Distributed Algorithms 2015/16
Self Stabilization

Reinhardt Karnapke | Communication and Operating Systems Group

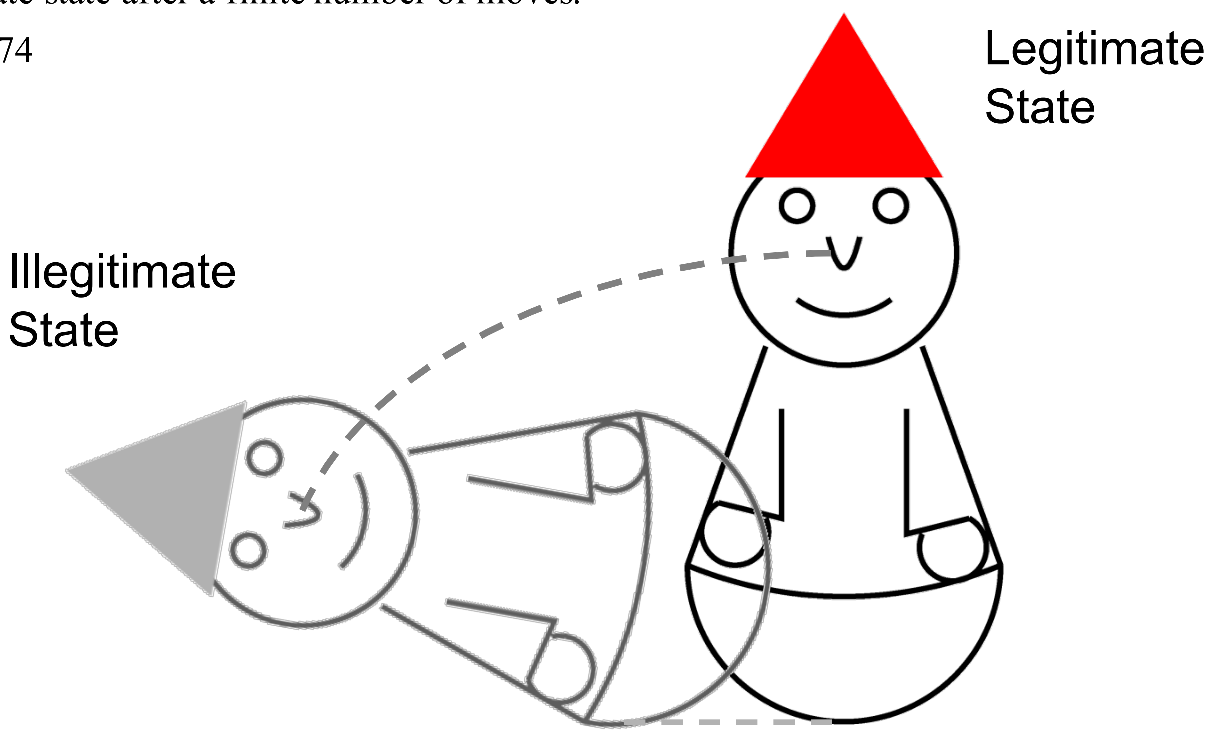
Overview

- Introduction (second to last lecture)
- Masking fault tolerance (last lecture)
  - Consensus and related problems
- Non-masking fault tolerance (this lecture)
  - Self-Stabilization

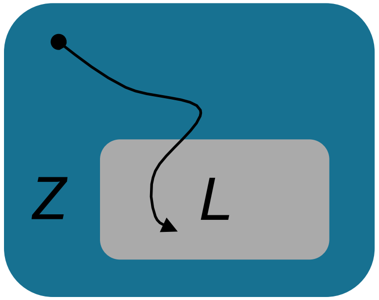
Self-Stabilizing Systems

„We call the system "**self-stabilizing**" if and only if, regardless of the initial state […], the system is guaranteed to find itself in a legitimate state after a finite number of moves.“

Dijkstra, 1974



Proof of Self-Stabilization



- Set of *all* states Z
- Set of the *legitimate* states $L \subseteq Z$
- To be proven: convergence and closure

- **Convergence**: Starting from a state $Z \setminus L$, after a limited number of steps a state in L is reached
  - Construct a function t (termination function) from Z to $\mathbb{N}$, that decreases with every step and indicates with $t = 0$ the stabilization in the end

- **Closure**: Starting in a state in L, each following state again is in L
  - Proven usually through an invariant

Recovery from Transient Errors

- Self-stabilizing systems recover from arbitrary *transient* faults if no new faults occur for a sufficient period of time
  - The state after the end of the last fault is regarded as „initial“ state → recovery guaranteed
- The class of transient faults contains among others
  - Temporary network faults
  - Crash and following restart of processes
  - Arbitrary corruption of data structures
- Note: Non-self-stabilizing systems fail possibly *permanently* even after transient faults!

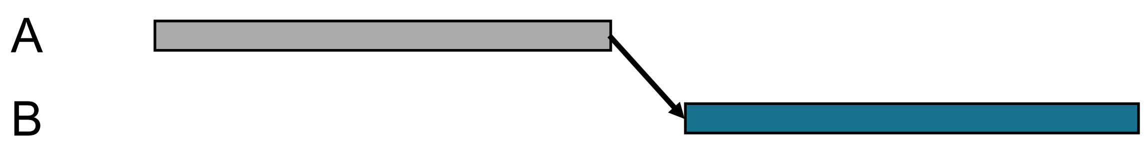
- *Arbitrary* large part of the resources can be affected by faults
  - Exception: program code and data in ROM cannot be corrupted

Self-Stabilizing Systems – Characteristics

- Do not need to be initialized because they reach a *legal* state from *every* starting state
- Tolerate arbitrary transient faults with one uniform mechanism
- Can not know for sure whether they are stabilized
- Must not terminate
- Adapt to dynamic changes of the typology if possible
- Do not necessarily need to detect faults to recover from them
- Offer efficient solutions for many problems
  - Information distribution, mutual exclusion, spanning tree construction, election, …

Composition of Algorithms

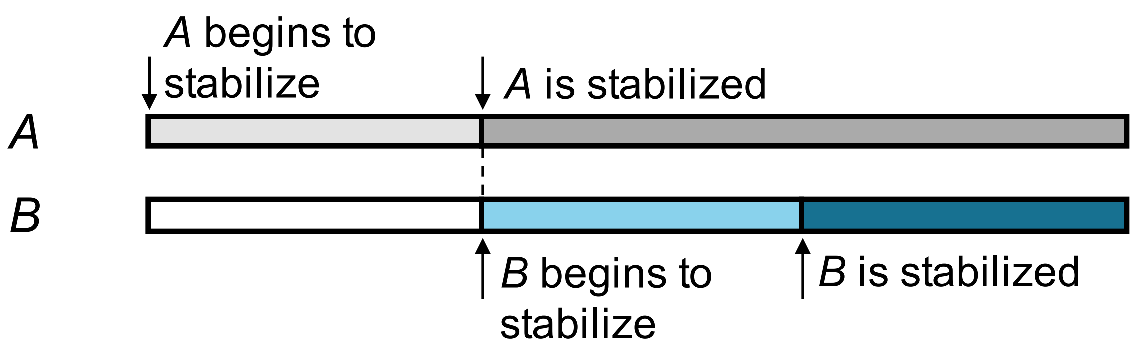
- Conventional composition of algorithms *A* and *B*
  - Composition of *A* and *B*
  - *B* is started when *A* has terminated
  - The output of *A* serves as input for *B*



Composition of Algorithms

- Composition of *self-stabilizing* algorithms
  - Simultaneous execution of *A* and *B*
  - If *A* has stabilized, *B* is stabilized afterwards
  - Precondition: *B* writes no data that *A* reads

- Stabilizing time of the composition is the sum of the stabilizing times of the single algorithms plus possible delays



Self-Stabilizing Token Ring (Dijkstra, 1974)

n + 1 processes are arranged in a unidirectional ring
Each process can take on one of k states ($k > n$)
→ Variable $s \in \{0, \ldots, k-1\}$
Each process can access the state of its left neighbor through a common variable *left*
Each process which can move anytime, will move at a time

```
ON bottom process (P0):
    WHILE TRUE DO
        IF (left == s) THEN
            <token>
            s := (s+1) mod k;
        FI
    END
END
```

```
ON other process (Pi, i≠0):
    WHILE TRUE DO
        IF (left != s) THEN
            <token>
            s := left;
        FI
    END
END
```

Self-Stabilizing Token Ring

Process	0	1	2	3
State	0	0	0	0
	1	0	0	0
	1	1	0	0
	1	1	1	0
	1	1	1	1
	2	1	1	1
	2	2	1	1
	2	2	2	1
	2	2	2	2
	3	2	2	2

Example trace without fault for n = 3 and k = 4

→ Process executes step and accesses token. State after step in next row

● Process could execute step and access the token.

Self-Stabilizing Token Ring

Example trace with recovery from a fault for n = 3 and k = 4

Process	0	1	2	3
State	3	→ **2** ●	**0** ●	**1** ●
	3	3	**0** ●	→ **1** ●
	3	3	→ **0** ●	0
	3	3	3	→ **0** ●
	→ **3** ●	3	3	3
	0	→ **3** ●	3	3
	0	0	→ **3** ●	3
	0	0	0	→ **3** ●
	→ 0 ●	0	0	0
	1	→ 0 ●	0	0

→ Process executes step and accesses token. State after step in next row

● Process could execute step and access the token.

Self-Stabilizing Token Ring

- Original Specification
  - Safety: There is at most one token in the system
  - Liveness: At least one token circulates in the ring
  - Fairness: If a process can exercise anytime, it will exercise after a finite time

- Self-stabilizing variant
  - Safety: After a finite number of steps, there is at most one token in the system
  - Liveness and Fairness as above

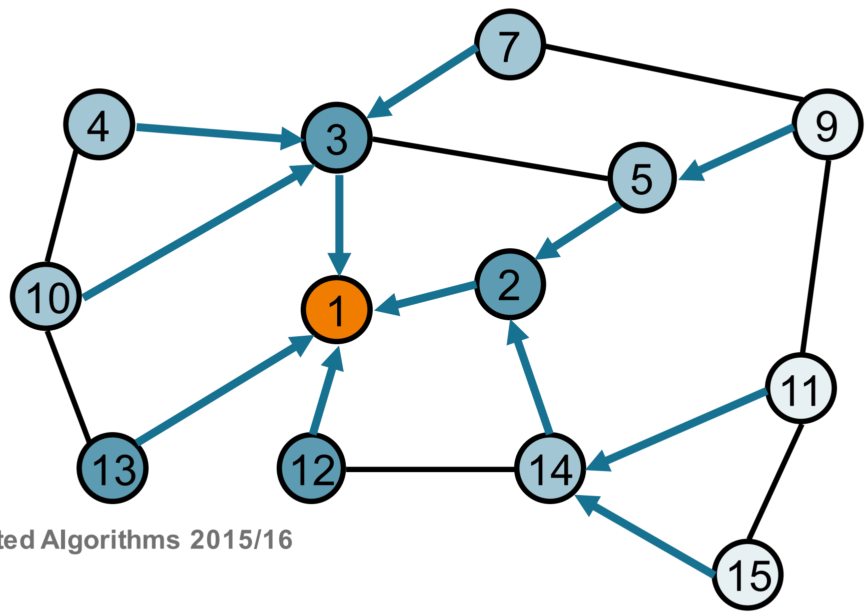
Self-Stabilizing Span Tree Construction

- Processes $\{P_1, \ldots, P_n\}$ are arranged in an arbitrary, connected topology

- Assumptions
  - Each process has a unique identity > 0, stored in its ROM
  - Each process has the same timeout-value ρ stored in its ROM
  - In the fault-free case, no messages get lost and messages have a limited message delay

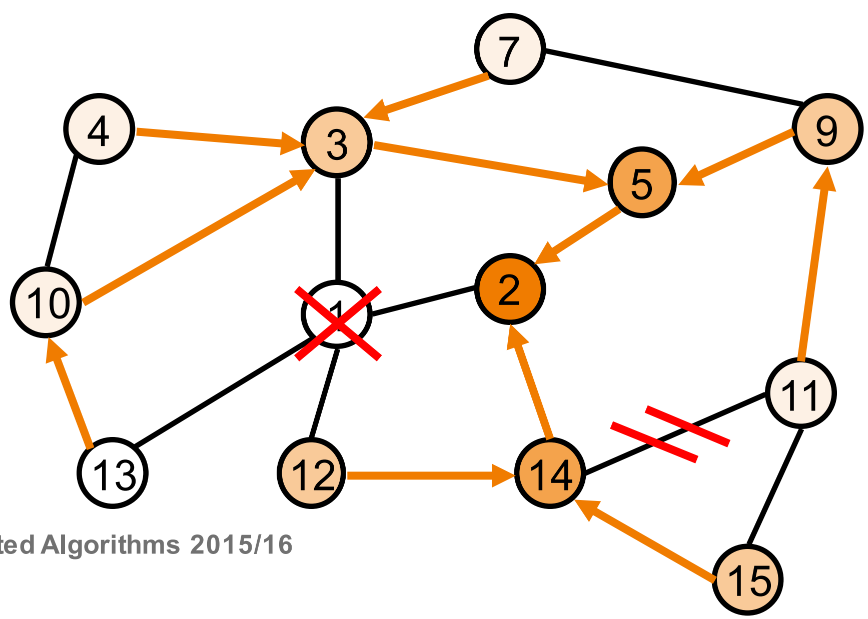
Self-Stabilizing Span Tree Construction – Basic Idea in the Fault-free Case

- Root (node with smallest ID) sends with period ρ heartbeats to all its neighbors
- Nodes relay received heartbeats to all other neighbors
- Each node elects the neighbor as father that lies closest to the root
- In case of equality, the node with smaller ID is elected
- Received heartbeat suppresses the desire of other nodes to become the root → delivery in time must be ensured



Self-Stabilizing Span Tree Construction – Basic Idea in Case of a Fault

- If a connection fails, another spanning tree with the same root node forms
- If the root fails or if the root is no longer reachable, another spanning tree with a different root node forms
- In both cases, the trigger for the formation of a new spanning tree is the occurrence of timeouts
- If the fault is transient, the original spanning tree forms again, after, e.g., the node is available again



Self-Stabilizing Span Tree Construction

- Each process P has
  - a variable P_V that points at one of its neighbors (its father)
  - a variable P_W that points at the current root
  - a variable P_L that indicates its level in the tree and
  - a variable P_F that is read out in case of a timeout and possibly changed

- Aim: After finite time the P_V –references of all nodes shall form a spanning tree

- Remark for the next slide:
  - $(v_1, v_2, v_3) < (w_1, w_2, w_3)$
    $\Leftrightarrow v_1 < w_1 \vee (v_1 = w_1 \wedge v_2 < w_2) \vee (v_1 = w_1 \wedge v_2 = w_2 \wedge v_3 < w_3)$

Self-Stabilizing Spanning Tree Construction

- A node P receives a message (w, l, i)
  - If $(P_W, P_L, P_v) < (w, l + 1, i)$ or $P < w$, the node ignores the message
    → **Message is not eligible**
  - If $(P_W, P_L, P_v) = (w, l + 1, i)$, it sets P_F to 2 und sends a message (P_W, P_L, P) to all other neighbors
    → **Eligible refresh message from current root**
  - Otherwise, it sets $(P_W, P_L, P_v) := (w, l + 1, i)$ and $P_F := 2$ and sends a message (P_W, P_L, P) to all other neighbors
    → **New root**

Fault-free Case

Faulty Case

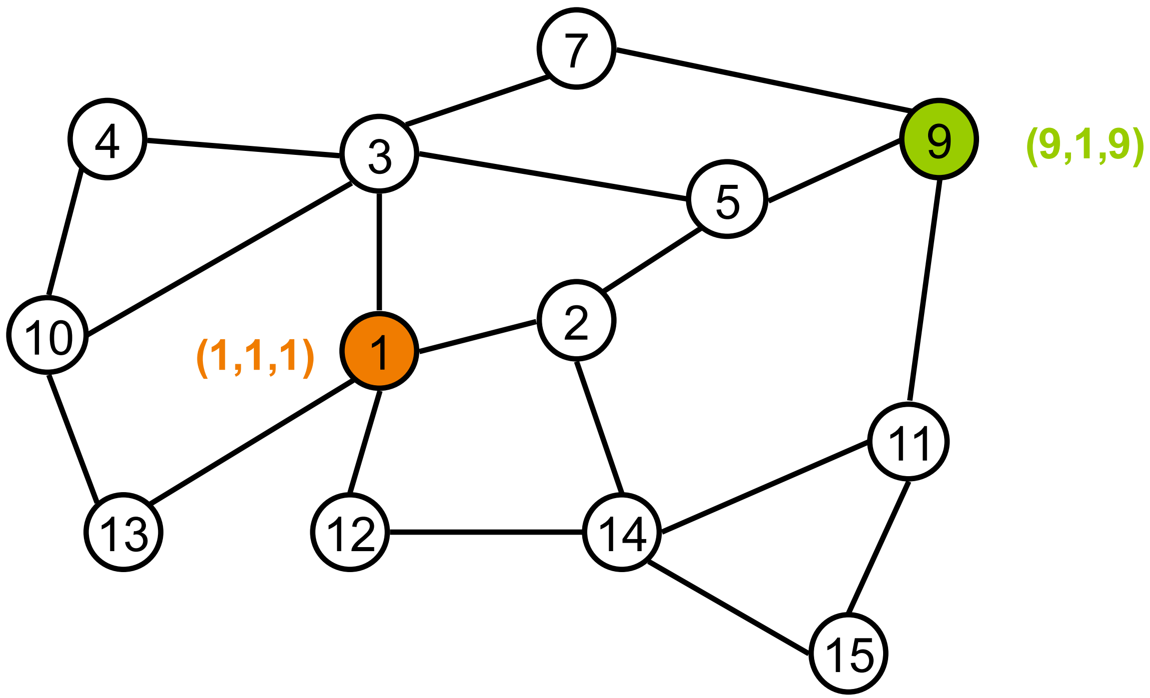
Self-Stabilizing Spanning Tree Construction

- When the timeout (ρ) occurs at process P:
  - If $P_F \leq 0$, it sets $(P_W, P_L, P_v) := (P, 1, P)$, leaves P_F unchanged and sends a message (P_W, P_L, P_v) to all neighbors
    → **Node declares itself to new root node**
  - **If $P_F = 1$, $P_F := 0$**
  - **If $P_F \geq 2$, $P_F := 1$**
  - In every case, the timer is reset and restarted

Fault-free Case

Faulty Case

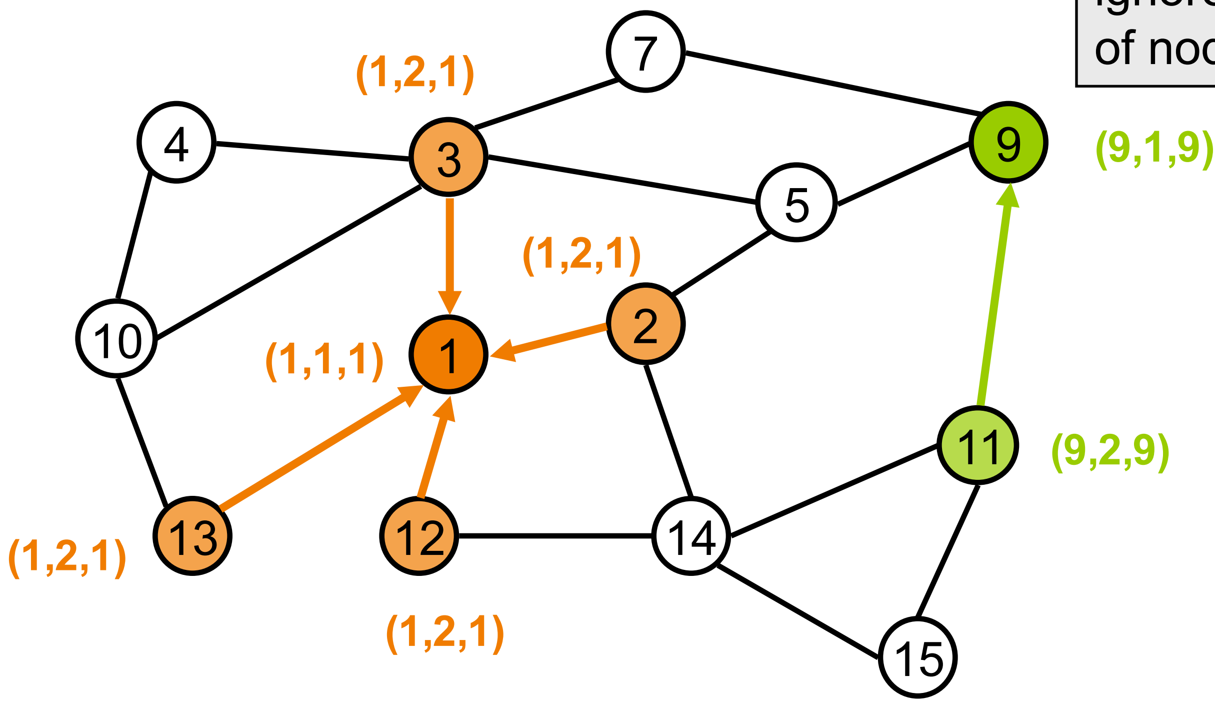
Self-Stabilizing Span Tree Construction



(Root, Level, Father)

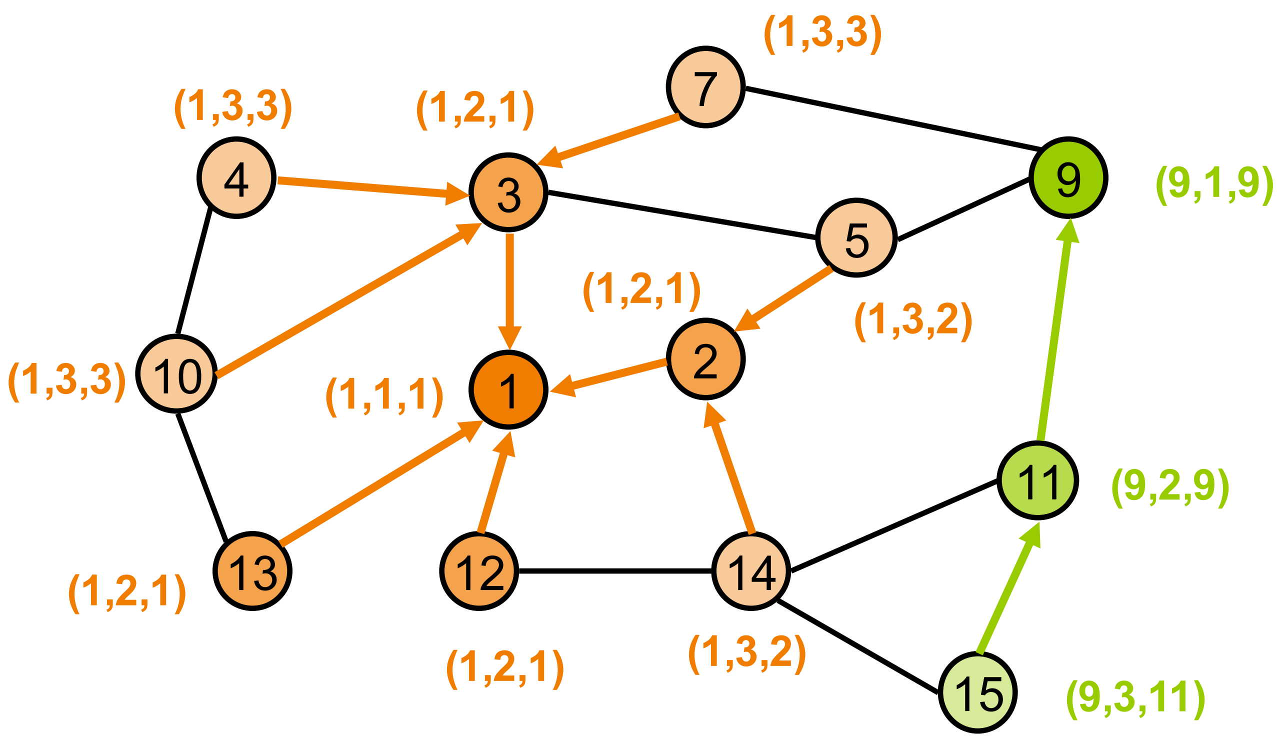
Self-Stabilizing Span Tree Construction

Nodes 5 and 7 ignore the message of node 9!



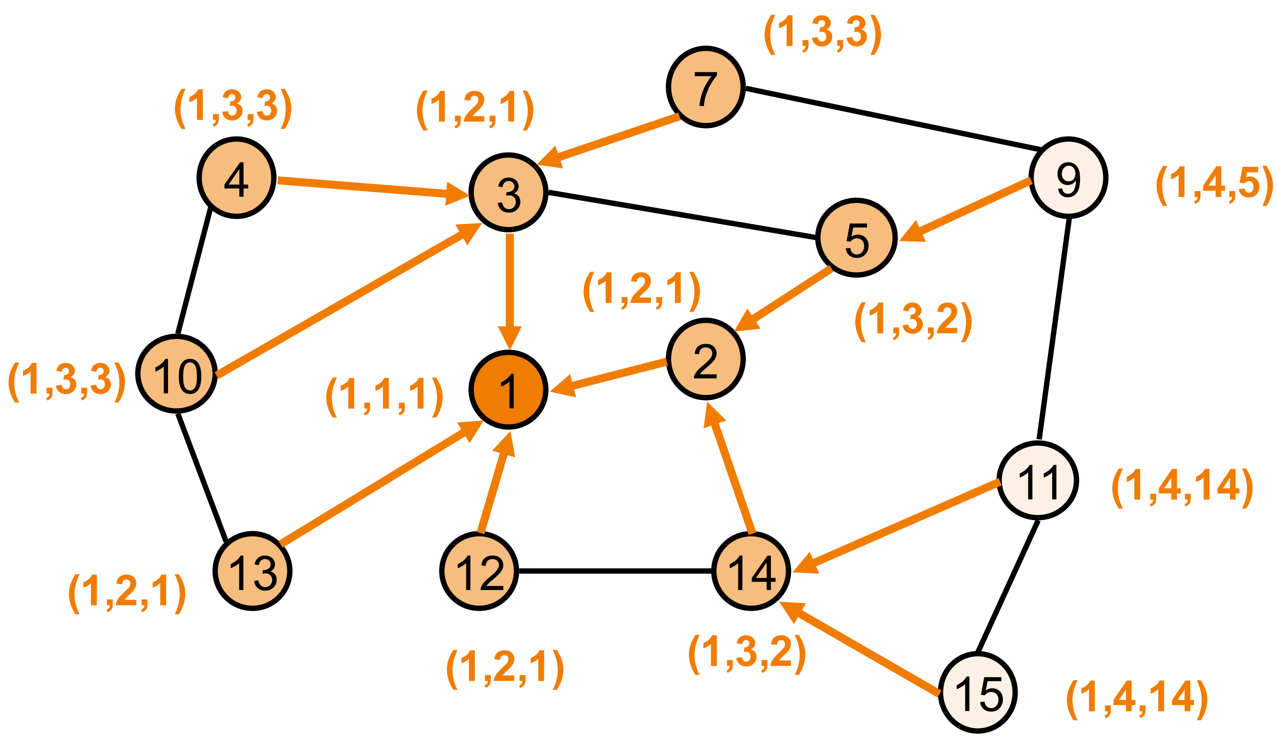
(Root, Level, Father)

Self-Stabilizing Span Tree Construction



(Root, Level, Father)

Self-Stabilizing Span Tree Construction



(Root, Level, Father)

Characteristics of the Constructed Spanning Tree / Proof of Convergence

- Constructed tree is unique in the fault-free case because
  - the process with the smallest identity becomes the only root and
  - each process, except for the root, elects that process as its predecessors that has the smallest identity among the neighbors with the smallest level
- In the fault-free case, only the refreshment messages triggered by the root are on the way

- The system is in a legal state if
  - The state of the processes conforms to the spanning tree introduced above and
  - No *faulty* messages are on the way anymore

- To prove *convergence* it has to be shown that the system, starting from an arbitrary state, reaches a legal state

Ensuring Closure

- Each node except for the root node always has to receive a refreshment message in time, otherwise it would declare itself as root

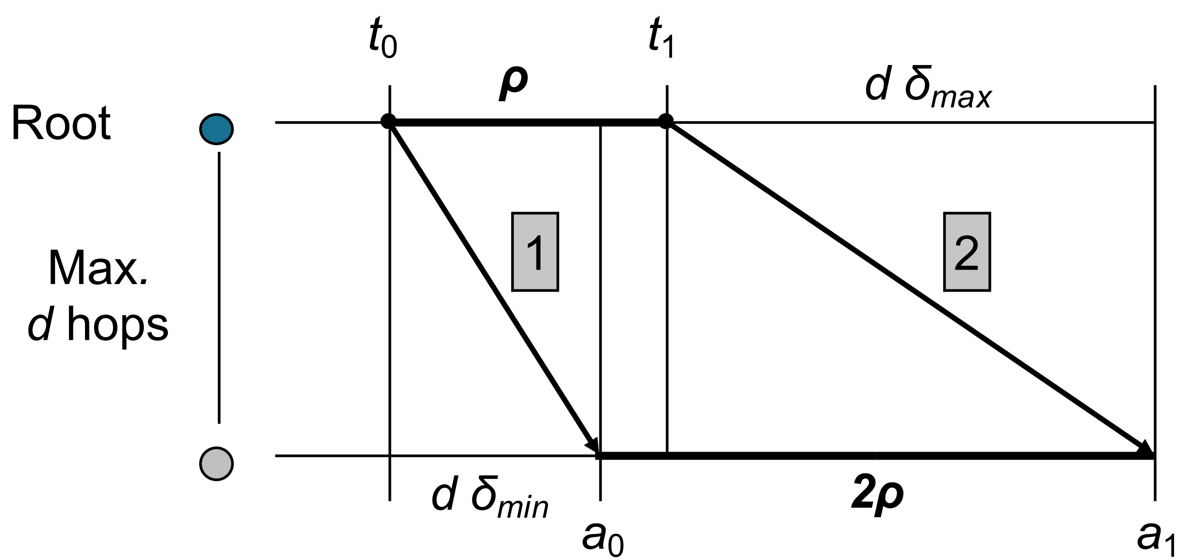
- Let δ_{min} be the minimal and δ_{max} the maximal message delay on a link and d the length of the *longest* path in the topology
- The height of the resulting tree is always less than or equal to d

- The maximal time between two refreshment messages occurs
  - if the root and the considered node are maximally far away from each other (max d hops) and
  - the 1st message is minimally (→ $d\ \delta_{min}$) and
  - the 2nd message is maximally (→ $d\ \delta_{max}$) delayed

Ensuring Closure

- The receipt of the first refreshment message sets P_F to 2
- The first timeout (which sets P_F to 1) can occur directly after the receipt
- The second refreshment message has to arrive before two further timeouts occur

$$a_0 = t_0 + d\,\delta_{min}$$
$$a_1 = t_1 + d\,\delta_{max}$$
$$t_1 = t_0 + \rho$$



$$2\rho > a_1 - a_0$$
$$\Downarrow$$
$$2\rho > \rho + d\,(\delta_{max} - \delta_{min})$$
$$\Downarrow$$
$$\boldsymbol{\rho > d\,(\delta_{max} - \delta_{min})}$$

Literature

1. **E. W. Dijkstra. Self-Stabilizing Systems in Spite of Distributed Control. Communications of the ACM, 17(11):643--644, 1974.**
2. S. Dolev. Self-Stabilization. MIT Press, 2000.
3. M. Schneider. Self-stabilization. ACM Computing Surveys, 25(1):45--67, 1993.
4. F. C. Gärtner. A survey of self-stabilizing spanning-tree construction algorithms. Technical Report 200338, Swiss Federal Institute of Technology (EPFL), School of Computer and Communication Sciences, Lausanne, Switzerland, June 2003.