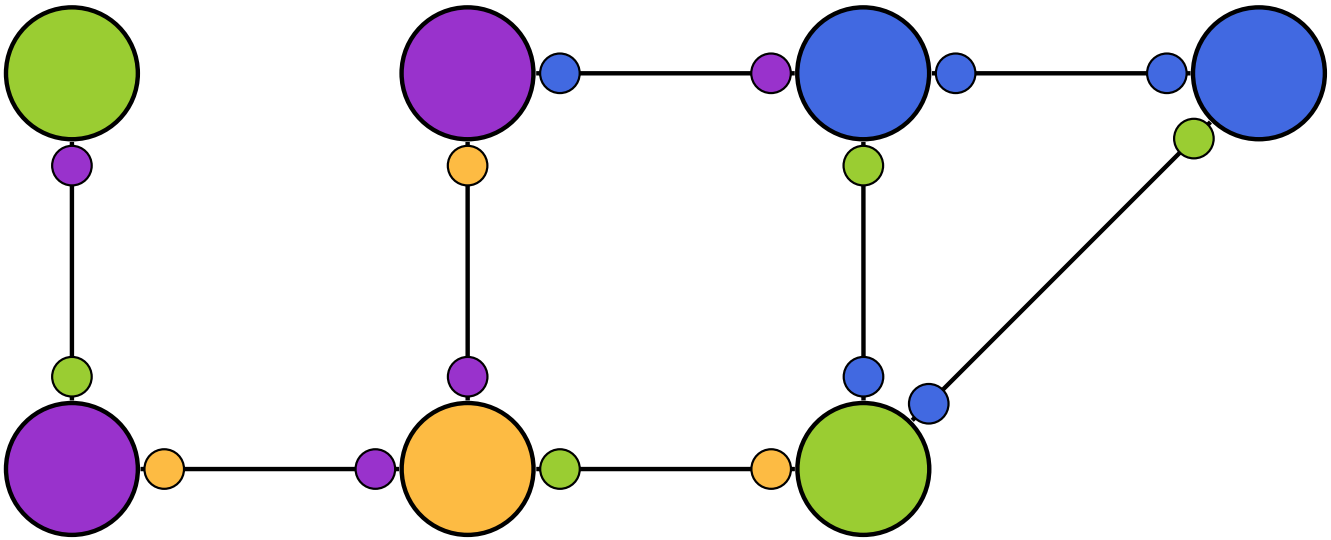
Configuration



Locally Shared Memory Model with Composite Atomicity

Atomic Step

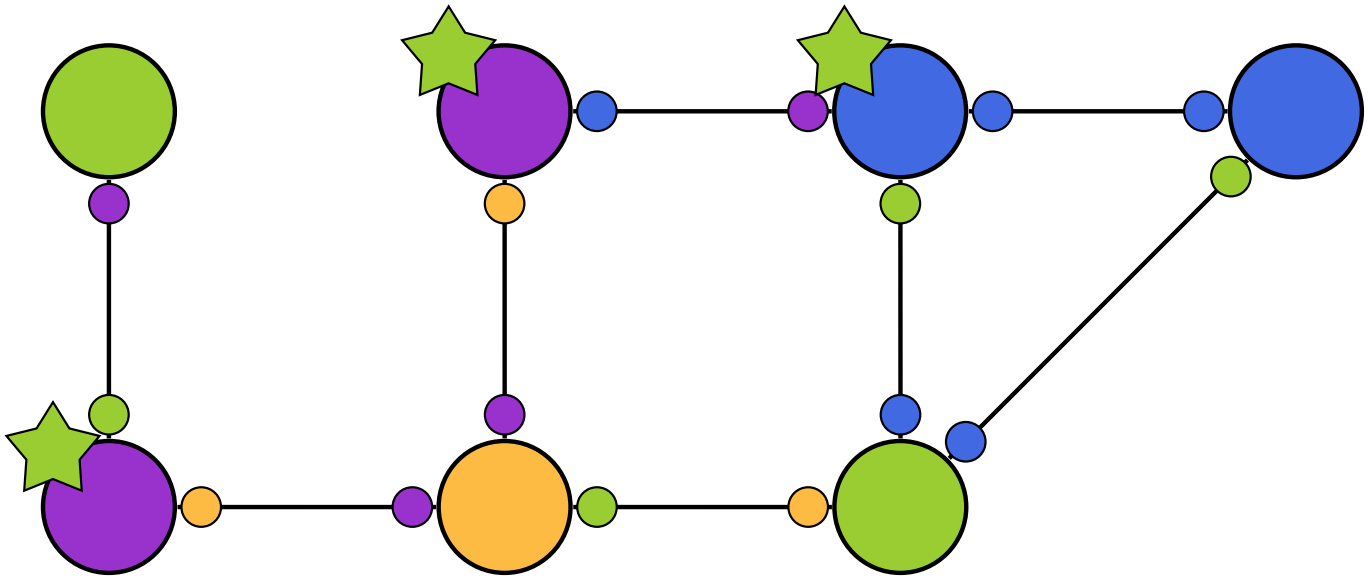
- Reading of the variables of the neighbors



Locally Shared Memory Model with Composite Atomicity

Atomic Step

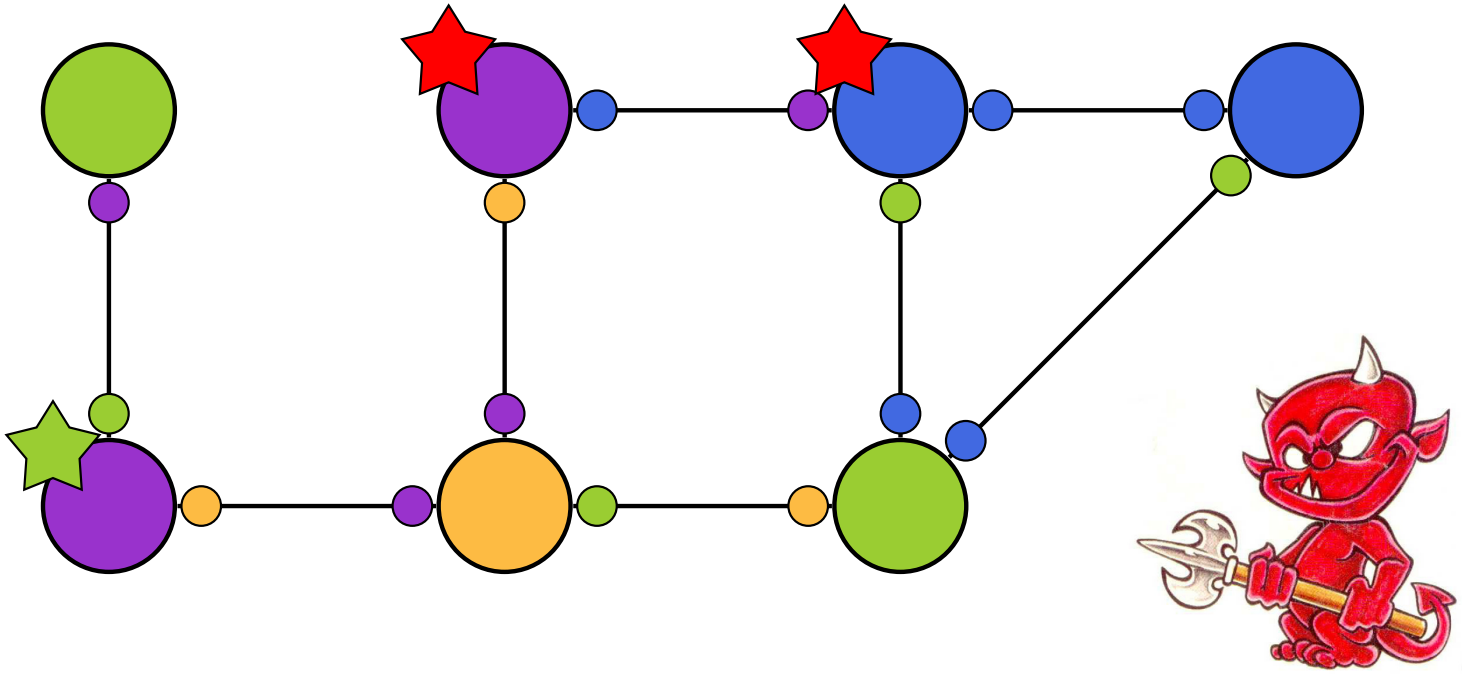
- Reading of the variables of the neighbors
- Enabled nodes



Locally Shared Memory Model with Composite Atomicity

Atomic Step

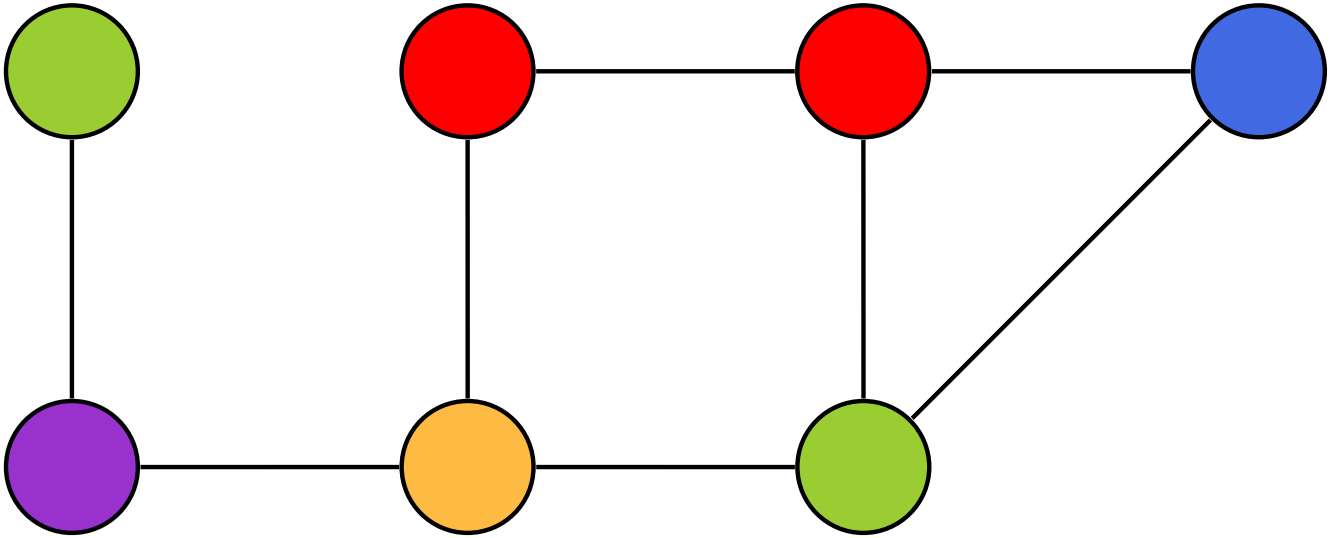
- Reading of the variables of the neighbors
- Enabled nodes
- Daemon selection: models the asynchronism



Locally Shared Memory Model with Composite Atomicity

Atomic Step

- Reading of the variables of the neighbors
- Enabled nodes
- Daemon selection: models the asynchronism
- Update of the local states



Daemons

- Synchronous
- Central / Distributed
- Fairness : Strongly Fair, Weakly Fair, Unfair

Distributed unfair daemon: no constraint, except progress!

Complexity

Space

Memory requirement in bits.

Time

(mainly stabilization time)

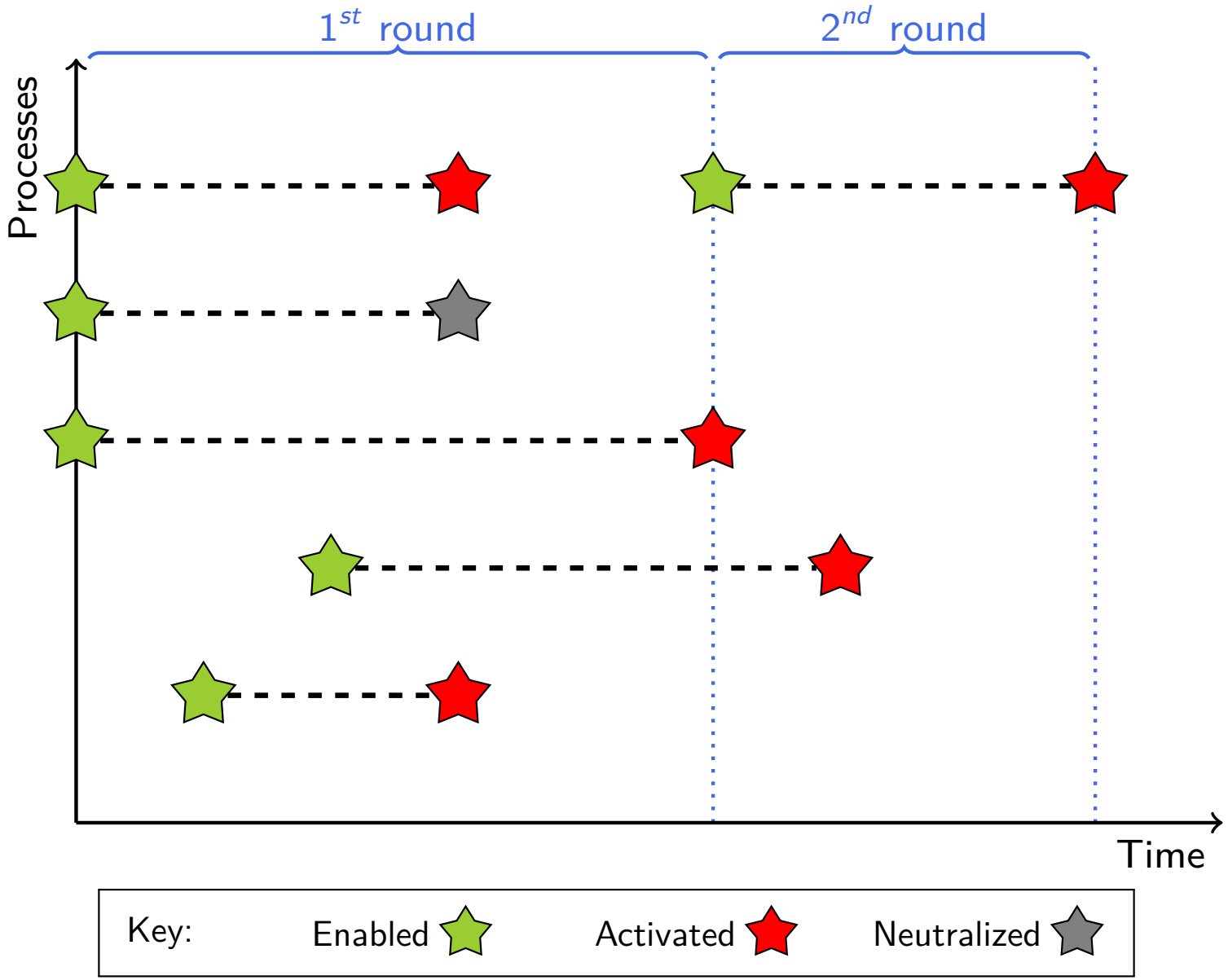
Rounds: execution time **according to the slowest process**.

Essentially similar to the notion of (asynchronous) rounds in message-passing models.

Moves: local state updates.

Rather unusual.

Rounds


1st round
2nd round
Processes
Time
Key:
Enabled
Activated
Neutralized

Complexity in moves: "a measure of energy"

The stabilization time in moves

- captures the amount of computations an algorithm needs to recover a correct behavior.

Complexity in moves: "a measure of energy"

The stabilization time in moves

- captures the amount of computations an algorithm needs to recover a correct behavior.
- can be bounded only if the algorithm is self-stabilizing under the unfair daemon.

Complexity in moves: "a measure of energy"

The stabilization time in moves

- captures the amount of computations an algorithm needs to recover a correct behavior.
- can be bounded only if the algorithm is self-stabilizing under the unfair daemon.

Contraposition: If an algorithm is self-stabilizing, for example, under a weakly fair daemon, but not under an unfair one, then its stabilization time in moves cannot be bounded.

Complexity in moves: "a measure of energy"

The stabilization time in moves

- captures the amount of computations an algorithm needs to recover a correct behavior.
- can be bounded only if the algorithm is self-stabilizing under the unfair daemon.

Contraposition: If an algorithm is self-stabilizing, for example, under a weakly fair daemon, but not under an unfair one, then its stabilization time in moves cannot be bounded.

This means that there are processes whose moves do not make the system progress in the convergence: **these processes waste computation power and so energy.**

Complexity in moves: unusual

Several ***a posteriori*** analyses show that (classical) self-stabilizing algorithms that work under a distributed unfair daemon have an exponential stabilization time in moves in the worst case.

- BFS spanning tree construction of Huang and Chen **[Devismes and Johnen, JPDC 2016]**
- Leader election of Datta, Larmore, Vemula **[Durand *et al.*, Inf. & Comp. 2017]**
- ...

General Schemes for Self-Stabilization

Related Work

- The general transformer of **[Katz & Perry, Dist. Comp. 93]**: not efficient, the purpose is only to demonstrate the feasability of the transformation (characterization).

Related Work

- The general transformer of **[Katz & Perry, Dist. Comp. 93]**: not efficient, the purpose is only to demonstrate the feasability of the transformation (characterization).

- Proof labeling scheme **[Korman *et al.*, Dist. Comp. 2010]**: restricted class of self-stabilizing algorithms (silent algorithms), stabilization time linear in n. No move complexity analysis.
- **[Devismes *et al.*, TAAS 2009]**: restricted class of self-stabilizing algorithms (wave algorithms), stabilization time linear in n and polynomial in moves.

Related Work

- The general transformer of **[Katz & Perry, Dist. Comp. 93]**: not efficient, the purpose is only to demonstrate the feasability of the transformation (characterization).

- Proof labeling scheme **[Korman *et al.*, Dist. Comp. 2010]**: restricted class of self-stabilizing algorithms (silent algorithms), stabilization time linear in n. No move complexity analysis.
- **[Devismes *et al.*, TAAS 2009]**: restricted class of self-stabilizing algorithms (wave algorithms), stabilization time linear in n and polynomial in moves.

Here, we restrict our study to silent spanning-tree-like data structure (*i.e.*, trees or forests).

Our contribution

Algorithm Scheme: a general scheme to compute spanning-tree-like data structures

Theorem 1

Scheme *is silent and self-stabilizing under the distributed unfair daemon in any bidirectional weighted networks of arbitrary topology.*[*]

[*] *n.b.*, the topologies are not necessarily connected. Disconnection may be due to a transient fault.

Algorithm Scheme: a general scheme to compute spanning-tree-like data structures

Theorem 1

Scheme *is silent and self-stabilizing under the distributed unfair daemon in any bidirectional weighted networks of arbitrary topology.*[*]

Theorem 2

The stabilization time in rounds of Scheme *is at most* $4n_{maxCC}$, *where* n_{maxCC} *is the maximum number of processes in a connected component.*

[*] *n.b.*, the topologies are not necessarily connected. Disconnection may be due to a transient fault.

Algorithm Scheme: a general scheme to compute spanning-tree-like data structures

Theorem 1

Scheme *is silent and self-stabilizing under the distributed unfair daemon in any bidirectional weighted networks of arbitrary topology.*[*]

Theorem 2

The stabilization time in rounds of Scheme *is at most* $4n_{maxCC}$, *where* n_{maxCC} *is the maximum number of processes in a connected component.*

Theorem 3

When all weights are strictly positive integers bounded by $W_{\max}$, *the stabilization time of* Scheme *in moves is at most* $(W_{\max}(n_{maxCC}-1)^2+5)(n_{maxCC}+1)n$.

[*] *n.b.*, the topologies are not necessarily connected. Disconnection may be due to a transient fault.

Results on particular instances of Algorithm Scheme

- In an identified network, leader election in each connected component (+ a spanning tree rooted at each leader): $O(n_{\texttt{maxCC}}{}^2 n)$ **moves**
  $\approx$ the best known move complexity **[Durand *et al.*, Inf & Comp 2017]**

[†] Every process in a connected component that does not contain the root should eventually take a special state notifying that it detects the absence of a root.

[‡] With explicit parent pointers.

Results on particular instances of Algorithm Scheme

- In an identified network, leader election in each connected component (+ a spanning tree rooted at each leader): $O({n_{\texttt{maxCC}}}^2 n)$ **moves**
  $\approx$ the best known move complexity **[Durand *et al.*, Inf & Comp 2017]**
- Given an input, spanning forest with non-rooted components detection[†]: $O(n_{\texttt{maxCC}} n)$ **moves**
  $\approx$ the best known move complexity for spanning tree construction **[Cournier, SIROCCO 2009]** [‡]

[†] Every process in a connected component that does not contain the root should eventually take a special state notifying that it detects the absence of a root.

[‡] With explicit parent pointers.

Results on particular instances of Algorithm Scheme

- In an identified network, leader election in each connected component (+ a spanning tree rooted at each leader): $O({n_{\mathtt{maxCC}}}^2 n)$ **moves**
  $\approx$ the best known move complexity **[Durand *et al.*, Inf & Comp 2017]**
- Given an input, spanning forest with non-rooted components detection[†]: $O(n_{\mathtt{maxCC}} n)$ **moves**
  $\approx$ the best known move complexity for spanning tree construction **[Cournier, SIROCCO 2009]** [‡]
- In assuming a rooted network, shortest-path spanning tree with non-rooted components detection: $O(\mathtt{W}_{\max} {n_{\mathtt{maxCC}}}^3 n)$ **moves** ($\mathtt{W}_{\max}$ is the maximum weight of an edge)
  $\approx$ the best known move complexity **[Devismes *et al.*, OPODIS 2016]**

[†] Every process in a connected component that does not contain the root should eventually take a special state notifying that it detects the absence of a root.

[‡] With explicit parent pointers.

The problem

Inputs (constants)

$canBeRoot_u$: true if u is candidate to be root.

Inputs (constants)

$canBeRoot_u$: true if u is candidate to be root.

In a terminal configuration, every tree root satisfies *canBeRoot*, but the converse is not necessarily true.

Inputs (constants)

$canBeRoot_u$: true if u is candidate to be root.

For every connected component GC, if there is at least one candidate $u \in GC$, then at least one process of GC should be a tree root in a terminal configuration.

Inputs (constants)

$canBeRoot_u$: true if u is candidate to be root.

If there is no candidate in a connected component, all processes of the component should converge to a particular terminal state notifying that it detects the absence of candidate.
(non-rooted components detection)

Inputs (constants)

$canBeRoot_u$: true if u is candidate to be root.

$pname_u$: the name of u.

Inputs (constants)

$canBeRoot_u$: true if u is candidate to be root.

$pname_u$: the name of u.

$pname_u \in IDs$, where $IDs = \mathbb{N} \cup \{\perp\}$ is totally ordered by $<$ and $\min_<(IDs) = \perp$.

Inputs (constants)

$canBeRoot_u$: true if u is candidate to be root.

$pname_u$: the name of u.

Two considered cases:

- $\forall v \in V, pname_v = \perp$.
- $\forall u, v \in V, pname_u \neq \perp \wedge (u \neq v \Rightarrow pname_u \neq pname_v)$

Weights

- $\omega_u(v) \in DistSet$ denotes the weight of the arc (u, v)
- $(DistSet, \oplus, \prec)$ is an ordered magma:
  - $\oplus$ is a closed binary operation on $DistSet$
  - $\prec$ is a total order on this set
  - $\forall (u, v), \forall d \in DistSet,\ d \prec d \oplus \omega_u(v)$
- $distRoot(u)$: the distance value of u is u is a root
- $P\_nodeImp(u)$ is a local predicate which is true is u should move to improve the solution

Variables

$st_u \in \{I, C, EB, EF\}$

$parent_u \in \{\perp\} \cup Lbl$: parent in the tree

$d_u \in DistSet$: distance to the root

Variables

$st_u \in \{I, C, EB, EF\}$

Normal behavior

I : *Isolated*

C : *Correct* (belong to a tree)

$parent_u \in \{\perp\} \cup Lbl$: parent in the tree

$d_u \in DistSet$: distance to the root

Variables

$st_u \in \{I, C, EB, EF\}$

In a terminal configuration, if V_u contains a candidate, then $st_u = C$, otherwise $st_u = I$.

$parent_u \in \{\perp\} \cup Lbl$: parent in the tree

$d_u \in DistSet$: distance to the root

Variables

$st_u \in \{I, C, EB, EF\}$

Correction mechanism

EB: Error Broadcast
EF: Error Feedback

$parent_u \in \{\perp\} \cup Lbl$: parent in the tree

$d_u \in DistSet$: distance to the root

Instances

Shortest-Path Spanning Tree

Inputs:

- $canBeRoot_u$ is false for any process except for $u = r$,
- $pname_u$ is $\perp$, and
- $\omega_u(v) = \omega_v(u) \in \mathbb{N}^*$, for every $v \in \Gamma(u)$.

Ordered Magma:

- $DistSet = \mathbb{N}$,
- $i1 \oplus i2 = i1 + i2$,
- $i1 \prec i2 \equiv i1 < i2$, and
- $distRoot(u) = 0$.

Predicate:

- $P\_nodeImp(u) \equiv$
  $(\exists v \in \Gamma(u) \mid st_v = C \wedge d_v \oplus \omega_u(v) \prec d_u)$
  $\vee$
  $canBeRoot_u \wedge distRoot(u) \prec d_u$

Leader Election

Inputs:

- $canBeRoot_u$ is true for any process,
- $pname_u$ is the identifier of u ($n.b.$, $pname_u \in \mathbb{N}$)
- $\omega_u(v) = (\perp, 1)$ for every $v \in \Gamma(u)$

Ordered Magma:

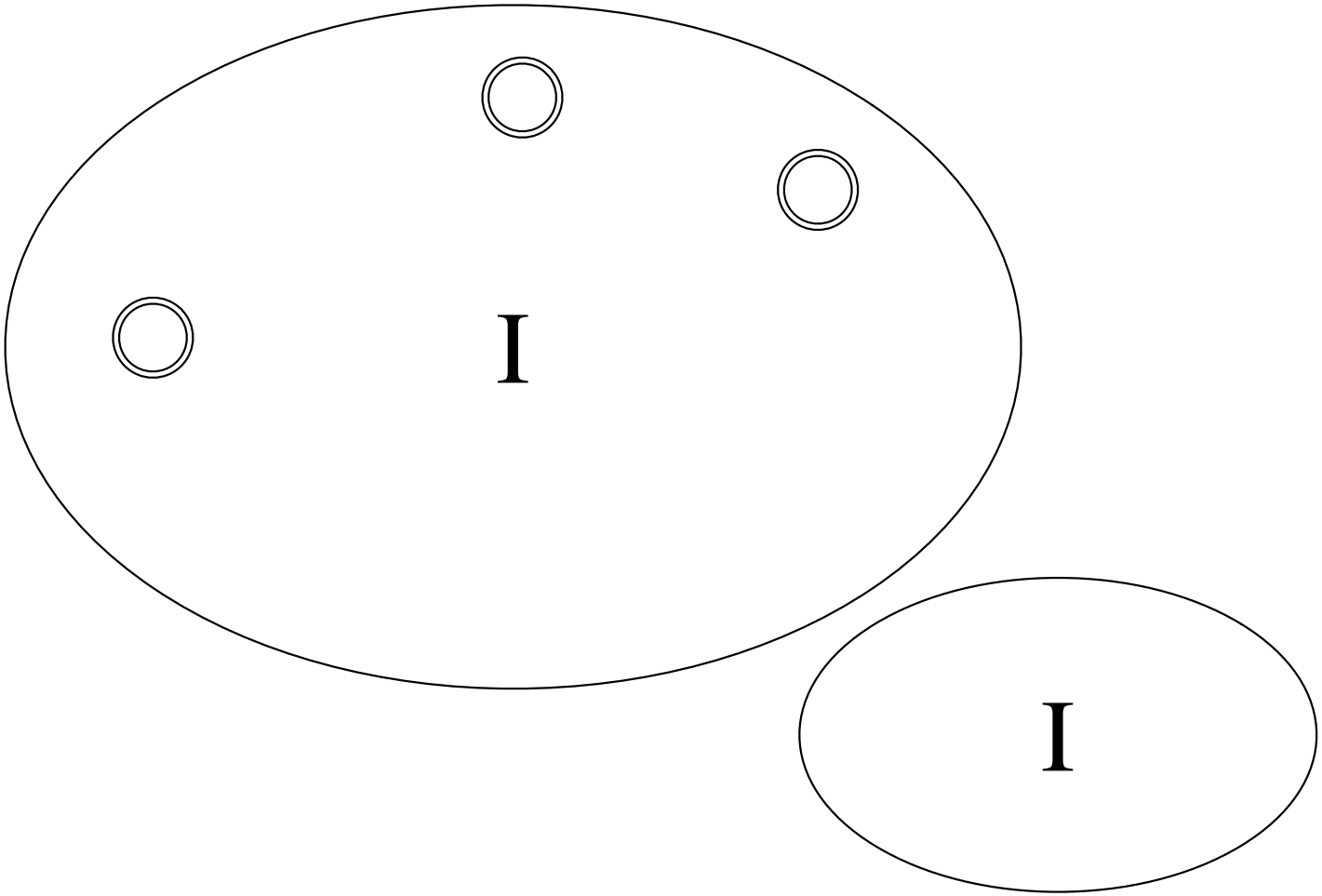
- $DistSet = IDs \times \mathbb{N}$
  for every $d = (a, b) \in DistSet$, we let $d.id = a$ and $d.h = b$.
- $(id1, i1) \oplus (id2, i2) = (id1, i1 + i2)$;
- $(id1, i1) \prec (id2, i2) \equiv$
  $(id1 < id2) \vee [(id1 = id2) \wedge (i1 < i2)]$
- $distRoot(u) = (pname_u, 0)$

Predicate:

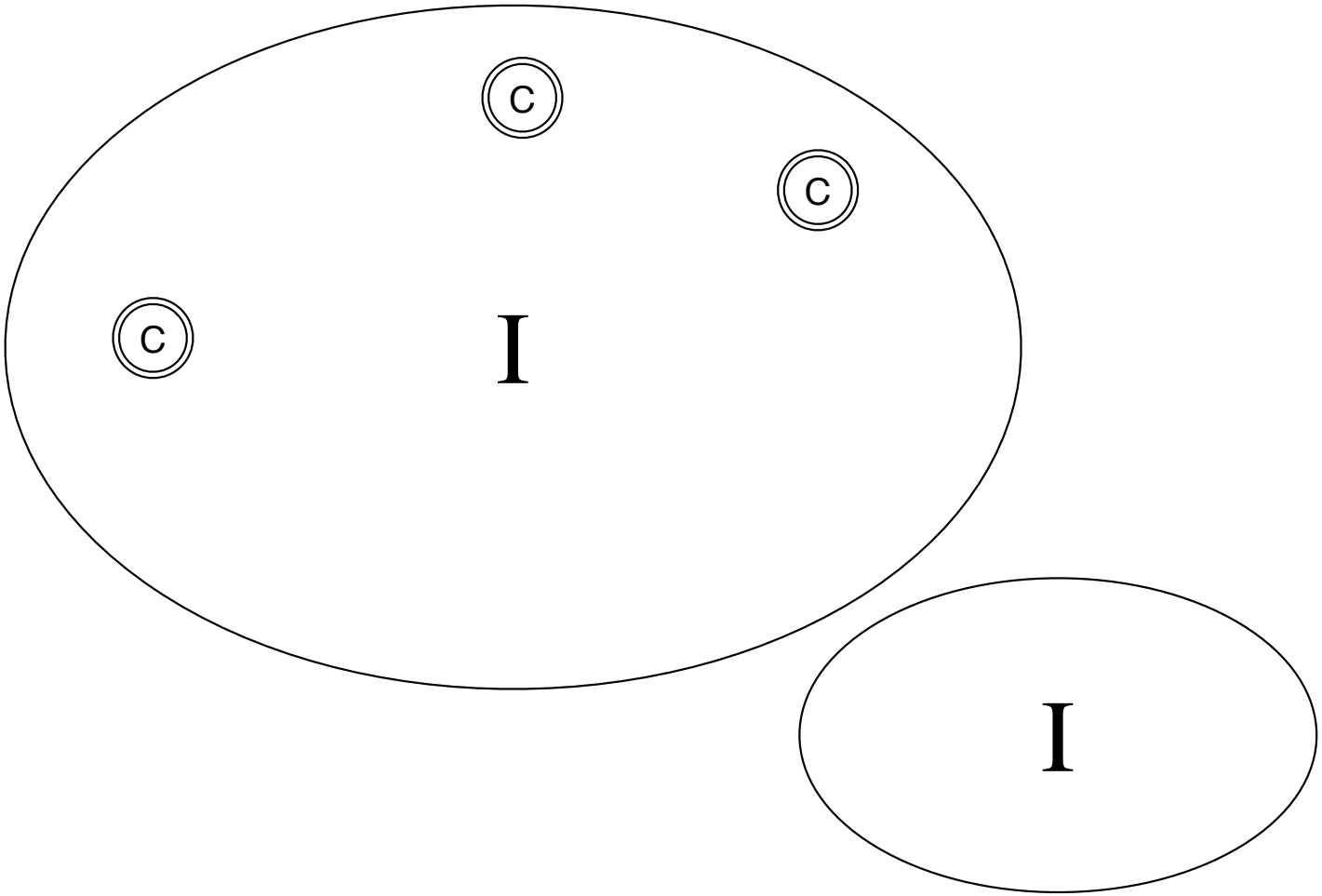
- $P\_nodeImp(u) \equiv (\exists v \in \Gamma(u) \mid st_v = C \wedge$
  $d_v.id < d_u.id) \vee distRoot(u) \prec d_u$

Our solution in a nutshell

Typical Execution

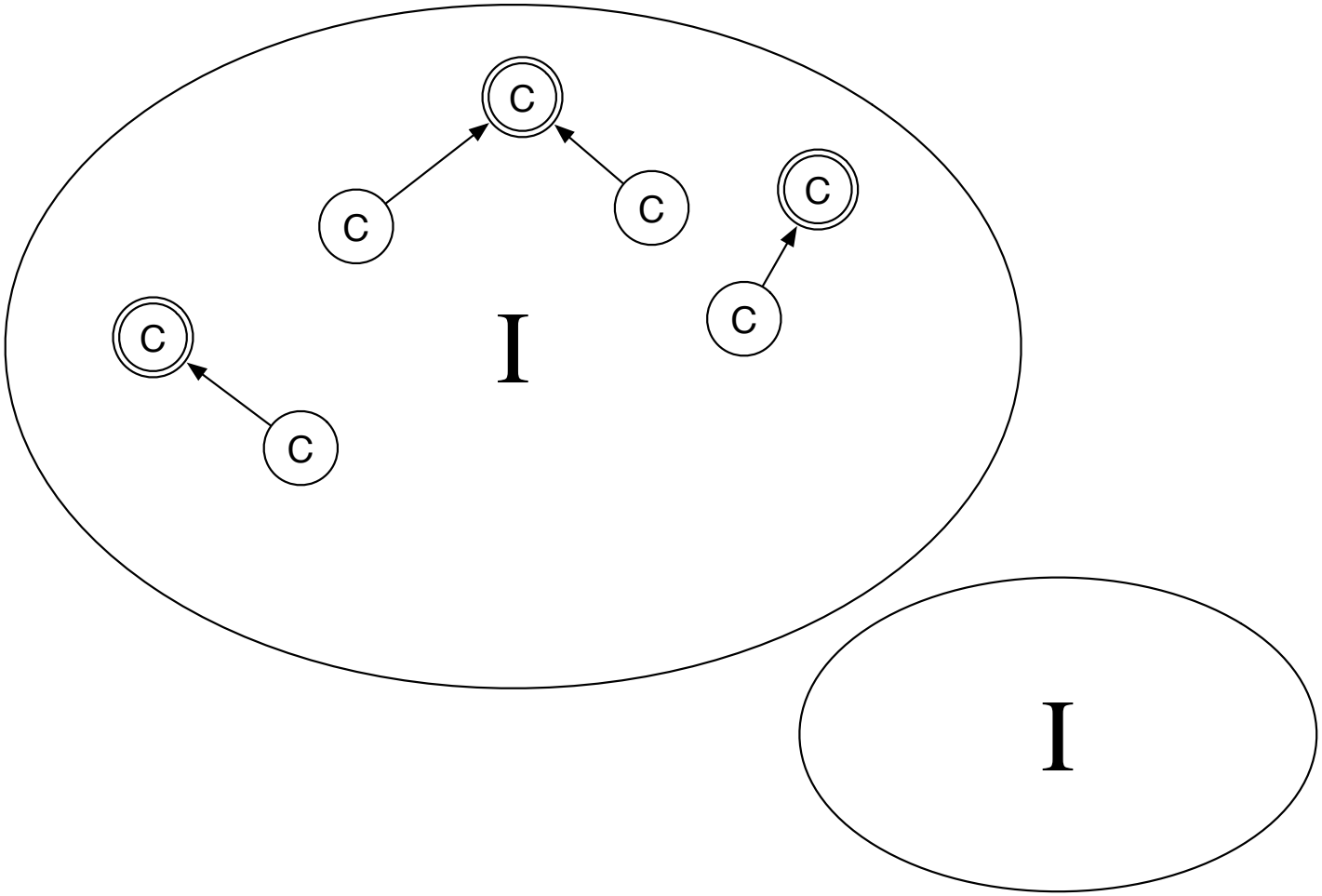


Typical Execution



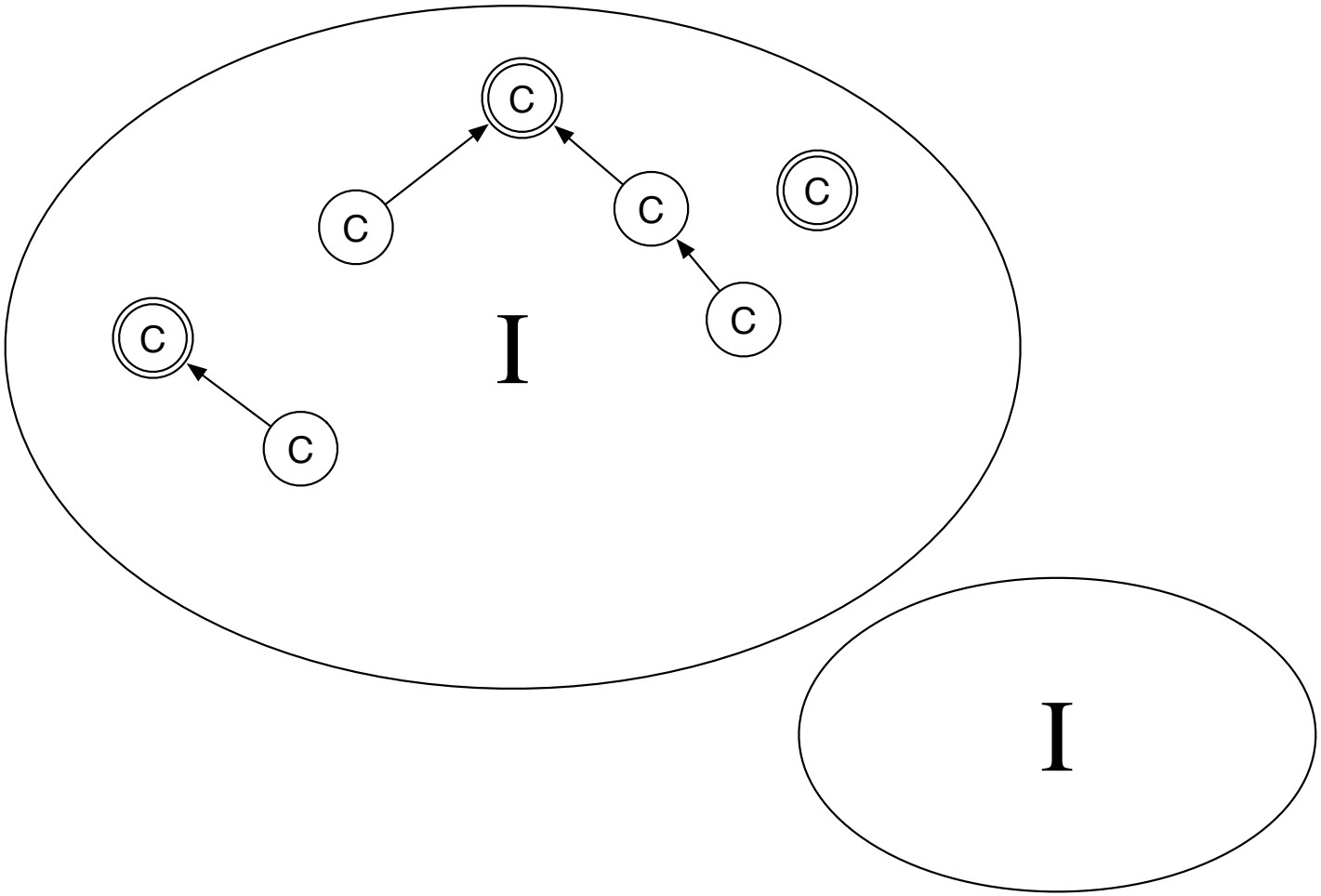
Any candidate u executes $\mathbf{R_R}$: either u becomes a root ($st_u \leftarrow C$, $d_u \leftarrow distRoot(u)$, $parent_u \leftarrow \perp$) or join an existing tree rooted to another candidate, if it is "better" (u selects a neighbor v in that tree as parent, and $d_u \leftarrow d_v \oplus \omega_u(v)$)

Typical Execution



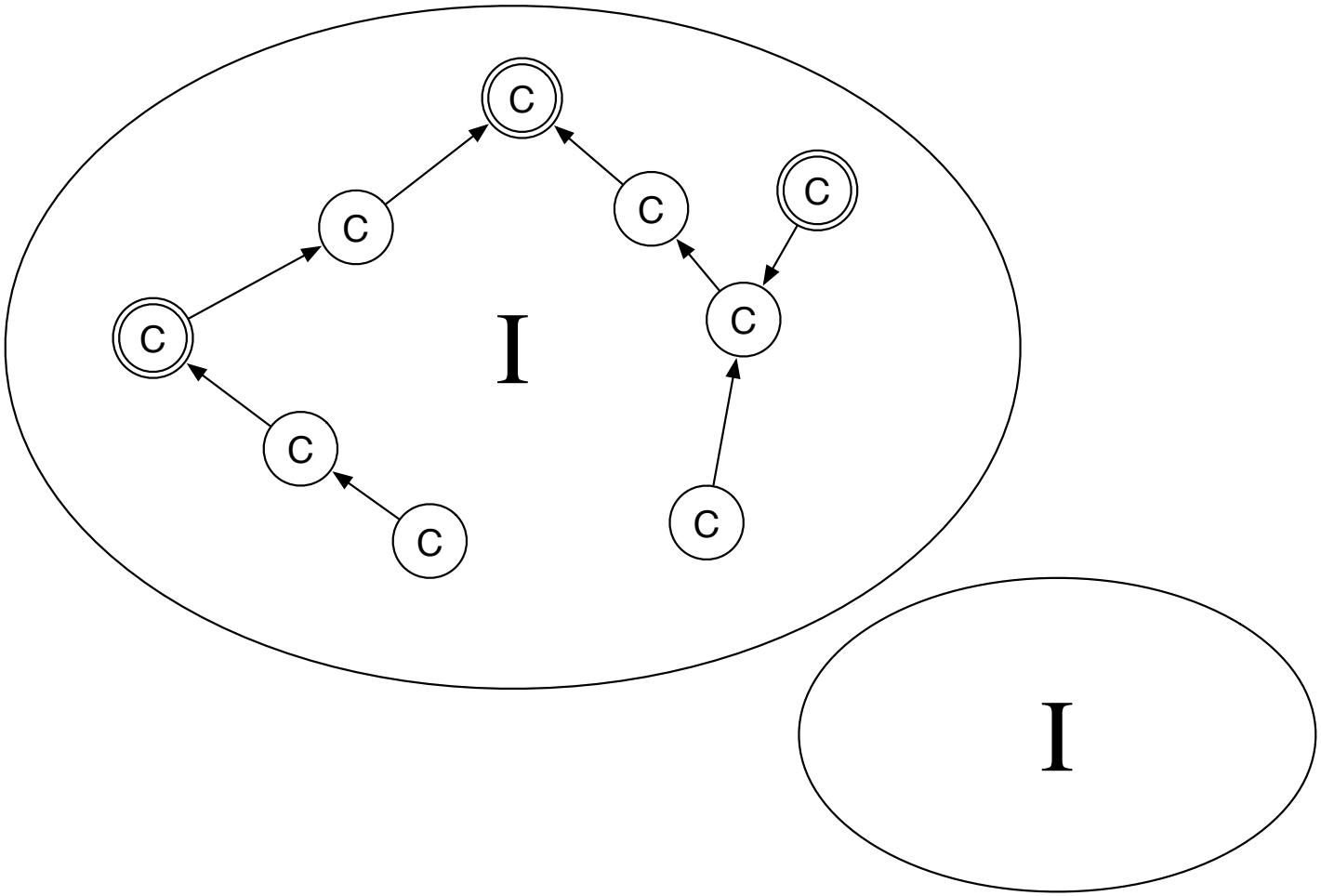
Any non-candidate v executes $\mathbf{R_R}$ when it finds a neighbor with status C: $st_u \leftarrow C$, select a parent v among its neighbors of status C, and $d_u \leftarrow d_v \oplus \omega_u(v)$

Typical Execution



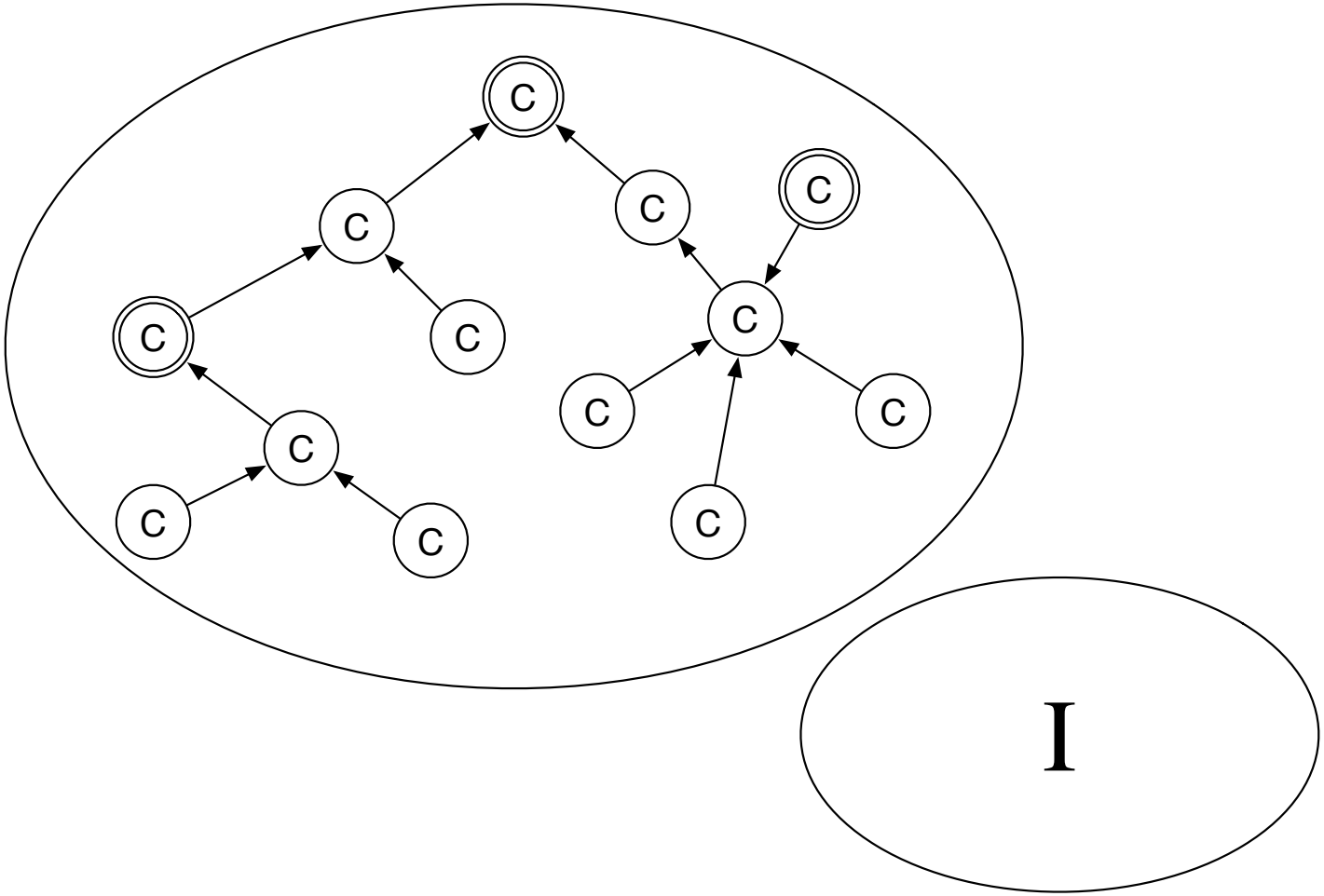
In parallel, rules $\mathbf{R_U}$ are executed to reduce the weight of the trees: when a process u with status C satisfies $P\_nodeImp(u)$, this means that u can reduce d_u by selecting another neighbor with status C as parent.

Typical Execution



A candidate can lose its tree root condition using $\mathbf{R_U}$, if it finds a sufficiently good parent in its neighborhood.

Typical Execution



Overall, within at most $n_{\texttt{maxCC}}$ rounds, a terminal configuration is reached.

Abnormal Roots

Inconsistencies are detected by some processes called Abnormal Roots

A process u is an abnormal root if u is not a normal root[§], $st_u \neq I$, and satisfies one of the following three conditions:

- its parent pointer does not designate a neighbor,
- its distance d_u is inconsistent with the distance of its parent, or
- its status is inconsistent with the status of its parent.

[§] A normal root is any process v such that $canBeRoot_v \wedge st_v = C \wedge parent_v = \perp \wedge d_v = distRoot(v)$.

Abnormal Roots

Inconsistencies are detected by some processes called Abnormal Roots

A process u is an abnormal root if u is not a normal root[§], $st_u \neq I$, and satisfies one of the following three conditions:

- its parent pointer does not designate a neighbor,
- its distance d_u is inconsistent with the distance of its parent, or
- its status is inconsistent with the status of its parent.

An abnormal root u is alive if $st_u \neq EF$

An abnormal tree is a tree rooted at an abnormal root.

An abnormal tree is alive if it contains a node v such that $st_v \neq EF$

[§] A normal root is any process v such that $canBeRoot_v \wedge st_v = C \wedge parent_v = \perp \wedge d_v = distRoot(v)$.

Abnormal Roots

Inconsistencies are detected by some processes called Abnormal Roots

A process u is an abnormal root if u is not a normal root[§], $st_u \neq I$, and satisfies one of the following three conditions:

- its parent pointer does not designate a neighbor,
- its distance d_u is inconsistent with the distance of its parent, or
- its status is inconsistent with the status of its parent.

An abnormal root u is alive if $st_u \neq EF$

An abnormal tree is a tree rooted at an abnormal root.

An abnormal tree is alive if it contains a node v such that $st_v \neq EF$

Main result: No abnormal alive root (resp. tree) is created during the execution.

[§] A normal root is any process v such that $canBeRoot_v \wedge st_v = C \wedge parent_v = \perp \wedge d_v = distRoot(v)$.

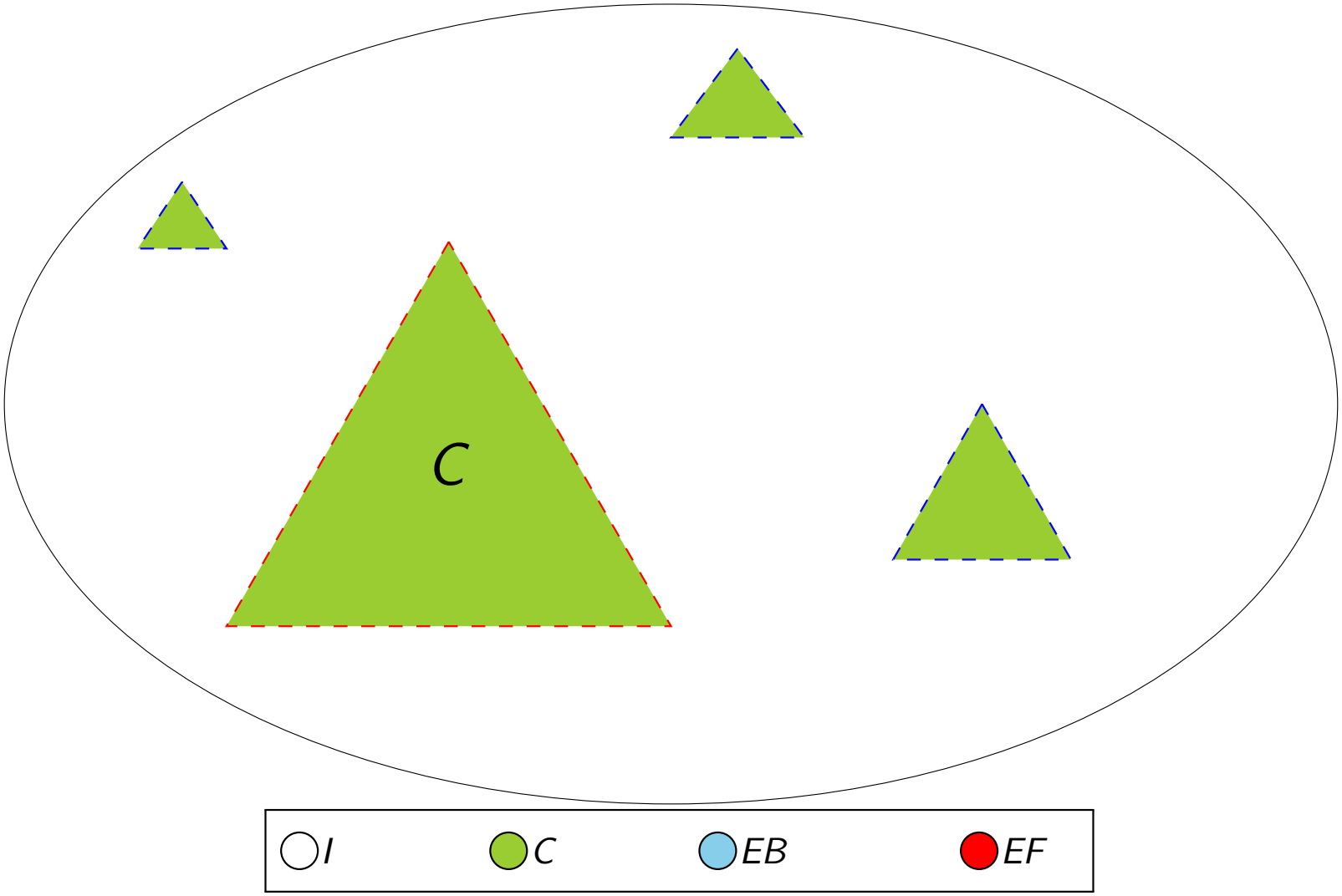
Abnormal trees removal

Freeze before Remove

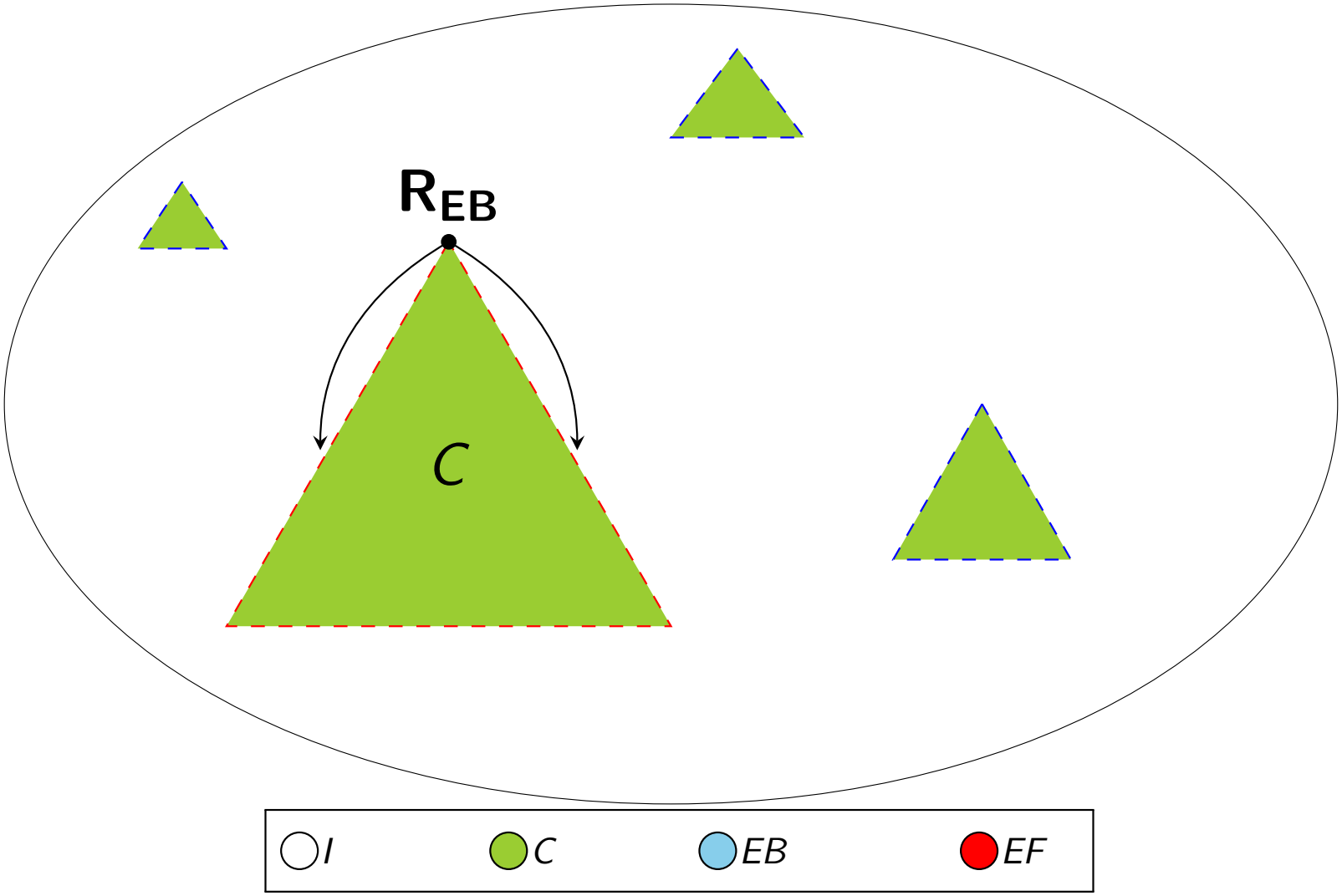
Variable $st_u \in \{I, C, EB, EF\}$

- I means *Isolated*
  - a process of status I can
    - join a tree only by choosing a neighbor of status C as parent, or
    - becoming a root
- C means *correct*
  - only processes of status C in a tree can modify their parent pointers and
  - only by choosing a neighbor of status C as parent
- EB: Error Broadcast
- EF: Error Feedback

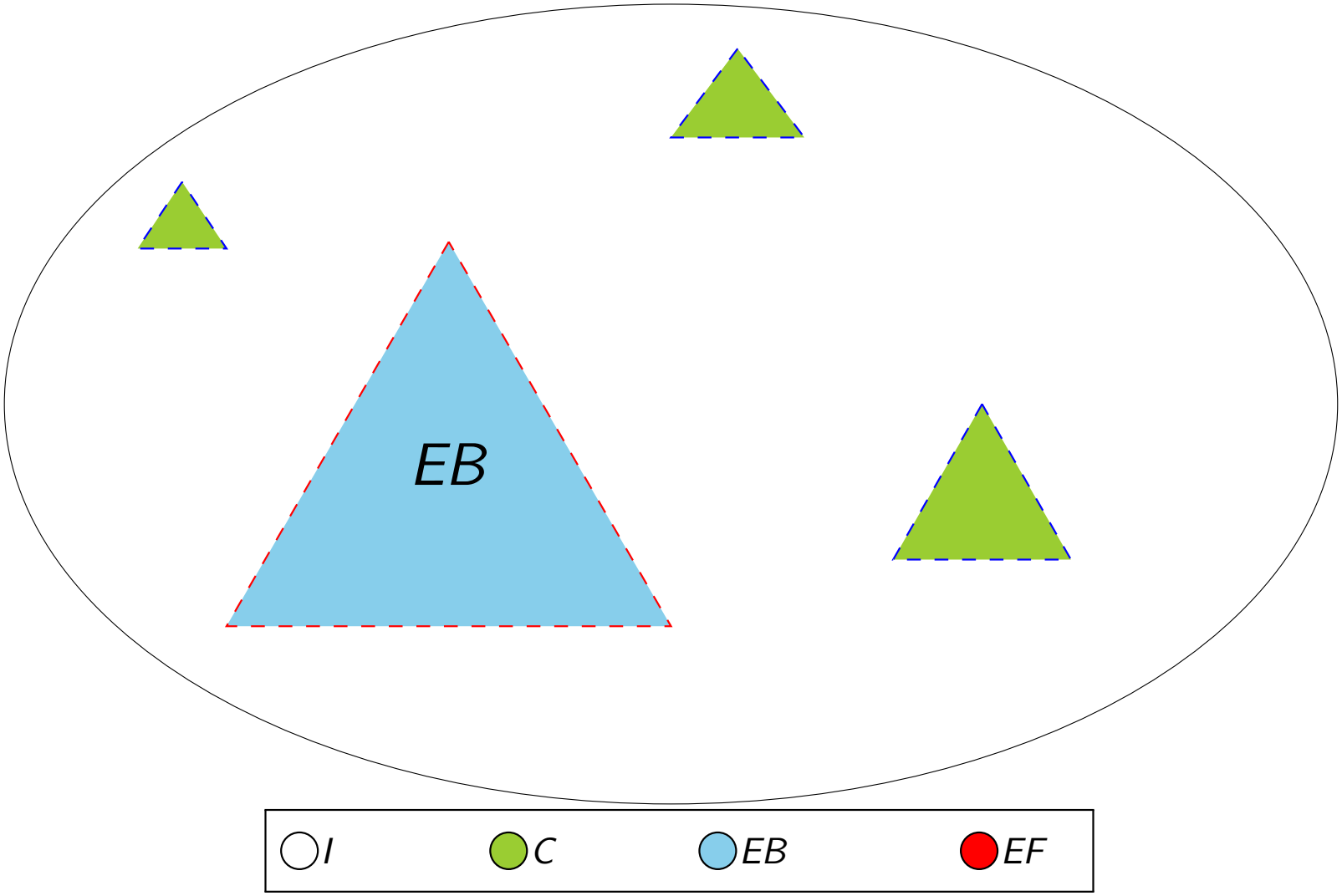
Freeze before remove



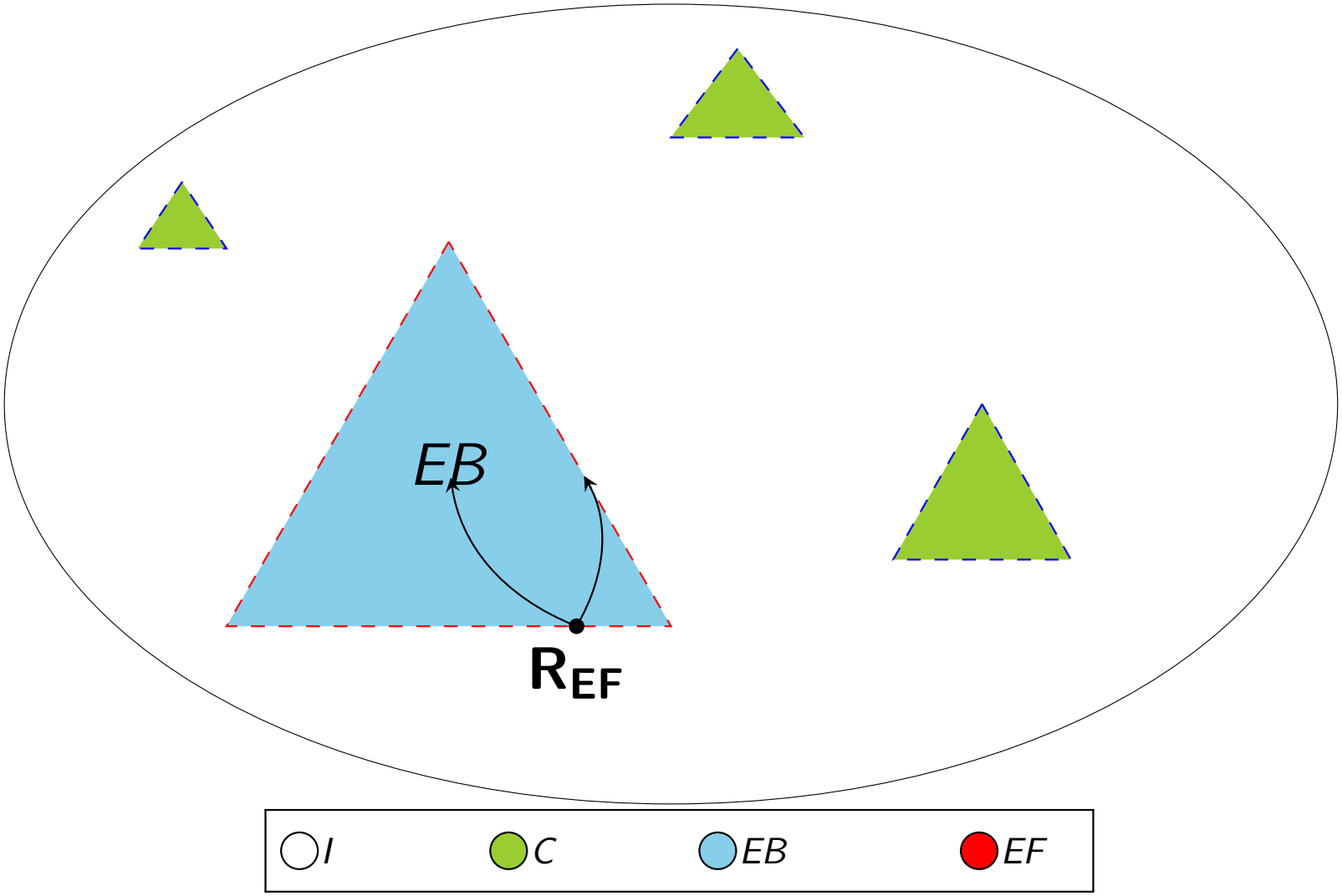
Freeze before remove



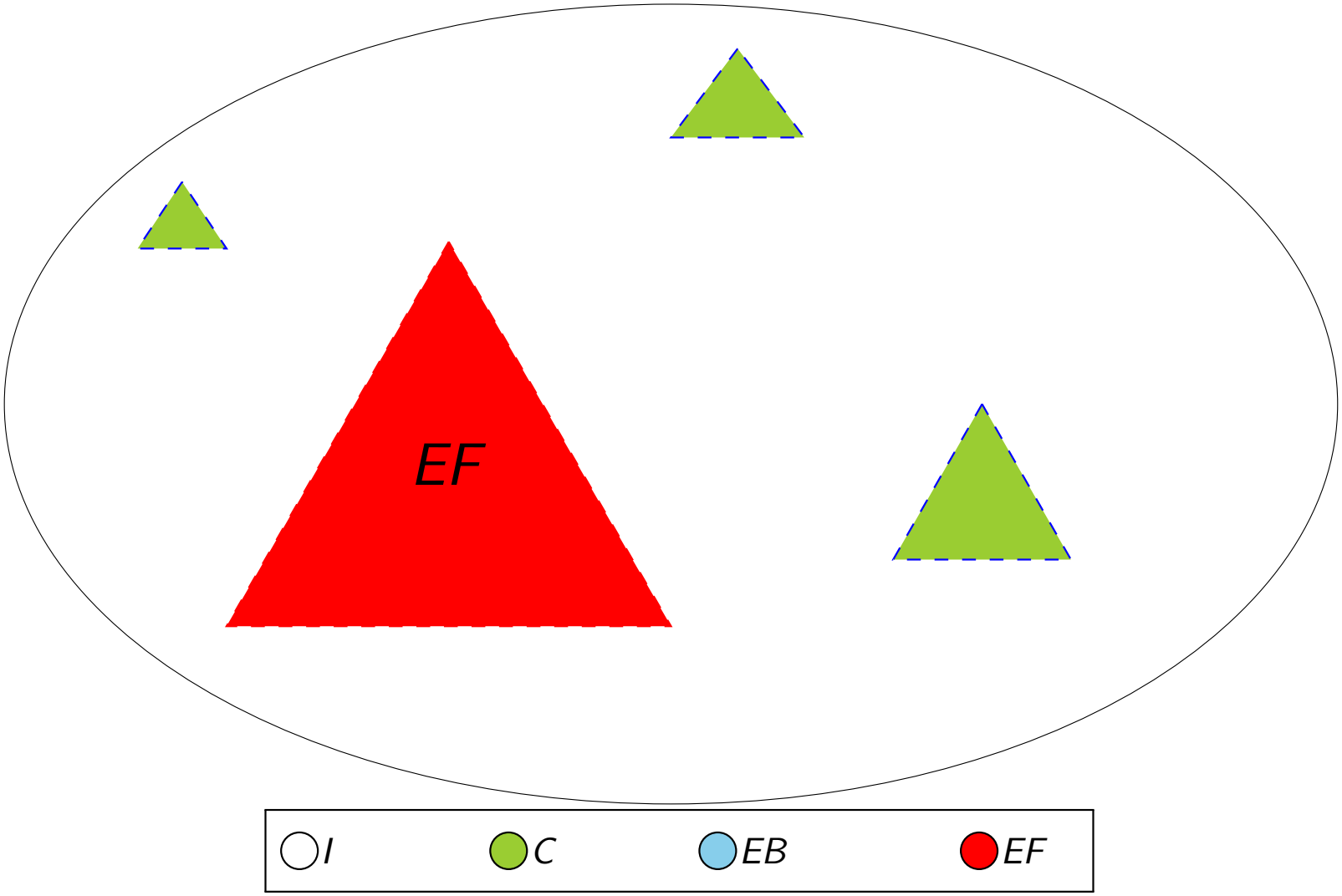
Freeze before remove



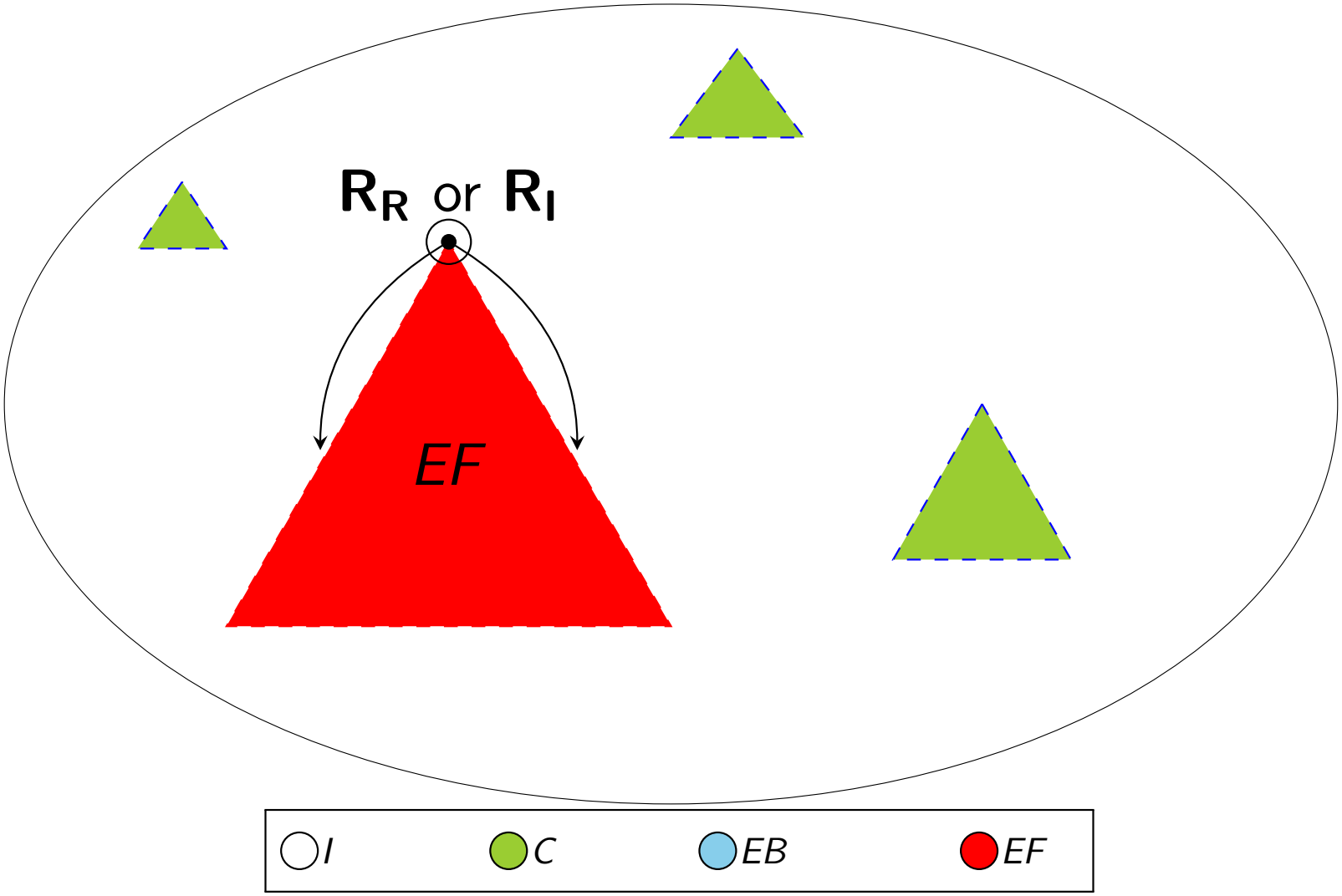
Freeze before remove



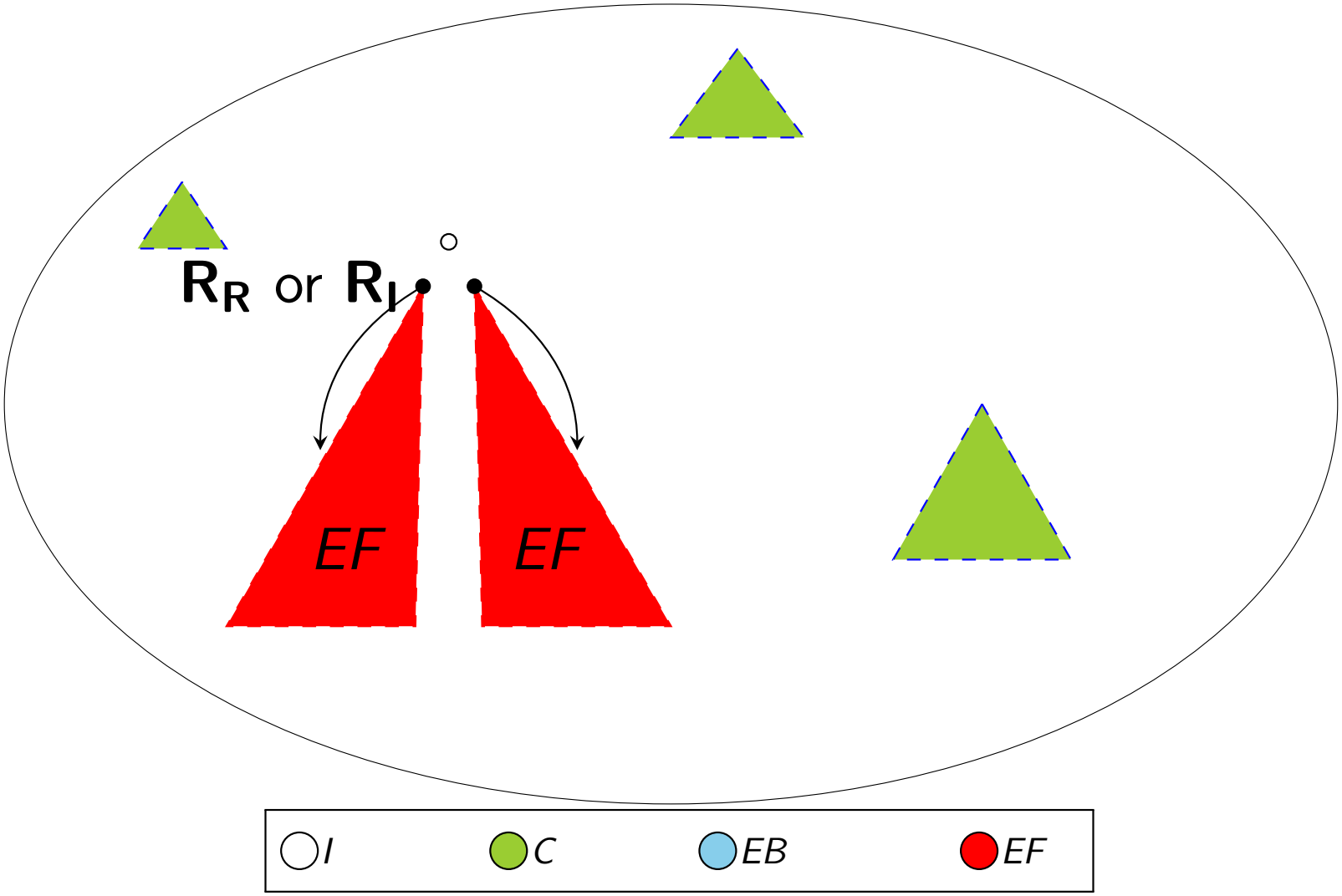
Freeze before remove



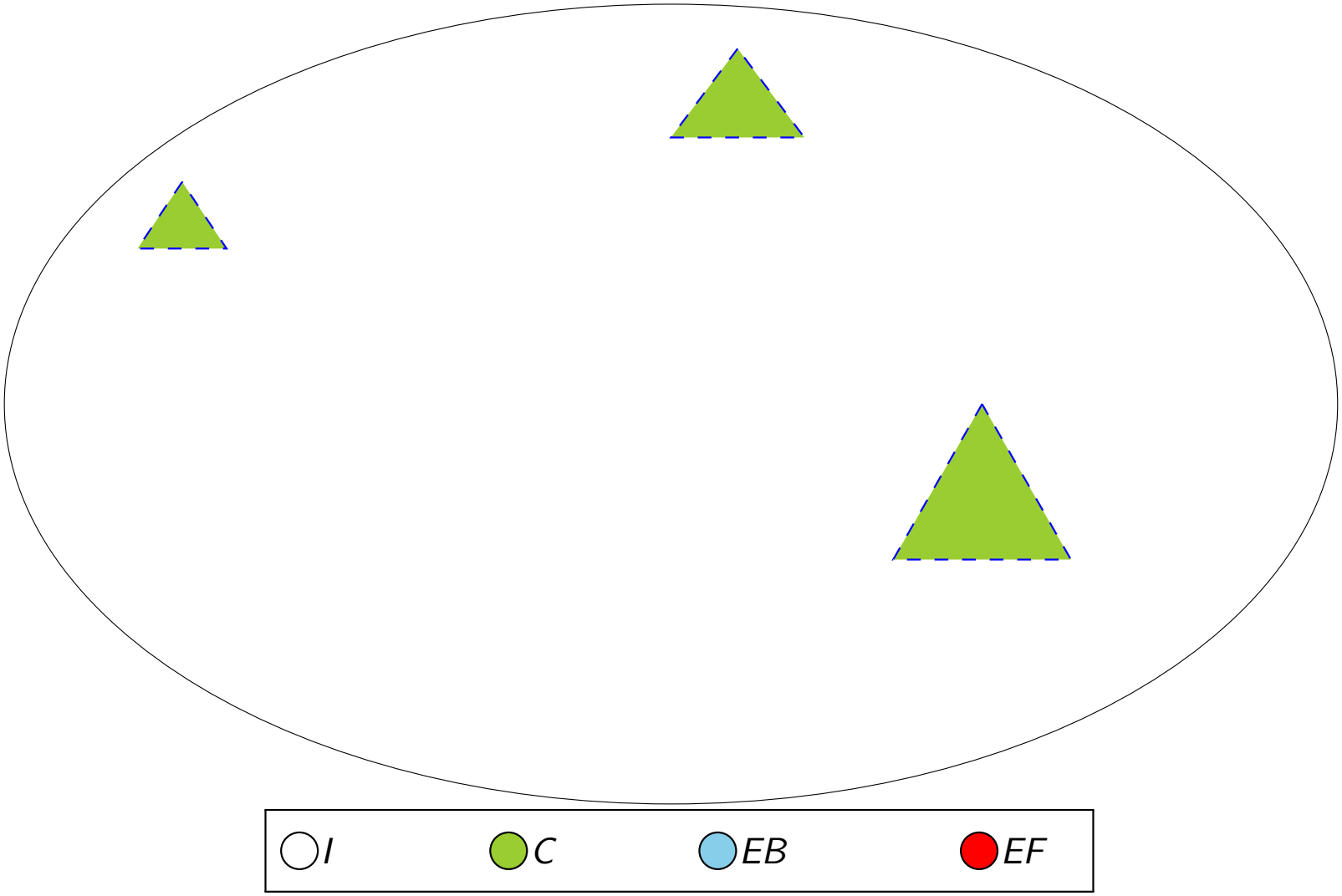
Freeze before remove



Freeze before remove



Freeze before remove



The definition of abnormal root should take to possible inconsistencies of variables *st* into account!

Stabilization Time in Rounds

- No alive abnormal tree created
- Height of an abnormal tree: at most $n_{\texttt{maxCC}}$

Stabilization Time in Rounds

- No alive abnormal tree created
- Height of an abnormal tree: at most $n_{\texttt{maxCC}}$
- **Cleaning:**
  - EB-wave : $n_{\texttt{maxCC}}$
  - EF-wave : $n_{\texttt{maxCC}}$
  - R-wave : $n_{\texttt{maxCC}}$

Stabilization Time in Rounds

- No alive abnormal tree created
- Height of an abnormal tree: at most $n_{\texttt{maxCC}}$
- **Cleaning:**
  - EB-wave : $n_{\texttt{maxCC}}$
  - EF-wave : $n_{\texttt{maxCC}}$
  - R-wave : $n_{\texttt{maxCC}}$
- **Building of the spanning tree/forest:** $n_{\texttt{maxCC}}$ (like in the typical execution)

Stabilization Time in Rounds

- No alive abnormal tree created
- Height of an abnormal tree: at most $n_{\texttt{maxCC}}$
- **Cleaning:**
  - EB-wave : $n_{\texttt{maxCC}}$
  - EF-wave : $n_{\texttt{maxCC}}$
  - R-wave : $n_{\texttt{maxCC}}$
- **Building of the spanning tree/forest:** $n_{\texttt{maxCC}}$ (like in the typical execution)

$O(4n_{\texttt{maxCC}})$ **rounds**

Stabilization Time in Moves (1/2)

Let GC be a connected component of G.

Let $SL(\gamma, GC)$ be the set of processes $u \in GC$ such that, in the configuration γ, u is an alive abnormal root, or $canBeRoot_u \wedge distRoot(u) \prec d_u \wedge st_u = C$ holds.

Second case: u is candidate and can improve by becoming a root ($\mathbf{R_U}$)

Stabilization Time in Moves (1/2)

Let GC be a connected component of G.

Let $SL(\gamma, GC)$ be the set of processes $u \in GC$ such that, in the configuration γ, u is an alive abnormal root, or $canBeRoot_u \wedge distRoot(u) \prec d_u \wedge st_u = C$ holds.

Second case: u is candidate and can improve by becoming a root ($\mathbf{R_U}$)

If a process satisfies one of these two conditions, then it does so from the beginning of the execution.

Let $e = \gamma_0, \cdots, \gamma_i$ be an execution: $SL(\gamma_{i+1}, GC) \subseteq SL(\gamma_i, GC)$.

Stabilization Time in Moves (1/2)

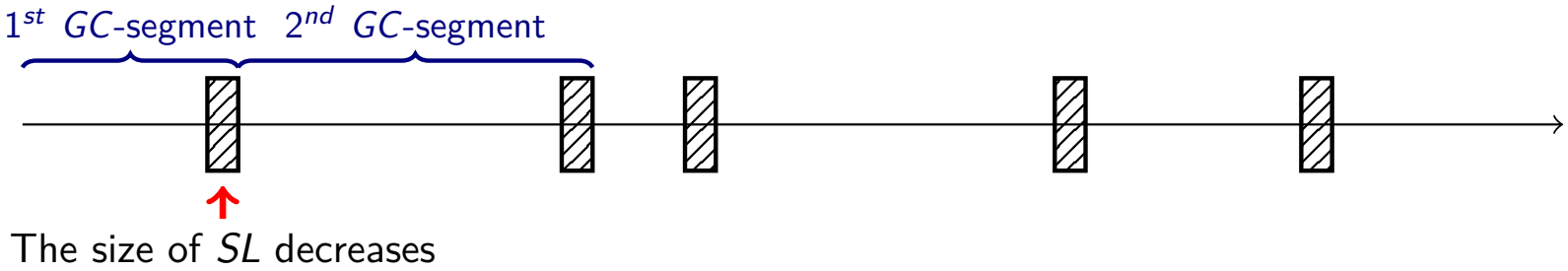
Let GC be a connected component of G.

Let $SL(\gamma, GC)$ be the set of processes $u \in GC$ such that, in the configuration γ, u is an alive abnormal root, or $canBeRoot_u \wedge distRoot(u) \prec d_u \wedge st_u = C$ holds.

Second case: u is candidate and can improve by becoming a root ($\mathbf{R_U}$)

If a process satisfies one of these two conditions, then it does so from the beginning of the execution.

Let $e = \gamma_0, \cdots, \gamma_i$ be an execution: $SL(\gamma_{i+1}, GC) \subseteq SL(\gamma_i, GC)$.



$\longrightarrow$ At most $n_{\texttt{maxCC}} + 1$ GC-segments in GC

Stabilization Time in Moves (2/2)

Let u be any process of GC. We proved that the sequence of rules executed by u during a GC-segment belongs to the following language:

$$(\mathbf{R_I} + \varepsilon)(\mathbf{R_R} + \varepsilon)(\mathbf{R_U})^*(\mathbf{R_{EB}} + \varepsilon)(\mathbf{R_{EF}} + \varepsilon).$$

Theorem 4

If the number of $\mathbf{R_U}$ executions during a GC-segment by any process of GC is bounded by nb_UN, then the total number of moves in any execution is bounded by $(nb\_UN + 4)(n_{maxCC} + 1)n$.

Stabilization Time in Moves (2/2)

Let u be any process of GC. We proved that the sequence of rules executed by u during a GC-segment belongs to the following language:

$$(\mathbf{R_I} + \varepsilon)(\mathbf{R_R} + \varepsilon)(\mathbf{R_U})^*(\mathbf{R_{EB}} + \varepsilon)(\mathbf{R_{EF}} + \varepsilon).$$

Theorem 4

If the number of $\mathbf{R_U}$ executions during a GC-segment by any process of GC is bounded by nb_UN, then the total number of moves in any execution is bounded by $(nb\_UN + 4)(n_{maxCC} + 1)n$.

nb_UN is necessarily defined because d_u decreases at each $\mathbf{R_U}(u)$ in a GC-segment.

Theorem 5

When all weights are strictly positive integers bounded by $W_{\max}$ $nb\_UN \leq W_{\max}(n_{maxCC} - 1)^2 + 1$

Conclusion

Conclusion

Our scheme is versatile since it can be applied to build **efficient silent self-stabilizing algorithms** for

- leader election (+ spanning tree of type arbitrary, BFS, DFS, ...), or
- spanning tree or forest constructions of type:
  - arbitrary
  - BFS
  - DFS
  - Shortest-Path
  - ...

in

- identified or
- semi-anonymous (*e.g.*, rooted)

networks.

Thank you for your attention