Computer Science 425/ECE 428/CSE 424 Distributed Systems (Fall 2009)

Lecture 20

Self-Stabilization

Reading: Chapter from Prof. Gosh's book

Klara Nahrstedt

Acknowledgement

- **The slides during this semester are based on ideas and material from the following sources:**
  - **Slides prepared by Professors M. Harandi, J. Hou, I. Gupta, N. Vaidya, Y-Ch. Hu, S. Mitra.**
  - **Slides from Professor S. Gosh's course at University o Iowa.**

Administrative

- MP2 posted October 5, 2009, on the course website,
  - Deadline November 6 (Friday)
  - Demonstration, 4-6pm, 11/6/2009
  - Tutorial for MP2 planned for October 28 evening if students send questions to TA by October 25. Send requests what you would like to hear in the tutorial.

Plan for Today

- Motivation for Self-Stabilization
- Self-Stabilization Concepts/Definitions
- Dijkstra’s stabilization of mutual exclusion in unidirectional ring
- Chord’s stabilization protocol
- Stabilizing graph coloring

Motivation

- As the number of computing elements increase in distributed systems failures become more common
- Fault tolerance (FT) should be automatic, without external intervention
- two kinds of fault tolerance
  - masking: application layer does not see faults, e.g., redundancy and replication
  - non-masking: system deviates, deviation is detected and then corrected: e.g., feedback, roll back and recovery
- self-stabilization is a general technique for non-masking FT distributed systems

Self-stabilization

- **Technique for spontaneous healing**
- **Guarantees eventual safety following failures**
- ***Feasibility demonstrated by Dijkstra (CACM `74)***

E. Dijkstra

Configurations of Distributed Systems

- **Two classes of configurations (or behaviors)**
  - **Legitimate configuration**
    - » **In non-reactive system 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 reactive system 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 codes are not corrupted**

Assumption:

- **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**

Reasons for illegal configurations

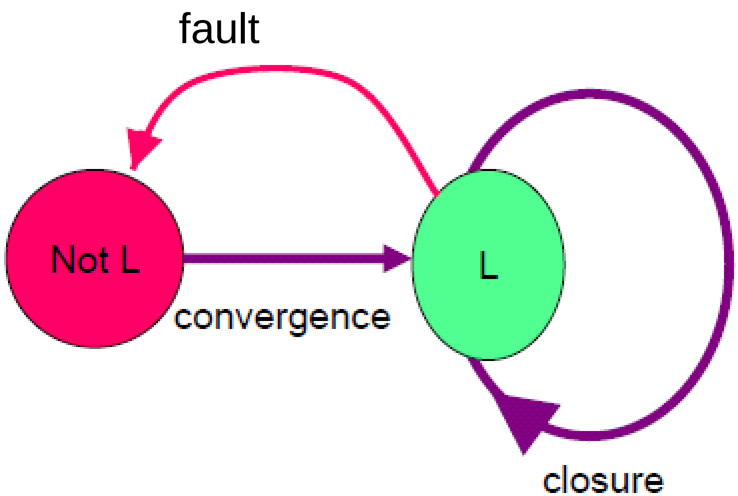
- **Transient failures perturb the global state. The ability to spontaneously recover from any initial state implies that no initialization is ever required.**
  - **Example: disappearance of the only circulating token in token ring; data corruption due to radio interference or power supply variations;**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

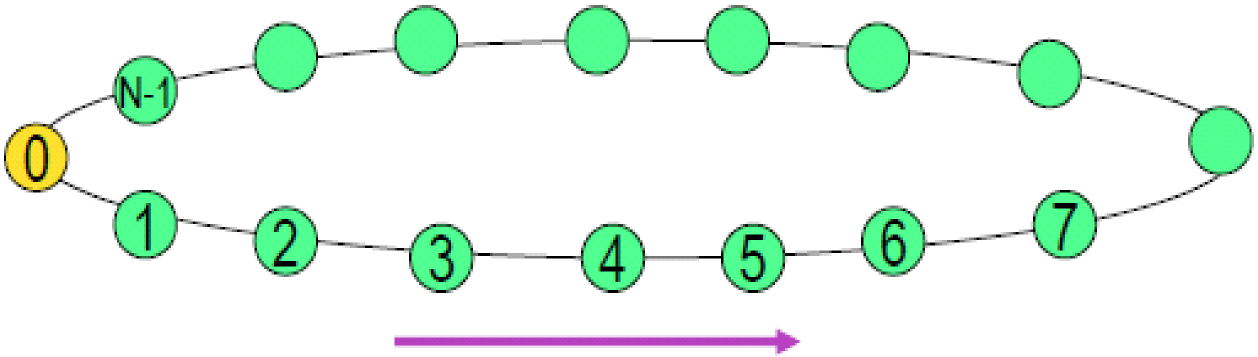
    regardless of initiate state, the system eventually returns to legal configuration
  - **closure**

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



L: Legitimate configuration
Non L: Illegitimate configruation

Example1: Stabilizing mutual exclusion in unidirectional ring



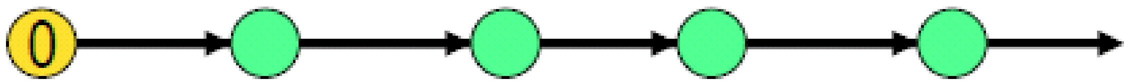
consider a unidirectional ring of processes.

Legal configuration = exactly one token in the ring desired “normal” behavior: single token circulates in the ring

Only the node that holds the token can access the critical region!!!

Dijkstra's stabilizing mutual exclusion

N processes: 0, 1, …, N-1
state of process j is $x[j] \in \{0, 1, 2, K-1\}$, where $K > N$



p_0 **if** $x[0] = x[N-1]$ **then** $x[0] := x[0] + 1$

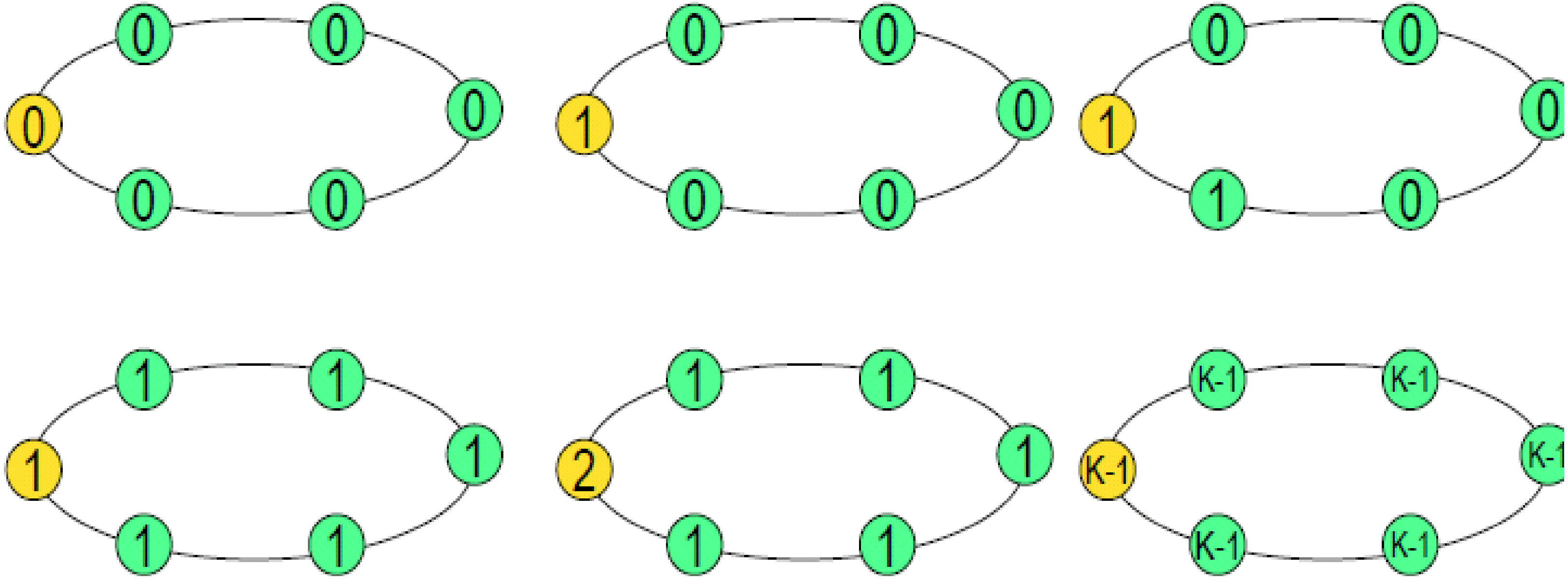
p_j $j > 0$ **if** $x[j] \neq x[j-1]$ **then** $x[j] := x[j-1]$

(TOKEN = if condition is true)

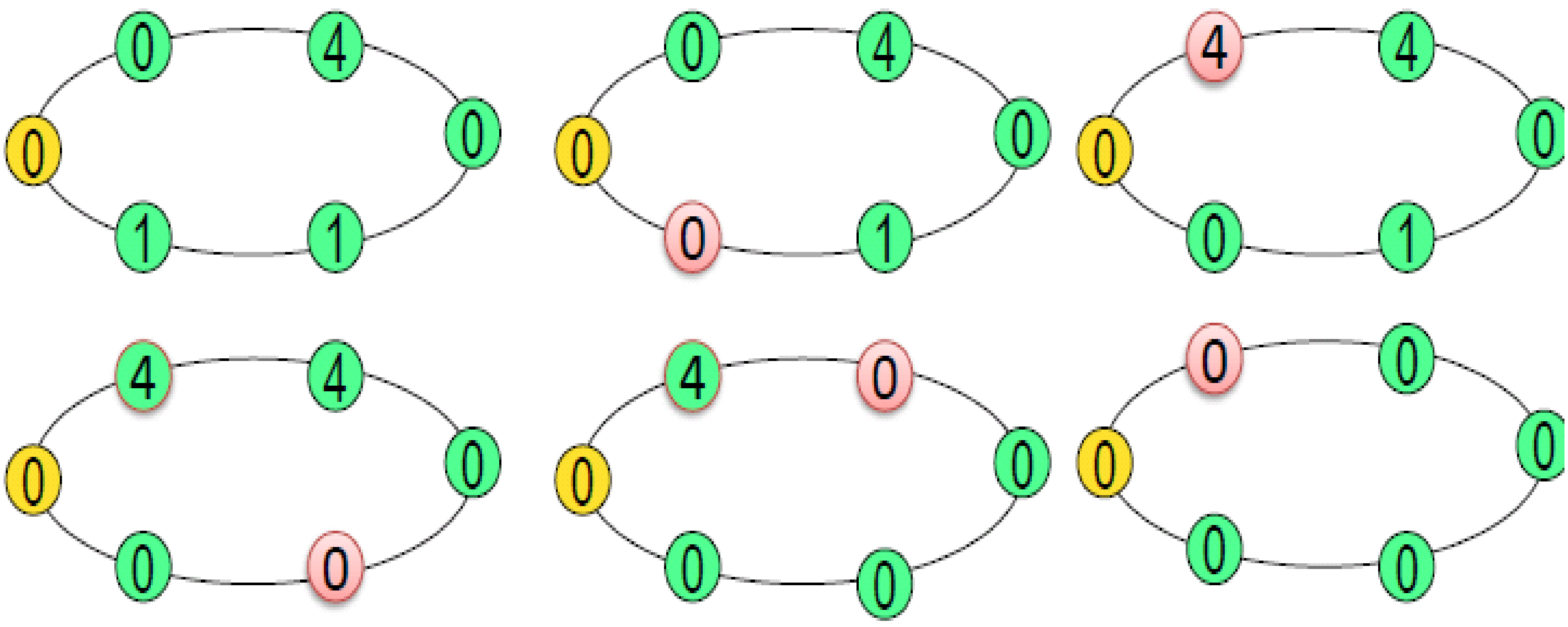
Legal configuration: only one process has token

start the system from an arbitrary initial configuration

Example execution



Example stabilizing execution



Why does it work?

1. at any configuration, at least one process can make a move (has token)
   - suppose $p_1,\ldots,p_{N-1}$ *cannot make a move*
   - then $x[N-1] = x[N-2] = \ldots\ x[0]$
   - then p_0 *can make a 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

- if only p_0 can make a move then for all i,j x[i] = x[j] and after p_0's move, only p_1 can make a move
- if only pi (i≠0) can make a move
  - for all j < i, x[j] = x[i-1]
  - for all k ≥ i, x[k] = x[i], and
  - x[i-1] ≠ x[i]

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

Why does it work?

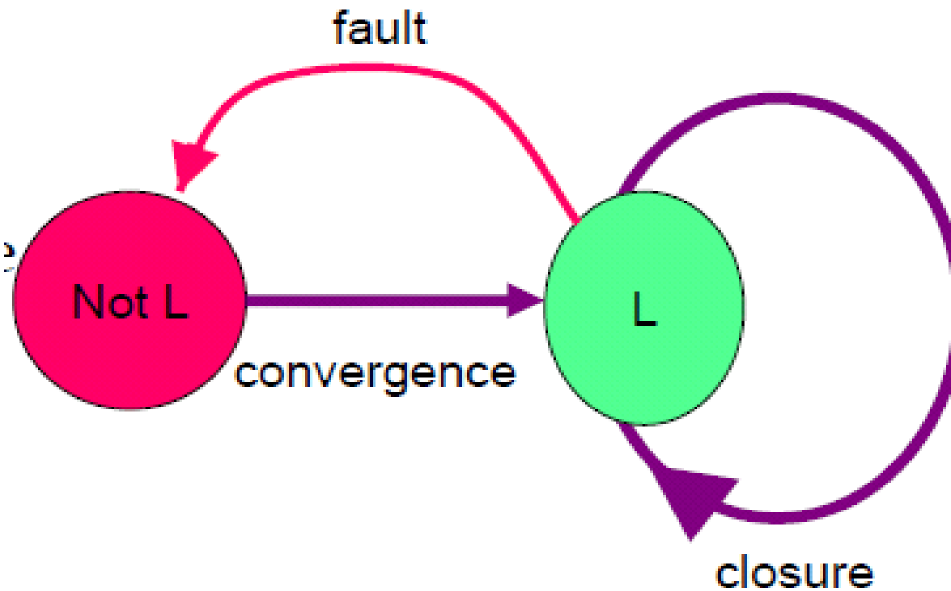
1. **at any configuration, at least one process can make a move (has token)**
2. **set of legal configurations is closed under all moves**
3. **total number of possible moves from (successive configurations) never increases**
   – **any move by p_i *either enables a move for p_{i+1} or none at all***

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. all illegal configuration C converges to a legal configuration in a finite number of moves
   - there must be a value, say v, that does not appear in C
   - except for p_0, *none of the processes create new values*
   - p_0 *takes infinitely many steps, and therefore, eventually sets x[0] = v*
   - all other processes copy value v and a legal configuration is reached in N-1 steps

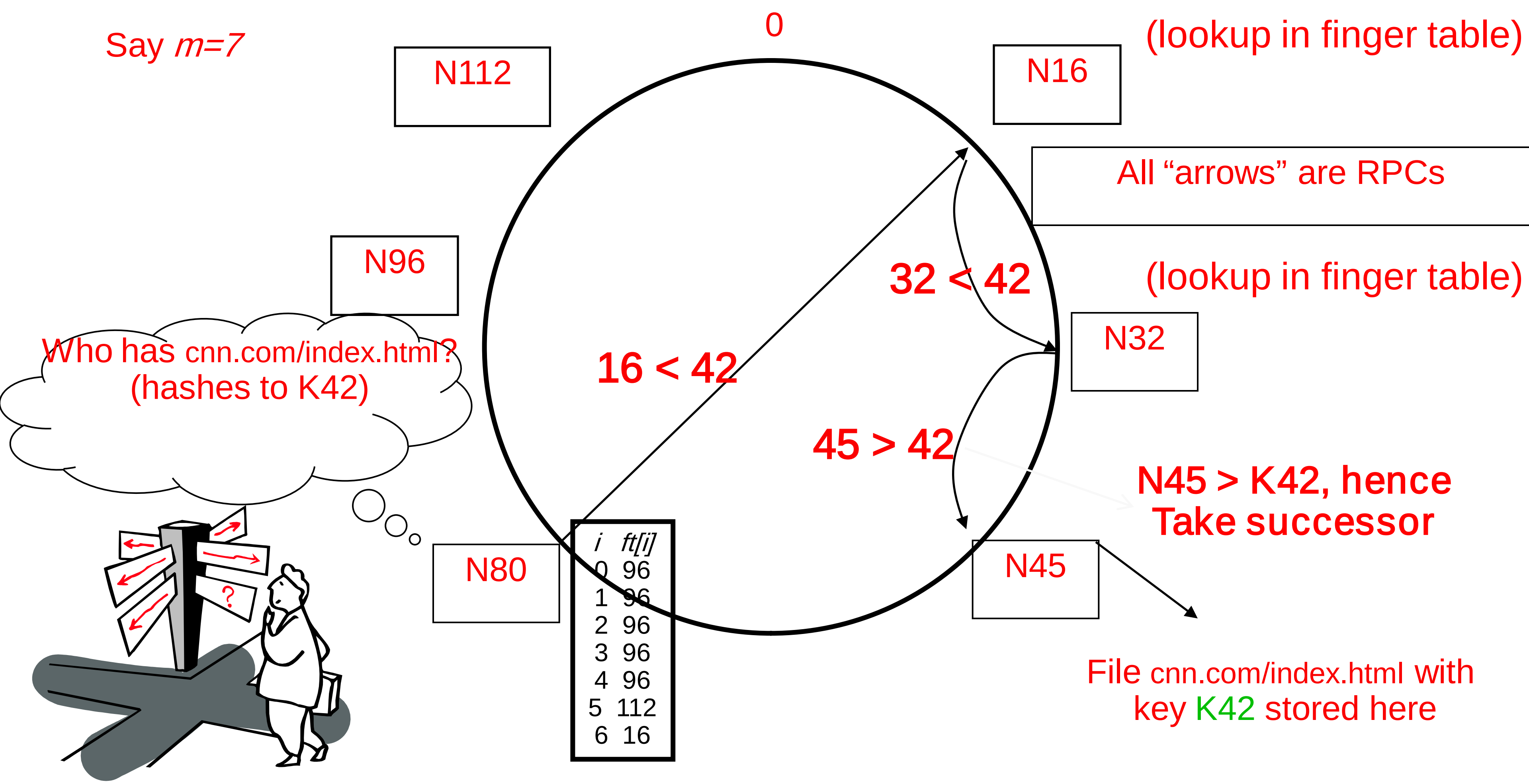
Putting it all together

- **Legal configuration = a configuration with a single token**
- **Perturbations or failures take the system to configurations with multiple tokens**
  - **e.g. mutual exclusion property may be violated**
- **Within finite number of steps, if no further failures occur, then the system returns to a legal configuration**



P2P Systems - Chord Search

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

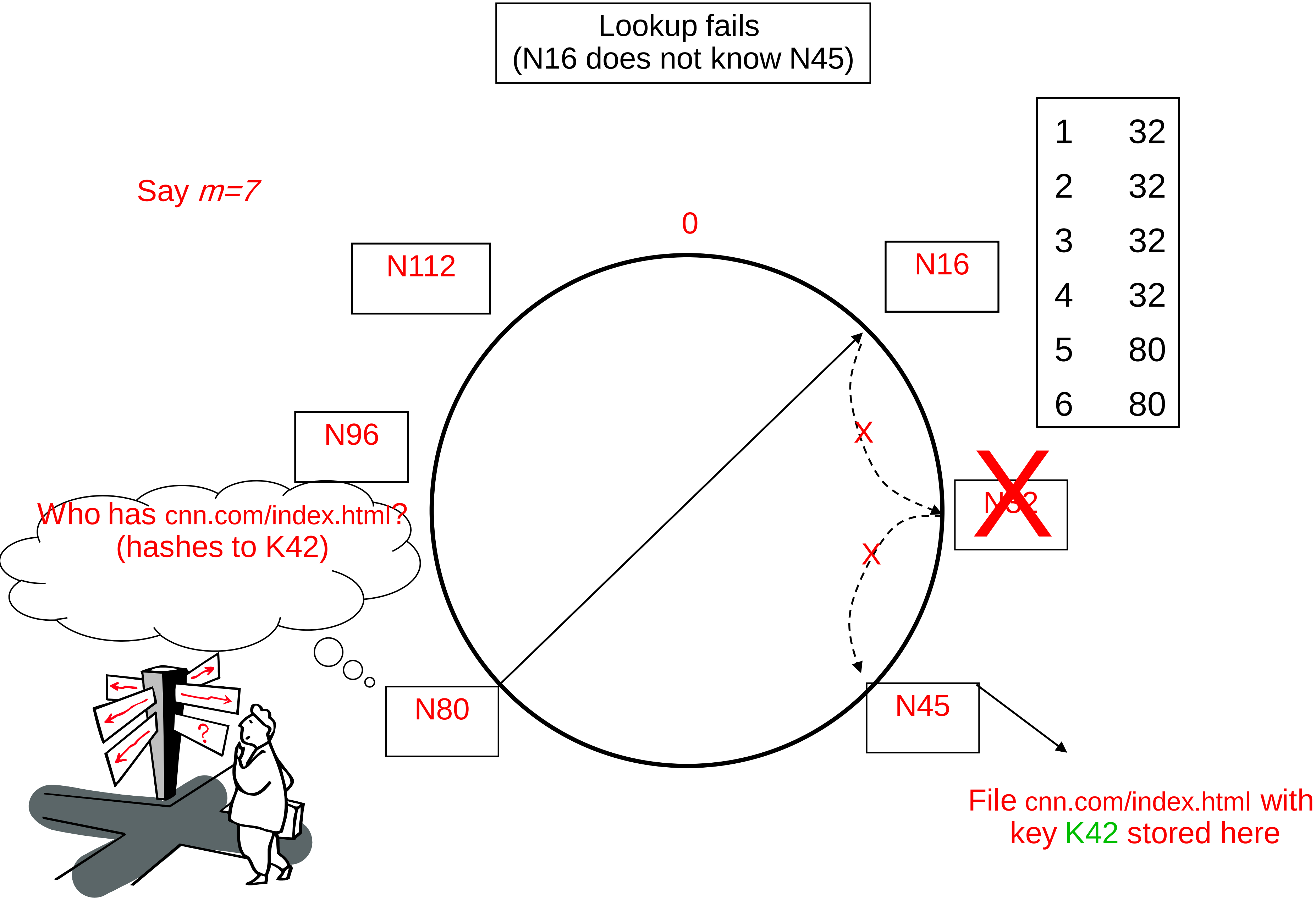


i	*ft[i]*
0	96
1	96
2	96
3	96
4	96
5	112
6	16

Stabilization 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

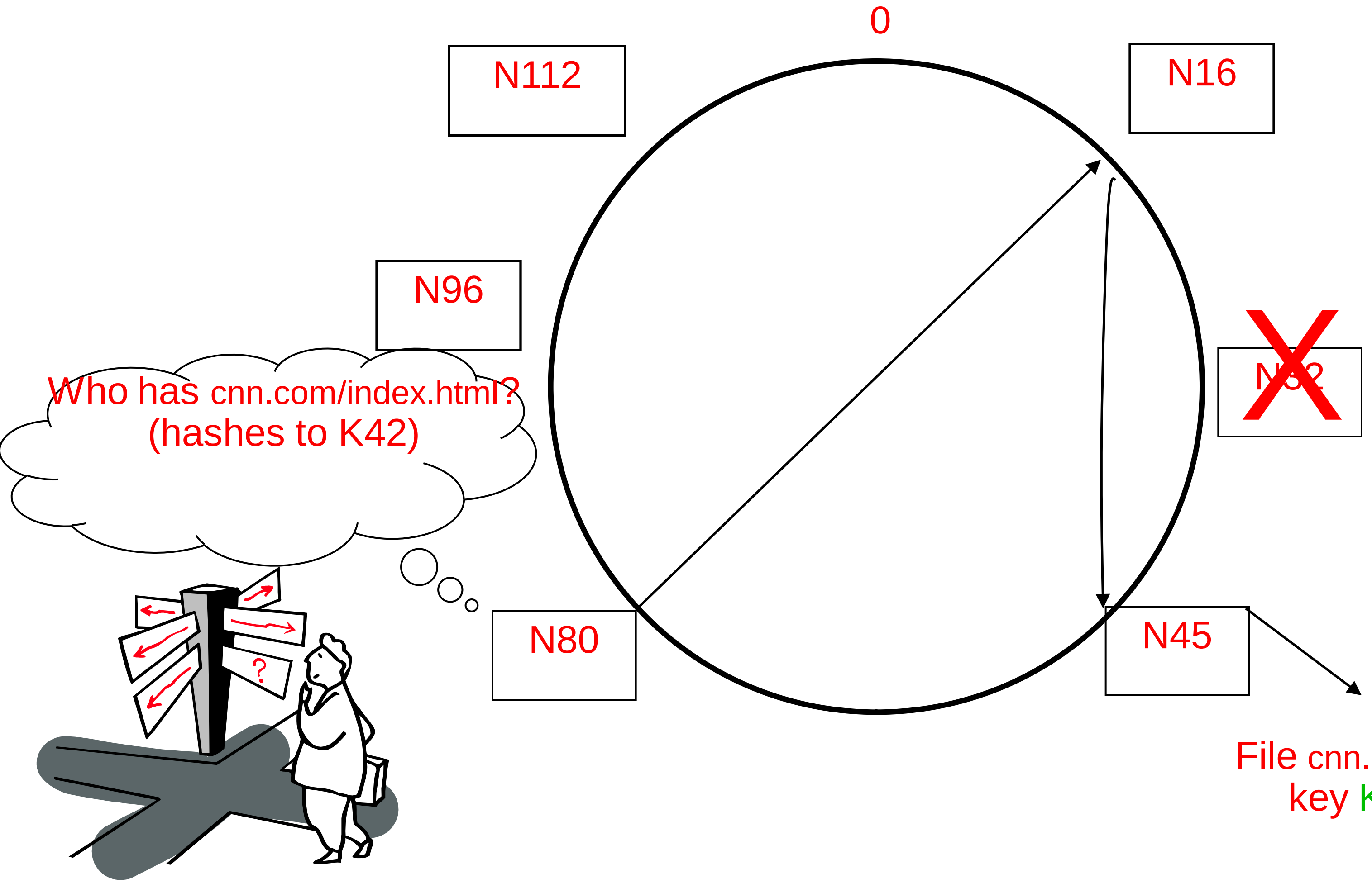


1	32
2	32
3	32
4	32
5	80
6	80

Search under peer failures

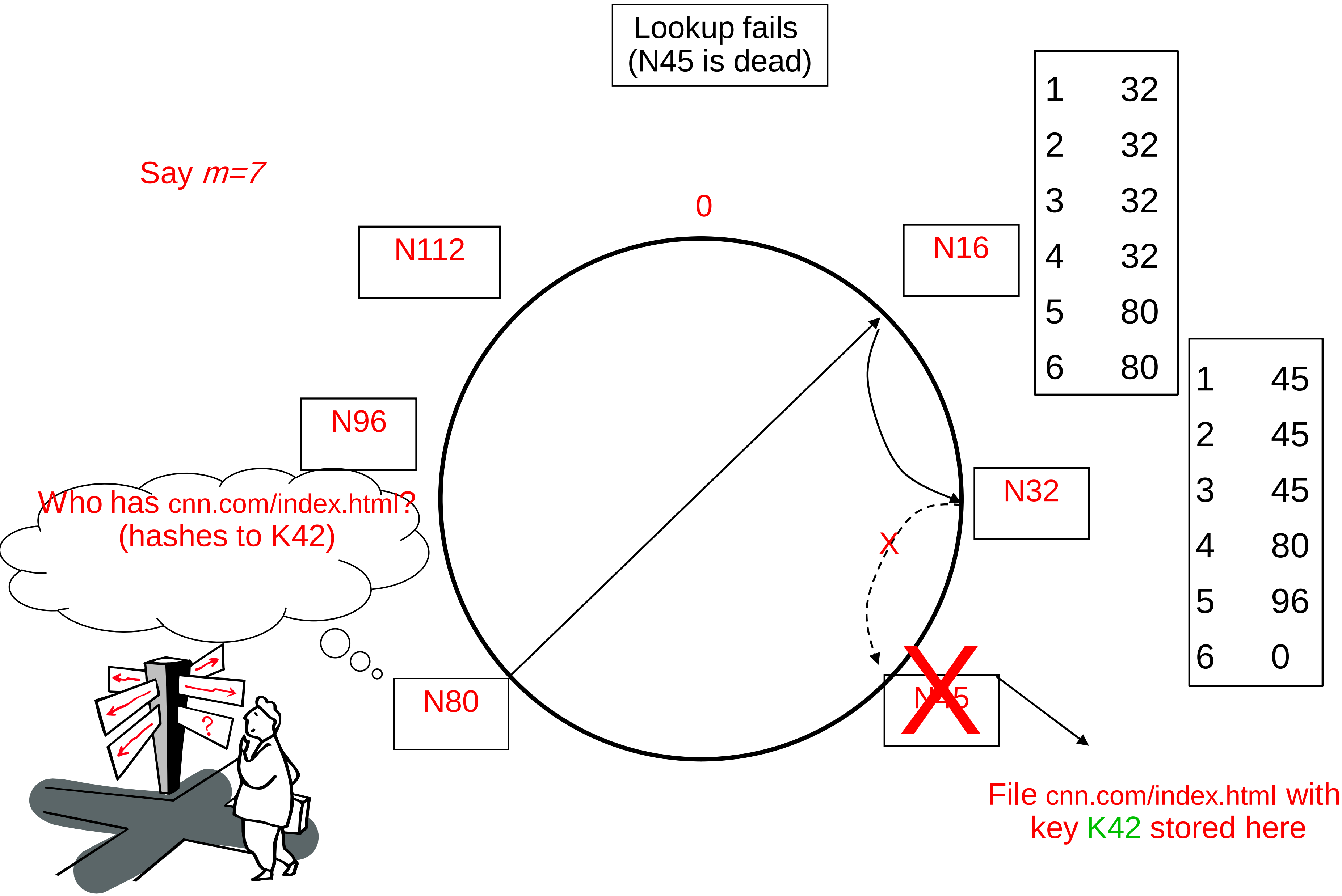
One solution: maintain *r* multiple *successor* entries
in case of failure, use successor entries

Say *m=7*



File cnn.com/index.html with
key K42 stored here

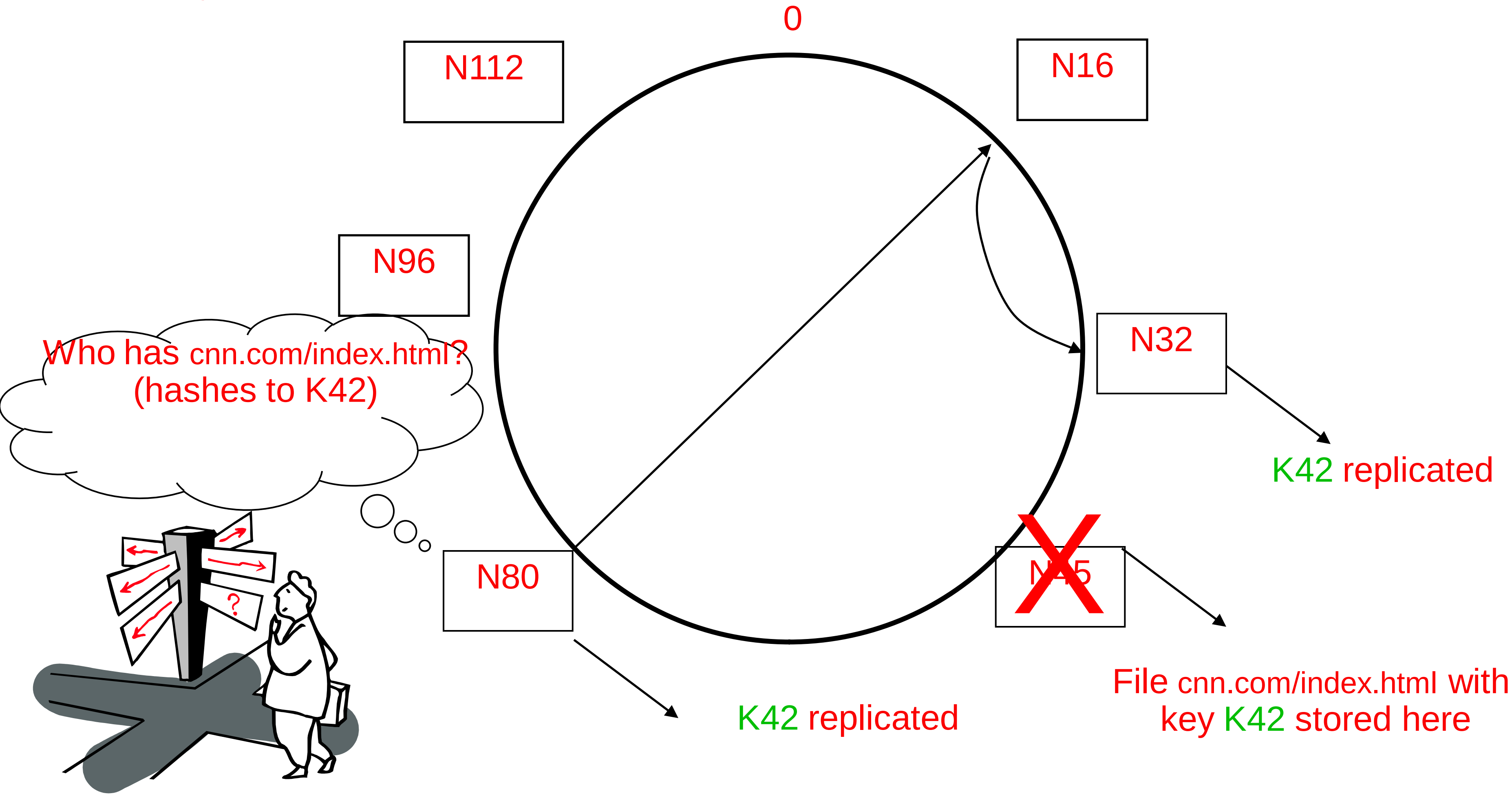
Search under peer failures (2)



Search under peer failures (2)

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

Say *m=7*



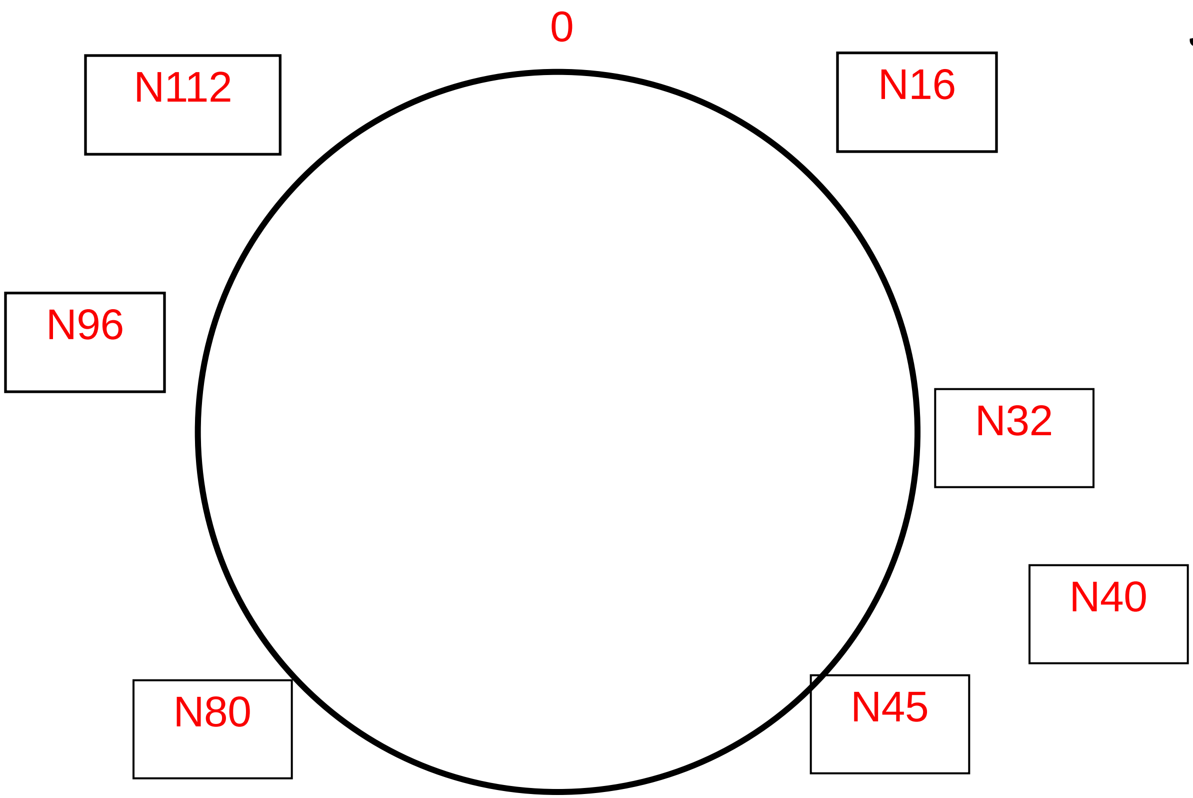
New peers joining

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

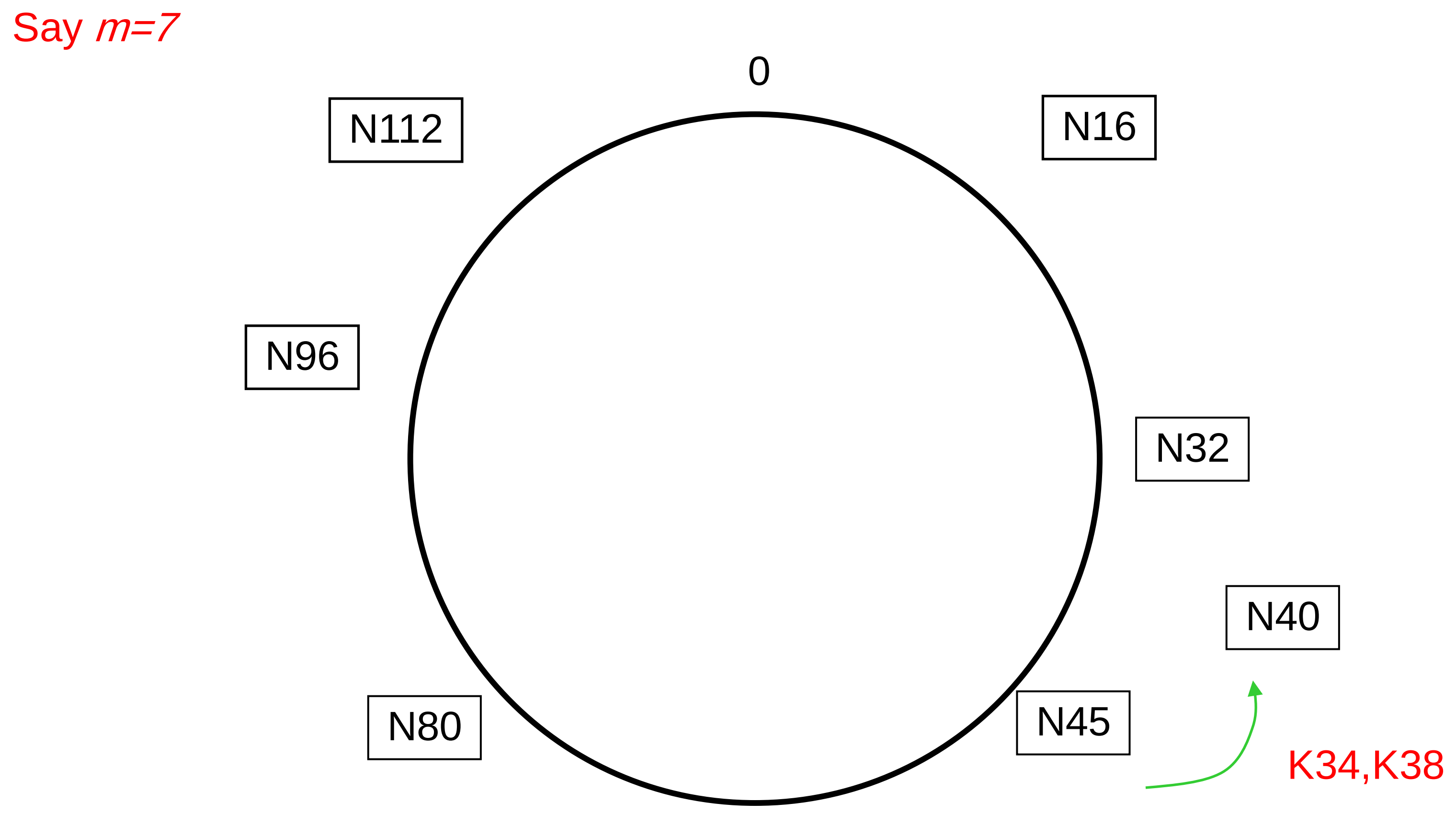
Stabilization protocol

Say *m=7*



New peers joining (2)

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



Chord Stabilization Protocol

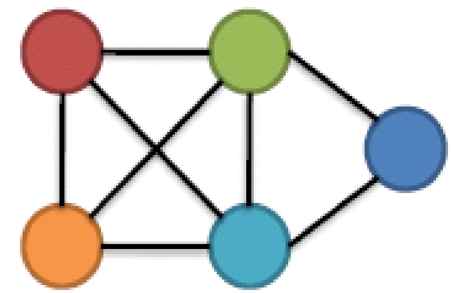
- **Concurrent peer joins, leaves, failures might cause loopiness of pointers, and failure of lookups**
  - **Chord peers periodically run a *stabilization* algorithm that checks and updates pointers and keys**
  - **Ensures non-loopiness of fingers, eventual success of lookups and *O(log(N))* lookups**
  - **[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 (!)**

Stabilizing graph coloring

- **Simple coloring problem and algorithm**

Graph coloring problem

- shared memory distributed system with N processes $p_0, \ldots, p_{N-1}$
  - induced undirected graph G = (V,E)
  - N_i: set of neighbors of p_i
  - $|N_i| \le D$, maximum degree of any node D
  - set of all colors C, $|C| = D + 1$



- initially nodes are assigned arbitrary colors
- design an algorithm such that for all i, j
  - if $j \in N_i$ then $color_i \neq color_j$
- application: choosing broadcast frequencies in a wireless network in order to reduce interference

Simple coloring algorithm

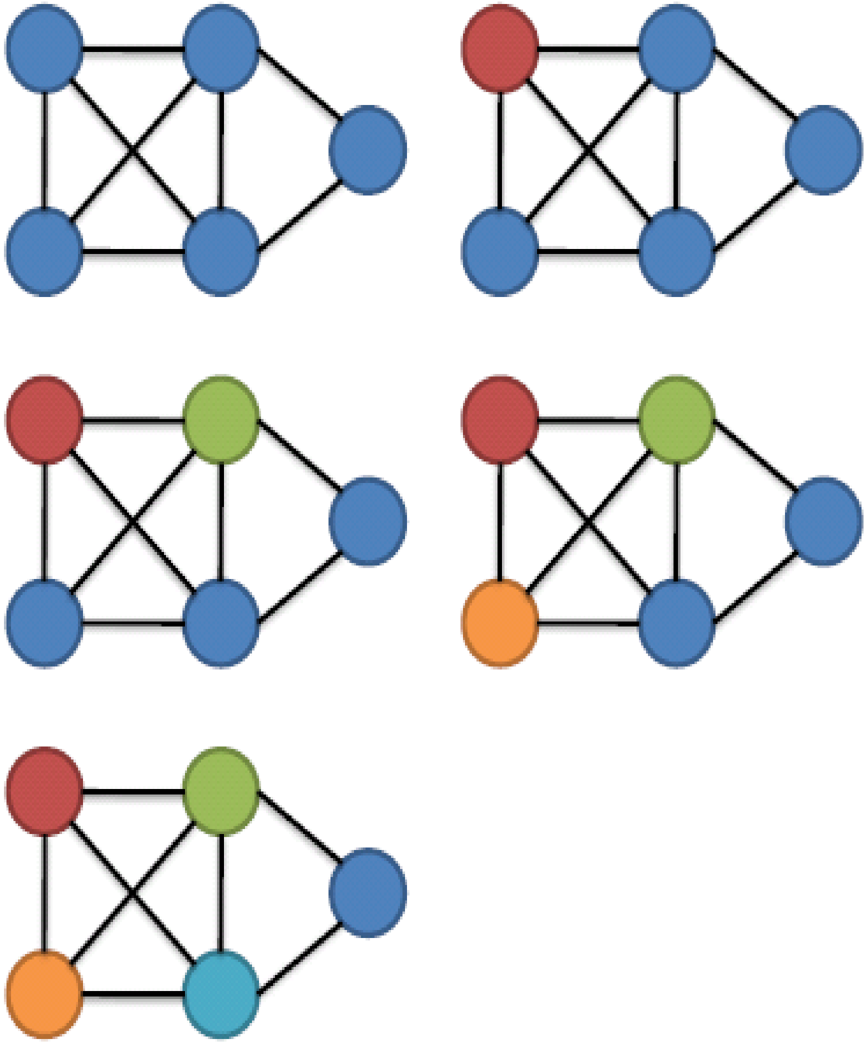
- program for process pi
  - NC = {c ∈ C | exists $j \in N_i$, $color_j$ = c}
  - **if** there exists $j \in N_i$ such that $color_i = color_j$

    then $color_i$:= choose from C \ NC

- shared memory program: p_i can read $color_j$, $j \in N_i$ and set $color_i$ in a single atomic step

Correctness of simple coloring (SC)

- **each action resolves the color of a node w.r.t. its neighbors**
- **once a node gets a distinct color, it never changes its color**
- **each node changes color at most once, algorithm terminates after N-1 steps**



Properties of SC

- Legal configuration = for all i, j, if j $\in N_i$ then $color_i \neq color_j$
- is SC self-stabilizing?
  - YES, does not require any initialization
  - from any initial coloring converges to a legal configuration, i.e., with correct coloring, in N-1 steps

- requires D+1 colors!
  - very suboptimal

Summary

- What is self-stabilization?
- Self-stabilization systems
- Simple coloring problem and algorithm