Computer Networks

EDA387/DIT663

Fault-tolerant Algorithms for Computer Networks

Super-stabilization (Ch.7)

Chapter 7: roadmap

7.1 Superstabilization

7.2 Self-Stabilizing Fault-Containing Algorithms

Dynamic System & Self Stabilization

Dynamic System	Self Stabilization
Algorithms for *dynamic systems* are designed to cope with failures of processors with no global re-initialization. Such algorithms consider only global states reachable from a predefined initial state under a restrictive sequence of failures and attempt to cope with such failures with as few adjustments as possible.	Self-stabilizing algorithms are designed to guarantee a particular behavior finally. Traditionally, changes in the communications graph were ignored.

Superstabilizing algorithms **combine the benefits of both self-stabilizing and dynamic algorithms**

Definitions

A Superstabilizing Algorithm:

- Must be self-stabilizing
- Must preserve a "*passage predicate*"
- Should exhibit fast convergence rate

Passage Predicate - Defined with respect to a class of topology changes (A topology change falsifies legitimacy and therefore the passage predicate must be weaker than legitimacy but strong enough to be useful).

Passage Predicate - Example

In a token ring:

Passage Predicate	Legitimate State
At most one token exists in the system. (e.g. the existence of 2 tokens isn't legal)	Exactly one token exists in the system.

A processor crash can lose the token but still not falsify the passage predicate

Evaluation of a Super-Stabilizing Algorithm

a. **Time complexity**

The maximal number of rounds that have passed from a legitimate state through a single topology change and ends in a legitimate state

b. **Adjustment measure**

The maximal number of processors that must change their local state upon a topology change, in order to achieve legitimacy

Motivation for Super-Stabilization

A self-stabilizing algorithm that does not ignore the occurrence of topology changes ("*events*") will be initialized in a predefined way and react better to dynamic changes during execution

Question:

Is it possible, for the algorithm that detects a fault, when it occurs, to maintain a "*nearly legitimate*" state during convergence?

Motivation for Super-Stabilization

While transient faults are rare (but harmful), a dynamic change in the topology may be frequent.

Thus, a super-stabilizing algorithm has a lower worst-case time measure for reaching a legitimate state again, once a topology change occurs.

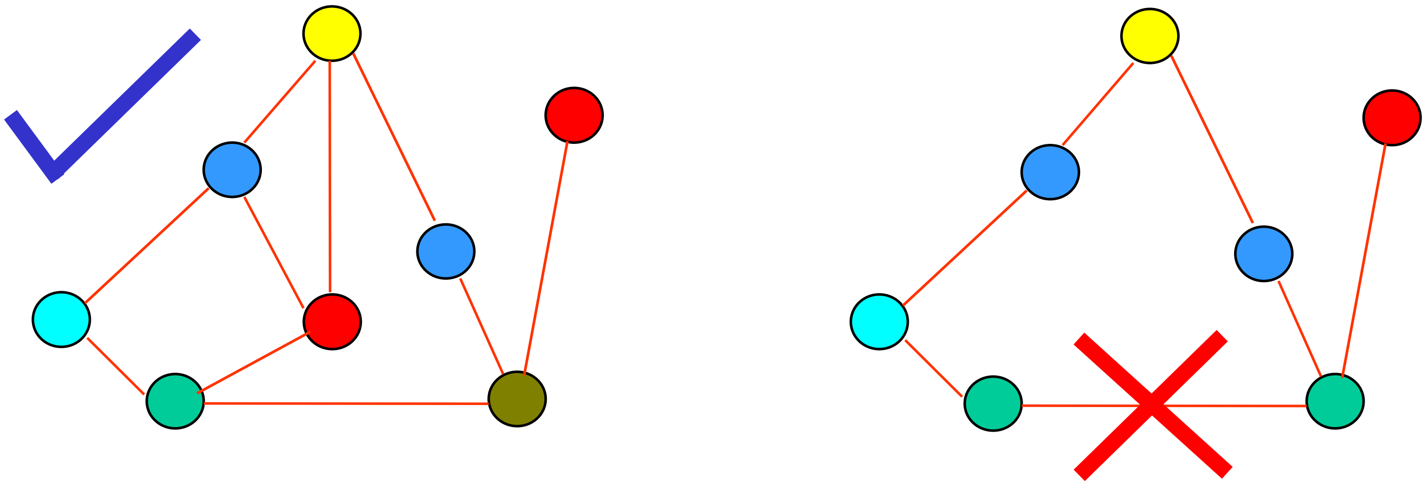
In the following slides we present a self-stabilizing and a super-stabilizing algorithm for the graph coloring task.

Graph Coloring

a. The coloring task is to assign a color value to each processor, such that no two neighboring processors are assigned the same color.

b. Minimization of the colors number is not required. The algorithm uses Δ+1 colors, where Δ is an upper bound on a processor's number of neighbors.

c. For example:



Graph Coloring - A Self-Stabilizing Algorithm

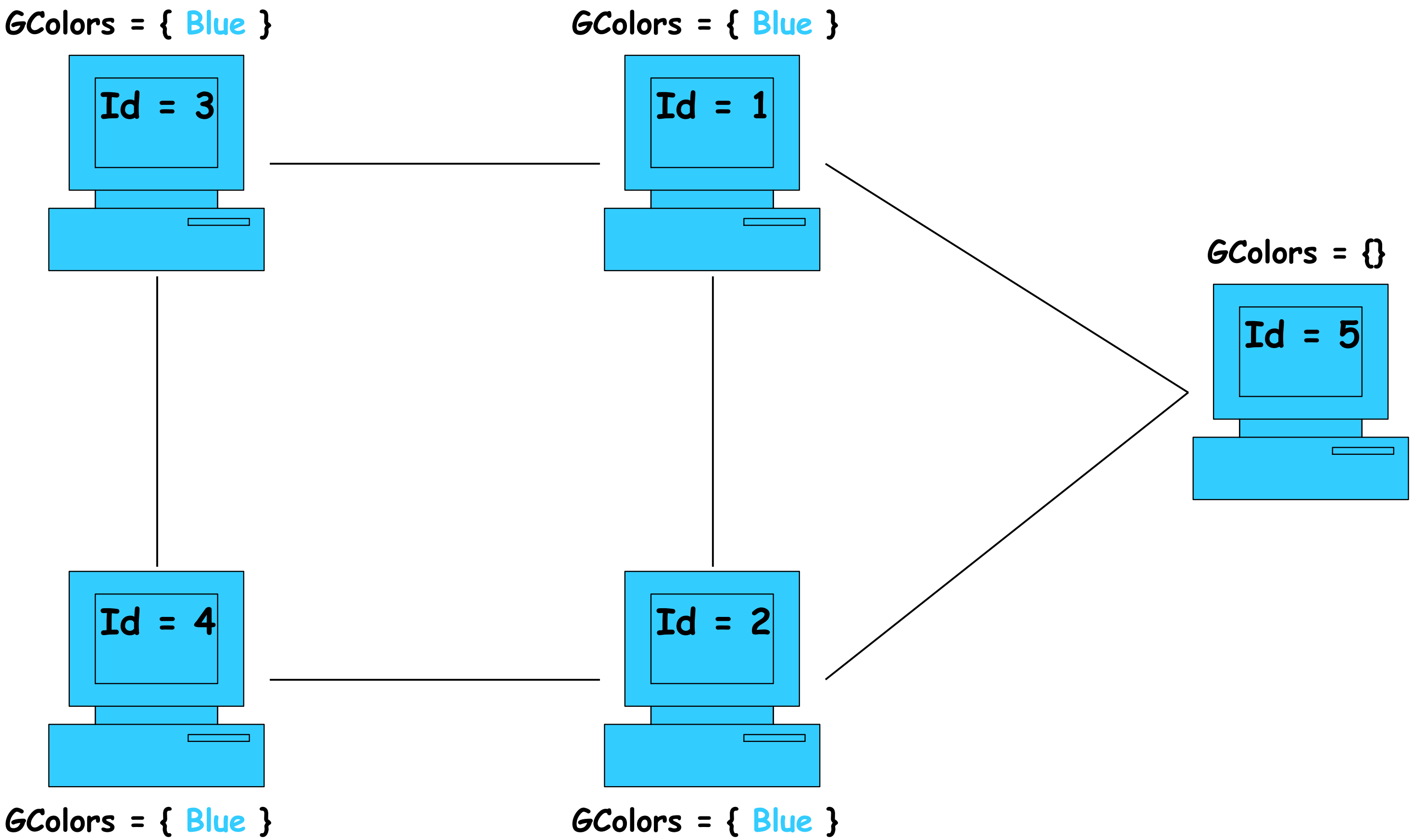
Do forever
    $\boxed{\text{GColors}} := \emptyset$
    For $m:=1$ to δ do
        $lr_m:=read(r_m)$
        If $ID(m)>i$ then
          $\boxed{\text{GColors} := \text{GColors} \cup \{lr_m.color\}}$
    od
    If $color_i \in$ GColors then
      $color_i:=choose(\backslash\backslash$ GColors$)$
    Write $r_i.color := color_i$
od

Colors of P_i's neighbors

If P_i has the color of one of its neighbors with a higher ID, it chooses another color and writes it.

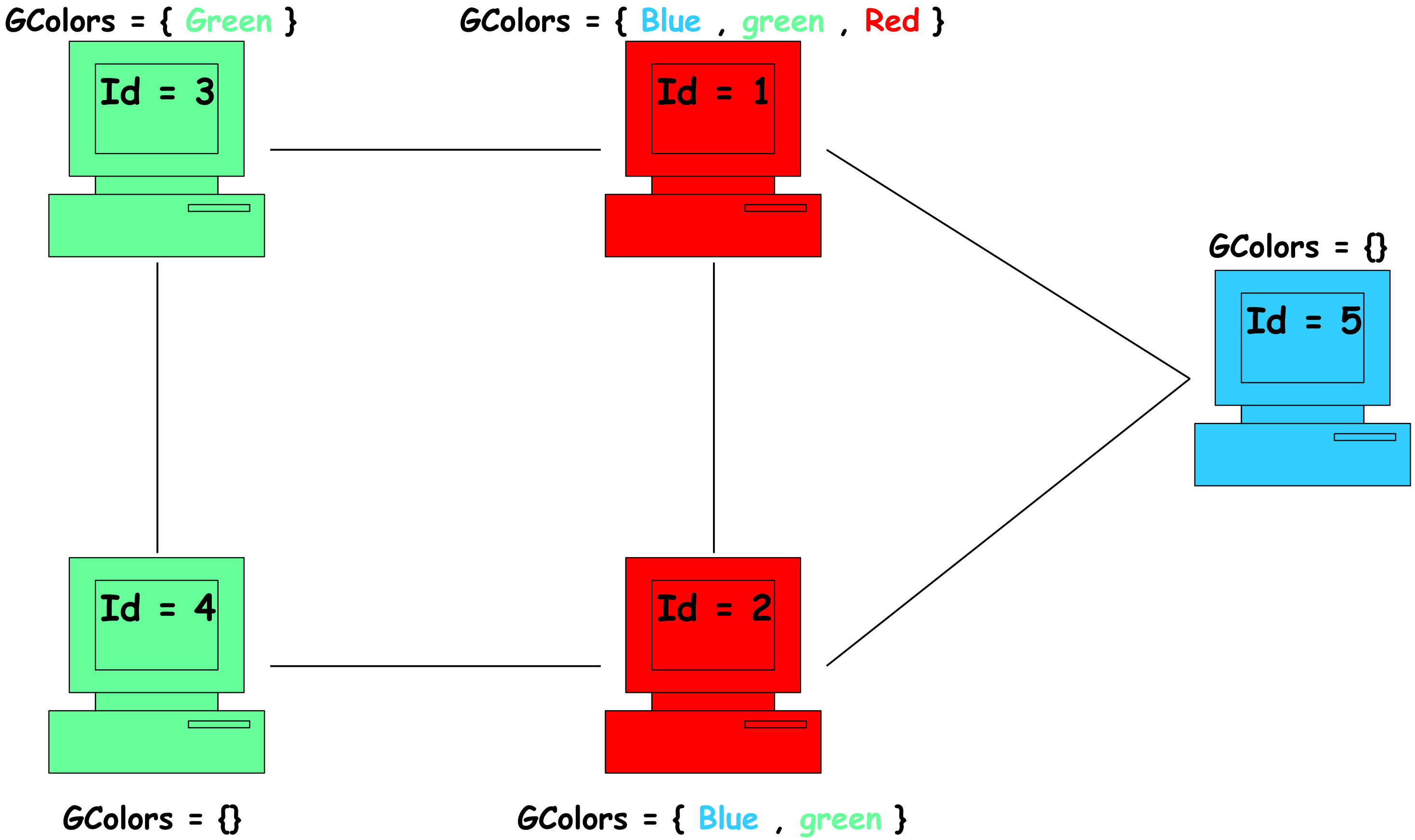
Gather only the colors of neighbors with greater ID than P_i's.

Graph Coloring - Self-Stabilzing Algorithm - Simulation



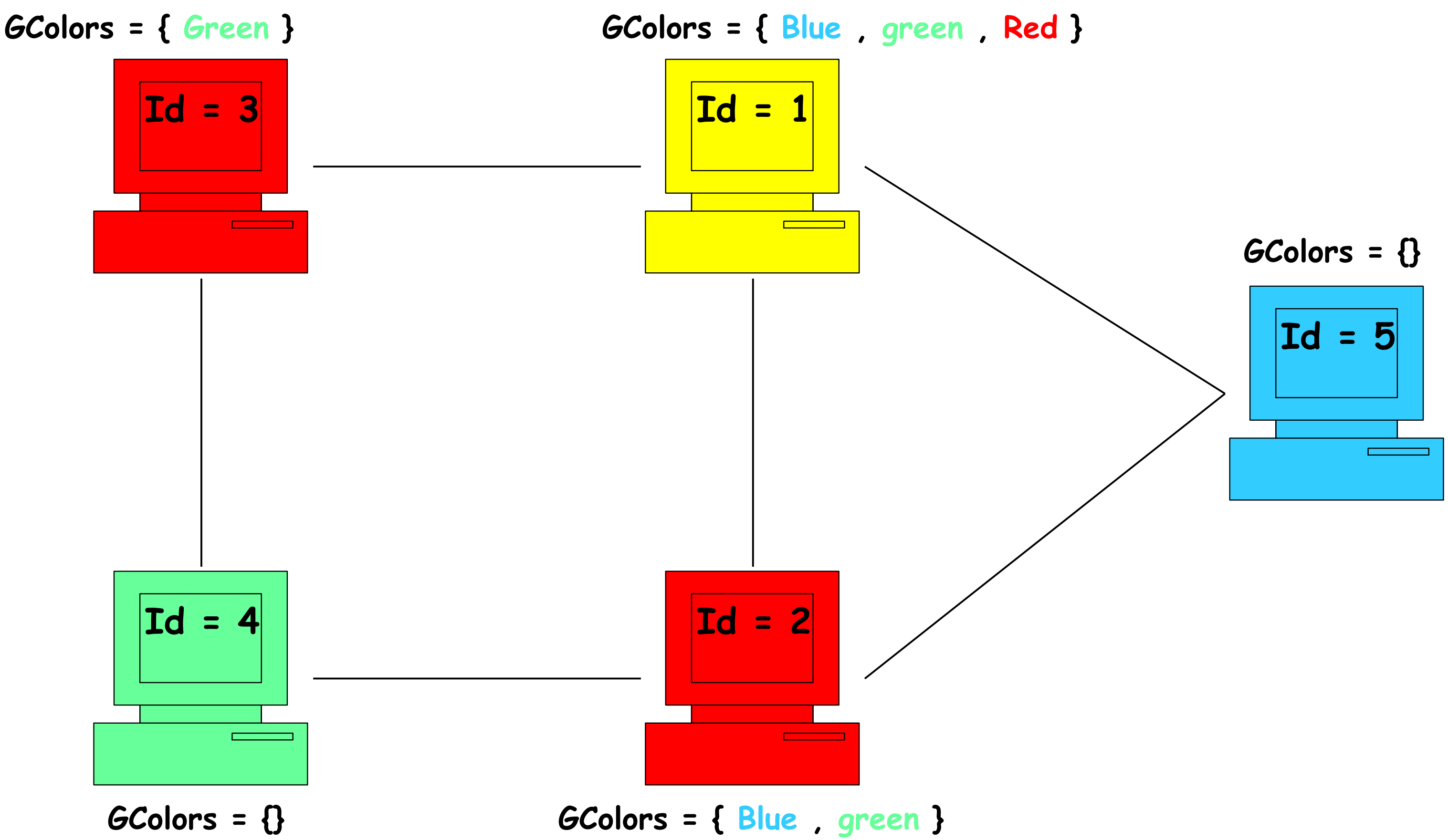
Phase I

Graph Coloring - Self-Stabilzing Algorithm - Simulation



Phase II

Graph Coloring - Self-Stabilzing Algorithm - Simulation



Stabilized

Phase III

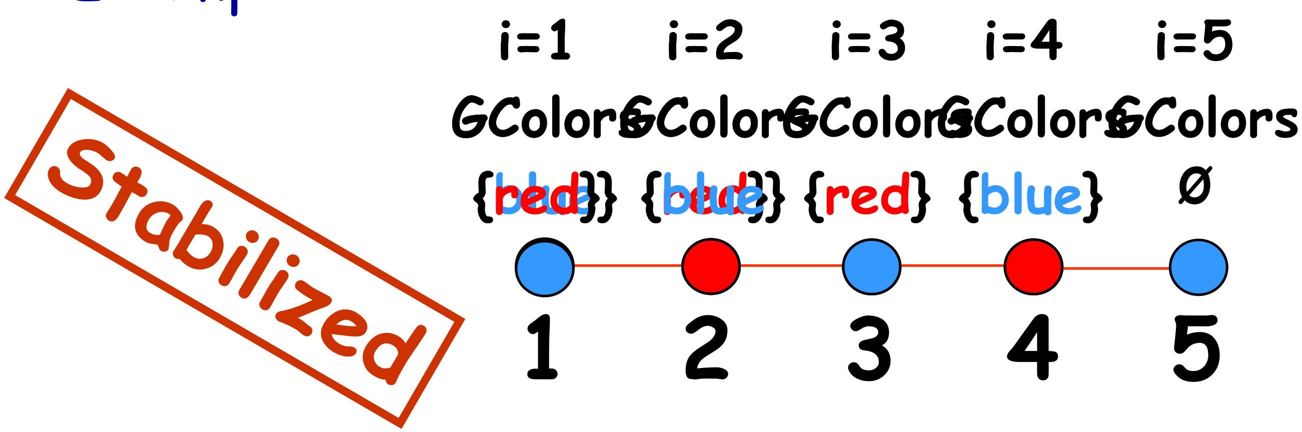
Graph Coloring - Self-Stabilizing Algorithm (continued)

What happens when a change in the topology occurs ?

If a new neighbor is added, it is possible that two processors have the same color.

It is possible that during convergence every processor will change its color.

Example:



But in what cost ?

Graph Coloring – Super-Stabilizing Motivation

a. Every processor changed its color but only one processor really needed to.
b. If we could identify the topology change we could maintain the changes in its environment.
c. We'll add some elements to the algorithm:
    a. AColor – A variable that collects all of the processor neighbors' colors.
    b. Interrupt section – Identify the problematic area.
    c. $\perp$ - A symbol to flag a non-existing color.

Graph Coloring - A Super-Stabilizing Algorithm

```
Do forever
    AColors := ∅
    GColors := ∅
    For m:=1 to δ do
        lr_m:=read(r_m)
        AColors:=AColors U {lr_m.color}
        If ID(m)>i then GColors := GColors U {lr_m.color}
    od
    If color_i = ⊥ or color_i ∈ GColors then
        color_i:=choose(\\ AColors)
    Write r_i.color := color
od
Interrupt section
    If recover_ij and j > i then
        Color_i := ⊥
        Write r_i.color := ⊥
```

All of P_i neighbors' colors

Activated after a topology change to identify the critical processor

$recover_{i,j}$ is the interrupt which P_i gets upon a change in the communication between P_i and P_j

Graph Coloring - Super-Stabilizing Algorithm - Example

Note that the new algorithm stabilizes faster than the previous one (in some cases).

Let us consider the previous example, this time using the super-stabilizing algorithm:

$Color_4 = \perp$

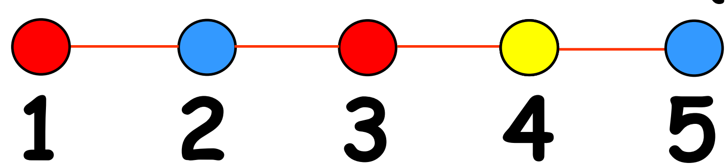
$r_4.color = \perp$

i=4

GColors = {blue}

AColors = {blue,red}

Stabilized



In O(1).

Graph Coloring - Super-Stabilizing Proof

Lemma 1: This algorithm is self-stabilizing.

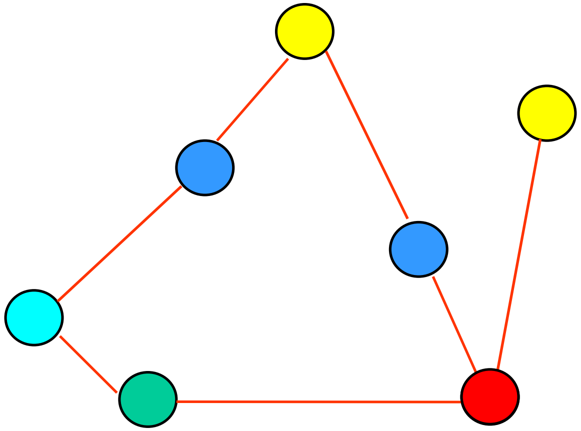
Proof by induction:

a. After the first iteration:
   a. The value $\perp$ doesn't exist in the system.
   b. P_n has a fixed value.
b. Assume that P_k has a fixed value $\forall$ $i<k\leq n$.
   If P_i has a P_k neighbor then P_i does not change to P_k's color, but chooses a different color.

Due to the assumptions we get that P_i's color becomes fixed for $1\leq i\leq n$, so the system stabilizes.

Graph Coloring - Super-Stabilizing

Passage Predicate - The color of a neighboring processor is always different in every execution that starts in a safe configuration, in which only a single topology change occurs before the next safe configuration is reached



Graph Coloring - Super-Stabilizing

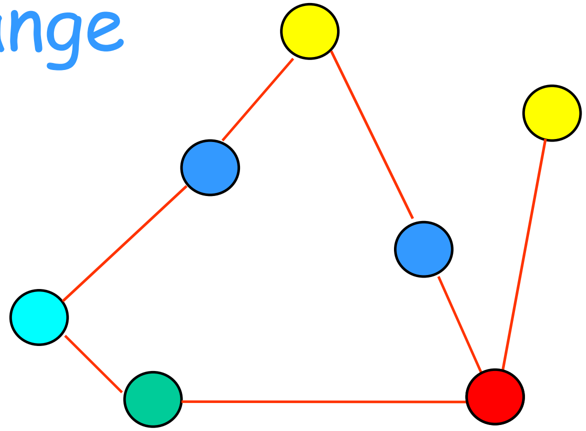
Super-stabilizing Time - Number of cycles required to reach a safe configuration following a topology change.

Super-stabilizing	vs.	Self-Stabilizing
$O(1)$		$O(n)$

Graph Coloring - Super-Stabilizing

Adjustment Measure - The number of processors that changes color upon a topology change.

The super-stabilizing algorithm changes one processor color, the one which had the single topology change



Reading Review

- Shlomi Dolev, Ted Herman: *Superstabilizing Protocols for Dynamic Distributed Systems*. Chicago J. Theor. Comput. Sci. 1997 (1997)

http://cjtcs.cs.uchicago.edu/articles/1997/4/cj97-04.pdf