Network Algorithms

Self-Stabilization
Robust Algorithms: A Concert for the Mayor
A Concert for the Mayor

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”!

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…
Self-Stabilizing Algorithms

The Vision:
Self-Stabilizing Algorithms

The Vision:
Self-Stabilizing System

A distributed system is self-stabilizing if, starting from an arbitrary initial state, it is guaranteed to converge to a legitimate state. If the system is in a legitimate state, it is guaranteed to remain there, provided that no further faults happen. A state is legitimate if the state satisfies the specifications of the distributed system.

A self-stabilizing system does not have to be initialized: “automatically correct”
The Adversary Model

The adversary can:
- crash nodes
- make nodes behave Byzantine
- even corrupt the volatile memory of a node (without the node noticing)
- corrupt messages on the fly (without anybody noticing)

… but it cannot change the ROM (the algorithm/code)

All failures are transient, and eventually all nodes must work correctly again: crashed nodes get resurrected, Byzantine nodes stop being malicious, messages are being delivered reliably, and the memory of the nodes is secure.
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)
Example: Topological Self-Stabilization

From chaos to order: self-stabilizing distributed datastructure

failures, adversary, attack, ...

t₀

adversary stops

t₀+D

desired structure
Example: Topological Self-Stabilization

Example: Hypercube (log+log)

failures, adversary, attack, ...

weakly connected

stabilized hypercube
Example: Topological Self-Stabilization

Example: Hypercube (log+log)

failures, adversary, attack, ...

\( t_0 \)

weakly connected

\( t_0 + D \)

stabilized hypercube

How?
Example: Topological Self-Stabilization

Configuration

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

Execution

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

Self-Stabilization

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

The time complexity of a self-stabilizing system is the time that passed after the last (transient) failure until the system has converged to a legitimate state again, staying legitimate.
Protocol correct iff exactly one token / active node in ring and token cycles.

Assume:
- topology fixed
- n known
- a leader $v_0$

How to design self-stabilizing token ring?! Adversary may add and remove many tokens anytime initially!
Self-Stabilizing Token Ring

Each node is in a state $S=\{1, \ldots, n\}$

- Each node informs its child continuously about its state

If $v = v_0$ then

- If $S(v) = S(c)$ then
  - $S(v) := S(v) + 1 \pmod{n}$
- Else $S(v) := S(c)$

Token Ring
Self-Stabilizing Token Ring

Each node is in a state $S = \{1, \ldots, n\}$

Each node informs its child continuously about its state.

Token Ring

If $v = v_0$ then
  If $S(v) = S(c)$ then
    $S(v) := S(v) + 1 \pmod n$
  End If
Else $S(v) := S(c)$

Leader chooses next ID, all other nodes copy from parent!
Self-Stabilizing Token Ring

Token Ring

The algorithm stabilizes correctly.

Proof:
- Each node except for leader $v_0$ will attain and forward the state of its parent.
- The leader can only get the next larger value when its parent has the leader value too.
- At some point the leader will reach a state $s$ that no other node had at time $t_0$. (There are $n$ nodes and $n$ states.)
- Then one node after the other will learn the current state of the leader.
- The leader itself does not push the next value until the previous value travelled the entire ring!
- So the system stabilizes after at most $n$ time units after the leader increased the value.
- At most one node active at any time: Token passed implicitly with the switching state.

QED
Self-Stabilizing Independent Sets

How to design self-stabilizing Maximal Independent Sets?
Self-Stabilizing Independent Sets

Assume: node have unique IDs

Independent Sets

Every node $v$ executes the following code:

1: do atomically (forever)
2: Leave MIS if a neighbor with a larger ID is in the MIS
3: Join MIS if no neighbor with larger ID joins MIS
4: Send (node ID, MIS or not MIS) to all neighbors
5: end do

Why does it work?
Self-Stabilizing Independent Sets

Assume: node have unique IDs

**Independent Sets**

Every node \( v \) executes the following code:

1: do atomically (forever)
2: Leave MIS if a neighbor with a larger ID is in the MIS
3: Join MIS if no neighbor with larger ID joins MIS
4: Send (node ID, MIS or not MIS) to all neighbors
5: end do

**Why does it work?**

Same argument as for our Slow-MIS algorithm. Only difference: need to run it forever now…

Can I make any local algorithm self-stabilizing?! Desired criteria?
Transformation

Given:
Any deterministic local algorithm $A$ that computes a solution of a given problem in $k$ synchronous communication rounds.

Output:
A self-stabilizing system with time complexity $k$, i.e.:
- if the adversary does not corrupt the system for $k$ time units, the solution is stable
- if the adversary does not corrupt any node or message closer than distance $k$ from a node $u$, node $u$ will be stable (locality)
Transformation

Given:
Any deterministic local algorithm $A$ that computes a solution of a given problem in $k$ synchronous communication rounds.

Output:
A self-stabilizing system with time complexity $k$, i.e.:
- if the adversary does not corrupt the system for $k$ time units, the solution is stable
- if the adversary does not corrupt any node or message closer than distance $k$ from a node $u$, node $u$ will be stable (locality)
Transformation: Local Checking

- Each node maintains tables for the k different rounds of the execution of A:
  - Store all local variables of the last k rounds (for local emulation!)
  - Store all messages received for the last k rounds
  - All info that is not known is simply “?”

- Simulate all the k phases of the local algorithm A in parallel
  - As I do not even know in which round I and others are…
  - Exchange the above tables forever

- Each node checks whether the received tables from the neighbors as well as the messages are consistent

- Also known as Local Checking
Transformation: Local Checking

- Let $L^i_u$ be the state of the local variables of node $u$ after round $i$, and let $m^i_{uv}$ be the message sent from $u$ to $v$ in round $i$.

- Each node continuously sends its $L^i_u$ tables to the neighbors, together with the messages $m^i_{uv}$.

- Neighbor updates its message table accordingly and sends corresponding messages.

- Proof by induction:
  - At time $t_0$, nodes correctly initialize their local variables
  - At time $t_0+1$, they send the correct messages to neighbors
  - At time $t_0+2$, ...

With this transformation:

design turned from Art to Craft!
Transformation: Local Checking

- Let $L^i_u$ be the state of the local variables of node $u$ after round $i$, and let $m^i_{uv}$ be the message sent from $u$ to $v$ in round $i$.

- Each node continuously sends its $L^i_u$ tables to the neighbors, together with the messages $m^i_{uv}$.

- Neighbor updates its message table accordingly and sends corresponding messages.

- Proof by induction:
  - At time $t_0$, nodes correctly initialize their local variables
  - At time $t_0+1$, they send the correct messages to neighbors
  - At time $t_0+2$, ...

With this transformation: design turned from Art to Craft!
What is the overhead?
What about randomized algorithms?
Discussion

- How to do it for randomized algorithms?
  - Do not know k, the number of rounds!
  - But can just simulate more rounds, no problem.
  - Careful about adversary: should not compromise randomness of choices (e.g., have nodes produce random bits until it’s what he wanted)
  - Problem: can also not just stick to given random choices once and forever! Adversary may have corrupted the variables before.

- Some additional memory overhead, but usually bearable.
  - Memory overhead depends on k, the number of rounds, which is low.

- Good for mobile environments: if k-neighborhood does not change, nothing changes
In a little town, each evening citizens call their friends to ask whether they vote for Democrats or Republicans. Then they decide themselves for majority (assume odd number of friends). Does this system «converge» or «stabilize»?
In a little town, each evening citizens call their friends to ask whether they vote for Democrats or Republicans. Then they decide themselves for the majority (assume odd number). Does this system «converge» or «stabilize»?
Example

t: 

\[ \begin{array}{c}
\text{(Red nodes)} \\
\text{(Blue nodes)}
\end{array} \]

\[ \begin{array}{c}
t+1:
\end{array} \]
Example
What do you think?

- Is eventually everybody voting for the same party?

- Will each citizen eventually stay with the same party?

- Will citizens that stayed with the same party for some time, stay with that party forever?

- And if their friends also constantly root for the same party?

- Will this beast stabilize at all? 😊
Democrats / Republicans

Eventually each citizen will vote for the same party every other day.

Why?
Democrats / Republicans

Eventually each citizen will vote for the same party every other day.

Why?

- Friendship = directed edges
- Bad edge: to node which did not follow the advisor’s opinion: Example?
Democrats / Republicans

Eventually each citizen will vote for the same party every other day.

Why?

- Friendship = directed edges
- **Bad edge**: to node which does not follow the advisor’s opinion on next day!
- Example?
Democrats / Republicans

Eventually each citizen will vote for the same party every other day.

Continued (1)

- Consider a citizen c (Democrat) with g good and b bad out-edges on a day t.
- Degree of citizen c is hence g + b.
- So g friends of c root for the Democrats on day t + 1, and b friends root for the Republicans.
- So in evening of t + 1, c will receive g recommendations for Democrats, and b for Republicans.
- So what will citizen vote at day t + 2?
- If g > b at t, then same again, otherwise opposite.
Democrats / Republicans

Eventually each citizen will vote for the same party every other day.

Continued (2)

$g > b$: same party

$g < b$: opposite party
Democrats / Republicans

Eventually each citizen will vote for the same party every other day.

Continued (3): if \( g > b \)

- At day \( t+1 \), (blue) citizen \( c \):
  - \( g > b \) neighbors blue, b red

- So citizen \( c \) will be blue (still/again) at \( t+2 \)

- So \( b \) (red) neighbors pointing to \( c \) are bad at \( t+1 \) (from neighbor’s perspective), since \( c \) will be blue at \( t+2 \).

Bad out-edges of \( c \) at time \( t \) will be bad edges to \( c \) at time \( t+1 \)! Total number of bad edges remains the same. (No matter what color of \( c \) is at time \( t+1 \).)
Democrats / Republicans

Eventually each citizen will vote for the same party every other day.

Continued (4): if \( b > g \)

- At day \( t+1 \), (blue) citizen \( c \):
  - \( b > g \) neighbors blue, \( g \) red

- So citizen \( c \) will be red at \( t+2 \)

- So \( g < b \) (blue) neighbors pointing to \( c \) are bad at \( t+1 \) (from neighbor’s perspective), since \( c \) will be red at \( t+2 \).

Bad out-edges of \( c \) at time \( t \) will be good edges to \( c \) at time \( t+1 \)! Total number of bad edges decreases. (No matter what color of \( c \) is at time \( t+1 \).)
Democrats / Republicans

Eventually each citizen will vote for the same party every other day.

Continued (5)

- In both cases, the number of bad edges does not increase.

- In fact, it decreases if any node switches the party.

- Since the number of bad edges cannot be negative, the system will stabilize for a certain number of bad edges.

- Once number of bad edges stabilized, each node either stabilizes to a party or switches back and forth between times t and t+2.

QED
What kind of equilibrium is this?

- Is eventually everybody voting for the same party? No.
- Will each citizen eventually stay with the same party? No.
- Will citizens that stayed with the same party for some time, stay with that party forever? No.
- And if their friends also constantly root for the same party? No.
Related to Conway’s Game of Life

- Turing-complete game: **LIFE**
- 2d cell grid, each cell *dead* or *alive*
- Every cell interacts with its eight neighbors:
  - Any live cell with fewer than two live neighbors dies (loneliness).
  - Any live cell with more than three live neighbors dies, as if by overcrowding.
  - Any live cell with >2 live neighbors lives on to the next generation.

Can model complex things: gun + glider: