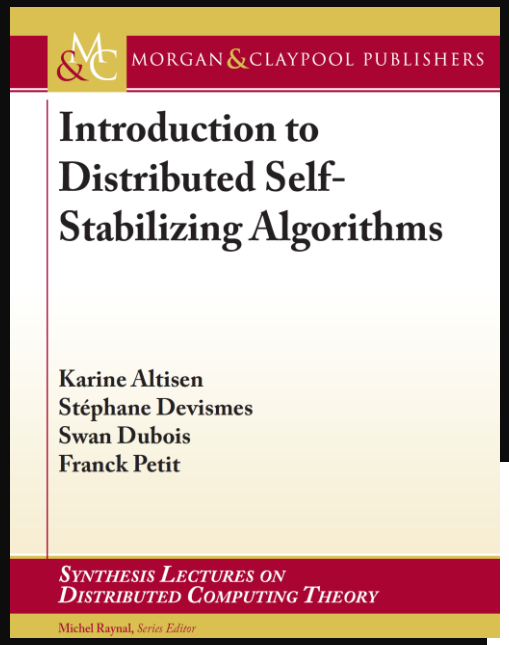
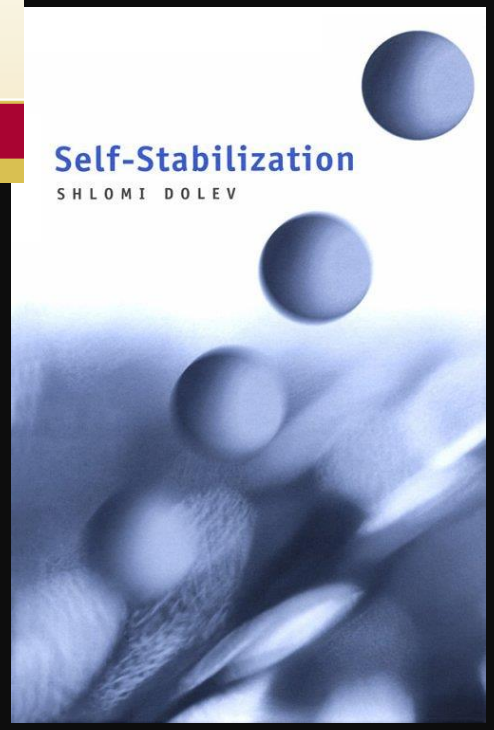
Self-Stabilization

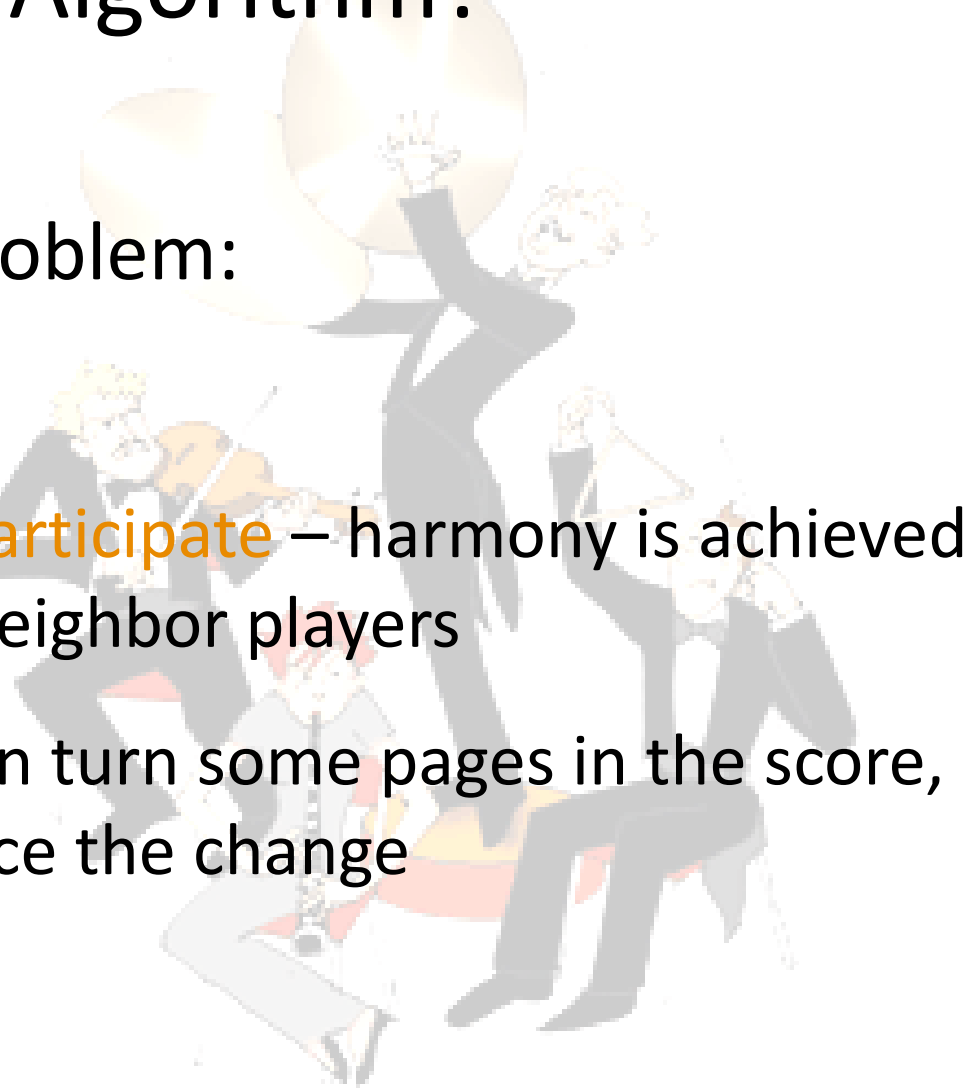
Slides based on S. Dolev Slides, Klara Nahrstedt





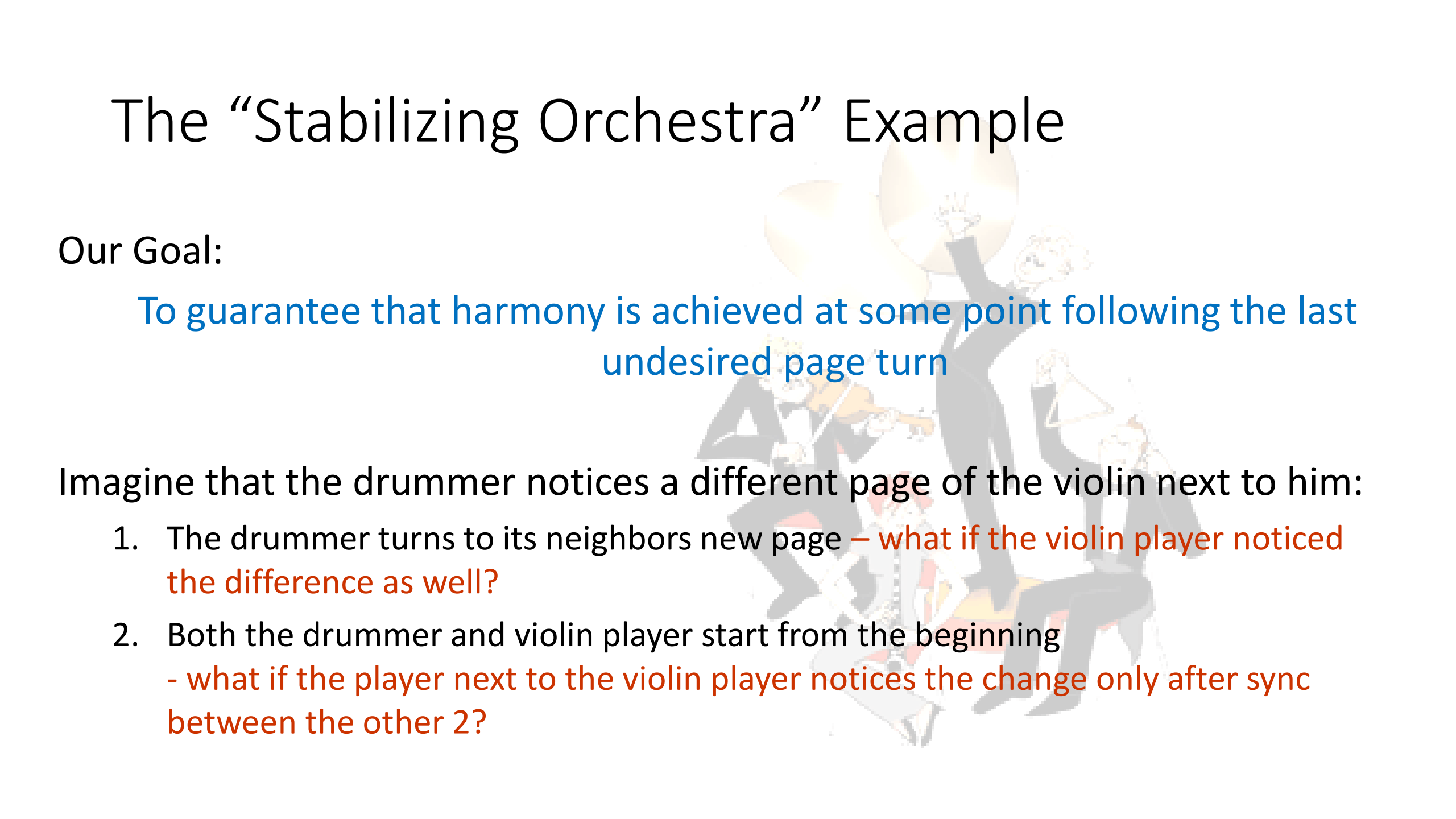
What is a Self-Stabilizing Algorithm?

The “Stabilizing Orchestra” problem:



- The conductor is unable to participate – harmony is achieved by players listening to their neighbor players
- Windy evening – the wind can turn some pages in the score, and the players may not notice the change

The “Stabilizing Orchestra” Example



Our Goal:

To guarantee that harmony is achieved at some point following the last undesired page turn

Imagine that the drummer notices a different page of the violin next to him:

1. The drummer turns to its neighbors new page – what if the violin player noticed the difference as well?
2. Both the drummer and violin player start from the beginning
   - what if the player next to the violin player notices the change only after sync between the other 2?

The “Stabilizing Orchestra” Example – the Self-Stabilizing Solution

Every player will join the neighboring player who is playing the earliest page (including himself)

- Note that the score has a bounded length. What happens if a player goes to the first page of the score before harmony is achieved?

- In every long enough period in which the wind does not turn a page, the orchestra resumes playing in synchrony

Motivation

- As the number of computing elements increase in distributed systems failures become more common

- Fault tolerance (FT) should be automatic, without external intervention

Some Properties of Distributed Systems…

A Safety Property:

"bad things never happen to the system"

e.g., Consensus:

$p_1: propose(v_1) \rightarrow decide(v)$

$p_2: propose(v_2) \rightarrow decide(v)$

$p_3: propose(v_3) \rightarrow decide(v)$

Safety properties:

1. The decided value is the same for all processes
2. The decided value is proposed by some process

Some Properties of Distributed Systems…

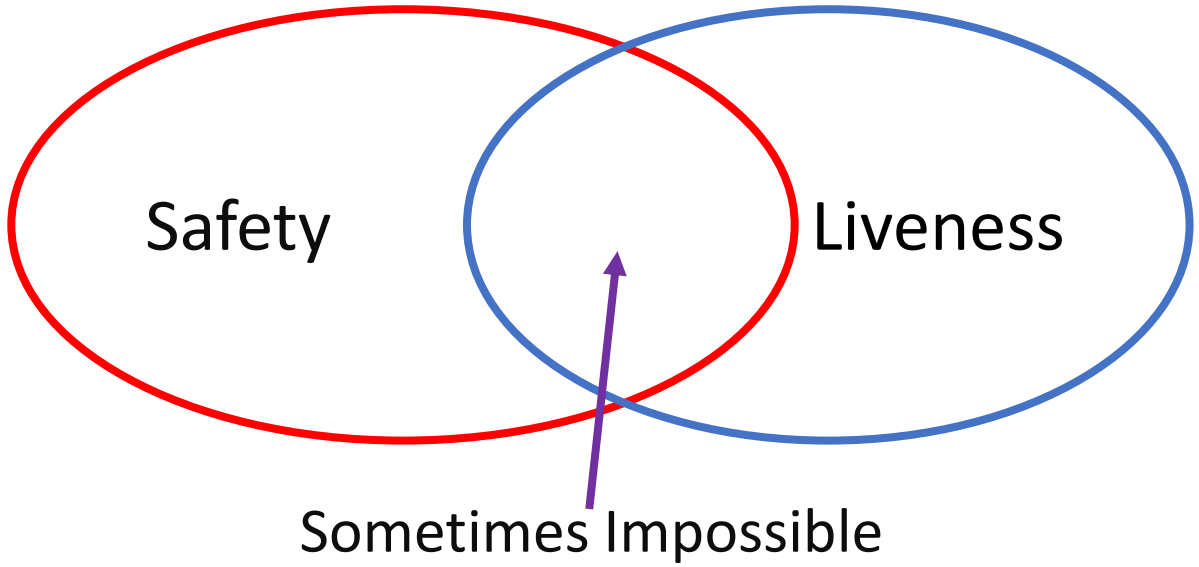
A Liveness Property:

“good things eventually happen”

Consensus Liveness:

1. All processes eventually decide

Safety and Liveness



In a fully asynchronous system, is there a deterministic consensus algorithm that can be safe, live, and fault tolerant? (The FLP impossibility Theorem)

It is impossible to have a deterministic algorithm that achieves safety and liveness

However: Randomized consensus algorithms can circumvent the FLP impossibility result by achieving both safety and liveness with overwhelming probability, even under worst-case scheduling scenarios such as an intelligent denial-of-service attacker in the network.

Some Types of Fault Tolerance Handling

- masking: application layer does not see faults, e.g., redundancy and replication (safety and liveness properties are unaffected)
- non-masking: system deviates, deviation is detected and then corrected: e.g., feedback, roll back and recovery (safety is affected and not liveness, eventually safety is restored)
- fail-safe: the system does not respond but with time-outs we can recover (the liveness property is compromised but not its safety properties)
- graceful degradation: some nodes failed in the Chord protocol, but we were able to recover with an aggravation in efficiency (affects safety but not liveness, and the system eventually recovers to a weaker state that is acceptable to the system).

Configurations of Distributed Systems

Two classes of configurations (or behaviors)

- Legitimate configuration
  - In a non-reactive system, it is represented by invariant over global state of the system
    - Example: legal state of network routing: no cycle in a route between pair of nodes
  - In a reactive system, it is determined by a state predicate and by behavior.
    - Example: in token ring, legitimate config: When (i) there is exactly one token in the network; (ii) in infinite behavior of the system, each process receives the token infinitely often.
- Illegitimate configuration
  - Example: if process grasps token, but does not release it, then the first criterion of the legitimate config is true, but the second criterion is not satisfied, hence configuration becomes illegitimate.

Self-stabilizing Systems

Recover from any initial configuration to a legitimate configuration in a bounded number of steps, as long as the protocols (codes) are not corrupted

Assumptions:

- failures affect the state (and data) but not the program since program executes the self-stabilization;
- Such systems can be deployed ad hoc, and are guaranteed to function properly in bounded time
- Guarantees fault tolerance when the Mean Time Between Failures (MTBF) >> Mean Time To Recovery (MTTR)
  - Stabilization provides solution when failures are infrequent and temporary malfunctions are acceptable or unavoidable

Reasons for Illegal Configurations

- Transient failures perturb the global state. The ability to spontaneously recover from any initial state implies that no initialization is ever required.
  - Example: disappearance of the only circulating token in token ring; data corruption due to radio interference or power supply variations;

- Topology changes: topology of network changes at run time when node crashes or new node is added to the system
  - Example: peer-to-peer networks and their churn rate (dynamic networks) – see stabilization protocol in Chord

- Environmental changes: environment of a program may change without notice
  - Example: traffic lights in city may run different programs depending on volume and distribution of traffic. If system runs “early morning program” in the afternoon rush hours, we have illegal configuration.

Self-Stabilizing Systems

Self-stabilizing systems exhibits non-masking fault-tolerance

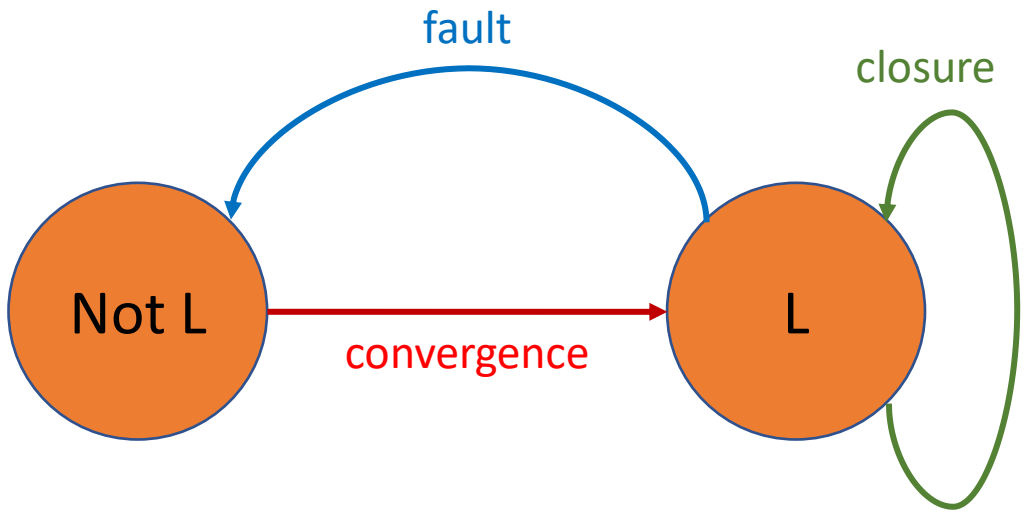
They satisfy the following two criteria

- convergence

  regardless of initiate state, the system eventually returns to legal configuration

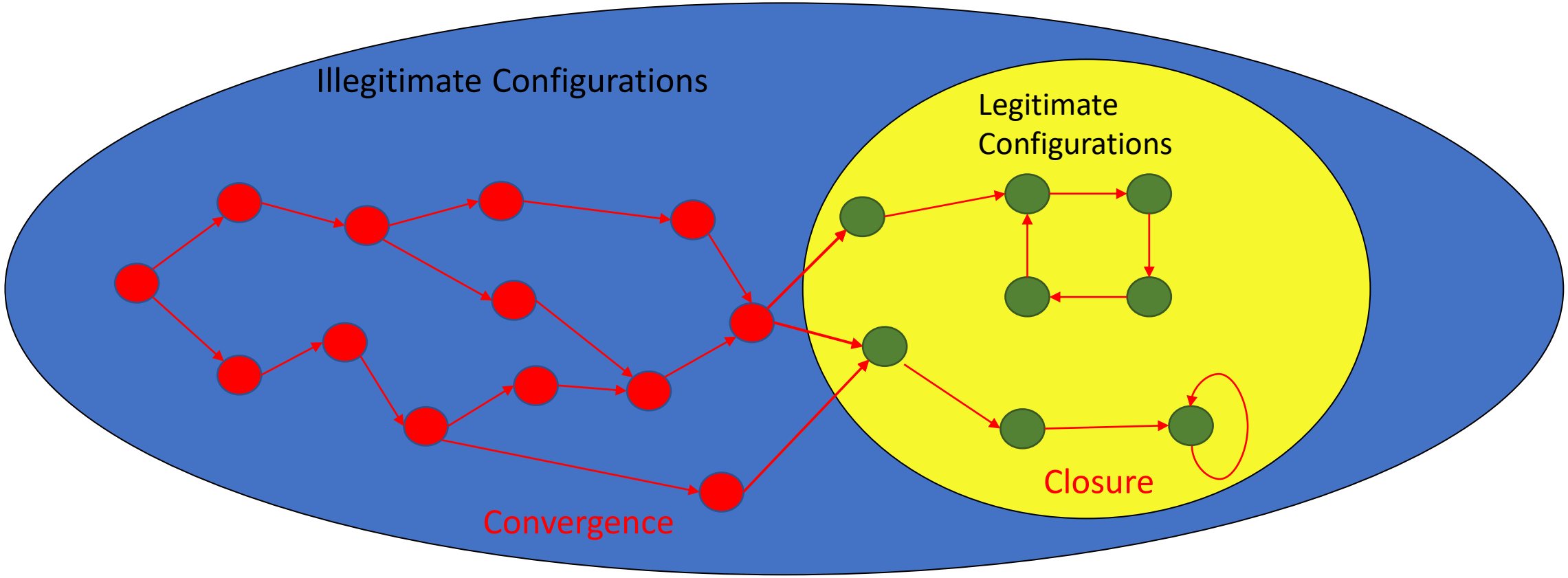
- closure

  once in legal configuration, system continues in legal configuration unless failure or perturbation corrupts data memory



L: Legitimate configuration
Not L: Illegitimate configuration

Definition: Convergence + Closure



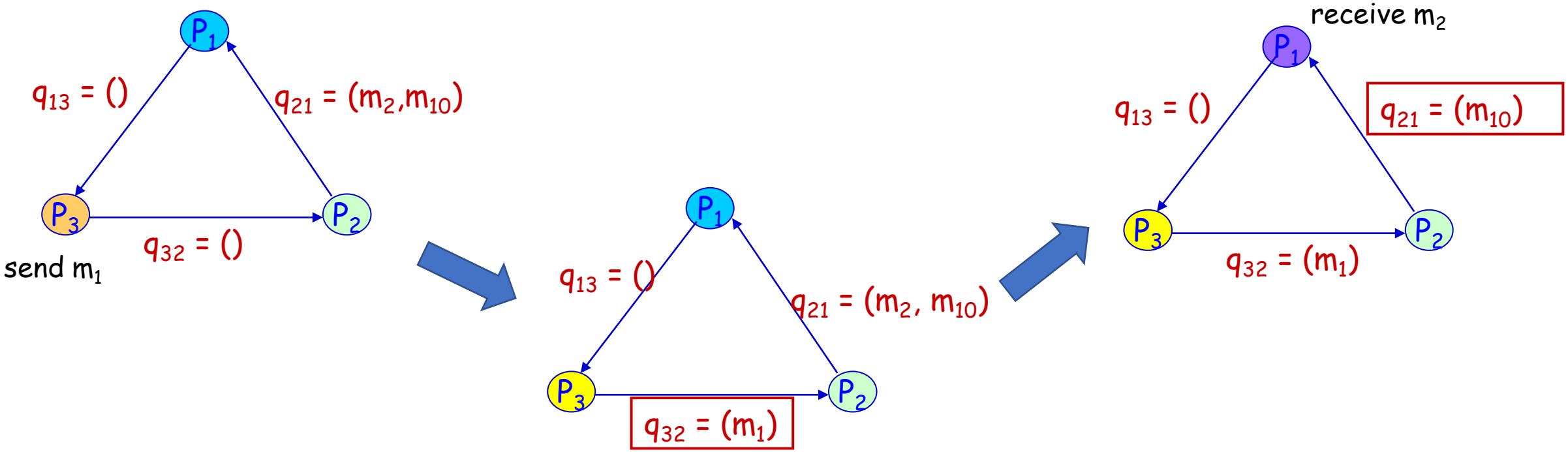
Configurations of the System

Back To Models ☺

Asynchronous Computation

Asynchronous Distributed Systems – Message Passing

- A communication link which is unidirectional from P_i to P_j transfers message from P_i to P_j
- For a unidirectional link we will use the abstract q_{ij} (a FIFO queue)



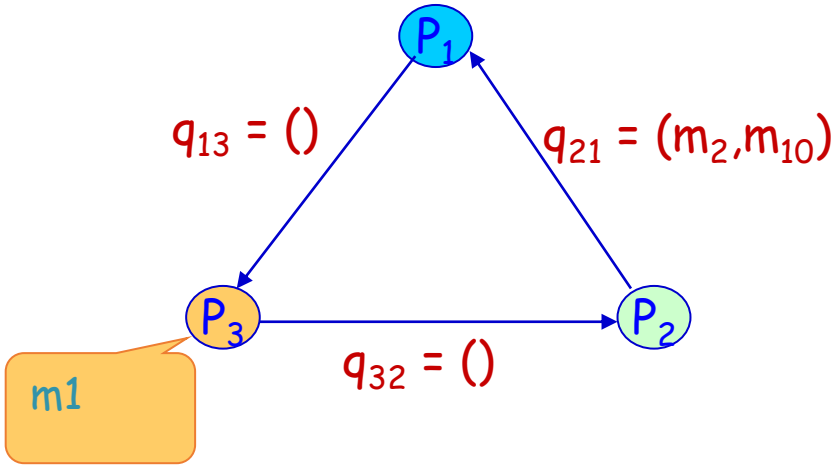
Asynchronous Distributed Systems - Message passing

System configuration: Description of a distributed system at a particular time.

A configuration will be denoted by

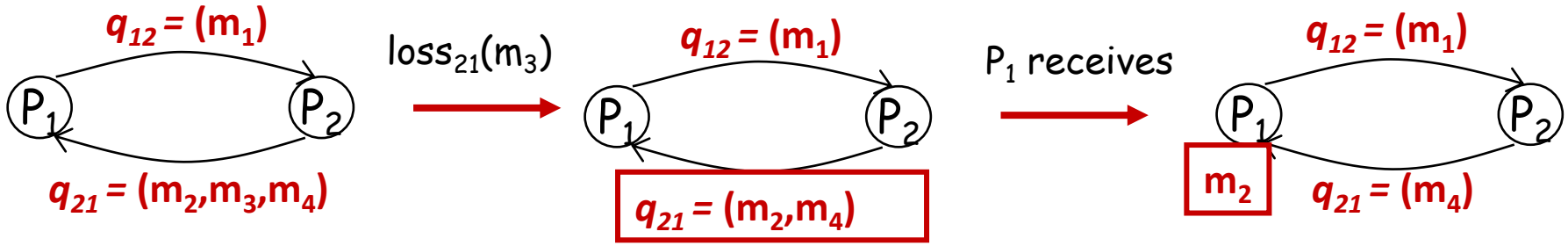
C = $(s_1, s_2, \ldots, s_n, q_{12}, q_{13}, \ldots, q_{ij}, \ldots, q_{n,n-1})$, where

- s_i =State of P_i
- $q_{i,j}$ $(i \neq j)$ the message queue



The Distributed System – A Computation Step

- Computation step (atomic step): Internal Computation + Single communication operation
- Every state transition of a process is due to communication-step execution
- A step will be denoted by a
- $c_1 \rightarrow^a c_2$ denotes the fact that c_2 can be reached from c_1 by a single step a
- An execution $E = (c_1, a_1, c_2, a_2, \dots)$, is an alternating sequence such that $c_{i-1} \rightarrow^{a_i} c_i \ (i > 1)$



The Daemon

(in a less spooky manner: Scheduler)

The asynchronism of the system is modelled by a non-deterministic adversary called daemon.

- Decides which running/enabled nodes are activated in each step

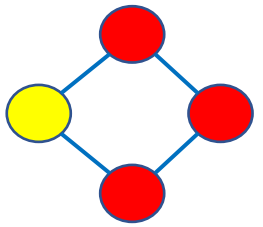
Progress Property (Proper Daemon):

The configuration of the distributed system changes in every step.

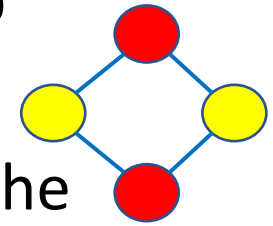
They are defined by their spreading and fairness.

Spreading of a Daemon

The choice of the daemon at each step is oblivious

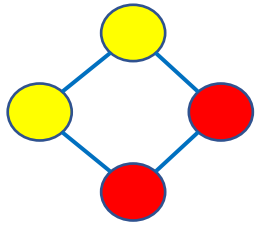


- Central (sequential) Daemon: Only one enabled node activated per step

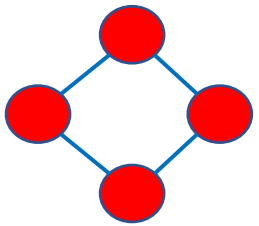


- Locally Central Daemon: It does not activate two enabled neighbors in the same step

- Distributed: at least one enabled node in each step without restrictions



- Synchronous: all enabled nodes are activated in each step

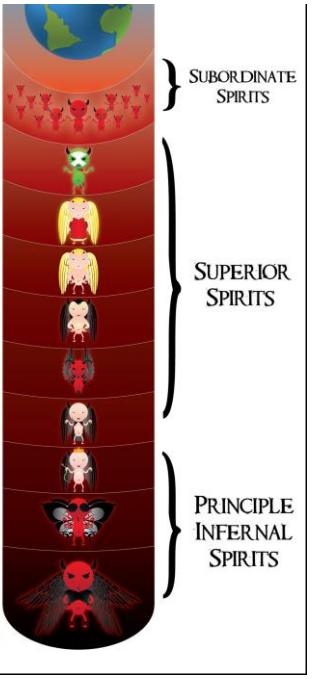


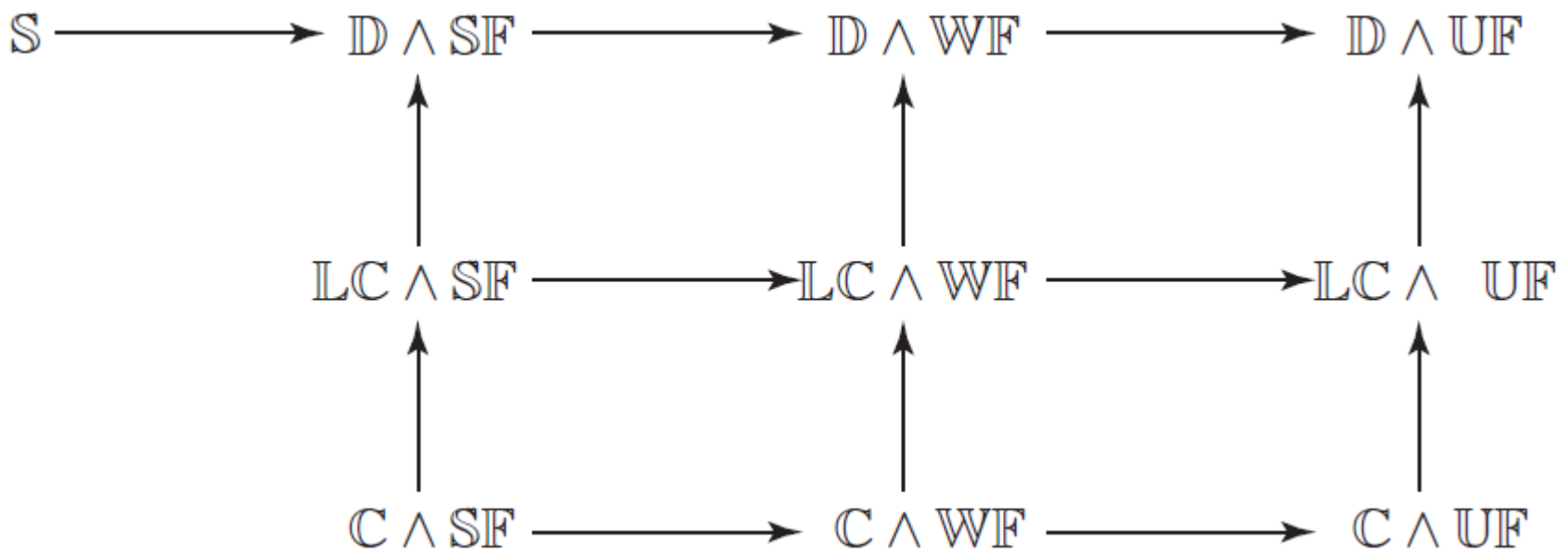
Fairness of a Daemon

Regulates the relative activation rate of nodes by taking past actions into account.

- Strongly Fair Daemon: It activates infinitely often all nodes that are enabled infinitely often
- Weakly Fair Daemon: It eventually activates any continuously enabled node
- Unfair Daemon: No restrictions (it will activate a node if it is the only enabled – due to progress property)

A Daemon Hierarchy





Time Complexity

- The first asynchronous round (round) in an execution E is the shortest prefix E' of E such that each node executes at least one step in E', E=E'E''.
- The number of rounds = time complexity

- A Self-Stabilizing algorithm is usually a do forever loop
- The number of steps required to execute a single iteration of such a loop is $O(\Delta)$, where Δ is an upper bound on the number of neighbors of P_i

- Asynchronous cycle (cycle) the first cycle in an execution E is the shortest prefix E' of E such that each node executes at least one complete iteration of it's do forever loop in E', E=E'E''.
- Note : each cycle spans $O(\Delta)$ rounds

- The time complexity of a synchronous algorithm is the number of pulses/ticks in the execution

Space complexity

- The space complexity of an algorithm is the total number of memory bits used to implement the algorithm

Some Formalities

Self-Stabilization (Definition)

A is self-stabilizing over the terminal predicate SP in network G under daemon D if there exists a non-empty subset of all configurations $L \subseteq C$, called the legitimate configurations (the rest are the illegitimate configurations) such that:

1. L is closed (by A in G under D): $\forall \delta \in L$ and $\forall \delta' \in C$: if $\delta \rightarrow \delta'$, then $\delta' \in L$
2. A converges under D to L *in* G: $\forall e \in E, \exists \delta \in e$ such that $\delta \in L$
3. Under D, SP is satisfied from L: $\forall e \in E(L)$, $SP(e)$ holds, where $E(L)$ is the subset of executions of E that starts from a configuration in L

Silence

A self-stabilizing algorithm is silent if it converges within finite time to a configuration from which the values of the communication variables used by the algorithm remain fixed.

In our setting: All executions in the considered networks under the considered daemon are finite.

A is silent and self-stabilizing for the configuration predicate SP in network G under daemon D if:

1. (Termination) $\forall e \in E, e$ is finite
2. (Partial Correctness) Every terminal configuration satisfies SP

Self-Stabilizing Algorithm: A Scaffolding

General Self-Stabilizing Algorithm

1. If $\langle predicate \rangle$ then
    1. the system is in a legitimate state
2. else
    1. If $\langle guard \rangle$ then
        1. $\langle action \rangle$

Self-Stabilization Algorithms - Coloring

(not again ☹)

Coloring under a Locally Central Unfair Daemon

Assumptions:

- No Ids
- Knowledge of Max Degree

If the power of the Daemon is not restricted, then the problem cannot be solved by any deterministic self-stabilizing algorithm.

RESTRICT THE DAEMON!!!

Algorithm

Inputs:

$p.N$: the set of p's neighbors

K : an integer such that $K \geq \Delta$

Local Variable:

$p.c \in \{0,1,\dots,K\}$: the color of p

Macros:

$Used(p)$: $\{q.c : q \in p.N\}$

$Free(p)$: $\{0,\dots,K\} \backslash Used(p)$

Guard:

$Conflict(p)$: $\exists q \in p.N : q.c = p.c$

Action:

$Color$: $Conflict(p) \rightarrow \big(p.c \leftarrow \min(Free(p))\big)$

Terminal Configuration (predicate):

$Colored$: $\forall p \in V, \forall q \in p.N : p.c \neq q.c$

Variant(Potential) Function

A threshold below which we are in a safe configuration

$$|VF(c)| \geq |VF(c_1)| \geq |VF(c_2)| \geq |VF(c_3)| \geq \ldots \geq |VF(c_{safe})| \geq \ldots \geq \text{bound}$$

- Used for proving convergence
- Can be used to estimate the number of steps required to reach a legitimate/safe configuration

Analysis (1)

Partial Correctness:

1. The predicate $Colored$ holds in every terminal configuration.

Termination (let an execution $e = \delta_0, \delta_1, \ldots, \delta_i, \ldots$):

1. Let p a node. In any configuration, for any node p, $Free(p) \neq \emptyset$.
2. Variant function (def): $Energy(\delta_i) = |\{p \in V : Conflict(p) \in \delta_i\}|$
3. If $Conflict(p)$ holds in δ_i then there exists $q \in p.N$ such that $Conflict(q)$ holds in δ_i as well (δ_i is a configuration)
4. Either $Energy(\delta_i) = 0$ or $2 \leq Energy(\delta_i) \leq n$
5. $Energy(\delta_i) = 0$ if and only if δ_i is terminal
6. For every node p and every step $\delta_i \rightarrow \delta_{i+1}$, if $\neg Conflict(p)$ holds in δ_i, then $\neg Conflict(p)$ holds in δ_{i+1}.

Analysis (2) - Termination

7. For every 2 configurations δ_i and δ_j such that $i \leq j$ it holds that $Energy(\delta_i) \geq Energy(\delta_j)$
8. For every node p and every step $\delta_i \rightarrow \delta_{i+1}$, if node p is activated in $\delta_i \rightarrow \delta_{i+1}$, then $Conflict(p)$ holds in δ_i but not in δ_{i+1}.
9. For every step $\delta_i \rightarrow \delta_{i+1}$, if $Energy(\delta_i) = 2$, then exactly one process is activated in $\delta_i \rightarrow \delta_{i+1}$ and $Energy(\delta_{i+1}) = 0$.
10. For every step $\delta_i \rightarrow \delta_{i+1}$, we have that $Energy(\delta_i) \geq 2$ and $Energy(\delta_{i+1}) < Energy(\delta_i)$
11. The execution e terminates after at most $n - 1$ steps.
12. For every execution e, e is finite (Termination)

Time Complexity

1. The Stabilization Time for this algorithm is one round ☺

2. The Stabilization Time for this algorithm is $n - 1$ steps

Self-Stabilization Algorithms - MIS

(not again ☹)

MIS – Spot The Difference (Nodes have IDs)

Non-stabilizing Algorithm for MIS:

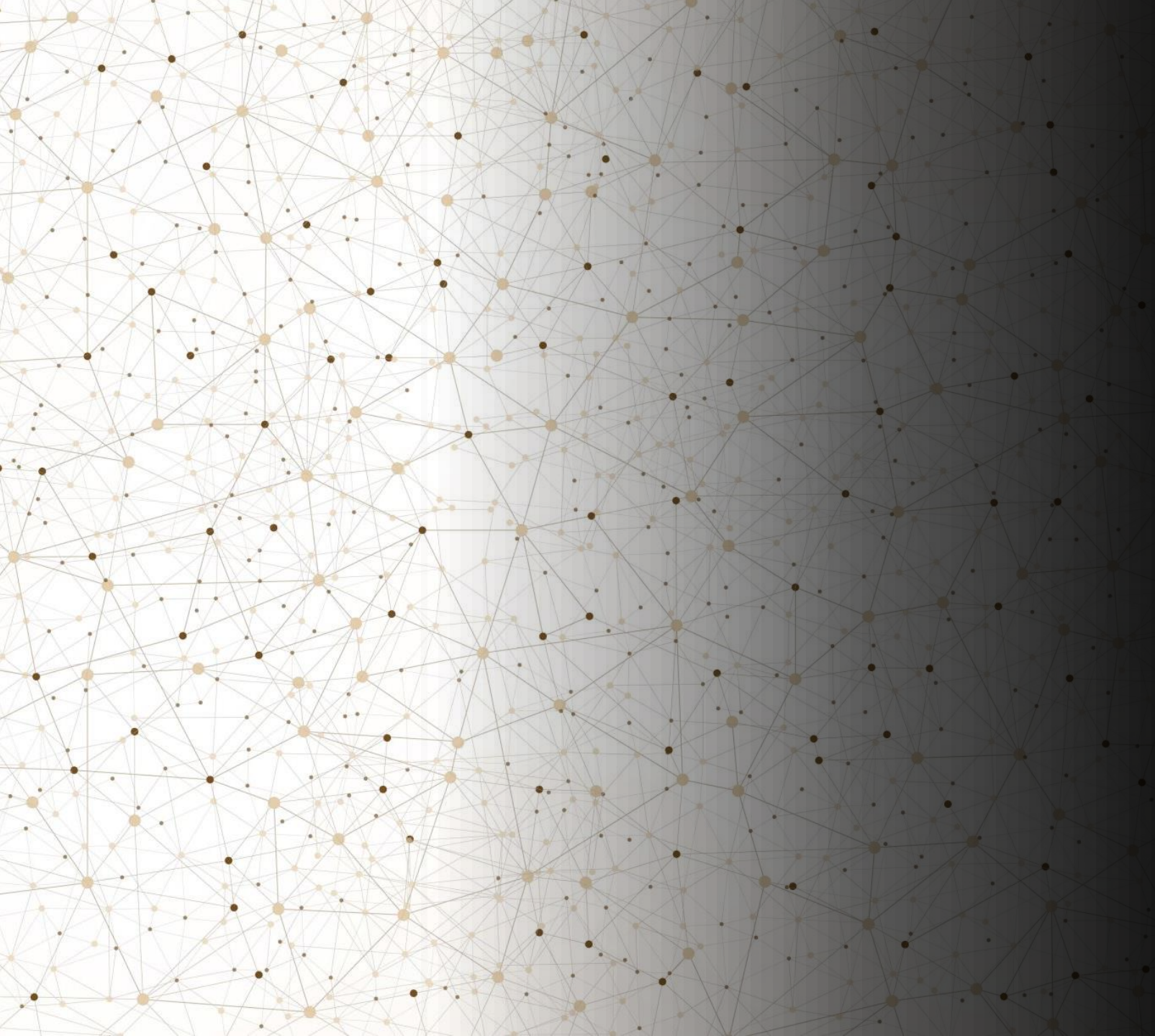
Every node v executes the following code:

1. If all neighbors of v with larger identifiers have decided not to join the MIS, then
    1. v decides to join the MIS

Stabilizing Algorithm for MIS:

Every node v executes the following code:

1. do iteratively:
    1. Leave MIS if a neighbor with a larger ID is in MIS
    2. Join MIS if no neighbor with larger ID joins MIS
    3. Send (node ID, MIS or not MIS) to all neighbors



Self-Stabilization Algorithms – Mutual Exclusion

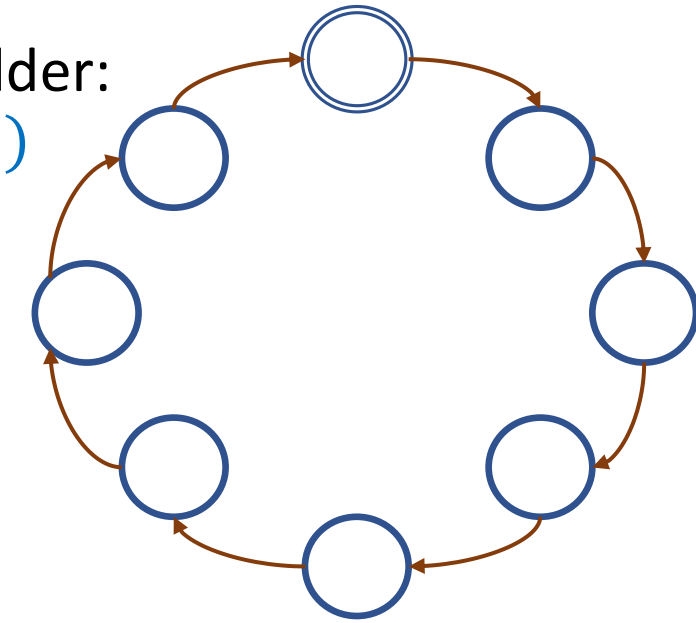
What?

The Problem

Requirement for the infinite circulation of a token in a ring network.

Predicate:

1. (Safety) In each configuration, there is at most one token holder:
$$\forall i \geq 0, \forall p, q \in V: \big(Token(p) \wedge Token(q)\big) \rightarrow (p = q)$$
2. (Liveness) Each process holds the token infinitely often:
$$\forall i \geq 0, \forall p \in V, \exists j \geq i: Token(p) \in \delta_j$$



Assumptions:

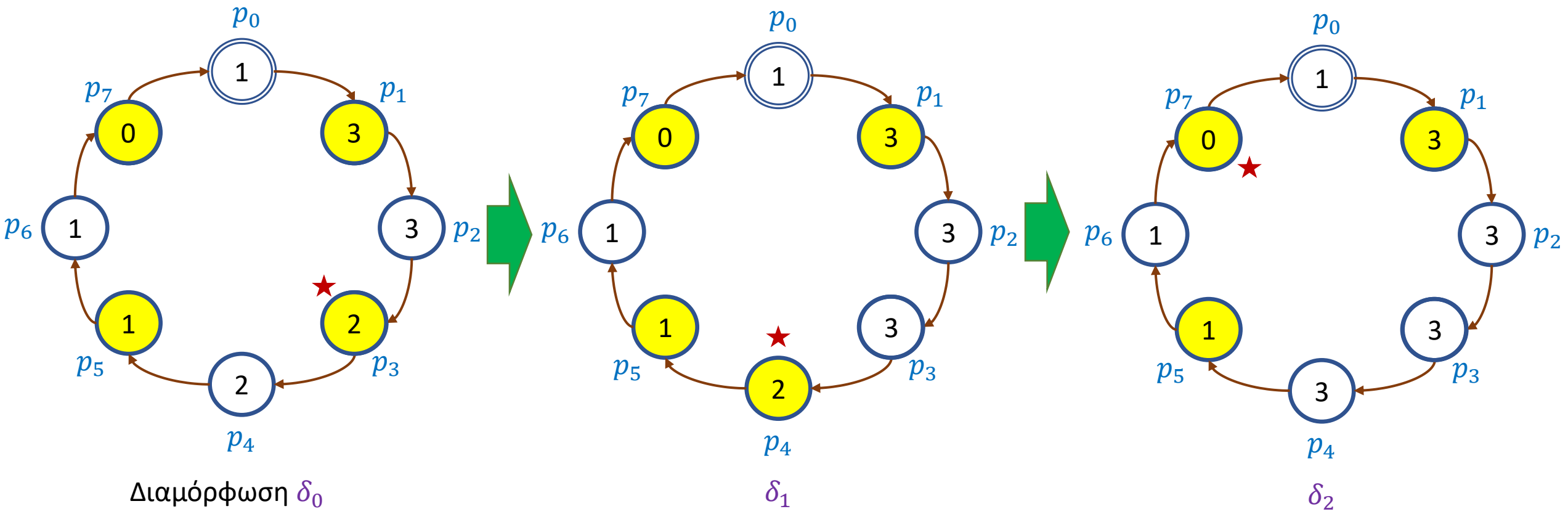
1. Distributed Unfair Daemon
2. Ring is rooted
3. Ring is oriented (consistent orientation: successor of predecessor of p is always itself)

Two Algorithms

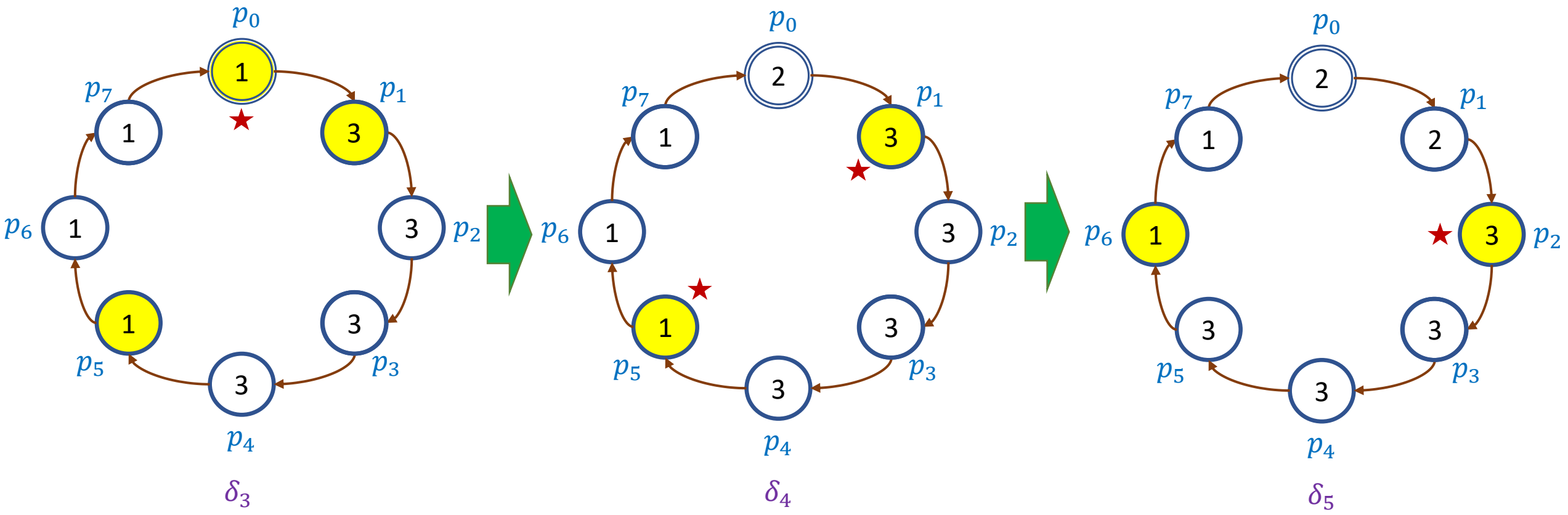
The root p_0 holds the token if $Token(p_0)$ holds.
The non-root node p holds the token if $Token(p)$ holds.

Algorithm for the Root p_0	Algorithm for non-root node p
Inputs: $p_0.Pred$: the predecessor of p_0 in the ring K : a positive integer $K > n$	Inputs: $p.Pred$: the predecessor of p in the ring K : a positive integer $K > n$
Local Variable: $p_0.v \in \{0,1,\dots,K-1\}$	Local Variable: $p.v \in \{0,1,\dots,K-1\}$
Guard: $Token(p_0) : p_0.v = p_0.Pred.v$	Guard: $Token(p) : p_0.v \neq p_0.Pred.v$
Action: $T : Token(p_0) \rightarrow (p_0.v \leftarrow (p_0.v + 1) mod\ K)$	Action: $T : Token(p) \rightarrow (p.v \leftarrow p.Pred.v)$

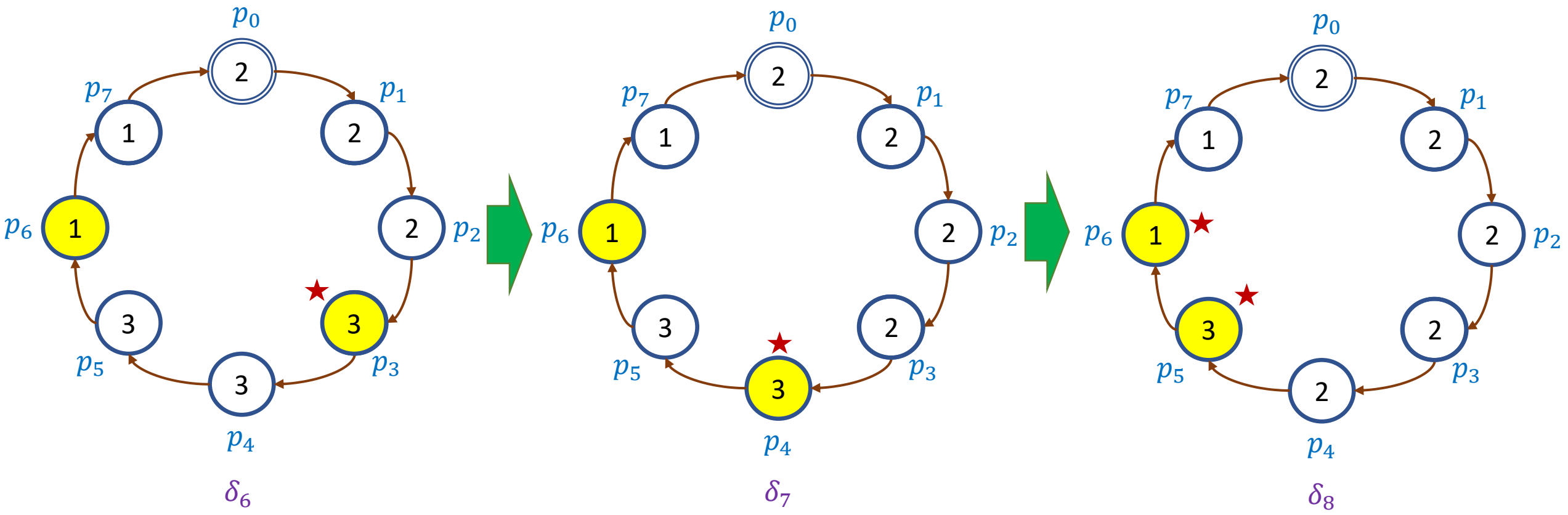
Convergence $(n = 8, K = 9)$



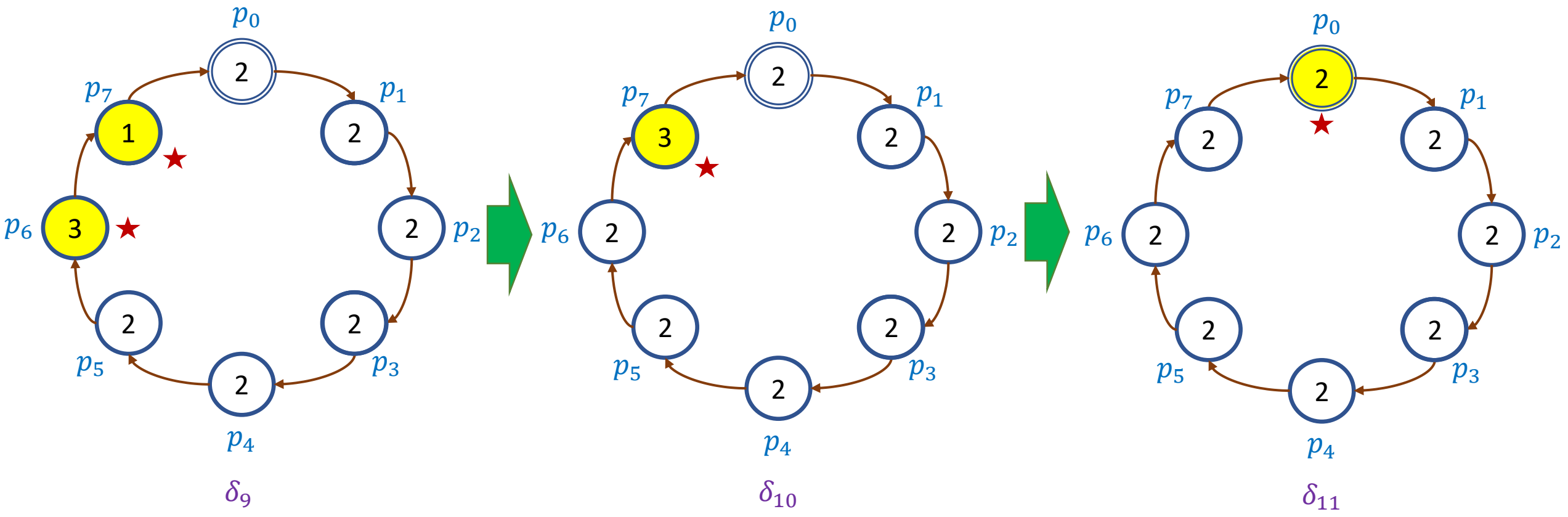
Convergence $(n = 8, K = 9)$



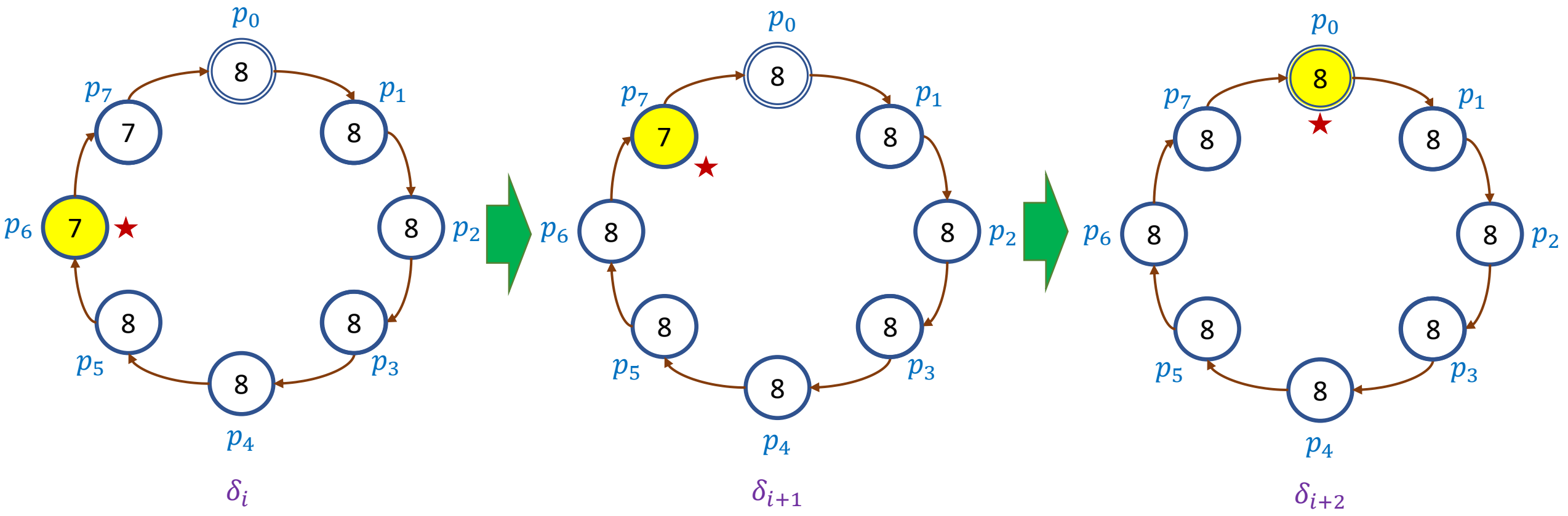
Convergence $(n = 8, K = 9)$



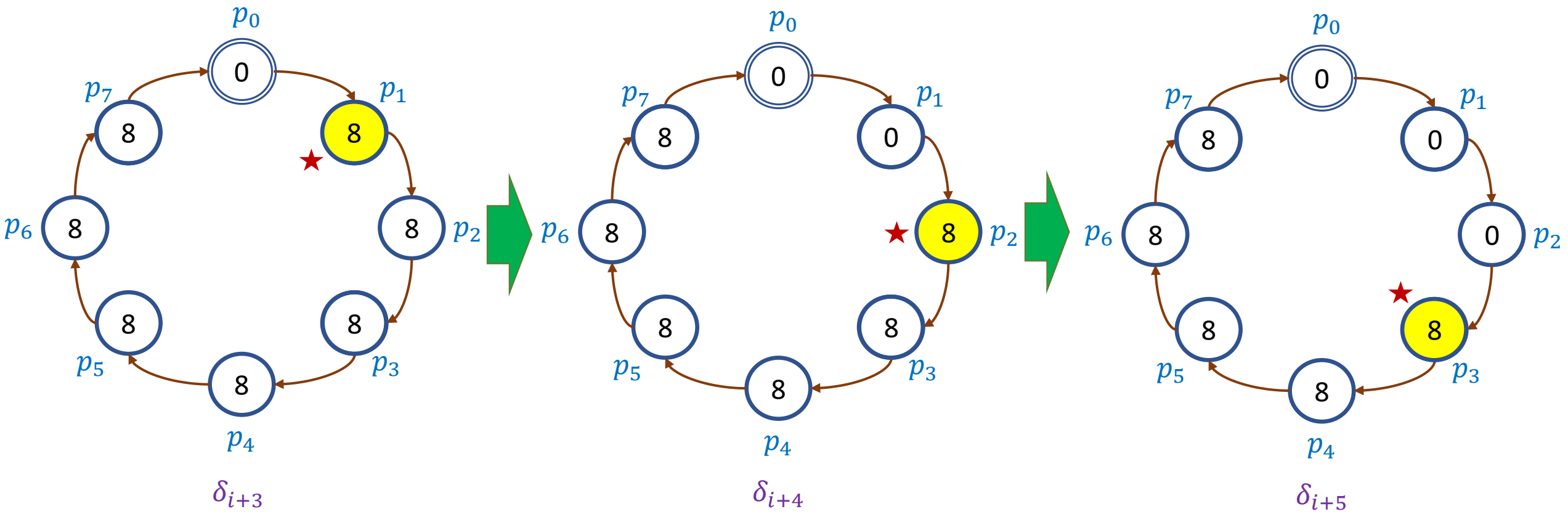
Convergence $(n = 8, K = 9)$



Closure $(n = 8, K = 9)$



Closure $(n = 8, K = 9)$



Why does it work ?

1. At any configuration, at least one process can make a move (has token)
2. Set of legal configurations is closed under all moves
3. Total number of possible moves from (successive configurations) never increases
4. Any illegal configuration C converges to a legal configuration in a finite number of moves

Why does it work ?

1. At any configuration, at least one process can make a move (has token), i.e., if condition is false at all processes

Proof by contradiction: suppose no one can make a move

- Then $p_1, \ldots, p_{n-1}$ cannot make a move
- Then $p_1.v = p_2.v = p_{N-1}.v = p_0.v$
- But this means that p_0 can make a move. Contradiction

Why does it work ?

1. At any configuration, at least one process can make a move (has token)
2. Set of legal configurations is closed under all moves
   - If only p_0 can make a move, then $\forall i, j: p_i.v = p_j.v$. After p_0's move, only p_1 can make a move
   - If only p_i $(i \neq 0)$ can make a move
     - $\forall i, j: (j < i) \rightarrow (p_j.v = p_{i-1}.v)$
     - $\forall i, k: (k \geq i) \rightarrow (p_k.v = p_i.v)$
     - $p_{i-1}.v \neq p_i$ $(i \neq 0)$
     - $p_o \neq p_{n-1}$

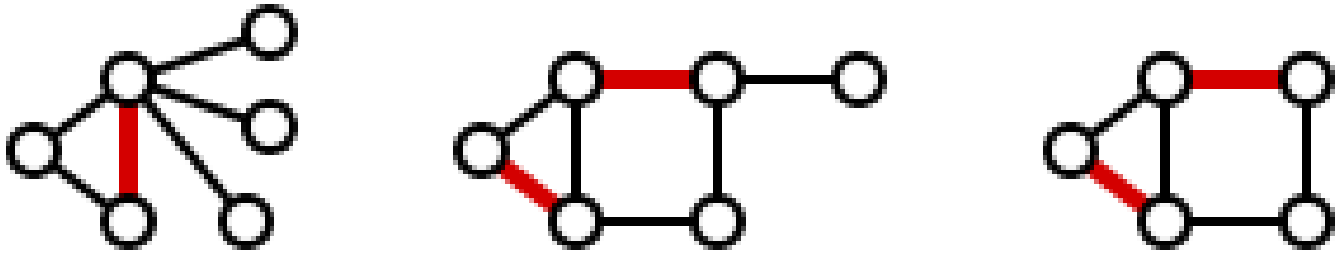
     in this case, after p_i's move only p_{i+1} can move

Why does it work ?

1. At any configuration, at least one process can make a move (has token)
2. Set of legal configurations is closed under all moves
3. Total number of possible moves from (successive configurations) never increases
   - any move by p_i either enables a move for $p_{i+1\ (mod\ n)}$ or none at all

Why does it work ?

1. At any configuration, at least one process can make a move (has token)
2. Set of legal configurations is closed under all moves
3. Total number of possible moves from (successive configurations) never increases
4. Any illegal configuration C converges to a legal configuration in a finite number of moves
   - There must be a value, say x, that does not appear in C (since the set of values is in the range $\{0,1,\dots,K\}$ and $K \geq n$)
   - Except for p_0, none of the processes create new values (since they only copy values)
   - Thus, p_0 takes infinitely many steps, and since it only self-increments, it eventually sets $p_0.v = x$
   - Soon after, all other processes copy value x and a legal configuration is reached



Self-Stabilization Algorithms – Maximal Matching

Wait a minute… ☺

Self-Stabilizing Maximal Matching

Every node P_i tries to find a matching neighbor P_j

Program for P_i:

01 **do** forever

02 **if** $(pointer_i = null)$ **and** $(\exists P_j \in N(P_i): pointer_j = i)$ **then**

03 $pointer_i = j$

04 **if** $(pointer_i = null)$ **and** $(\forall P_j \in N(P_i): pointer_j \neq i)$ **and**

05 $(\exists P_j \in N(P_i): pointer_j = null)$ **then**

06 $pointer_i = j$

07 **if** $(pointer_i = j)$ **and** $(pointer_j = k)$ **and** $(k \neq i)$ **then**

08 $pointer_i = null$

09 **od**

Remarks - Assumptions

- The algorithm should reach a configuration in which $pointer_i = j$ implies that $pointer_j = i$
- We will assume the existence of a central daemon
- The set of legal executions MM for the maximal matching task includes every execution in which the values of the pointers of all the nodes are fixed and form a maximal matching (legitimate configurations)

Some Definitions

Program for P_i:

01 **do** forever

02 **if** $(pointer_i = null)$ **and** $(\exists P_j \in N(P_i): pointer_j = i)$ **then**

03 $pointer_i = j$ matched

04 **if** $(pointer_i = null)$ **and** $(\forall P_j \in N(P_i): pointer_j \neq i)$ **and**

free

05 $(\exists P_j \in N(P_i): pointer_j = null)$ **then**

06 $pointer_i = j$ waiting

07 **if** $(pointer_i = j)$ **and** $(pointer_j = k)$ **and** $(k \neq i)$ **then** chaining

08 $pointer_i = null$

09 **od**

Correctness

- The variant function VF(c) returns a vector (m+s,w,f,c)

  m - matched, s – single, w – waiting, f – free, c – chaining

- Values of VF are compared lexicographically

- VF(c) = (n,0,0,0) $\Leftrightarrow$ c is a safe configuration with relation to MM and to our algorithm

- Once a system reaches a safe configuration, no node changes the value of its pointer

Correctness

- In every non-safe configuration, there exists at least one node that can change the value of its pointer
- Every change of a pointer-value increases the value of VF
- The number of such pointer-value changes is bounded by the number of all possible vector values.

  The first three elements of the vector (m+s,w,f,c)

  imply the value of c, thus there at most $O(n^3)$ changes.

Advanced Stabilization

A glimpse from decentralization…

Opinion Dynamics

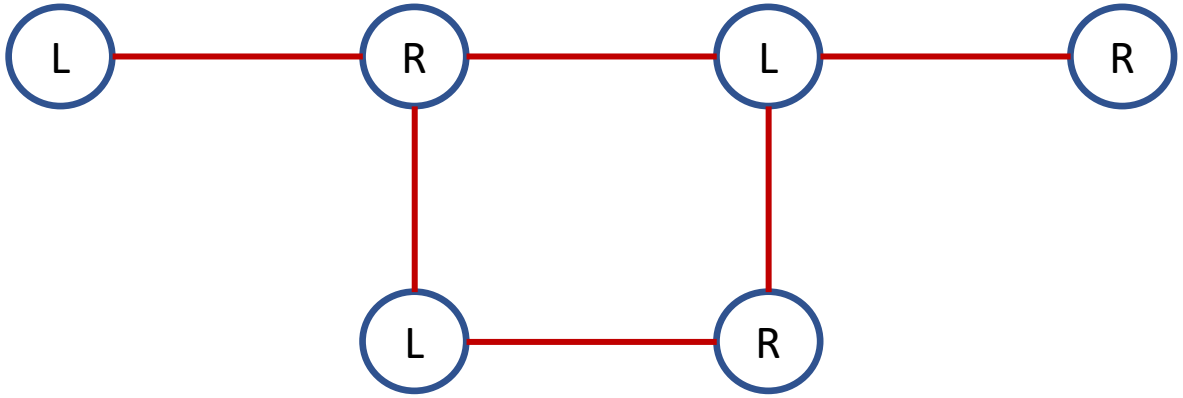
In Kalavryta, every evening each citizen calls all his (or her) friends, asking them whether they will vote for the Left or the Right party at the next election. In our village, citizens listen to their friends, and everybody re-chooses his or her vote according to the majority of friends. Is this process going to stabilize" (in one way or another)?

Remarks:

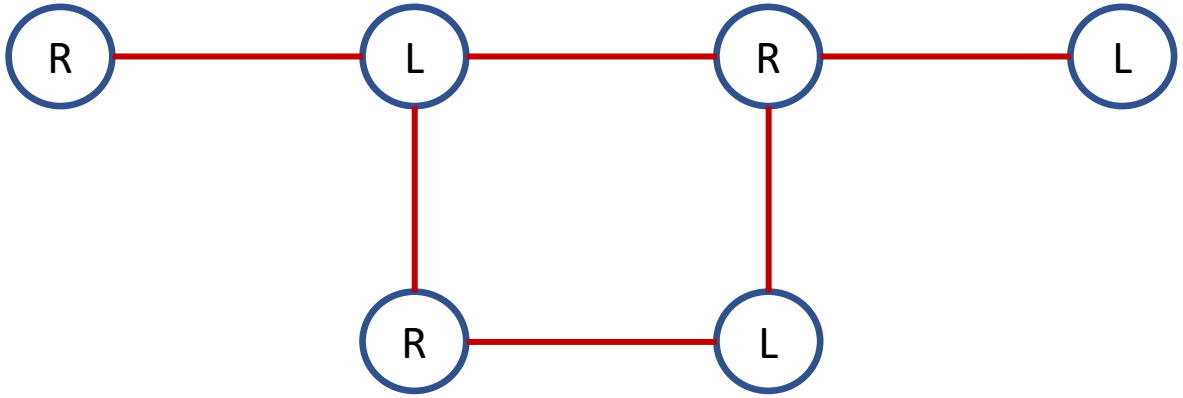
- Is eventually everybody voting for the same party? No.
- Will each citizen eventually stay with the same party? No.
- Will citizens that stayed with the same party for some time, stay with that party forever? No.
- Will this beast stabilize at all?!? Yes!

Theorem: Eventually every citizen is rooting for the same party every other day.

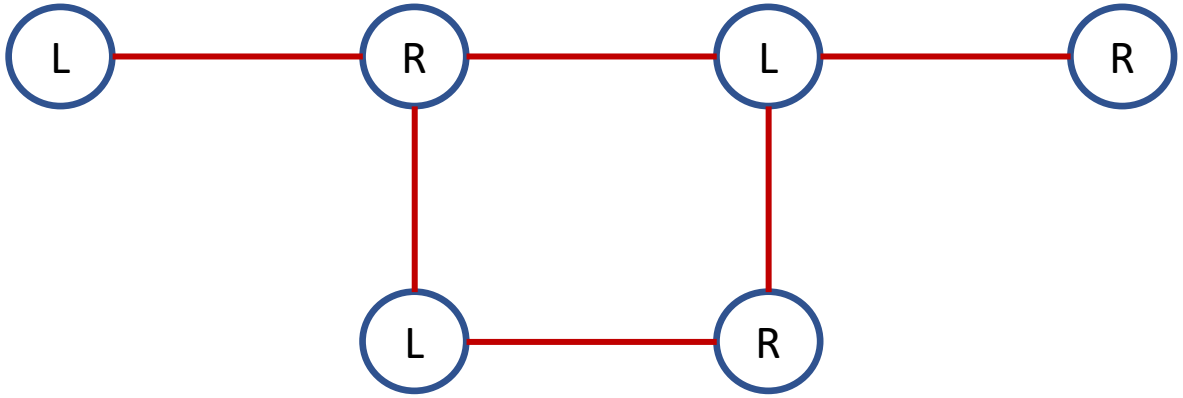
Example that flip-flops



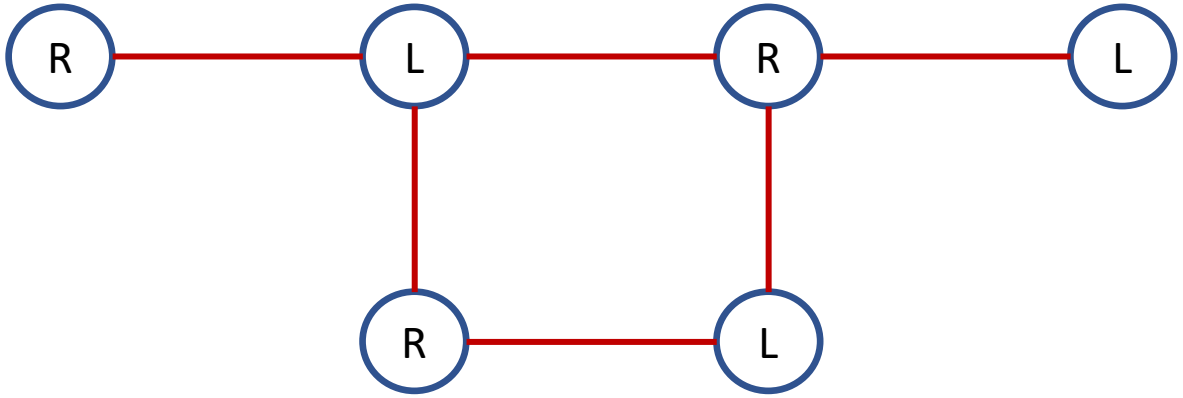
Example that flip-flops



Example that flip-flops



Example that flip-flops

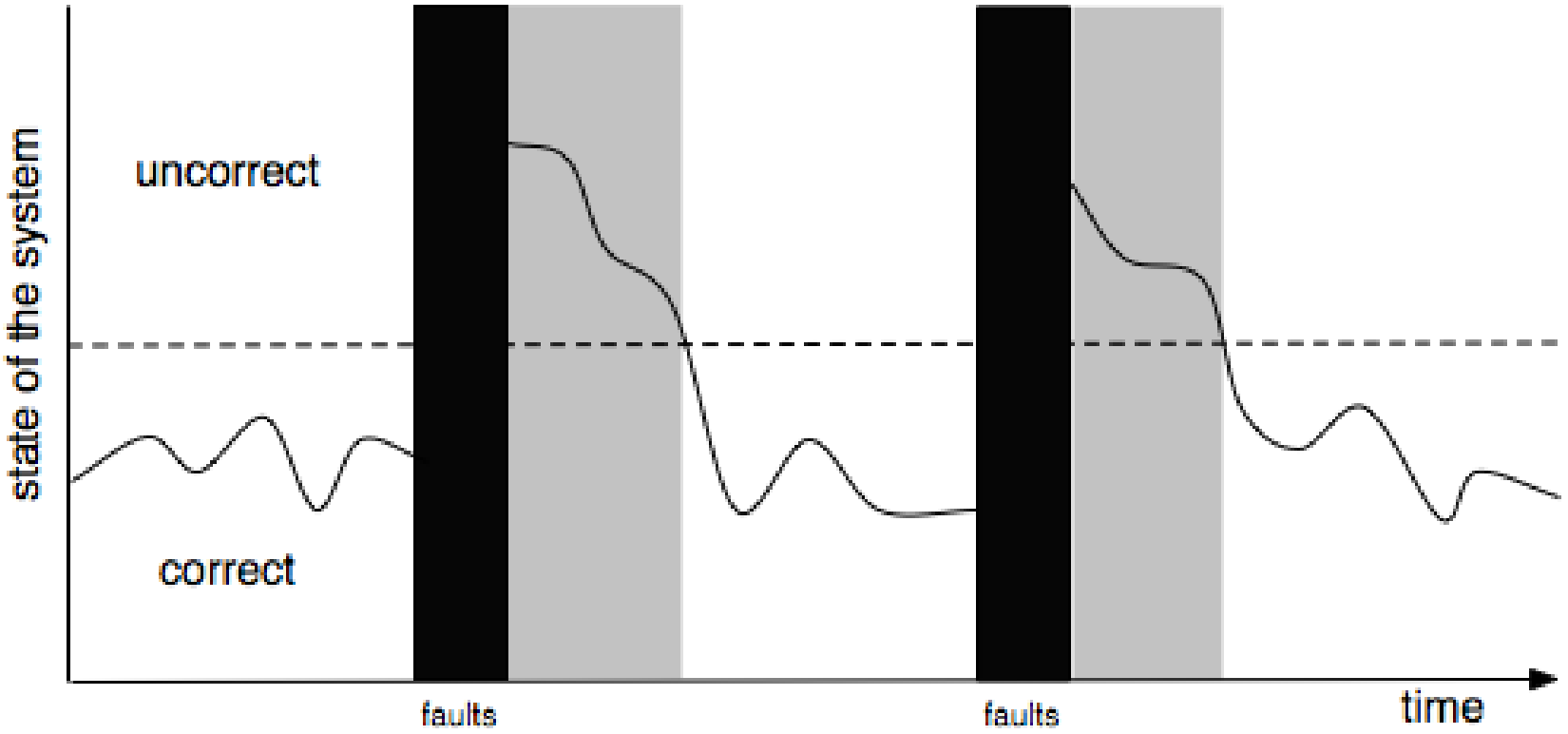



Just Before Course
Stabilization

Advantage of self-stabilization (1/3)

- Tolerance to *any* transient fault
- Transient fault:
  - Duration: *finite*
  - Periodicity: *rare*
  - Effect: *alter the contain of some component(s) of the network (processes and/or links)*
  - E.g., *memory/message corruption, crash-recover, lose of messages…*

Advantage of self-stabilization (1/3)

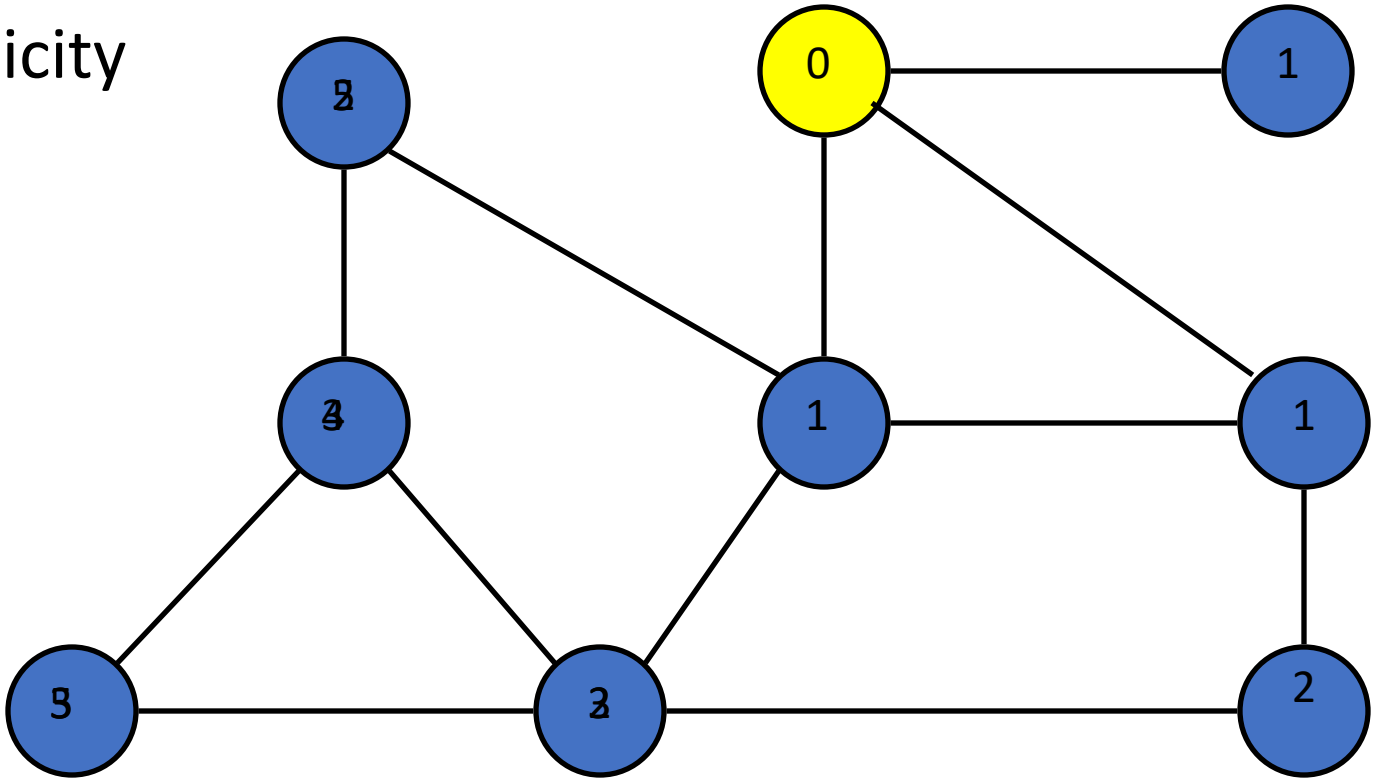


Advantage of self-stabilization (2/3)

- No initialization
  - Large-scale network
  - Self-organization in sensor network

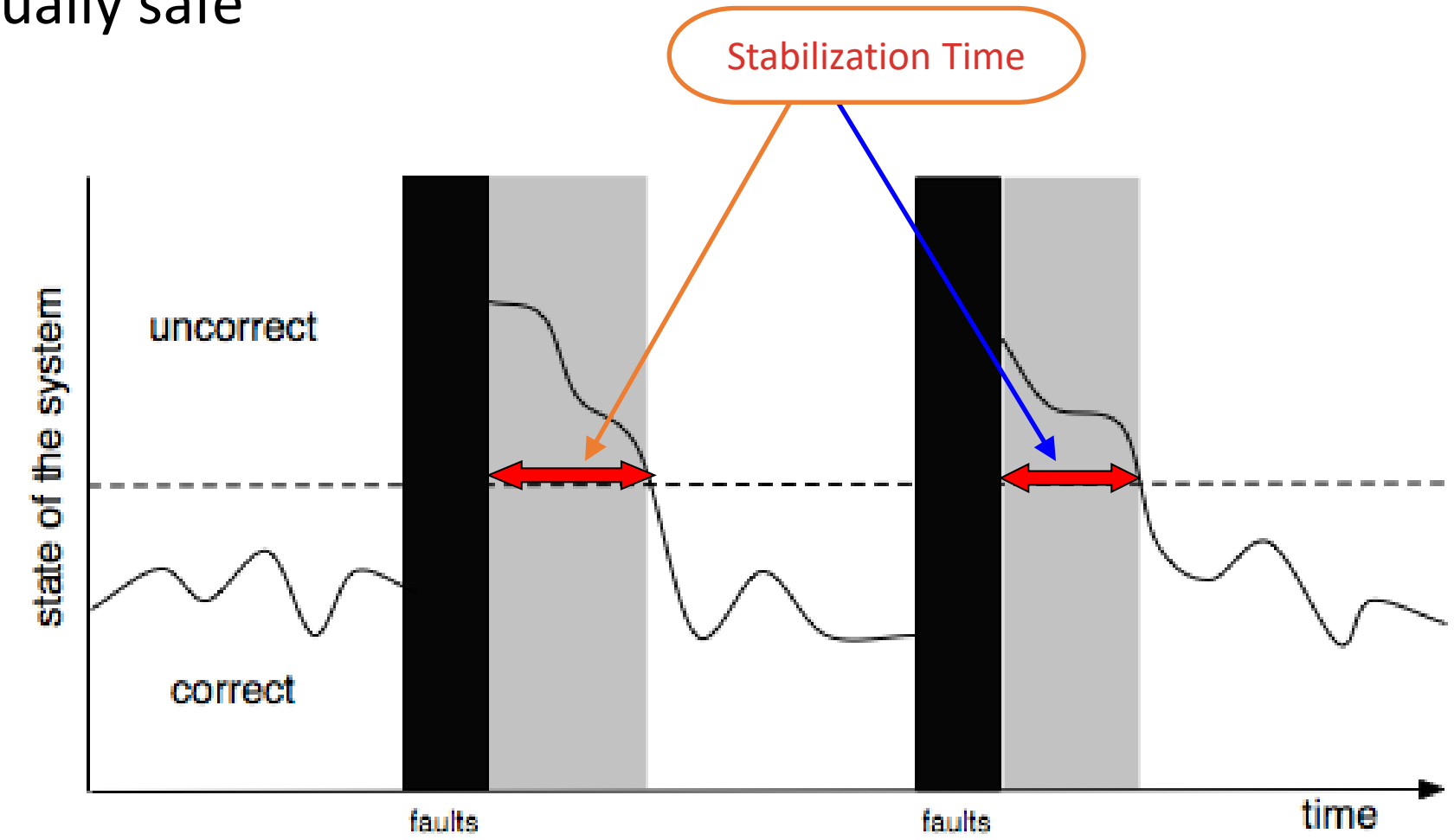
Advantage of self-stabilization (3/3)

- Dynamicity



Drawbacks of self-stabilization (1/2)

- Eventually safe

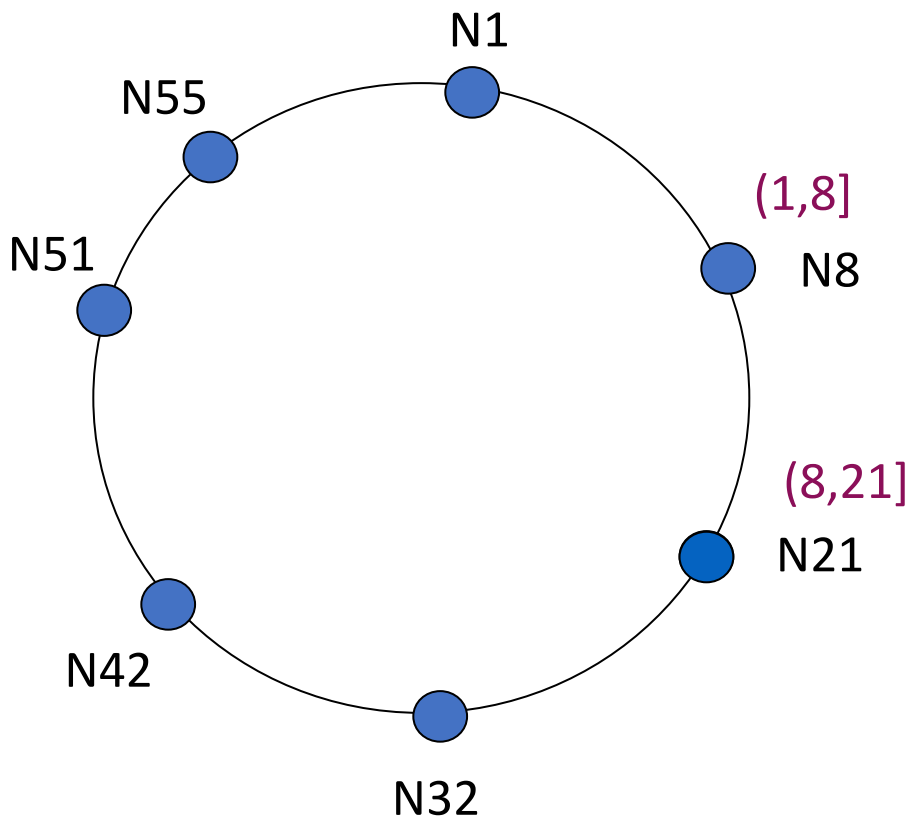


Drawbacks of self-stabilization (2/2)

- Do not tolerate *any kind* of faults, e.g.:
  - Crash
  - Byzantine faults


Making Faults
with Chord …

Chord Ring



- Associate nodes to *m*-bit identifiers

  hash(IP Addr) ⇒ Node ID

- Associate keys to *m*-bit identifiers

  hash(File Name) ⇒ key

- The key is given to the 1st node with *Node ID ≥ key*

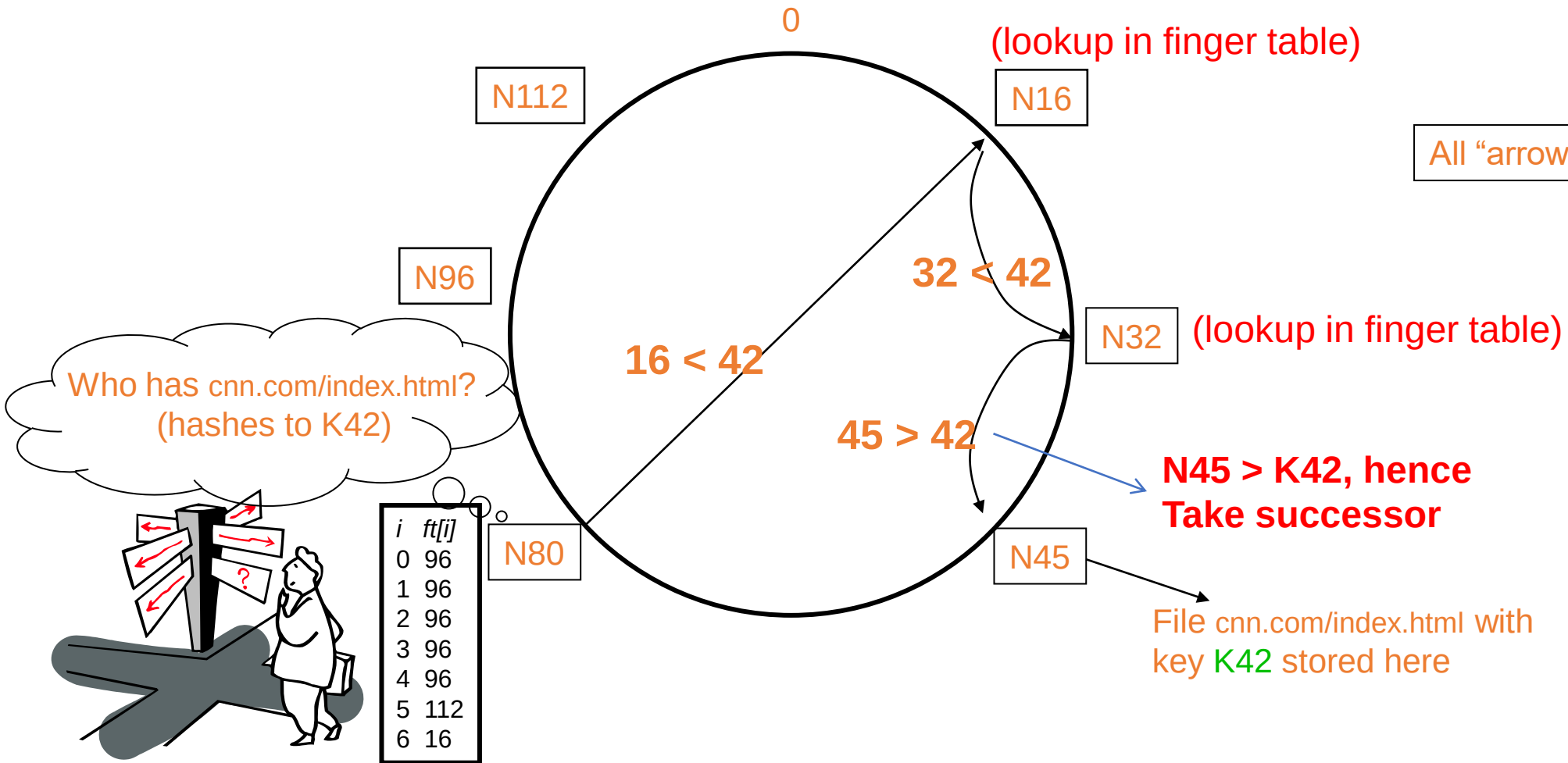
- *Search(key)*

  Search(9) ⇒ N21

P2P Systems - Chord Search

At node *n*, send query for key *k* to largest successor/finger entry < *k* (mod *m*)
If none exist, send query to *successor(n)*

Say *m*=7

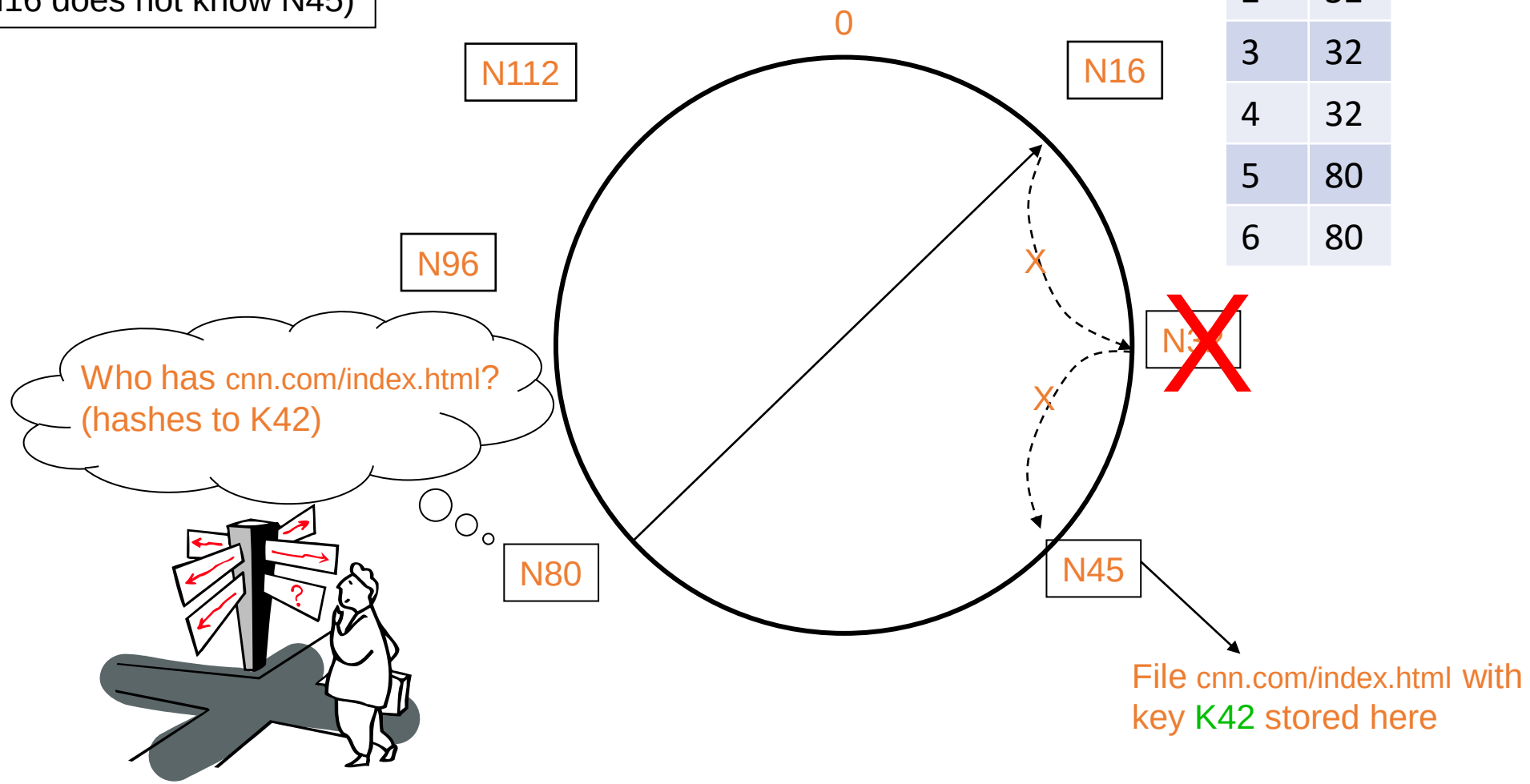


Self-Organizing Protocol in Chord

- Chord has to deal with peer churns – topological changes!!!
- Maintaining finger tables only is expensive in case of dynamic joint and leave nodes
- Chord therefore separates correctness from performance goals via stabilization protocols
- Basic stabilization protocol
  - Keep successor's pointers correct!
  - Then use them to correct finger tables

Search under peer failures

Lookup fails
(N16 does not know N45)

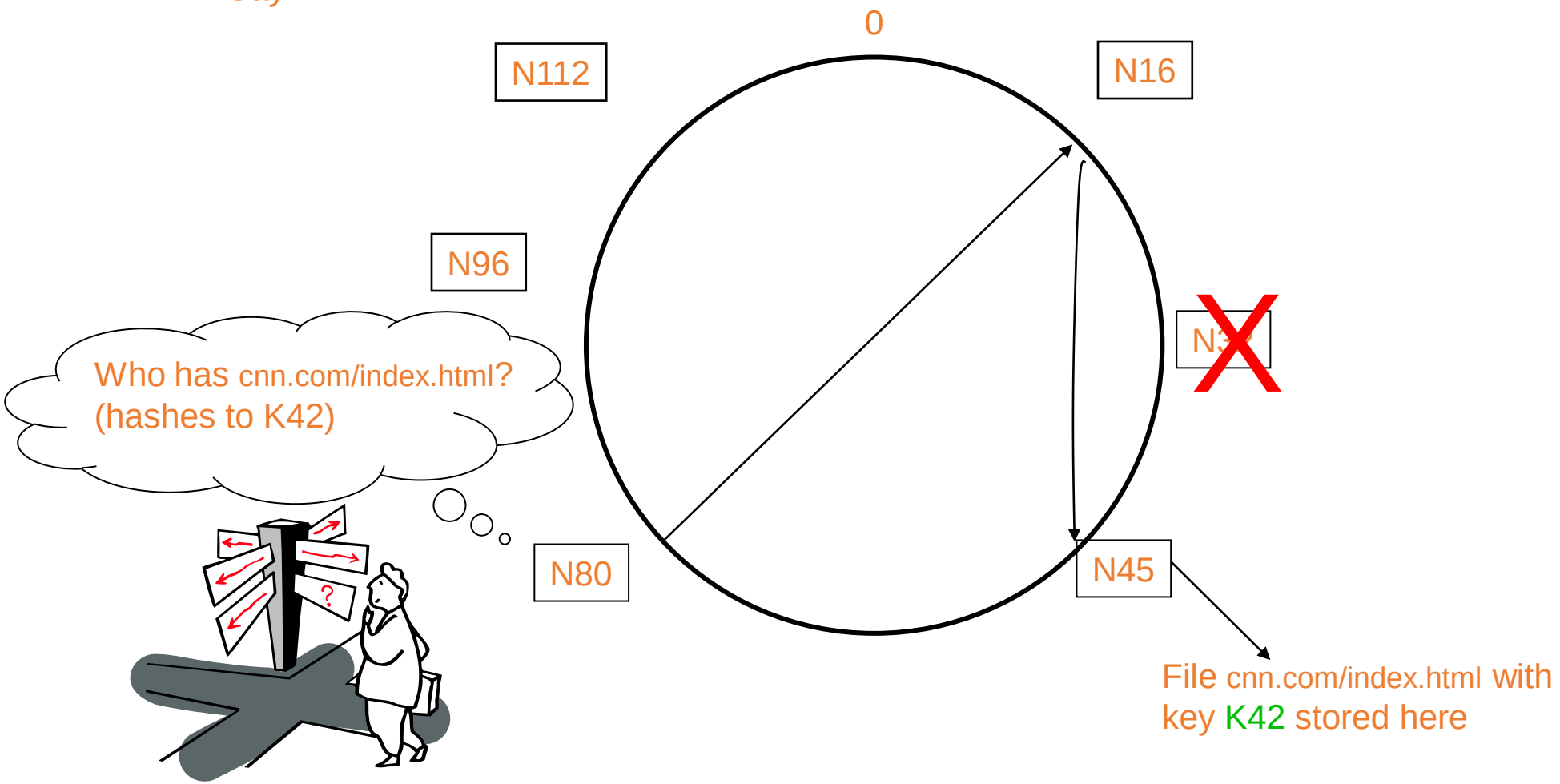


0	32
1	32
2	32
3	32
4	32
5	80
6	80

Search under peer failures

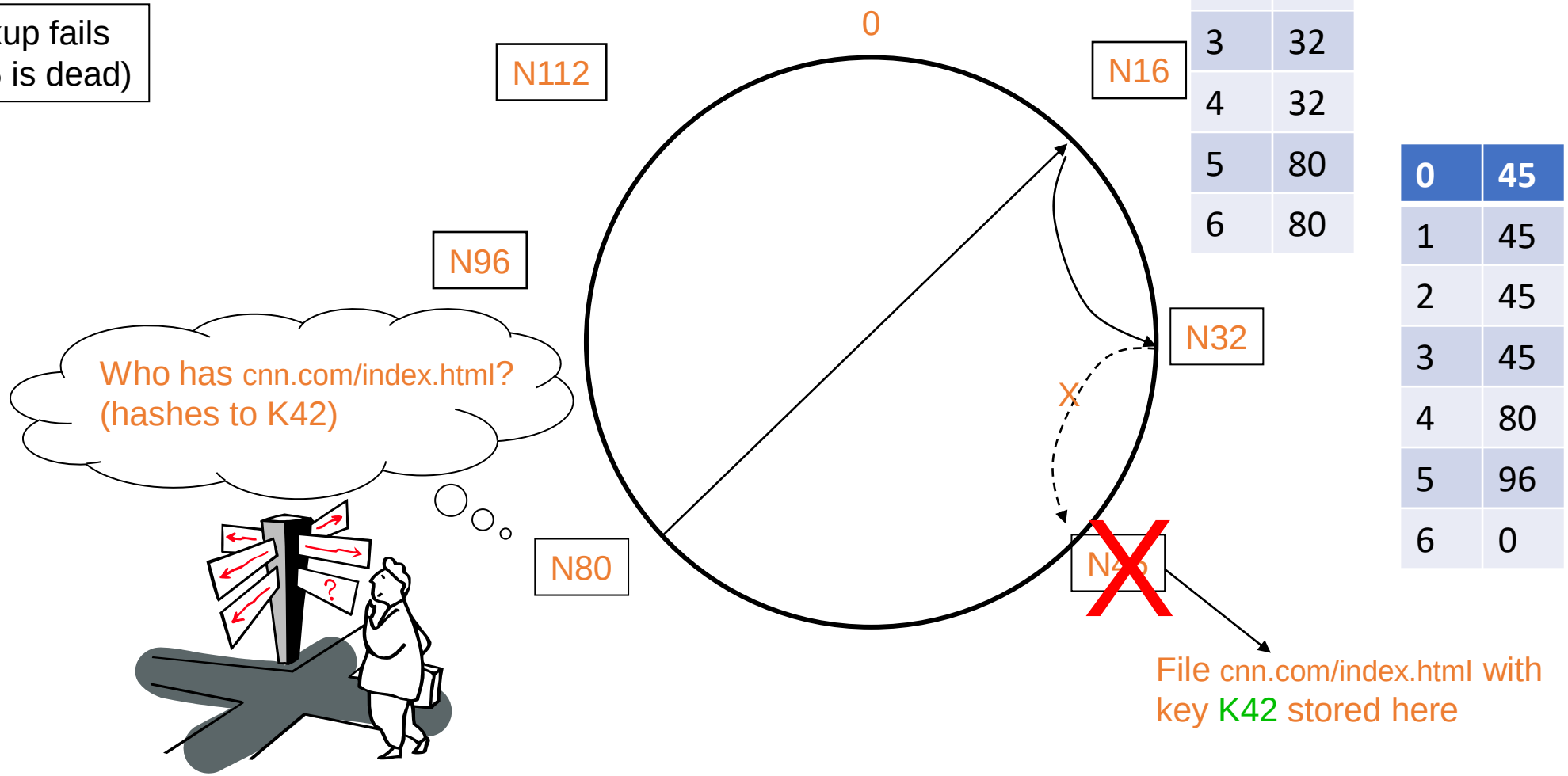
One solution: maintain *r* multiple *successor* entries in case of failure, use successor entries

Say *m=7*



Search under peer failures (2)

Lookup fails
(N45 is dead)

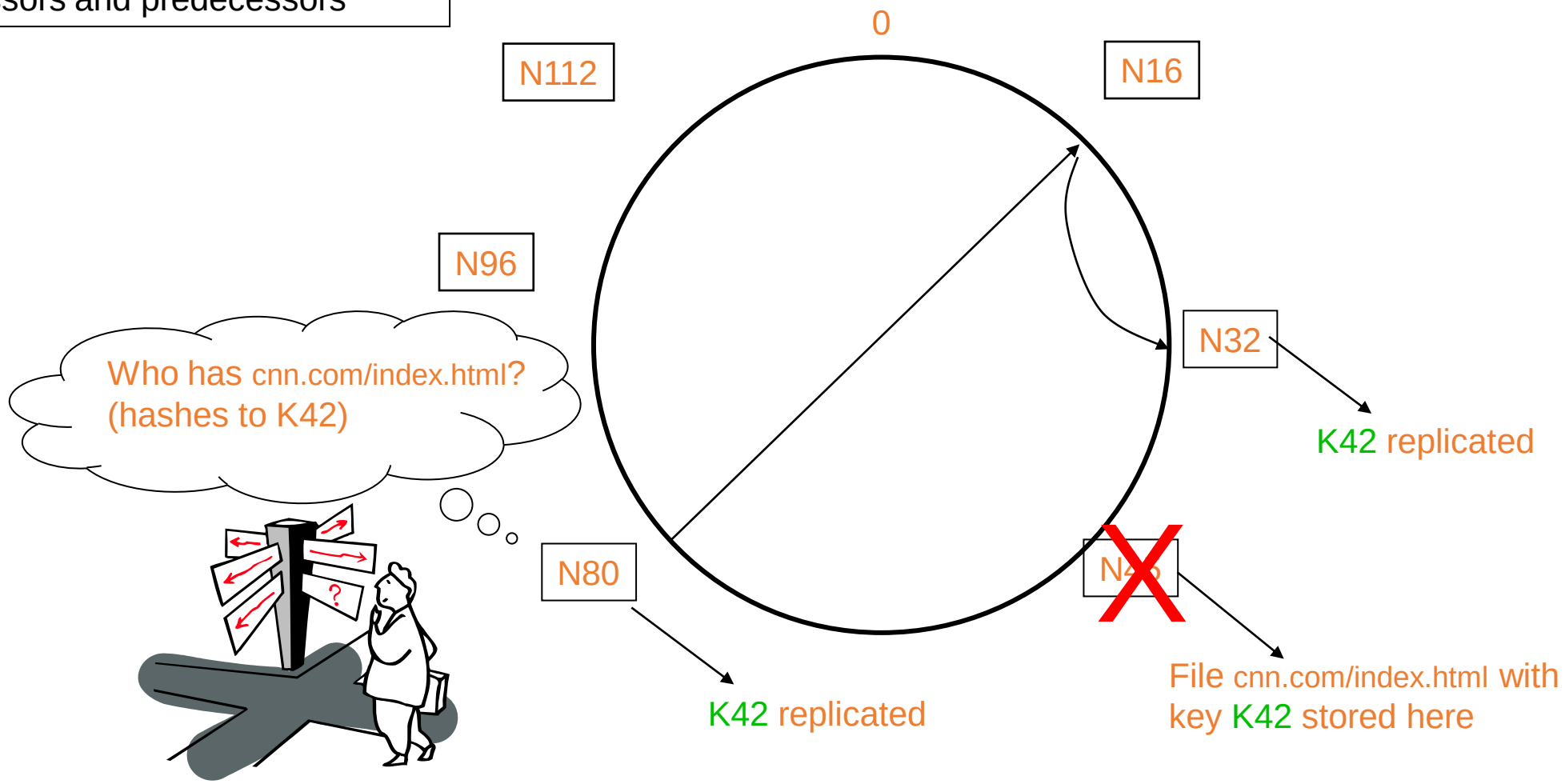


0	32
1	32
2	32
3	32
4	32
5	80
6	80

0	45
1	45
2	45
3	45
4	80
5	96
6	0

Search under peer failures (2)

One solution: replicate file/key at *r* successors and predecessors

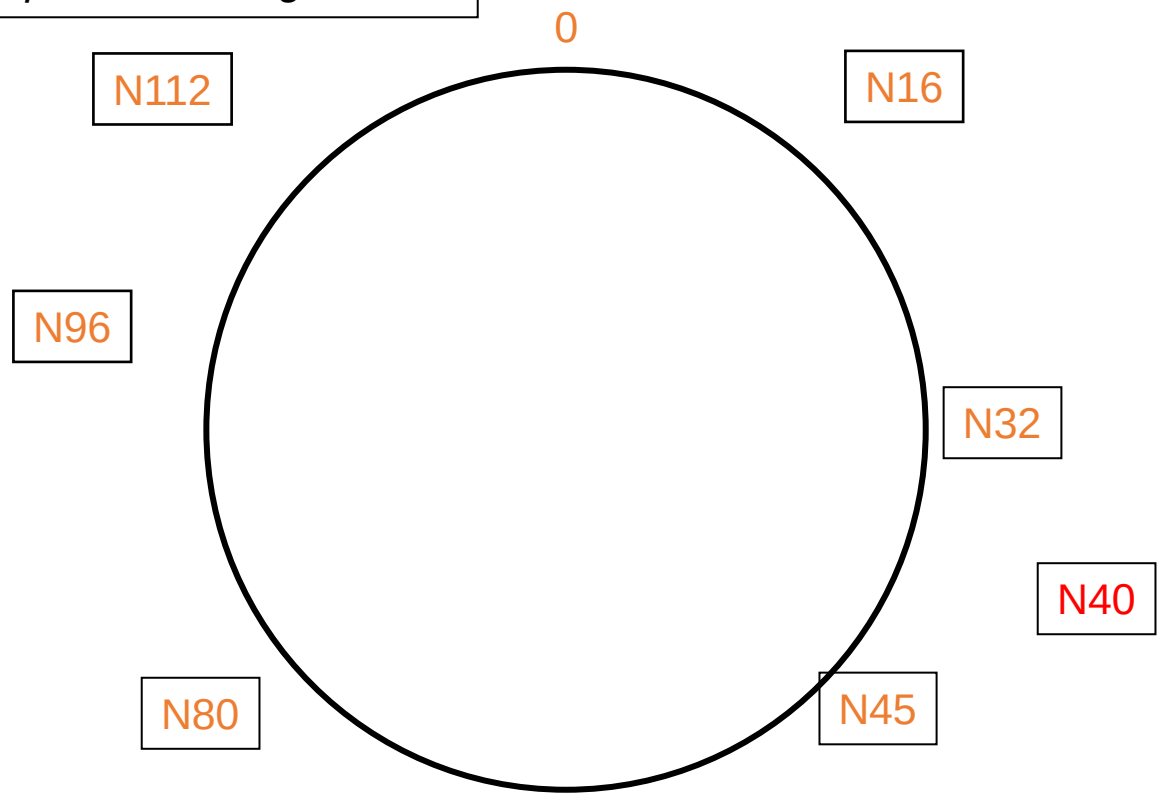


New peers joining

1. N40 acquires that N45 is its successor
2. N45 updates its info about predecessor to be N40
3. N32 runs stabilizer and asks N45 for predecessor
4. N45 returns N40
5. N32 updates its info about successor to be N40
6. N32 notifies N40 to be its predecessor
N40 periodically talks to neighbors to update own finger table

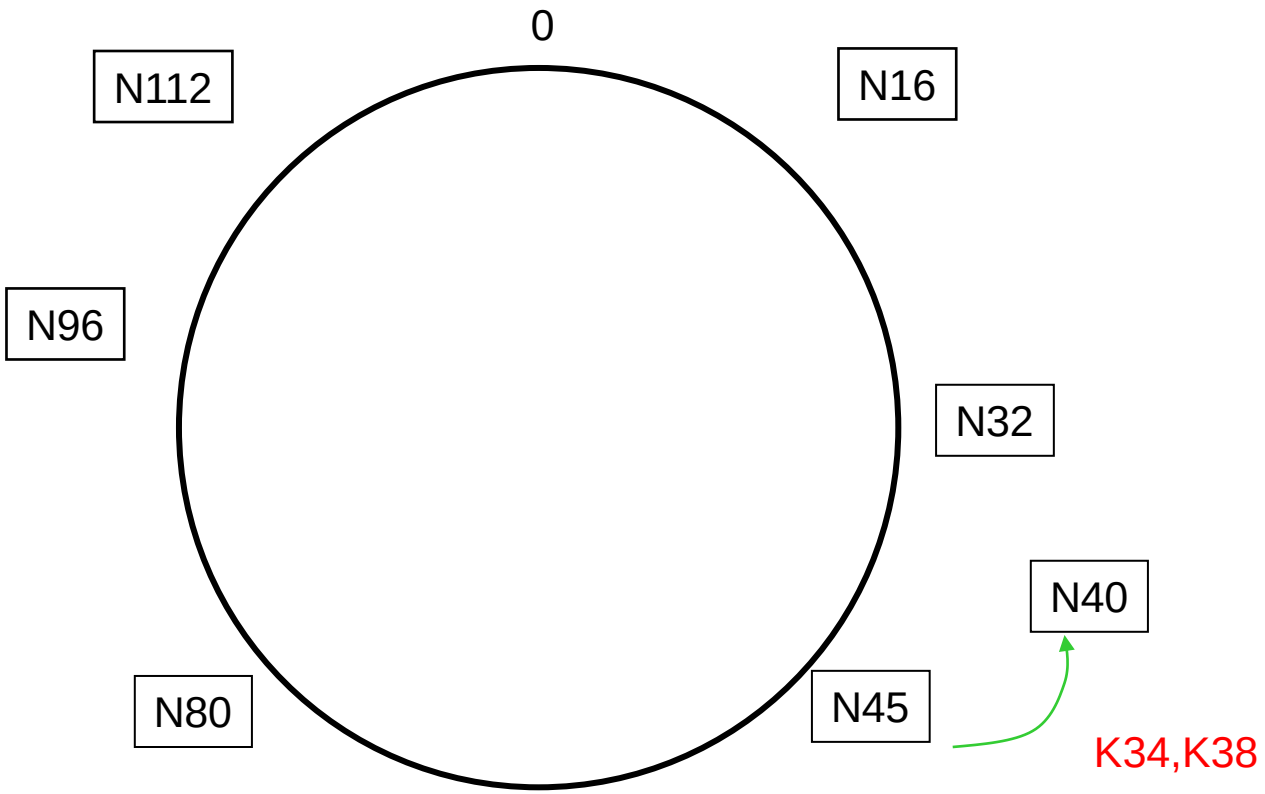
Peers also keep info about their predecessors to deal with dynamics

Stabilization protocol



New peers joining (2)

N40 may need to copy some files/keys from N45
(files with fileid between 32 and 40)



Chord Stabilization Protocol

Concurrent peer joins, leaves, failures might cause loopiness of pointers, and failure of lookups

- Chord peers periodically run a *stabilization* algorithm that checks and updates pointers and keys
- Ensures non-loopiness of fingers, eventual success of lookups and *O(log(N))* lookups
- [TechReport on Chord webpage] defines weak and strong stability
- Each stabilization round at a peer involves a constant number of messages
- Strong stability takes $O(N^2)$ stabilization rounds (!)

References

1. *Introduction to Distributed Self-Stabilizing Algorithms*. Karine Altisen; Stéphane Devismes; Swan Dubois; Franck Petit. Morgan & Claypool, 2019.
2. Self-Stabilization. S. Dolev. MIT Press. 2000.
3. S. Schmid and P.S. Mandal. Distributed Network Algorithms. Lecture Notes for GIAN Course, Chapter 13: Self-Stabilization, 2016.