Self-stabilizing and Self-optimizing
Distributed Data Structures

Stefan Schmid (TU Berlin & T-Labs)

Collaborators:
Riko Jacob, Andrea Richa, Christian Scheideler, Hanjo Täubig, …
Self-stabilizing and Self-optimizing Distributed Data Structures

Synchronous model alert!

Stefan Schmid (TU Berlin & T-Labs)

Collaborators:
Riko Jacob, Andrea Richa,
Christian Scheideler, Hanjo Täubig, …
Bremen Stadtmusikanten On A Windy Evening

© Shlomi’s book
Requirements:

- Play “happy birthday” again and again
- Wind changes pages without players knowing!
- When wind stops, harmonize eventually!
How to achieve?

Idea 1: If out of sync, just change to the page of a nearby player!

But what if the neighbor does the same? Do not know who was right! May never converge…

Idea 2: Go to start when asynchrony detected!

But players further away detect it later and restart later! May never converge…
What about synchronizing to the neighbor with the smallest page number?
What about…?
We need a “self-stabilizing algorithm”!

Idea 1: If out of sync, just change to the page of a nearby player!
But what if the neighbor does the same? Do not know who was right! May never converge…

Idea 2: Go to start when asynchrony detected!
But players further away detect it later and restart later! May never converge…
A Historical Note

Self-stabilizing algorithms pioneered by **Dijkstra** (1973): for example **self-stabilizing mutual exclusion**.

“I regard this as Dijkstra’s most brilliant work. Self-stabilization is a very important concept in fault tolerance.”

Leslie **Lamport** (PODC 1983)
This Talk: Topological Self-Stabilization

From chaos to order: self-stabilizing distributed datastructure (e.g., p2p)

failures, adversary, attack, ...

t_0

adversary stops

t_0 + D

desired structure
Example: Hypercube (log+log)

failures, adversary, attack, ...

weakly connected

stabilized hypercube

Stefan Schmid (T-Labs)
failures, adversary, attack, ...

How:
- Distributed: local algorithm
- Fast: minimize D, and stay there!
Model & Terminology

Configuration

- **Constants**: identifiers
- **Variables**: neighborhoods (set of identifiers)
- Union over all nodes

Execution

- **Scheduler**: execute enabled actions
- Gives next configuration
- In parallel, or “scalably”

Rules

- **Condition**: on local state
- **Action**: propose new link in neighborhood
- Careful: stay connected!

Self-Stabilization

- **Convergence**: eventually we end up in desired configuration
- **Closure**: once there, stays there
Performance Metrics

Parallel Time complexity
- Number of parallel rounds until stabilization in the worst case
- Depends on scheduler (scalable: only constant number of enabled actions per node)

Work
- Number of changed edges

Input-sensitive
- Local repairs and joins/leaves

weakly connected → stabilized

Number of parallel rounds until stabilization in the worst case
Depends on scheduler (scalable: only constant number of enabled actions per node)

Number of changed edges
Local repairs and joins/leaves
Local Algorithms (Local Model)

Send...

... receive...

... compute.
Talk Overview

- Primo Piatto: Linearization
- Main Dish: The Skip+ Graph
- Dessert: Delaunay Graphs & Co.
- Digestive: From Self-Stabilization to Self-Optimization
Linearization

Input: Weakly Connected Graph

Output: Sorted Network (wrt IDs!)

How?
- Local neighborhood changes only
- Preserve connectivity
- Once there, stay there
A First Insight: Local Checkability

Basic requirement:

At least one node must observe (and continue changing)!

Local checkability:

\[ F(\text{No!}) = \text{No!} \]
Most Simple *Undirected* Linearization

**Linearize left (x<y<z):**

```
 x -- y -- z
```

**Linearize right (x<y<z):**

```
 x -- y -- z
```

**Correctness:**
- **Connectivity** preserved: paths via missing edge still exist
- **Closure:** no changes in linearized setting
- **Convergence:**
  - Triple always exists if not linearized
  - Firing triple reduces potential: $\Phi = \sum \text{len}(e)$, edges get shorter

**Complexity?**
Types of Schedulers

The **FULL Scheduler**: full set of triples okay

Problem: many changes at single node (e.g., two new edges at node 2, but up to n-1)

**Only maximal independent set (MIS Scheduler):**

Each node involved in at most one triple!
Concrete MIS Schedulers (Hypothetical!)

Greedy MIS Scheduler

- E.g., select highest (remaining) degree node first (“least linearized guy”)
- And for this node, fire triple with most remote neighbors on side with higher degree (“most progress”)

Worst / Best Case MIS Scheduler

- Worst/best sets of MIS triples such that complexity max/minimized

Random MIS Scheduler

- Random MIS triples
The LIN-MAX Algorithm: each node proposes furthest triple on each side

Under a greedy MIS scheduler, LIN-MAX has a time complexity of $O(n \log n)$. 
Under a greedy MIS scheduler, LIN-MAX has a time complexity of $O(n \log n)$.

Proof

- Consider potential function $\Phi = \sum \text{len}(e)$.
- Clearly, initially $\Phi < O(n^3)$, each edge at most $n$ long, and when fully linearized, $\Phi = O(n)$.
- We show: in each round where triple still exists, potential is multiplied by factor $1 - \Omega(1/n)$.
- When triple right-linearized by $x$, $\Phi$ reduced by at least $\text{dist}(x,z) - \text{dist}(y,z) = \text{dist}(x,y)$.
- Due to greedy degree scheduling, and since LIN-MAX takes furthest neighbors: $\text{dist}(x,y) \geq \deg(x)/2 - 1 \geq \deg(x)/4$.
- Due to this triple, how many other triples cannot be fired in this round ("blocked potential"): overall potential at most $O(n) \cdot \deg(x)$. So we reduce a $1/n$ fraction of the total potential. QED.
“Due to the triple \((x,y,z)\), at most \(O(n) \times \text{deg}(x)\) remaining potential is blocked.”

- Look at remaining components and neighbors \(w\) of \(x, y,\) or \(z\)
  - Case A: if remaining component is line, cannot linearize further in this step, but line has blocked potential \(n\), plus potential for edge to \(w\) (at most \(n\) as well)
  - Case B: if remaining component still has triples that can fire in this round, account for them later. But lose edge to \(w\) (potential \(n\)).
  - Since \(\max(\text{deg}(y), \text{deg}(z)) \leq \text{deg}(x)\), max 6 \(\text{deg}(x)\) neighbors on both sides
  - So we block at most 6*\(\text{deg}(x)\) edges and components of potential 2n.
A Lower Bound

Even under an *optimal* MIS scheduler, LIN-MAX has a time complexity of $\Omega(n)$.

Proof

Length of edge $e$ is reduced by one only per round: no parallelism.

QED
Under worst scheduler, time $\Omega(n^2)$ for LIN-MAX algorithm.

Initially complete bipartite graph.

QED
Talk Overview

- Primo Piatto: Linearization
- Main Dish: The Skip+ Graph
- Desert: Delaunay Graphs & Co.
- Digestive: from Self-Stabilization to Self-Optimization
Skip Graphs

- Attractive distributed data structure: logarithmic height, logarithmic degree
- Distributed variant of skip list…: connect to nearest neighbors on log(n) many levels
- Nodes v have identifier v.id and random string v.rs
- Nodes sorted according to v.id (range search), and organized in layers according to v.rs
Skip+ Motivation: Local Checkability

- For fast self-stabilization, we use a different variant of the Skip graph
- Additional edges for (1) local checkability and (2) efficiency

Problem: The following graph looks “locally correct”!

Such a Skip graph does not work: node a only has two neighbors c (on level 0) and u (on level 1)
- But neither a, c, or u are aware of n: the graph looks correct locally: everyone has its nearest neighbors!
- Add additional edges to all nodes on this level, until nearest neighbor of the prefix.
- Node c can now realize that u is not a nearest neighbor of a, and tell it to a!
Definition of Skip+

Define predecessors and successors on each level

Define ranges in which nodes are interested on a level $i$: all up to end of opposite color

In words: a white node is interested in all nodes until first black node (inclusive); if white node does not have white neighbor yet on that side, it is interested in all black nodes until white again (exclusive).
Properties of Skip+

The diameter and degree of Skip+ is $O(\log n)$, w.h.p.

- The height and diameter is not larger than in the original Skip graph
- Interestingly, also the degree does not increase asymptotically
- Probability that there are $k$ neighbors on level $i$: $2^{-k}$
- Union bound over all possible distributions of degrees over levels:

\[
\Pr[X = d] \leq \sum_{k_0, \ldots, k_{H-1} \geq 0: \sum_{i=0}^{H-1} k_i = d} \prod_{i=0}^{H-1} \frac{1}{2^{k_i - 2}} \leq \left(\frac{d + H - 1}{H - 1}\right) \frac{1}{2^{d - 2H}}.
\]

If $d = c \cdot (H - 1)$, we get

\[
\left(\frac{d + H - 1}{H - 1}\right) \frac{1}{2^{d - 2H}} \leq \frac{[(c + 1)d]^{H-1}}{2^{c(H-1) - 2H}} \leq \frac{[(c + 1)d]^{H-1} \cdot 2^{4(H-1)}}{2^{c(H-1)}} \leq \frac{1}{n^{c'}}
\]
Distributed Self-Stabilization: Algorithm ALG+

Principles

- **Never delete** any edges! Only forward or merge with existing ones (preserves connectivity)
- **Four simple rules**: all executed at all times
- No phase changes (“first clique, then…”: not self-stabilizing)
- But **analysis in phases** okay!
- Preprocessing / transition step between rules: make things bi-directed, etc.
- Do not introduce unnecessary edges (**degree**!)

Stefan Schmid (T-Labs)
Pre-Processing / Transition

- Receive requests
- Make links bi-directed and send state / neighborhood
Rule 1: Range Reduction

- Distinguish: stable and temporary neighbor
- Stable = out-neighbor in-range on some level \( i \); temporary = not
- Note: in-range at level \( i \) implies in-range at level \( j > i \) (if prefix still fits: on higher levels less nodes as more prefix bits required)
- For every level \( i \), for any stable \( v \in N(u) \) and \( \text{pfx-}i(v) = \text{pfx-}i(w) \) and \( v \) interested in \( w \), \( u \) requests new stable \((v,w)\), plus if also stable: \((w,v)\)

Rule 1 ensures a fast “pointer doubling” until first interesting nodes are found! (Initially: unbounded ranges!)
Rule 2: Forward Edges

- Node u forwards non-interesting / temporary edge to (u,v) to the stable neighbor with the largest common prefix with v.
- W must exist, otherwise (u,v) would be stable.

Rule 2 used to quickly propagate edges to nodes where they are more useful (otherwise vanish / merge).
Rule 3: Local Closure («intro all»)

- Whenever the stable neighborhood of a node $u$ changes, e.g., even if only the level at which some edge is stable, $u$ introduces edges between all its neighbors.

Rule 3 quickly propagates new edges in neighborhood and ensures that already connected components stay connected in future.
Rule 4: Linearize

- Every node $u$ on every level $i$: for all stable neighbors, link neighbors according to order of identifier ID (not random string rs!)

Rule 4 sorts nodes according identifiers: gives desired search structure.
Proof Overview (1)

- Think in “phases”!
- **Bottom-up phase (time \( \log^2 n \))**
  - From layer 0 upward, \( G_\rho \) components arise:
    - Trivial for empty prefix (connected)
    - Induction: each node with \( \rho_0 \) finds **buddy**: node of opposite color \( \rho_1 \) on same level \( | \rho | \).
    - Given a buddy and connected \( V_\rho \), we quickly get connected graphs \( G_{\rho_0} \) and \( G_{\rho_1} \).

\[ G_\rho = (V_\rho, E_\rho) \text{ where } V_\rho \text{ is set of nodes with prefix } \rho \text{ and } E_\rho \text{ are edges between } V_\rho \text{ nodes.} \]
Proof Overview (1)

- Think in “phases”!
- **Top-down phase** (time $\log n$)
  - From level H downward
  - Level i contains all edges of $G_\rho$ (stable “little Skip graph”)
  - Level H trivial: single nodes
  - Then, by Rule 1, two i-finished components *zip together* to (i-1)-finished component

\[ G_\rho^0 \rightarrow G_\rho^1 \rightarrow G_\rho \]

$O(1)$

Stefan Schmid (T-Labs)
Lemma: If weakly connected at $t_0$, nodes will have buddy at $t_0+O(\log n)$ w.h.p.
- By Rule 1 (pointer doubling until in-range node!)
- Concept of pre-component / pre-connected:

\[(\sigma,k)\text{-pre-component}\]
Nodes $a$, $b$ with prefix $\sigma=\rho x$ but in different $G\sigma$ components are $(\sigma,k)$-pre-connected if (1) $G\rho$ is weakly connected, (2) every node in $G\rho_0$ and $G\rho_1$ has at least one neighbor in the opposite component, (3) $a$ and $b$ are either directly connected, $\sigma$-V-linked, or if there exists a stable $(\sigma,k')$-bridge with $k'\leq k$.

Shaded nodes are $(\rho_0,k)$-pre-component:
Bottom-Up Phase (2)

- **Lemma**: Once pre-connected, stays pre-connected.
- **Lemma**: If in $(\sigma,k)$-pre-component at $t$, $\sigma$-connected at $t+4$.
  - Mostly due to Rule 1 and 3
- **Lemma**: Evolution of bridges
  - The level of temporary bridge edge grows quickly: endpoints share larger prefix in each round (Forwarding Rule 2 plus existence of buddy)
  - Then, bridge edge stabilizes, and can serve for forwarding as well.
  - This yields new stable bridges at lower levels.
- **Lemma**: Once $G_{\rho}$ connected at time $t$, $G_{\rho 0}$ and $G_{\rho 1}$ also connected at time $t+O(\log n)$.
- So summing over all levels: $O(\log^2 n)$.

Stefan Schmid (T-Labs)
Other Features of Skip+

Individual joins/leaves can be handled locally, with polylog work.
Talk Overview

- Primo Piatto: Linearization
- Main Dish: The Skip+ Graph
- Desert: Delaunay Graphs & Co.
- Digestive: from Self-Stabilization to Self-Optimization
There exists a self-stabilizing algorithm for Delaunay graph with time complexity of $O(n^3)$.

Idea
- More geometric
- Always compute local Delaunay graph of (outgoing) neighbors plus “local hull”: stable edges
- Greedily route temporary edges towards node closest to edge destination (“distance compass routing”): maintain connectivity
Talk Overview

- Primo Piatto: Linearization
- Main Dish: The Skip+ Graph
- Desert: Delaunay Graphs & Co.
- Digestive: from Self-Stabilization to Self-Optimization
From “Optimal” Networks to Self-Adjusting Networks

- Networks become more and more dynamic (e.g., flexible SDN control)
- Vision: go beyond classic “optimal” static networks
- Example: Peer-to-peer

Chord, Pastry, SHELL
- Hypercubic
- Log diameter
- Log degree
- Log routing

Koorde, ...
- Constant degree
- Log routing

Pancake
- Log/loglog degree and log/loglog routing
From “Optimal” Networks to Self-Adjusting Networks

- Networks become more and more dynamic (e.g., flexible SDN control)
- Vision: go beyond classic “optimal” static networks
- Example: Peer-to-peer Chord, Pastry, SHELL, Koorde, Pancake
- Hypercubic
- Log diameter
- Log degree
- Log routing

What if networks could self-adjust depending on communication pattern?

Stefan Schmid (T-Labs)
An Old Concept: Move-to-front, Splay Trees, ...

- Classic data structures: lists, trees
- Linked list: move frequently accessed elements to front!
- Trees: move frequently accessed elements closer to root
An Old Concept: Move-to-front, Splay Trees, …

- Classic data structures: lists, trees
- Linked list: move frequently accessed elements to front!
- Trees: move frequently accessed elements closer to root

Stefan Schmid (T-Labs)
An Old Concept: Move-to-front, Splay Trees, …

- Classic data structures: lists, trees

- Linked list: move frequently accessed elements to front!

- Trees: move frequently accessed elements closer to root
The Vision: Splay Networks (“Distributed Splay Trees”)

- Most simple self-adjusting tree network: Binary Search Tree (BST)
Most simple self-adjusting tree network: Binary Search Tree (BST)
The Vision: Splay Networks ("Distributed Splay Trees")

- Most simple self-adjusting tree network: Binary Search Tree (BST)

Communication between peer pairs!
(Not only lookups from root…)

Stefan Schmid (T-Labs)
The Vision: Splay Networks ("Distributed Splay Trees")

- Most simple self-adjusting tree network: Binary Search Tree (BST)

Why BST?!
- Most simple generalization of classic data structure
- Allows for local routing!
- Allows for algebraic gossip
Model: Self-Adjusting SplayNets

Input:
- communication pattern: (static or dynamic) graph

Output:
- sequence of network adjustments

Cost metric:
- expected path length
- # (local) network updates
The Optimal Offline Solution

Dynamic program

- Binary search:
  decouple left from right!
- Polynomial time
  (unlike MLA!)
- So: solved M”BST”A

See also:

- Related problem of phylogenetic trees
The Online SplayNets Algorithm

From Splay tree to SplayNet:

Algorithm 1 Splay Tree Algorithm \( ST \)
1: (* upon lookup \((u, v)\) *)
2: splay \( u \) to root of \( T \)

Algorithm 2 Double Splay Algorithm \( DS \)
1: (* upon request \((u, v)\) in \( T \) *)
2: \( w := \alpha_T(u, v) \)
3: \( T' := \text{splay} \ u \) to root of \( T(w) \)
4: \( \text{splay} \ v \) to the child of \( T'(u) \)
The Online SplayNets Algorithm

From Splay tree to SplayNet:

**Algorithm 1** Splay Tree Algorithm

1: (* upon lookup (u) *)
2: **splay** u to root of T
The Online SplayNets Algorithm

From Splay tree to SplayNet:

Algorithm 2: Double Splay Algorithm

1: (*) upon request \((u, v)\) in \(T\) *
2: \(w := \alpha_T(u, v)\)
3: \(T' := \text{splay} u \text{ to root of } T(w)\)
4: \(\text{splay} v \text{ to the child of } T'(u)\)

Least Common Ancestor
Analysis: Basic Lower and Upper Bounds

**Upper Bound**

\[ \text{A-Cost} < H(X) + H(Y) \]

where \( H(X) \) and \( H(Y) \) are empirical entropies of sources resp. destinations

Adaption of Tarjan&Sleator

**Lower Bound**

\[ \text{A-Cost} > H(X | Y) + H(Y | X) \]

where \( H( | ) \) are conditional entropies.

Assuming that each node is the root for “its tree”

Therefore, our algorithm is optimal, e.g., if communication pattern describes a product distribution!
Properties: Convergence

Cluster scenario:

Nodes communicate within local clusters only!

Over time, nodes will form clusters in BST! No paths “outside”.

Stefan Schmid (T-Labs)
Properties: Optimal Solutions

Laminated scenario:

Will converge to optimum: Amortized costs 1.

Non-crossing matching (= “no polygamy”) scenario:

Will converge to optimum: Amortized costs 1.
Properties: Optimal Solutions

Multicast scenario (BST): Example

Invariant over “stable” subtrees (from right):
Improved Lower Bounds (and More Optimality)

Via interval cuts or conductance entropy:

Grid:

Cut of interval: entropy yields amortized costs!
Simulation Results

- Facebook component with 63k nodes and 800k edges
- SplayNet exploit random walk locality, to less extent also matching
Multiple BSTs: OBST

- **Static:**
  - Not much help for lookup model
  - Much help for routing model!

- **Dynamic:** yes 😊
Conclusion

- Topological self-stabilization
  - Linearization, Skip Graph, Delaunay
  - Take-home messages: local checkability, only one single phase possible, compute “local” version but ensure connectivity, …

- Self-optimization
  - Beating the lower bounds
  - First look into trees
  - Related to entropy


Thank you! Questions?