Robust Architectures for Distributed Systems and Topological Self-Stabilization







Stefan Schmid
(T-Labs / TU Berlin)

Joint work with:
Matthias Baumgart (TUM)
Dominik Gall (TUM)
Riko Jacob (TUM)
Fabian Kuhn (USI)
Andrea Richa (ASU)
Christian Scheideler (UPB)
Hanjo Täubig (TUM)
Roger Wattenhofer (ETH)

ACM WRAS 2010
Third ACM International Workshop on Reliability, Availability, and Security

Natural R&A by scalability and redundancy (but also more exposed)

Robust Architectures for *Distributed Systems* and Topological Self-Stabilization







How to design?

Stefan Schmid
(T-Labs / TU Berlin)

Joint work with:
Matthias Baumgart (TUM)
Dominik Gall (TUM)
Riko Jacob (TUM)
Fabian Kuhn (USI)
Andrea Richa (ASU)
Christian Scheideler (UPB)
Hanjo Täubig (TUM)
Roger Wattenhofer (ETH)

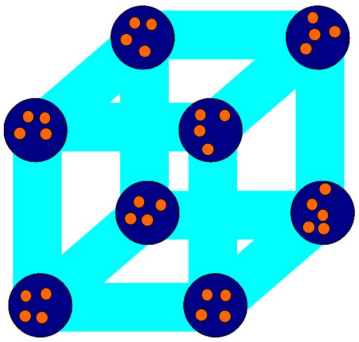
Talk outline: R&A Distributed Systems

1. Some design principles:
- Maintaining overlay topology under worst-case churn?
- Secure data repliation despite past-insider adversary?
- Connect to the seniors!

2. Towards self-repairing systems

... interactive...

Talk outline



1. Some design principles:
- Maintaining overlay topology under worst-case churn?
- Secure data repliation despite past-insider adversary?
- Connect to the seniors!

2. Towards self-repairing systems

Talk outline

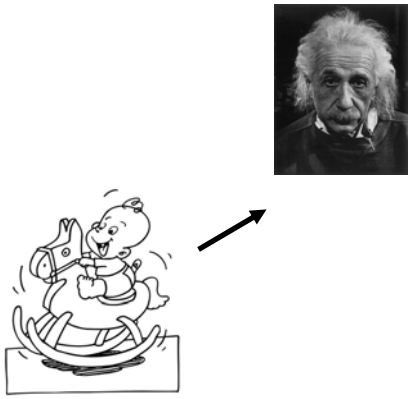
1. Some design principles:
- Maintaining overlay topology under worst-case churn?
- Secure data repliation despite past-insider adversary?
- Connect to the seniors!

2. Towards self-repairing systems

Talk outline

1. Some design principles:
- Maintaining overlay topology under worst-case churn?
- Secure data repliation despite past-insider adversary?
- Connect to the seniors!

2. Towards self-repairing systems



Talk outline

1. Some design principles:
- Maintaining overlay topology under worst-case churn?
- Secure data repliation despite past-insider adversary?
- Connect to the seniors!

2. Towards self-repairing systems
(Exciting, not well-understood field!)

Use redundancy 1: reliable dynamic topologies

Maintaining a System under Worst-Case Churn

Design of Dynamic Distributed System?

An algorithmic challenge: How many nodes can join and leave a network (e.g., a p2p system) *per unit time* (e.g., max transmission time) such that

- Network remains connected?
- Network maintains hypercubic structure?
- Network still allows for logarithmic time routing / search?

Idea: Simulated / redundant graphs!

Recipe

1. Take a graph with desirable properties

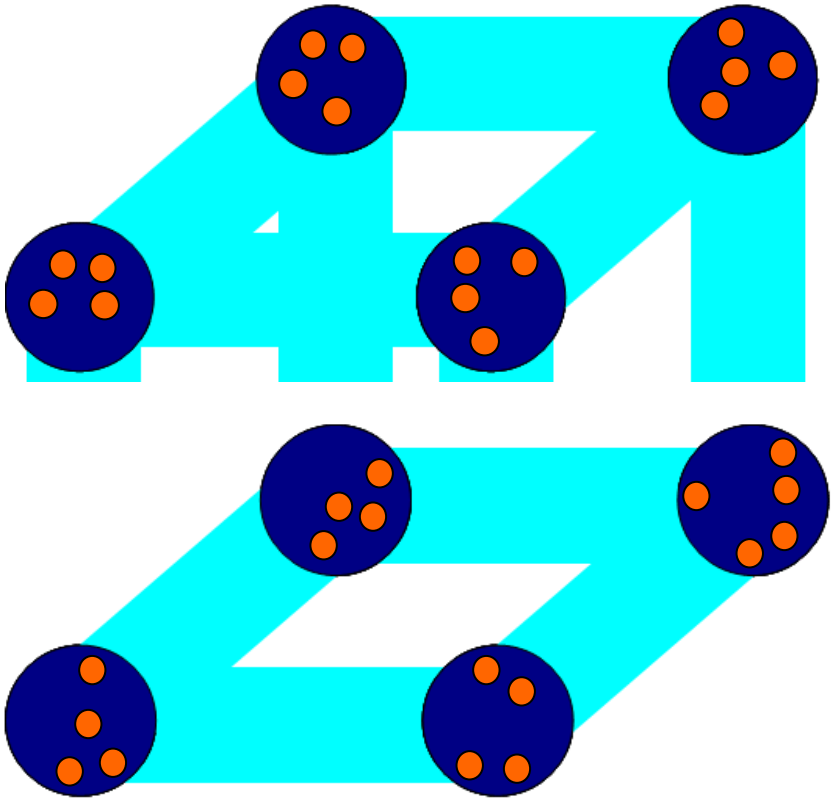
2. Simulate the graph by representing each vertex by a set of nodes (Goal: keep graph properties!)

3. Find a token (node) distribution algorithm on this graph

4. Find an algorithm to estimate the total number of nodes in the system

5. Find an algorithm to adapt the graph‘s dimension

Example: Hypercube



Result

Theorem:

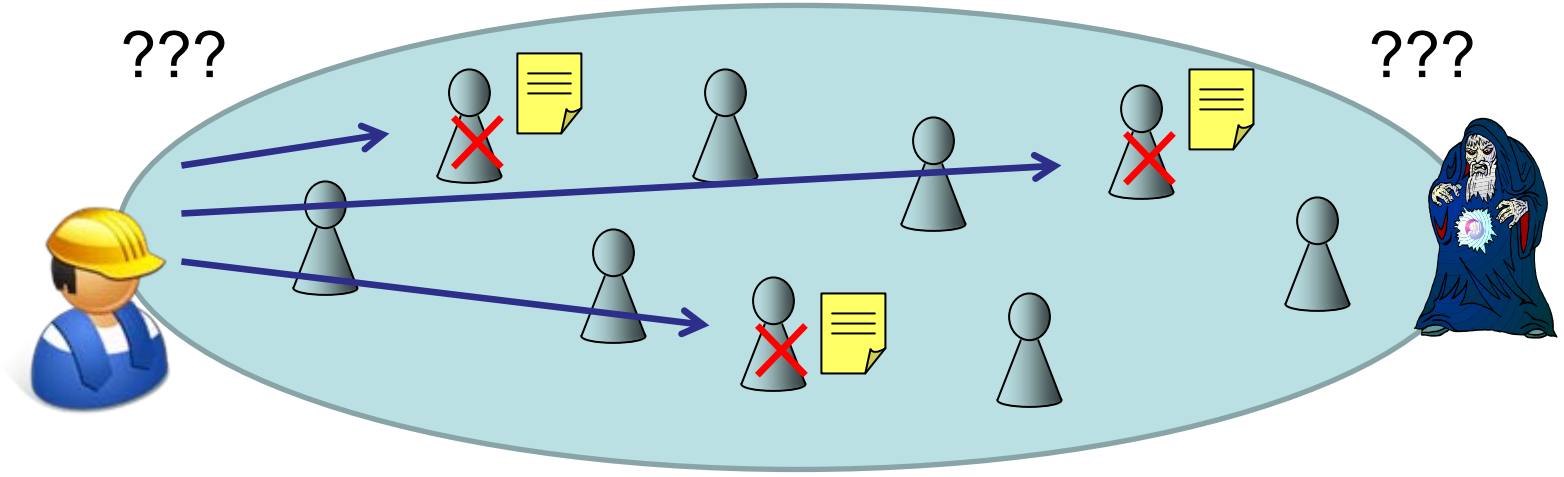
- Despite ADV(log n, log n, 1), topology can be maintained
- Peer degree and network diameter (connections: matching of cliques): O(log n)
- Asymptotically optimal (why?)
- Similarly for pancake graphs: replace log n by log n / loglog n (weaker adversary!)
- not self-stabilizing (see later)

Use redundancy 2: How to achieve reliable storage?

Chameleon: Robust data replication

Idea:

- Store data redundantly (goal: minimal blow-up factor)
- Even a *past insider* should not know where it is stored
  (As soon as he is out, newly added data is hard to find!)
- Thus: Information system must „change its appearance“ over time
- How?
- Idea: Randomization (no fixed locations, e.g., depending on file name)



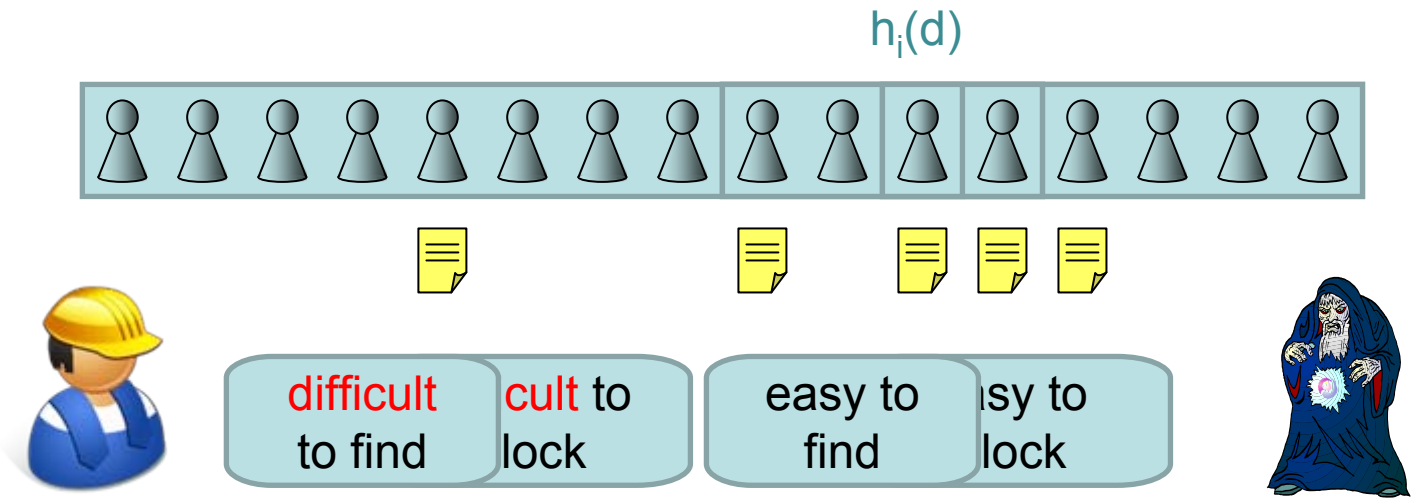
Don‘t know where
to attack – and search!

deterministic placement

randomized placement

Basic strategy: Random placements in increasing „vicinities“

- choose suitable hash functions $h_1,..,h_c:D \rightarrow V$
  (D: name space of data, V: set of servers)
- Store copy of item d for every i and j randomly in a set of servers of size 2^j that contains $h_i(d)$



Past-Insider-Attack: Attacker knows everything about system till (unknown) time t_0

Goal: scalable information system so that everything that was inserted after t_0 is safe (w.h.p.) against any past-insider DoS attack that can shut down any ε-fraction of the servers, for some $\varepsilon>0$, and create any legal set of put and get requests

What replication factor is needed to no data loss, efficient search and insertion if ε is a constant?

One would expect a linear number of replicas are needed, but polylog are enough!

It can be shown (see Awerbuch & Scheideler) that for sufficiently random placements (expansion property of hash functions), system is robust.

But what if attacker prevents proper replica placement during insertion?

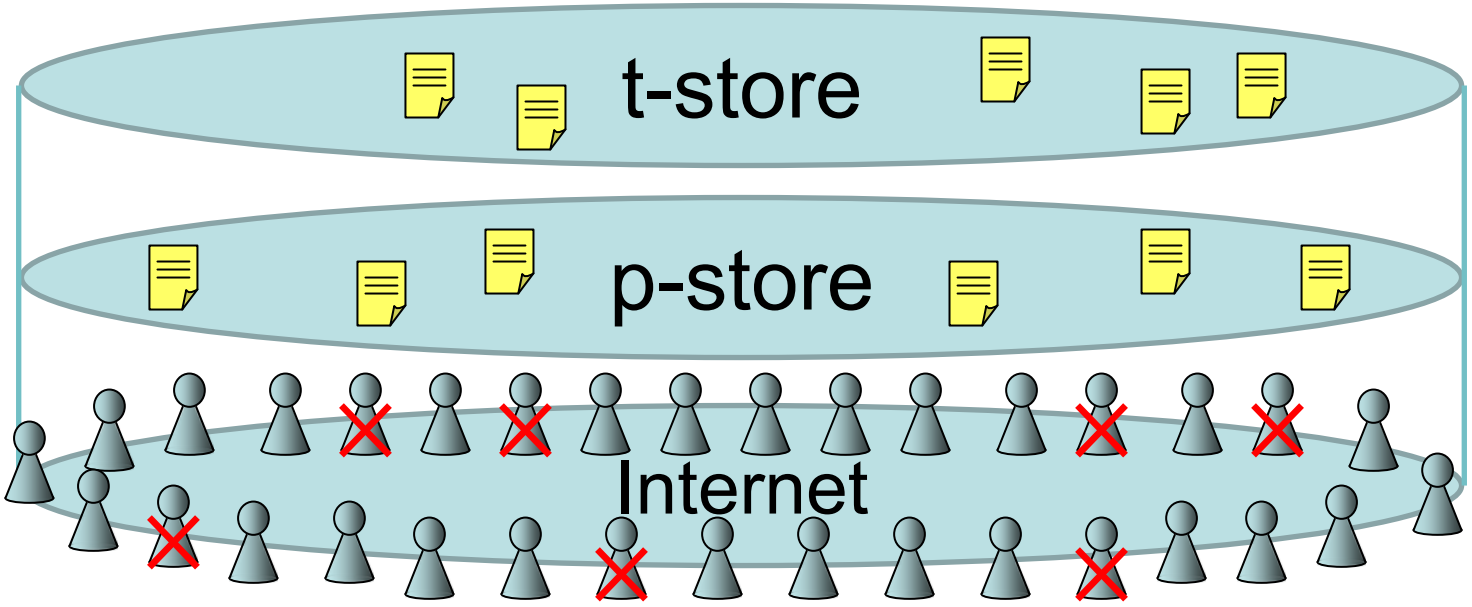
Idea:

- Use two stores (essentially DHTs!): a permanent p-store where data is replicated properly (robust to past insider)
- A temporary, always changing t-store where insert requests are buffered (not many!) until required replication level is guaranteed in p-store

Phase of Chameleon system:

1. Adversary blocks servers and initiates put & get requests
2. build new t-store, transfer data from old to new t-store (no quiet time as in DISC!)
3. process all put requests in t-store (de Bruijn like network)
4. process all get requests in t-store and p-store (detect block areas fast by sampling, no waste of resources to find)

1. try to transfer data items from t-store to p-store



Theorem: Under any ε-bounded past-insider attack (for some constant $\varepsilon>0$ that removes a constant fraction ε of all nodes), the Chameleon system can serve any set of requests (one per server) in $O(\log^2 n)$ time s.t. every get request to a data item inserted or updated after t_0 is served correctly, w.h.p.

[Baumgart, Scheideler, Schmid @ SPAA 2009]

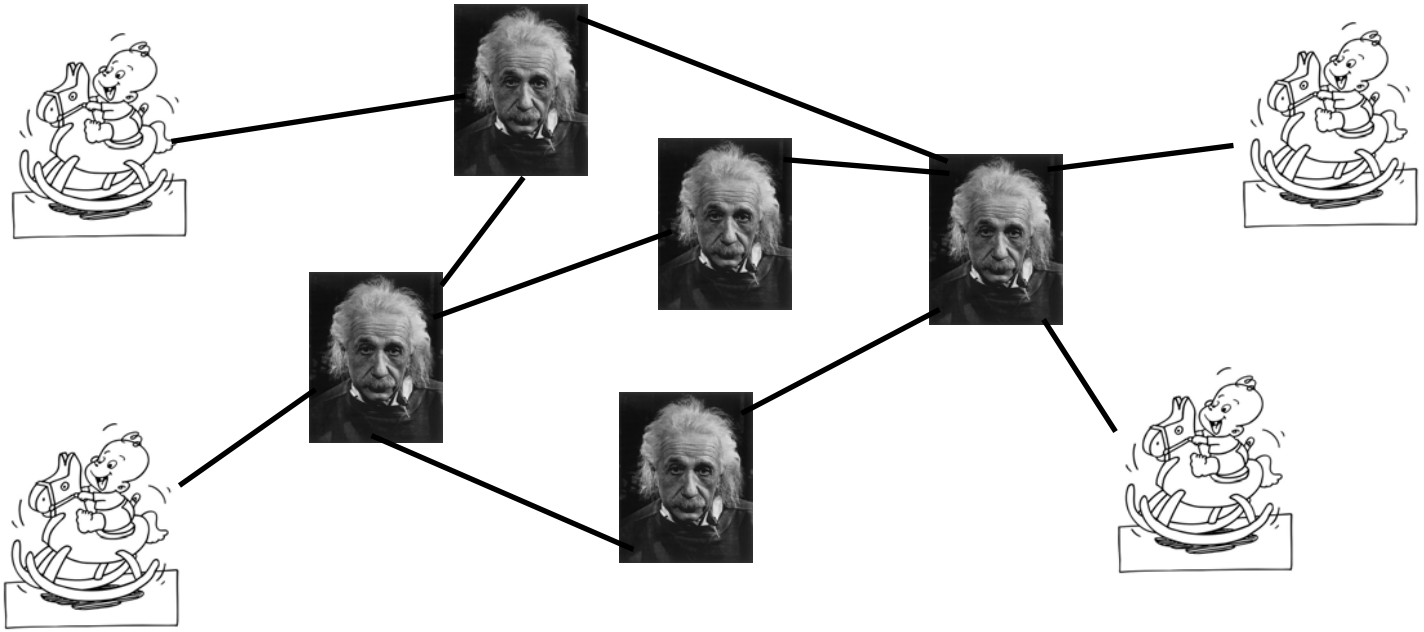
Reliability by connecting „to the seniors“:

Shell: Robust Distributed Heap

How to build robust systems? Connect to the seniors!

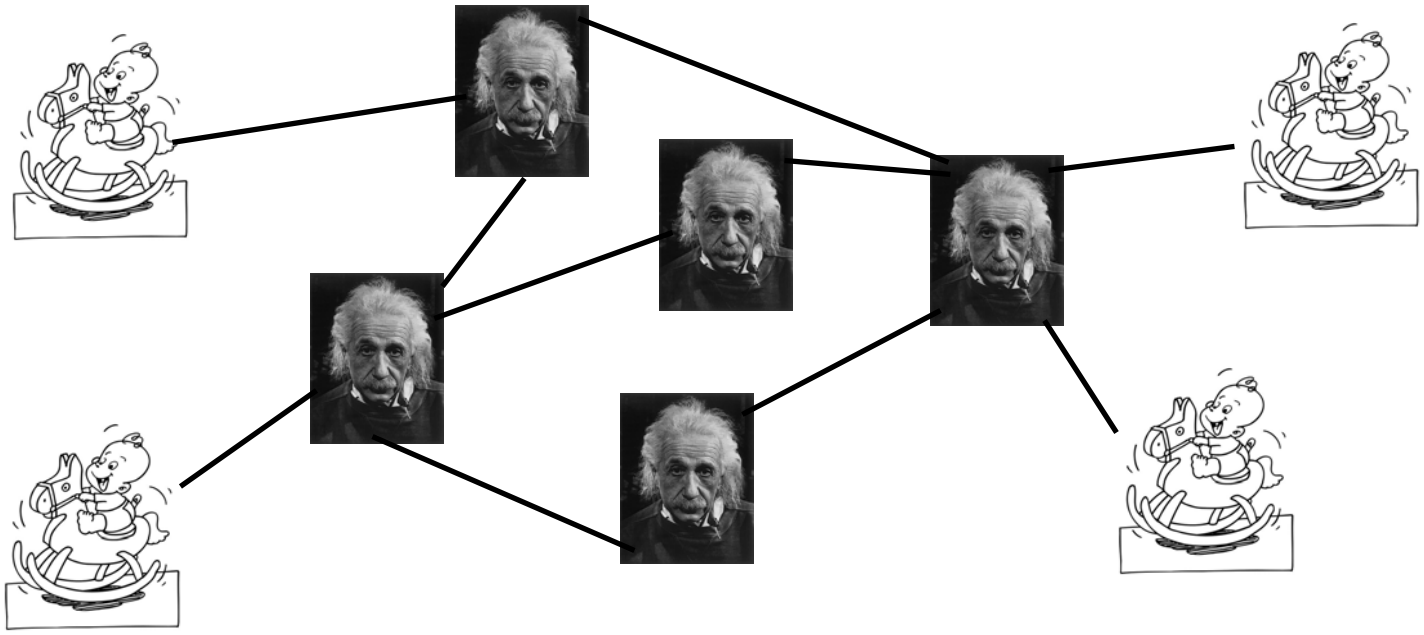
Idea: Older nodes typically less dynamic = more reliable?

Goal: Build a distributed system that allows for fast joins and leaves where younger peers only connect older peers!



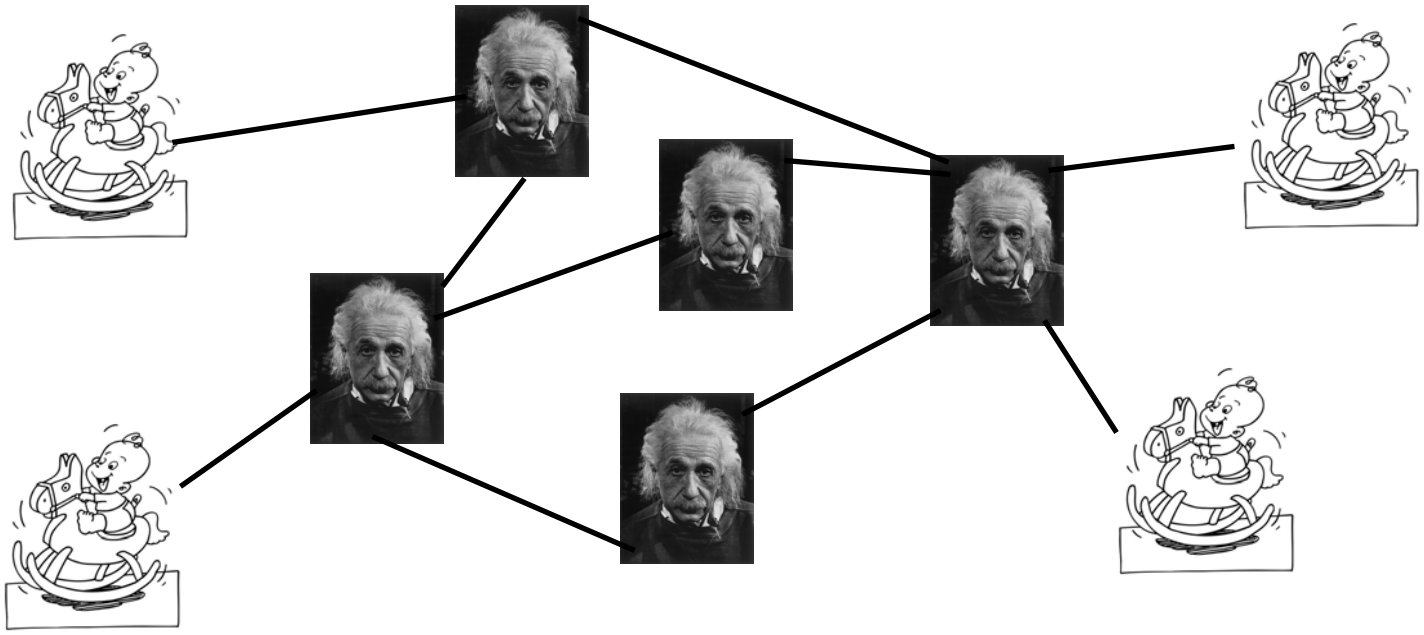
Idea (1)

- Idea: if everybody only connects to older network participants...
  - ... nodes would have stable neighborhoods!
  - ... one is resilient against attacks by „young troublemakers“



Idea (2)

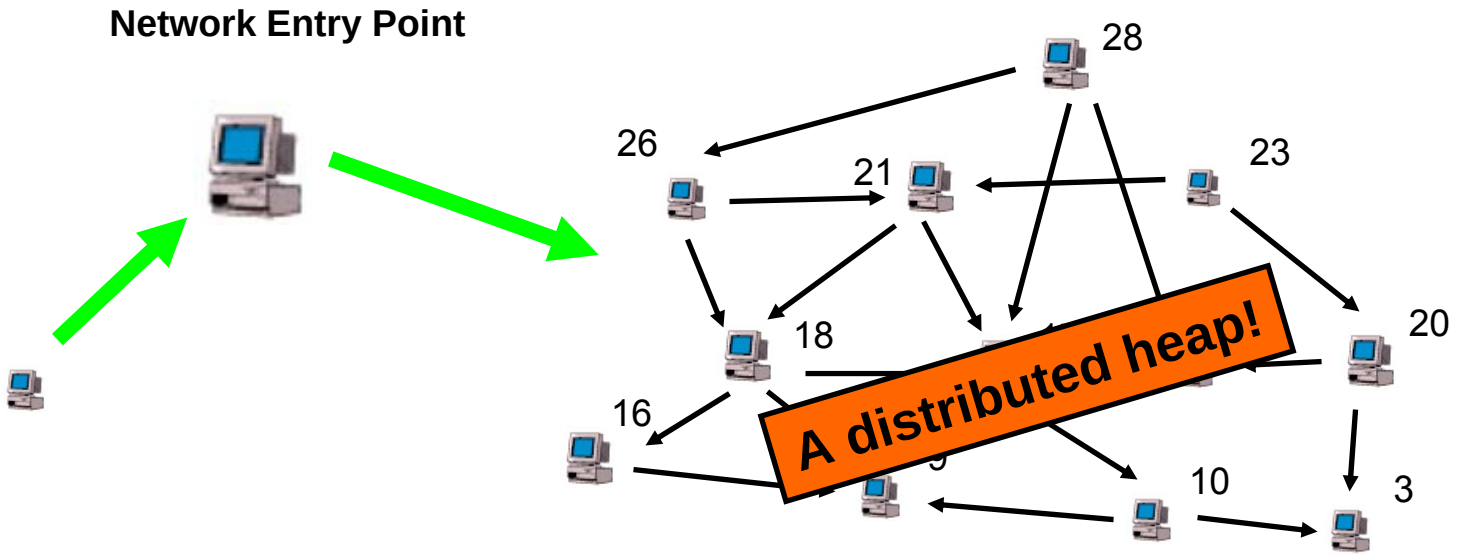
- Implications:
  - Communication paths of the „seniors“ never include younger nodes -
  - Young nodes cannot overload network (rate control in „core network“)



Model

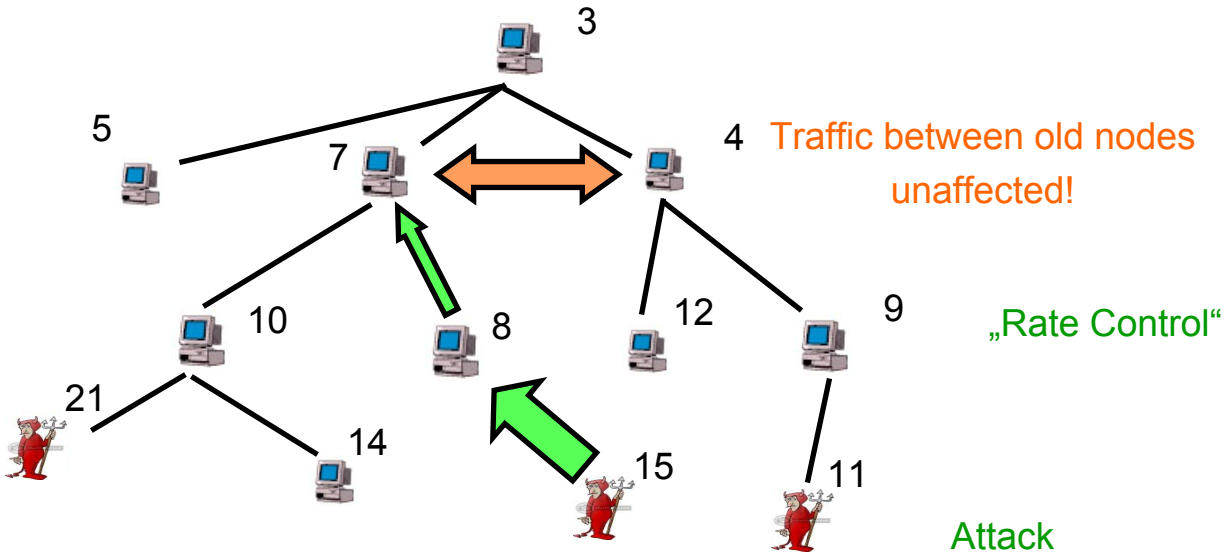
- How to implement such an idea?

- Idea: A central server assigns joining nodes a rank
  - Nodes only connect to nodes that arrived earlier (lower rank)



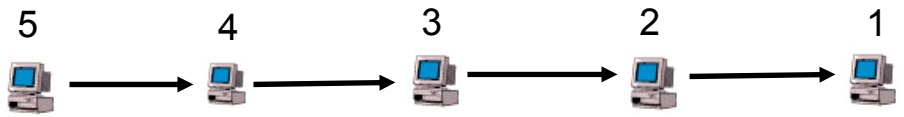
SHELL

Sybil attack by newcomers: No problem, traffic & access control to core network



A naive Solution

- Our goal is achieved with:



- Problem: Scalability
  - Large diameter, not robust to join/leave, etc.

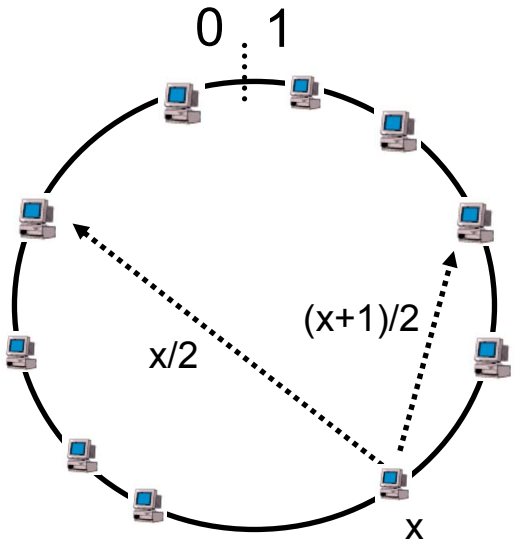
- Better topologies: hypercubes, pancake graphs, ...

Simple Approach for „good“ Peer-to-Peer Topologies

- Naor & Wieder: *The Continuous-Discrete Approach*

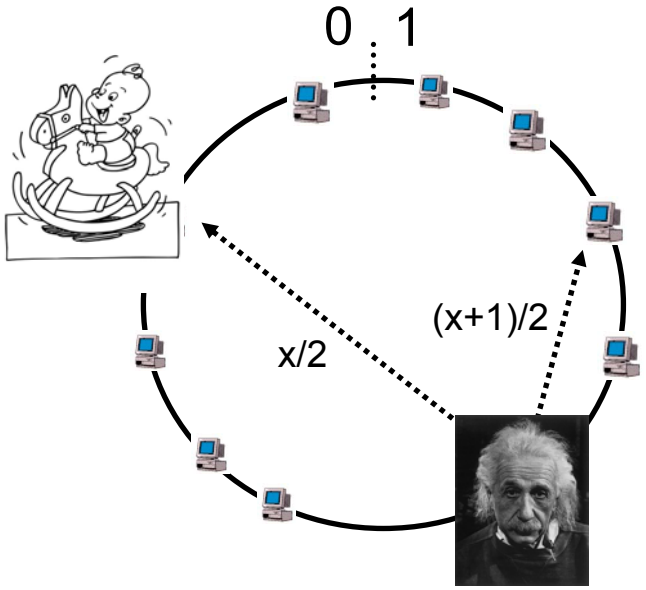
- Simplified version:
  - Nodes join at random position in **[0,1)**
  - Connects to points **x/2** and **(1+x)/2**
  - If there is no node, rounding is necessary („continuous => discrete“)
  - Details less interesting here

- Result: a kind of de Bruijn Graph
  - constant degree
  - logarithmic diameter
  - simple routing



Idea and Problem

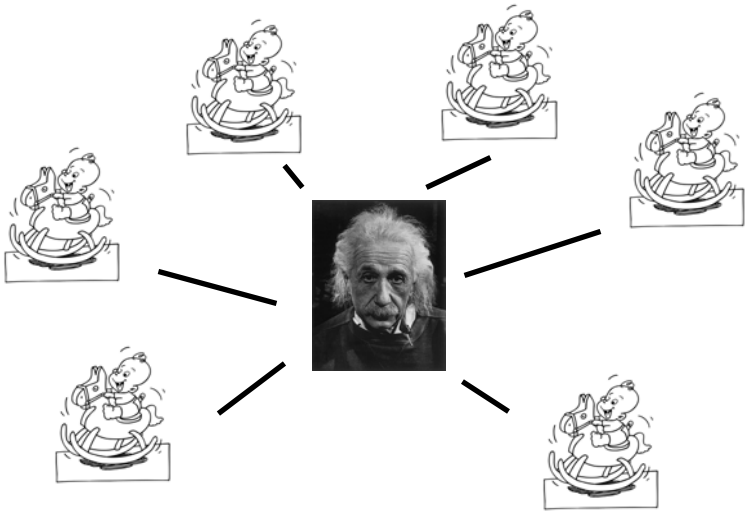
- Network Entry Point assigns random position [0,1)...
- ... and then build topology according to Continuous-Discrete Approach!
- Problem?



There could be a younger node there!
Solution?

Solution and another Problem

- Connect to corresponding older node close to this position
- Everything solved? Other problems?
- Analysis shows, that older nodes can be congested, as everybody tries to connect to them!

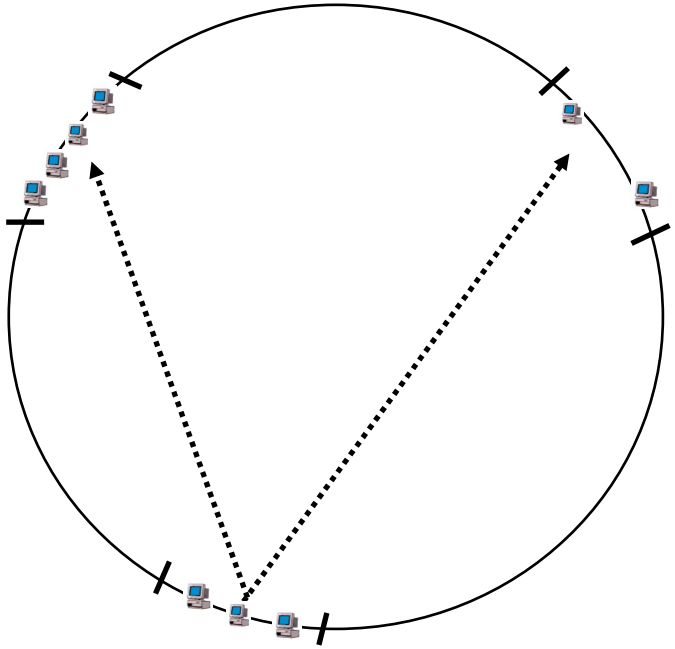


Idea?

Redundancy

- Solution: we connect to more than one node!
  (During routing, go to youngest neighbor among older ones...)

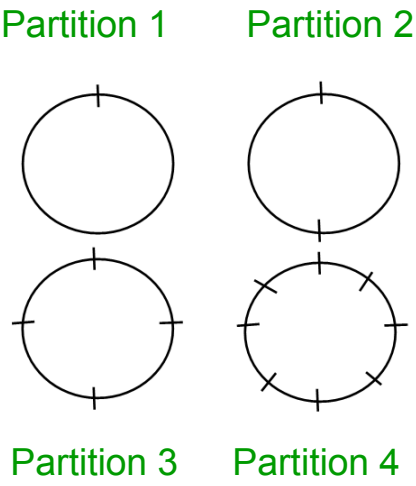
- Allows us to „load-balance“



The Algorithm (1)

- Assume, each node u knows n_u = # living nodes that are older
  (can be estimated, see later)

- Divide [0,1) Circle in fixed Intervals / Levels of exponentially decreasing sizes



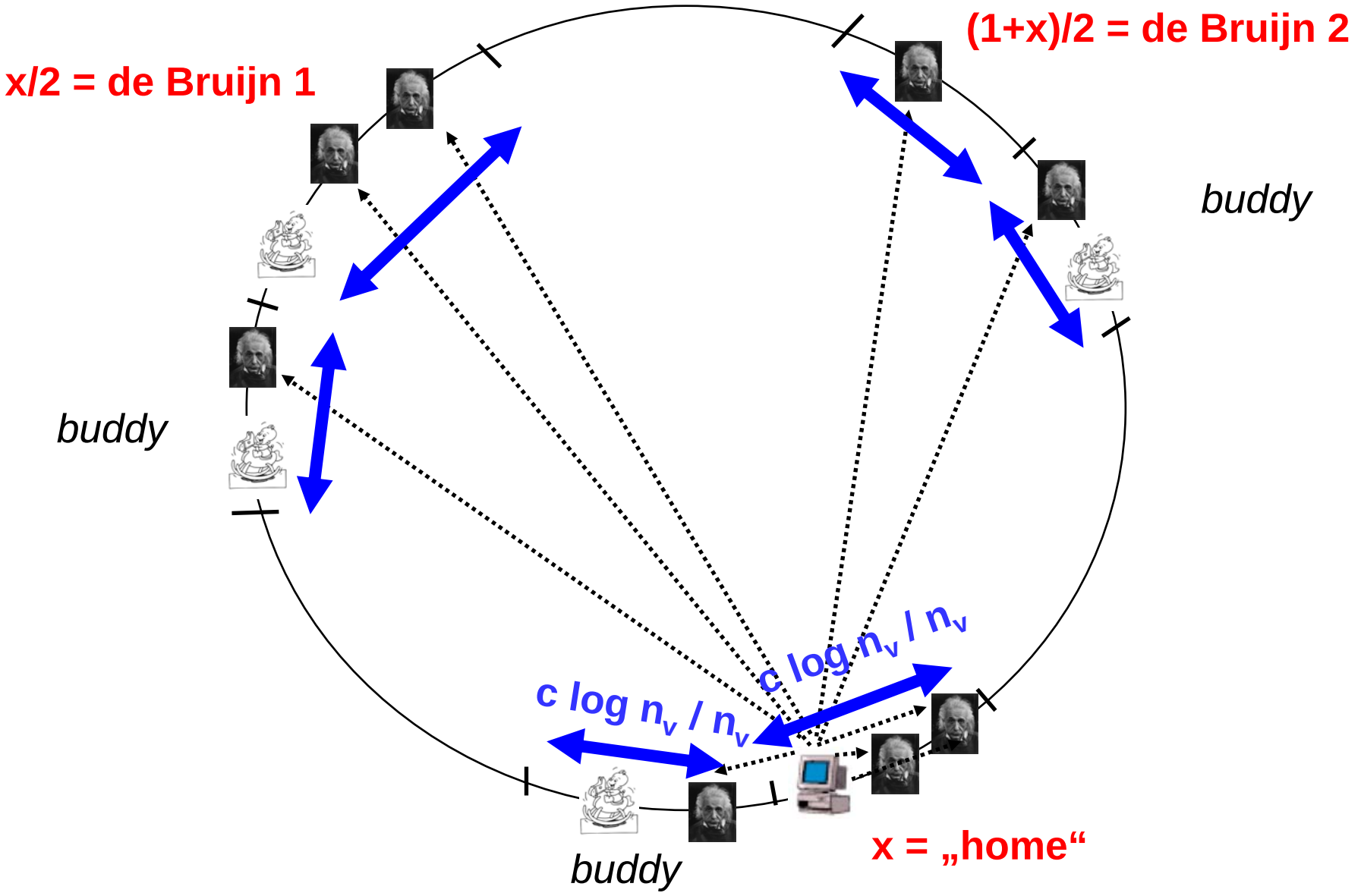
The Algorithm (2)

- Node v connects to three Intervals
  - $\mathbf{I_{v,0}}$ & buddy: Home-interval with position x
    plus other half of the (i-1)-interval
  - $\mathbf{I_{v,1}}$ & buddy: Interval with position x/2, plus buddy
  - $\mathbf{I_{v,2}}$ & buddy: Interval with position (1+x)/2, plus buddy

- Interval is chosen such that it includes at least $\mathbf{c \log n_v}$ older nodes (c = const.)! (If not possible, set level to 0.)

- Establish forward edges to these nodes. Store all incoming edges as backward edges!

Overview „Forward Edges“

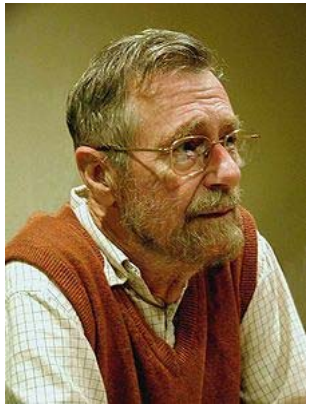


Unfortunately, not many distributed systems fulfill a very appealing robustness requirement:

Self-Stabilization

Self-Stabilization

- Important concept in fault-tolerance
- A self-stabilizing system (eventually) ends up in a correct state...
- ... independently of the initial state.



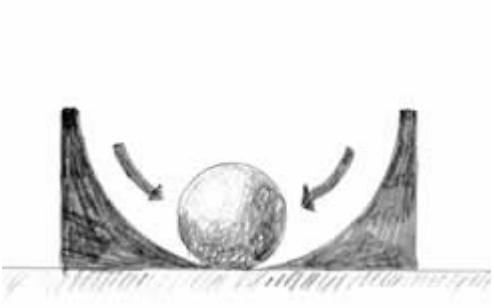
„All the designs I was familiar with were not self-stabilizing in the sense that when once (erroneously) in an illegitimate state, they could – and usually did! – remain so forever.“

E. W. Dijkstra (1974)

Self-Stabilization

- Model: Adversary can disturb the computations (shared variables in system state) arbitrarily

- Once the changes are over, algorithm converges towards desired state

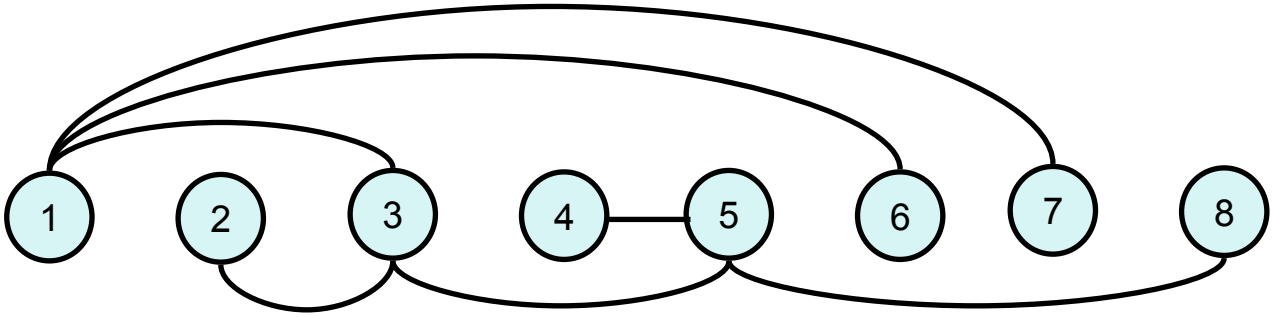


- Good news:
  „Awerbuch & Varghese 91“: (cf also Wattenhofer @ SSS 2009)
  local algorithms = self-stabilizing algorithms
- However:
  - if applied for dynamic topologies the overhead is large
  - randomized local algorithms less understood

Goal: *Terminator 2*!

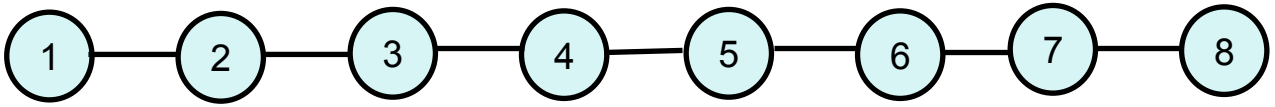
Most Simple Topological Stabilization: Graph Linearization

- INPUT: Arbitrary connected graph
  - nodes have arbitrary IDs



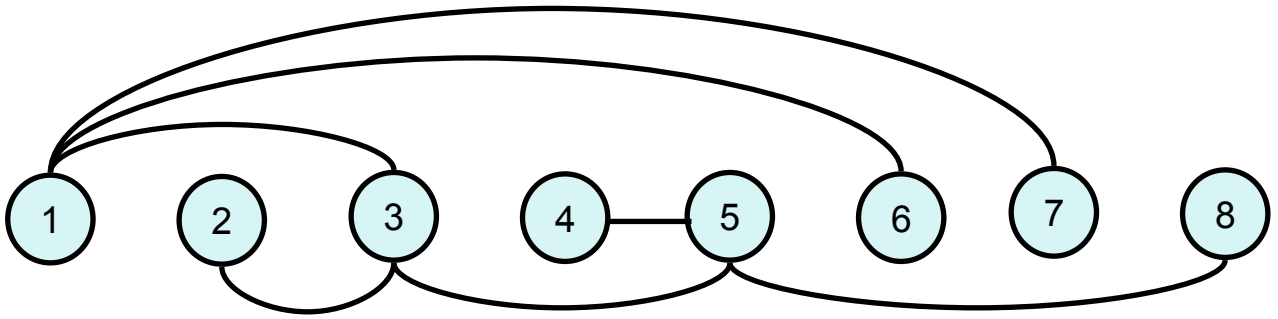
Most Simple Topological Stabilization: Graph Linearization

- OUTPUT: Sorted graph



Most Simple Topological Stabilization: Graph Linearization

- Ideas: How to linearize locally?
  To preserve connectivity?



Basic Linearization Step

- A basic linearization step involves a node triple

- Observe: Connectivity is preserved

LIN_{all} and LIN_{max}

- LIN_{all} proposes *all* possible triples to the scheduler (for node *u*)

$$\textbf{linearize left}(v,w): \quad (v,w \in u.L \ \wedge \ w < v < u) \ \rightarrow \ e(u,w) := 0, e(v,w) := 1$$

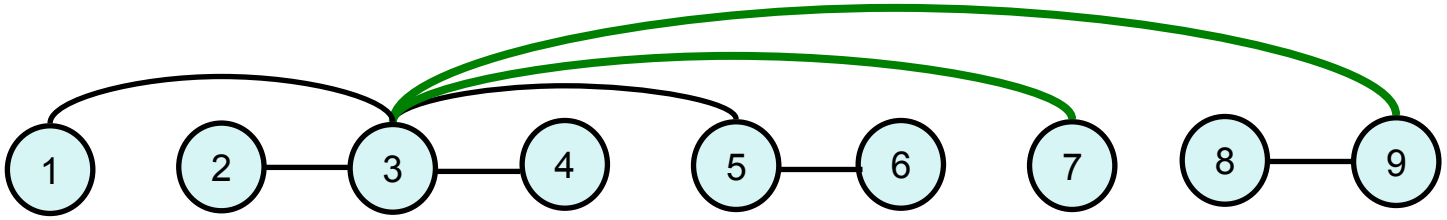
$$\textbf{linearize right}(v,w): \quad (v,w \in u.R \ \wedge \ u < v < w) \ \rightarrow \ e(u,w) := 0, e(v,w) := 1$$

- LIN_{max} proposes the *furthest* triple on each side (for node *u*)

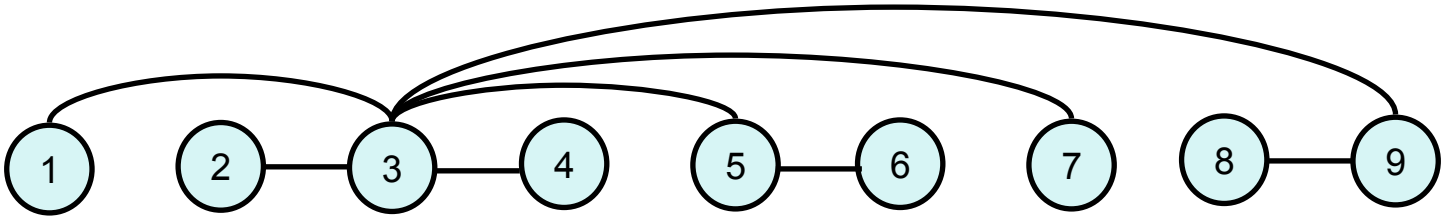
$$\textbf{linearize left}(v,w):$$
$$(v,w \in u.L) \ \wedge \ w < v < u \ \wedge \nexists x \in u.L \setminus \{w\} : x < v) \ \rightarrow \ e(u,w) := 0, e(v,w) := 1$$
$$\textbf{linearize right}(v,w):$$
$$(v,w \in u.R) \ \wedge \ u < v < w \ \wedge \nexists x \in u.R \setminus \{w\} : x > v) \ \rightarrow \ e(u,w) := 0, e(v,w) := 1$$

LIN_{all} and LIN_{max}

- **LIN_{all} proposes *all* possible triples to the scheduler (for node *u*)**

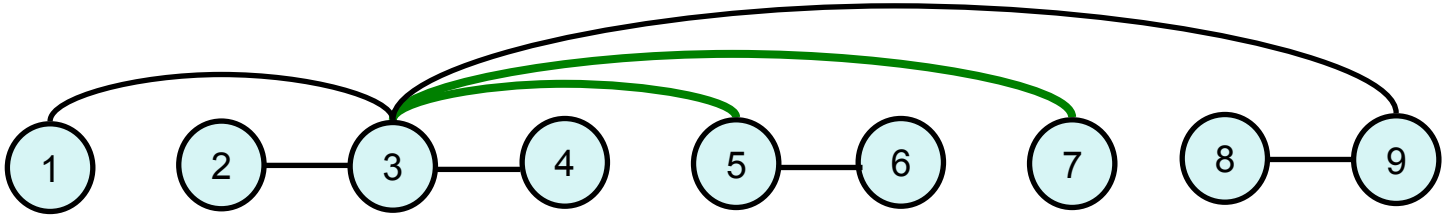


- LIN_{max} proposes the *furthest* triple on each side (for node *u*)

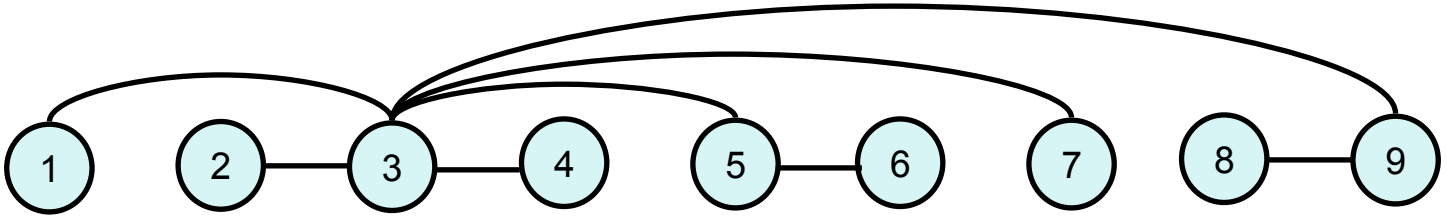


LIN_{all} and LIN_{max}

- **LIN_{all} proposes *all* possible triples to the scheduler (for node *u*)**

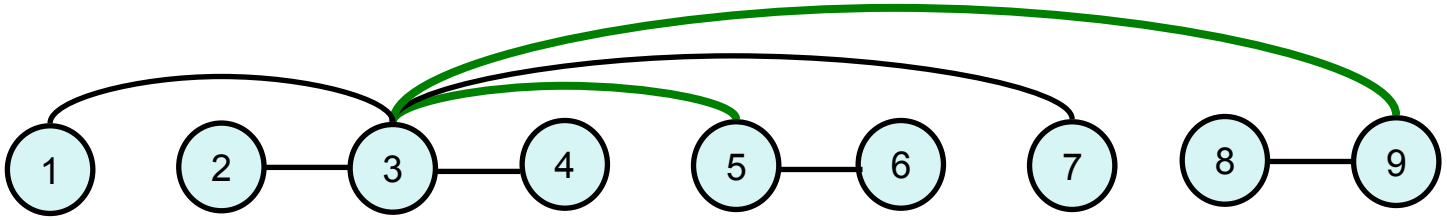


- LIN_{max} proposes the *furthest* triple on each side (for node *u*)

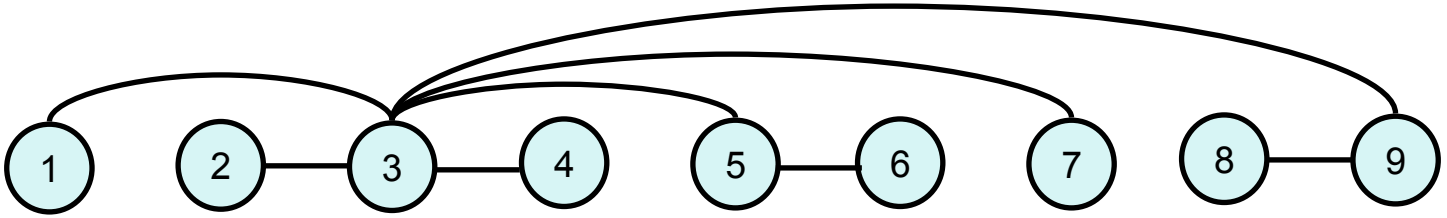


LIN_{all} and LIN_{max}

- **LIN_{all} proposes *all* possible triples to the scheduler (for node *u*)**

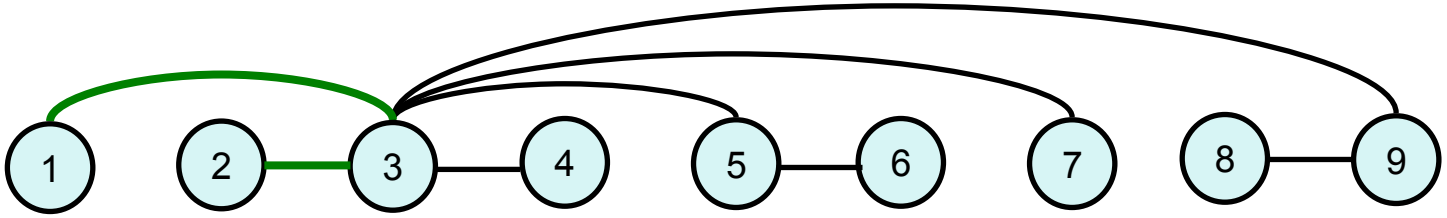


- LIN_{max} proposes the *furthest* triple on each side (for node *u*)

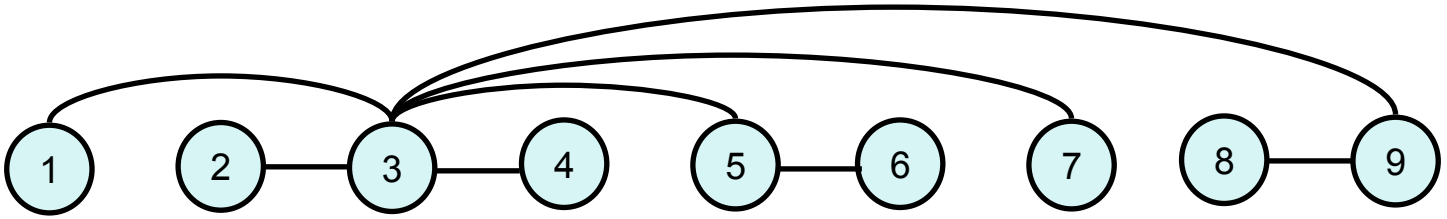


LIN_{all} and LIN_{max}

- **LIN_{all} proposes *all* possible triples to the scheduler (for node *u*)**

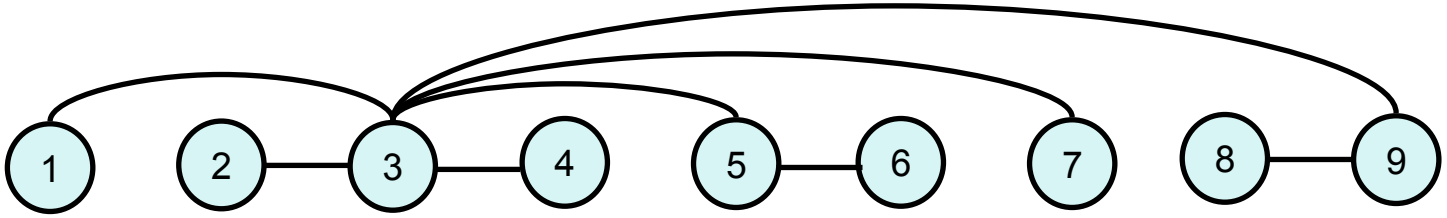


- LIN_{max} proposes the *furthest* triple on each side (for node *u*)

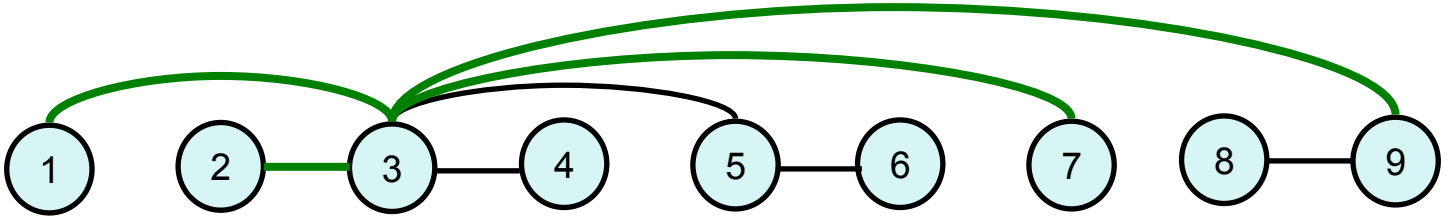


LIN_{all} and LIN_{max}

- LIN_{all} proposes *all* possible triples to the scheduler (for node *u*)



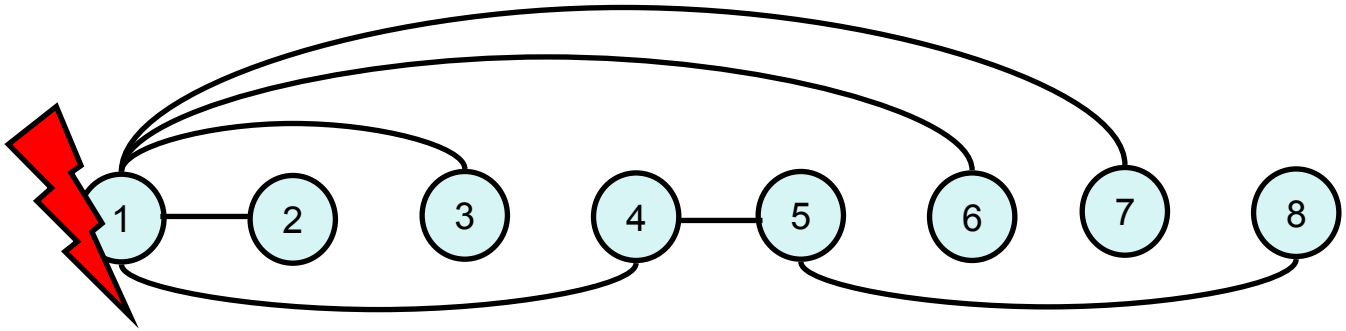
- **LIN_{max} proposes the *furthest* triple on each side (for node *u*)**



How to measure distributed execution time?

A Naïv Model

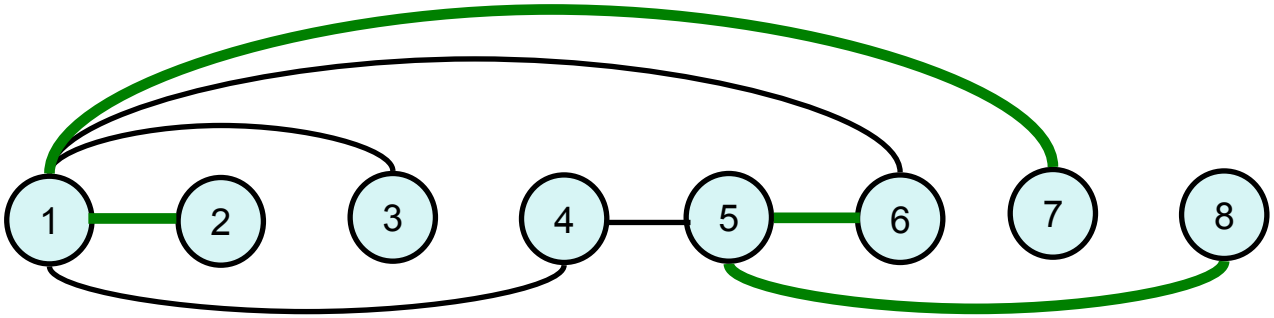
- There are different models for what can happen in one round!
- For example: Every node can fire one action per round
- Problem: Nodes can be involved in many changes
  - Therefore, this solution does not scale!



A Scalable Model

We propose the following, scalable model:

- Let V(A) be the nodes involved in an action A
- Two actions A and B are independent if V(A) ∩ V(B) = {}
- Only an independent set of actions is fired per round



Schedulers

- Nodes propose different enabled actions to the scheduler...
- ... – which one to choose?

Worst-case scheduler: chooses independent set of enabled actions which maximizes the runtime

Best-case scheduler: chooses independent set of enabled actions which minimizes the runtime

Randomized scheduler: chooses independent sets at random

Greedy scheduler: scheduler gives priority to nodes having a large degree

Analysis

- It turns out that already these simple algorithms are challenging!
- Overview of results:

Worst-case scheduler:
LIN_{max} requires $\Theta(n^2)$ rounds
LIN_{all} requires $O(n^2 \log n)$ rounds

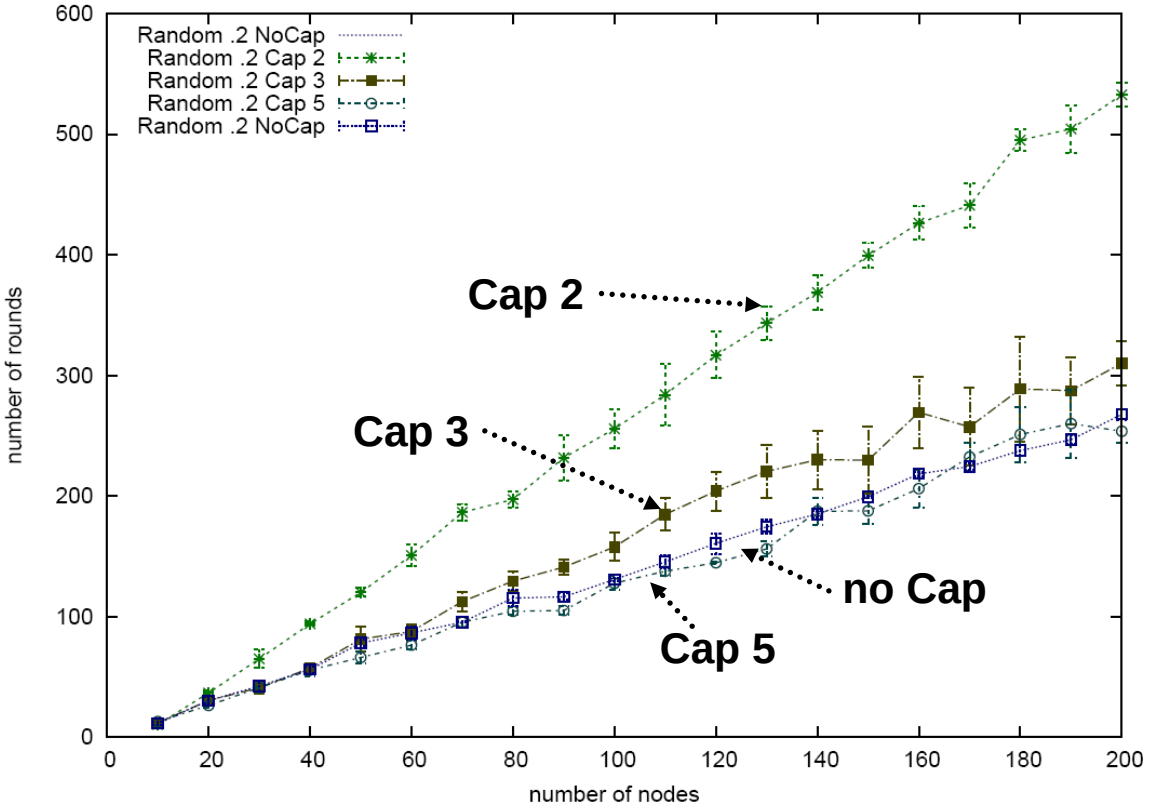
Greedy scheduler:
LIN_{all} requires $O(n \log n)$ rounds

Best-case scheduler:
LIN_{max} and LIN_{all} require $\Theta(n)$ rounds

With degree cap (worst-case scheduler):
LIN_{max} requires at most $O(n^2)$ and LIN_{all} at most $O(n^3)$ rounds

Degree Cap Phenomenon

- Degree cap constraint can sometimes improve the runtime! (Cap schedulers?)
  - degree grows only a „middle node“
  - cap: do not allow to increase degree if degree larger or equal cap
  - too small degree: blocks many options
  - however, small degree also forces execution on „good paths“



Sample Analysis (1)

Theorem: Under a greedy scheduler, LIN_{all} terminates after at most O(n log n) rounds.

Greedy scheduler: In each round, nodes are sorted w.r.t. remaining degree (remove fired triples with incident edges). Scheduler picks node *v* with largest degree, and schedules triple of *v* (to the larger degree side) with most distant neighbors.

Proof.

Consider the potential function $\Psi = \sum_{e \in E} \text{len}(e)$

Initially: $\psi_0 < n^3$

In the end: $\psi = n-1$

We will show that in each round, potential ψ is multiplied by a factor of at most 1-1/(24 n). *This implies the claim.*

Sample Analysis (2)

This implies the claim?

Lemma 3.4. *Let Ξ be any positive potential function, where Ξ_0 is the initial potential value and Ξ_i is the potential after the i^{th} round of a given algorithm ALG. Assume that $\Xi_i \leq \Xi_{i-1} \cdot (1 - 1/f)$ and that ALG terminates if $\Xi_j \leq \Xi_{stop}$ for some $j \in \mathbb{N}$. Then, the runtime of ALG is at most $O(f \cdot \log(\Xi_0/\Xi_{stop}))$ rounds.*

Proof. From $\Xi_i \leq \Xi_{i-1} \cdot (1 - 1/f)$, it follows that $\Xi_j \leq \Xi_0 \cdot (1 - 1/f)^j$.
Now consider $j = f \cdot \ln \frac{\Xi_0}{\Xi_{stop}}$, which leads to (using $\ln(1 + x) \leq x$ for all $x > -1$)

$$\Xi_j \leq \Xi_0 \cdot (1 - 1/f)^{f \cdot \ln \frac{\Xi_0}{\Xi_{stop}}} = \Xi_0 e^{f \cdot \left(\ln \frac{\Xi_{stop}}{\Xi_0}\right) \cdot \ln(1-1/f)} \leq \Xi_0 e^{f \cdot \left(\ln \frac{\Xi_0}{\Xi_{stop}}\right) \cdot (-1/f)} = \Xi_0 e^{-\ln \frac{\Xi_0}{\Xi_{stop}}} = \Xi_{stop}.$$

□

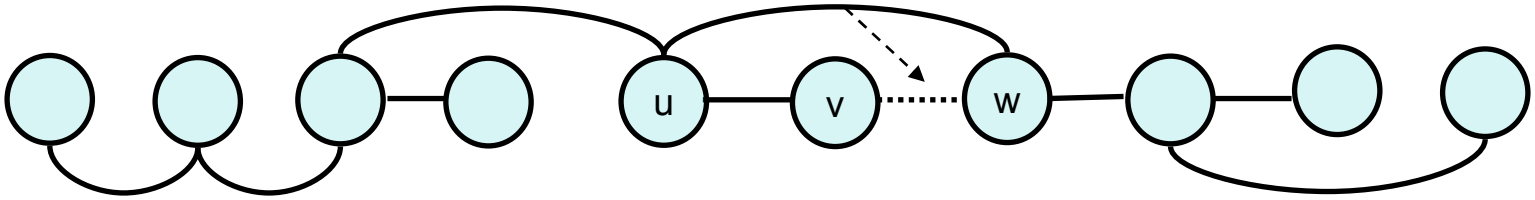
Sample Analysis (3)

Greedy scheduler: In each round, nodes are sorted w.r.t. remaining degree (remove fired triples with incident edges). Scheduler picks node *v* with largest degree, and schedules triple of *v* with most distant neighbors (to larger degree side).

Consider the potential function $\Psi = \sum_{e \in E} \text{len}(e)$

We will show that in each round, potential ψ is multiplied by a factor of at most 1-1/(24 n). This implies the claim.

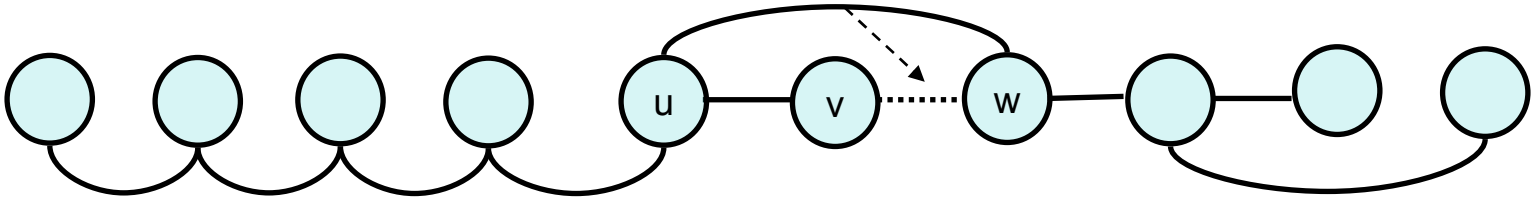
- Observe: firing a triple reduces potential ψ...
- ... but other nodes will be blocked in this round.



- Idea: we want to estimate the amount of blocked potential.

Sample Analysis (4)

- Consider the following right-linearization step



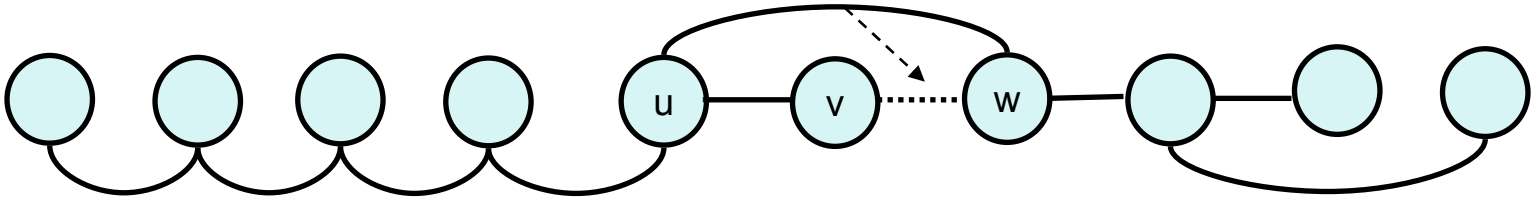
- Removing {u,w} and adding {v,w} reduces the potential by at least

$$dist(u,w)-dist(v,w) = dist(u,v)$$

- Since the greedy scheduler takes larger degree side:

$$dist(u,v) \geq deg(u)/2 - 1 \geq deg(u)/4$$

Sample Analysis (5)



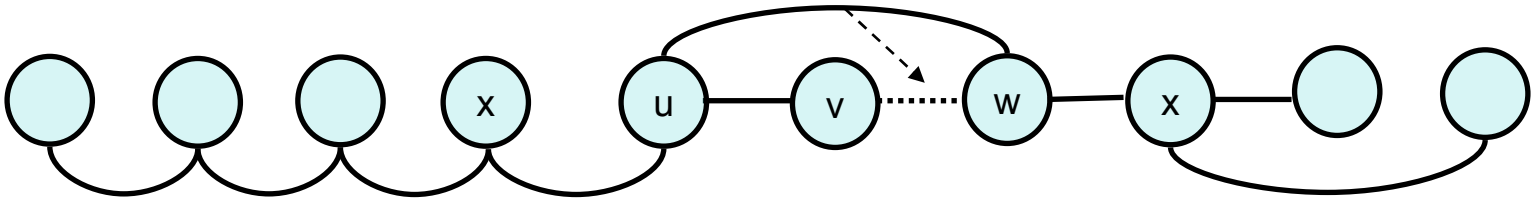
- Thus, potential reduced in one step by at least deg(u)/4

- How much potential is blocked?

- Consider remaining components after removing triple
- Consider neighbor *x* of *u, v* or *w*

  - if x is in ordered line component => blocked potential at most *n+n*
    (link length to x plus potential of ordered line)
  - if x is in different component => can still be linearized further (account for blocked component's potential later, only count link length potential: *n*)

Sample Analysis (6)



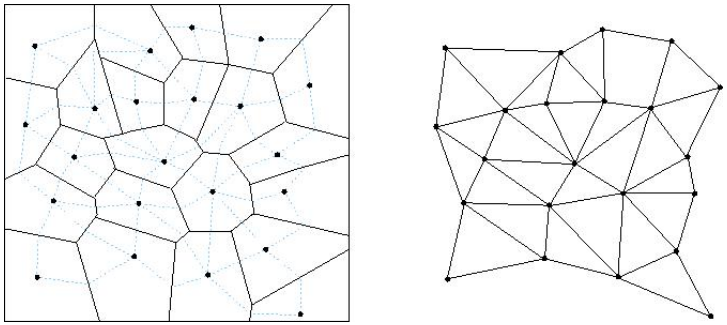
- The amount of blocked potential is at most $6 \cdot \deg(u) \cdot n$
  - since *u* has larger degree than *v* and *w*,
  - and since we have at most blocked potential $2 \cdot n$ per neighbor (*n* for component plus *n* for link to this neighbor)

- Thus, potential reduced by a factor at least $1-\Theta(1/n)$ per round.

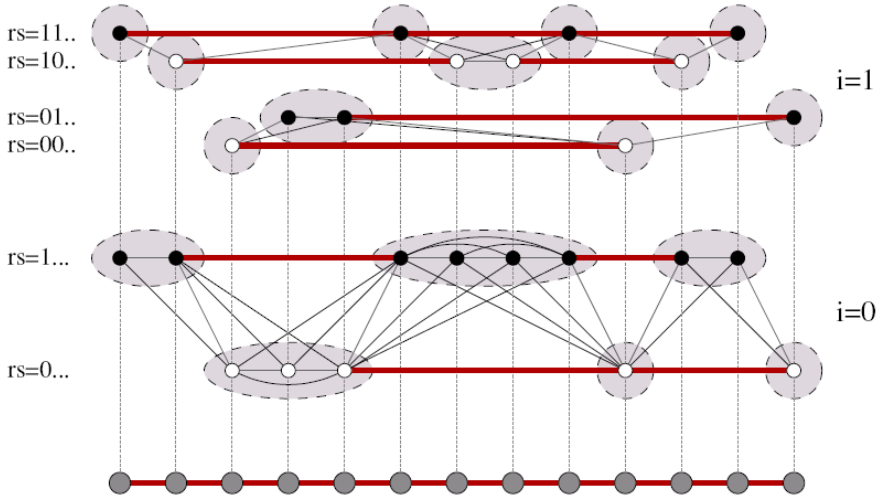
QED.

Other Algorithms for Self-Stabilizing Topologies

- „2-dimensional linearization“ („social network“): **Delaunay graphs**



- Hypercubic graphs („p2p network“): **Skip graphs** (see also PODC BA by Sriram et al.)



Thanks!

Thanks to my co-authors:

Matthias Baumgart (TUM)
Dominik Gall (TUM)
Riko Jacob (TUM)
Fabian Kuhn (USI)
Andrea Richa (ASU)
Christian Scheideler (UPB)
Hanjo Täubig (TUM)
Roger Wattenhofer (ETH)