Modern Distributed Computing

Theory and Applications

Ioannis Chatzigiannakis

Sapienza University of Rome

Lecture 11
Tuesday, May 21, 2013

Part 5: Stabilization

1. Self-stabilization, definitions.
2. Mutual Exclusion
3. Breadth First Search
4. Power Supply Technique

Robust Algorithms

- We have studied the correctness of algorithms when communication channels and/or processes are reliable.
- We have also studied the correctness of the algorithms
  - When process fail,
  - Communication channels are faulty.
- We have also studied fully dynamic networks.
- The algorithms achieve robustness
  - Trying to maintain a “stable” network state.
  - They achieve this by making certain assumptions (Consensus, number of failures, violation of properties, rate of changes).
  - End up being too complex (Two Phase and Three Phase Commit)

Self-Stabilizing Algorithms

- Self-stabilizing algorithms achieve robustness via a fundamentally different approach.
- Robust algorithms tend to be pessimistic
  - Assume that all kinds of failures that may occur, will eventually occur.
  - Every round they check certain properties in order to guarantee correctness.
  - For each failure they follow a specific, specialized rule to recover.
  - They try to keep the system under a “correct” operating condition.
- Stabilizing algorithms are by nature more optimistic
  - Failures are transient.
  - Processes may fail or act abnormally from time to time.
  - Correct processes may at some point behave inconsistently.
  - Yet, at some point, they will recover.

Self-Stabilizing Algorithms

- Main idea
  - The system is designed to converge within finite number of steps from any (unstable) state to a desired (stable) state.
  - ... the system will eventually self-stabilize.
- We accept that a correct state is eventually reached.
  - We abandon failure models and bounds on failure rates.
  - The combination and type of faults cannot be totally anticipated in on-going systems.
- We assume that all processes operate properly, but the execution may fail arbitrarily during a transient failure.
  - We do not monitor failed processes.
  - We assume that no further failures occur.
  - We let the processes manage themselves locally by following simple rules.

Self-Stabilizing Algorithms

- We do not need to examine faulty processes and the history of the system.
- We assume that the initial state of the algorithm is one where a failure has occurred.
- Then the algorithm is self-stabilizing (or stabilizing) if eventually it behaves correctly.
  - That is, eventually it adheres to the specifications, independently of its initial state.
- The concept of stabilization was introduced by Dijkstra
  - Limited progress until the end of the 80s.
  - Most significant findings during the 90s when the approach became widely known.
  - Recently, attracted even more interest.

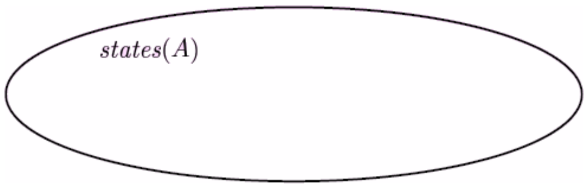
Definition

- Stabilizing algorithms are models as state-transition systems without initial state.
- For each pair of states κ, κ', $\kappa \rightsquigarrow \kappa'$ an action ϵ exists if $(\kappa, \epsilon, \kappa') \in \mathit{trans}(\mathcal{A})$
- An algorithm $\mathcal{A}$ stabilizes to specification Π if there is a subset of states $\mathcal{L} \subseteq \mathit{states}(\mathcal{A})$ such that
  - For every execution that starts in $\mathcal{L}$ it complies with Π (correctness)
  - Every possible execution includes a state in $\mathcal{L}$ (convergence)

Proving Stabilization

- In order to prove that an algorithm is a stabilizing algorithm we use the notion of "legal" or stable execution.
- Initially we assume that the algorithm starts from a stated in $\mathcal{L}$
- Then we identify a potential function (convergence function).

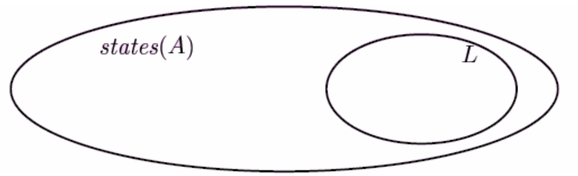
Execution Example



Proving Stabilization

- In order to prove that an algorithm is a stabilizing algorithm we use the notion of "legal" or stable execution.
- Initially we assume that the algorithm starts from a stated in $\mathcal{L}$
- Then we identify a potential function (convergence function).

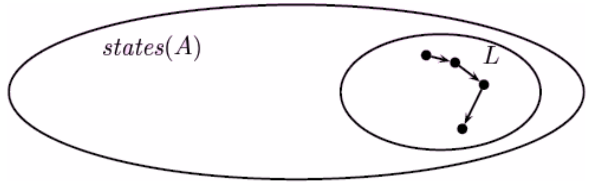
Execution Example



Proving Stabilization

- In order to prove that an algorithm is a stabilizing algorithm we use the notion of "legal" or stable execution.
- Initially we assume that the algorithm starts from a stated in $\mathcal{L}$
- Then we identify a potential function (convergence function).

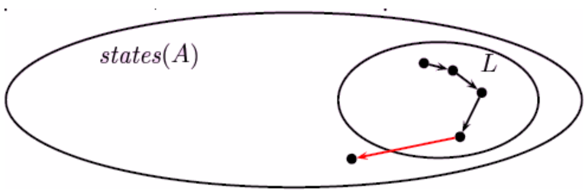
Execution Example



Proving Stabilization

- In order to prove that an algorithm is a stabilizing algorithm we use the notion of "legal" or stable execution.
- Initially we assume that the algorithm starts from a stated in $\mathcal{L}$
- Then we identify a potential function (convergence function).

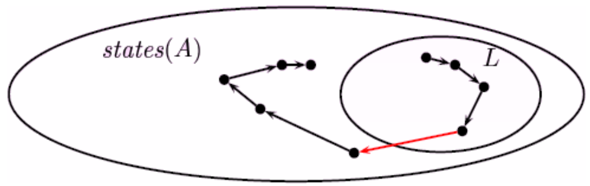
Execution Example



Proving Stabilization

- In order to prove that an algorithm is a stabilizing algorithm we use the notion of "legal" or stable execution.
- Initially we assume that the algorithm starts from a stated in $\mathcal{L}$
- Then we identify a potential function (convergence function).

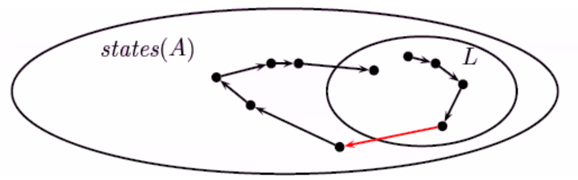
Execution Example



Proving Stabilization

- In order to prove that an algorithm is a stabilizing algorithm we use the notion of "legal" or stable execution.
- Initially we assume that the algorithm starts from a stated in $\mathcal{L}$
- Then we identify a potential function (convergence function).

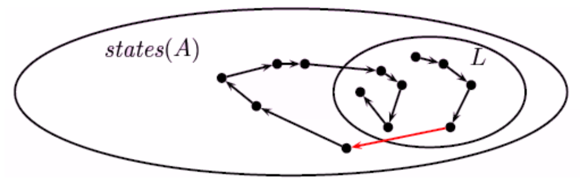
Execution Example



Proving Stabilization

- In order to prove that an algorithm is a stabilizing algorithm we use the notion of "legal" or stable execution.
- Initially we assume that the algorithm starts from a stated in $\mathcal{L}$
- Then we identify a potential function (convergence function).

Execution Example



Proving Stabilization

Our proofs examine executions that start from states in $\mathcal{L}$

Lemma

Let

- *All halting states be in $\mathcal{L}$, (i.e., $halt(\mathcal{A}) \subseteq \mathcal{L}$)*
- *There exists a function $f : states(\mathcal{A}) \to \mathcal{W}$ (where $\mathcal{W}$ well define set) such that if $\kappa \rightsquigarrow \kappa'$ then, either $f(\kappa) > f(\kappa')$ or $\kappa' \in \mathcal{L}$*

Then $\mathcal{A}$ guarantees convergence.

Properties of Stabilizing Algorithms

The benefits of stabilizing algorithms in contrast to robust algorithms

1. Fault Tolerance – they provide a complete and automatic tolerance to all kinds of transient failures since they eventually converge to a steady state.
2. Lack of Initialization – there is no need to initialize the algorithm at a predefined stated, the eventual behavior of the system is guaranteed.
3. Dynamic Topology – If a change occurs, the algorithm will eventually converge to a new working state.

Properties of Stabilizing Algorithms

The drawbacks of stabilizing algorithms in contrast to robust algorithms

1. Inconsistent State – until convergence is achieved, the algorithm may produce inconsistent output.
2. Increased Message Complexity – due to the continuous exchange of messages, stabilizing algorithms tend to be less efficient.
3. Termination Condition – it is impossible to identify if the algorithm has reached a final stated, thus the processes are usually unaware if the correct output has been produced.

Mutual Exclusion

- Processes share a common (critical) resource.
- Access to this resource requires exclusive access from only one process.
- The part of the process that handles the resource exclusively is called the "critical section" (CS).
- We need to coordinate the actions of the processes.
- In centralized systems, various primitives are available such as
    - semaphores, locks, monitors ...
- The problem of mutual exclusion was introduced by Edsger Dijkstra in 1965.

Minimum Requirements

- Safety – only and only one process may access the critical resource at any given time instance.
- Liveness –
    - If a process wishes to enter the critical section then it will eventually succeed.
    - If the common resource is not used, then any process requesting access will be granted access within a finite period of time.

Assumptions

1. Processes are assigned unique identifiers.
2. Each process a critical section.
3. Processes compete for 1 critical resource.
4. No global clock is available.
5. Processes communicate using messages.
6. Communication channels are reliable, FIFO.
7. The network is fully connected.

Performance Measures

1. Correctness – the conditions of safety, liveness, ordering are preserved.
2. Communication Complexity – processing of requests to enter critical section minimize total number of message exchanges.
3. Latency – time elapsed between the issue of a request and the access of the resource is minimized.

Stabilizing Mutual Exclusion Algorithm – Dijkstra, 1974

- Each process i maintains a counter x_i.
- Processes are positioned in a "virtual" ring, e.g., sorted by ID.
  - Let x_1 the counter of the process with the smaller ID.
  - Let x_n the counter of the process with the highest ID.
- Periodically, they transmit their counter.
- Process 1 can use the common resource when $x_1 = x_n$.
  - When it completes it sets $x_1 = (x_1 + 1) \mod (n + 1)$.
- Any other process i can use the common resource when $x_i \neq x_{i-1}$.
  - When it completes it sets $x_i = x_{i-1}$.

Stabilizing Mutual Exclusion Algorithm – Dijkstra, 1974

1. The process that has access to the common resource may change its state
   - after completing the execution of the critical section.
2. Changing the state of a process always results in losing access to the common resource.
3. Process $u \neq 1$ may set $x_u = x_{u-1}$
   - since it is the active process, it holds that $x_u \neq x_{u-1}$
4. Process u_1 may set x_0 to take a different value from x_{n-1} by setting $x_1 = (x_n + 1) \mod K$
   - since they equality initially holds.

Self-Stabilizing Mutual Exclusion Algorithm

Each process u holds a variable $x_u \in \{0, 1, \ldots, K-1\}$. Process u_1 gains access to execute its CS if $x_1 = x_n$. Each other process u gains access to execute its CS if $x_u \neq x_{u-1}$. The process that has access to the common resource may changes its state and release the resource by setting:

$$x_u = \begin{cases} x_{u-1} & \text{if } u \neq 1 \\ (x_n + 1) \mod K & \text{if } u = 1 \end{cases}$$

Example of Execution – Initial State

u	1	2	3	4	$\cdots$	$n-1$	n
x_u	0	0	0	0	$\cdots$	0	0

Self-Stabilizing Mutual Exclusion Algorithm

Each process u holds a variable $x_u \in \{0, 1, \ldots, K-1\}$. Process u_1 gains access to execute its CS if $x_1 = x_n$. Each other process u gains access to execute its CS if $x_u \neq x_{u-1}$. The process that has access to the common resource may changes its state and release the resource by setting:

$$x_u = \begin{cases} x_{u-1} & \text{if } u \neq 1 \\ (x_n + 1) \mod K & \text{if } u = 1 \end{cases}$$

Example of Execution – Intermediate States

u	1	2	3	4	$\cdots$	$n-1$	n
x_u	(0)	0	0	0	$\cdots$	0	0

Self-Stabilizing Mutual Exclusion Algorithm

Each process u holds a variable $x_u \in \{0, 1, \ldots, K-1\}$. Process u_1 gains access to execute its CS if $x_1 = x_n$. Each other process u gains access to execute its CS if $x_u \neq x_{u-1}$. The process that has access to the common resource may changes its state and release the resource by setting:

$$x_u = \begin{cases} x_{u-1} & \text{if } u \neq 1 \\ (x_n + 1) \mod K & \text{if } u = 1 \end{cases}$$

Example of Execution – Intermediate States

u	1	2	3	4	$\cdots$	$n-1$	n
x_u	1	(0)	0	0	$\cdots$	0	0

Self-Stabilizing Mutual Exclusion Algorithm

Each process u holds a variable $x_u \in \{0, 1, \ldots, K-1\}$. Process u_1 gains access to execute its CS if $x_1 = x_n$. Each other process u gains access to execute its CS if $x_u \neq x_{u-1}$. The process that has access to the common resource may changes its state and release the resource by setting:

$$x_u = \begin{cases} x_{u-1} & \text{if } u \neq 1 \\ (x_n + 1) \mod K & \text{if } u = 1 \end{cases}$$

Example of Execution – Intermediate States

u	1	2	3	4	$\cdots$	$n-1$	n
x_u	1	1	(0)	0	$\cdots$	0	0

Self-Stabilizing Mutual Exclusion Algorithm

Each process u holds a variable $x_u \in \{0, 1, \ldots, K-1\}$. Process u_1 gains access to execute its CS if $x_1 = x_n$. Each other process u gains access to execute its CS if $x_u \neq x_{u-1}$. The process that has access to the common resource may changes its state and release the resource by setting:

$$x_u = \begin{cases} x_{u-1} & \text{if } u \neq 1 \\ (x_n + 1) \mod K & \text{if } u = 1 \end{cases}$$

Example of Execution – Intermediate States

u	1	2	3	4	$\cdots$	$n-1$	n
x_u	1	1	1	(0)	$\cdots$	0	0

Self-Stabilizing Mutual Exclusion Algorithm

Each process u holds a variable $x_u \in \{0, 1, \ldots, K-1\}$. Process u_1 gains access to execute its CS if $x_1 = x_n$. Each other process u gains access to execute its CS if $x_u \neq x_{u-1}$. The process that has access to the common resource may changes its state and release the resource by setting:

$$x_u = \begin{cases} x_{u-1} & \text{if } u \neq 1 \\ (x_n + 1) \mod K & \text{if } u = 1 \end{cases}$$

Example of Execution – Intermediate States

u	1	2	3	4	$\cdots$	$n-1$	n
x_u	1	1	1	1	$\cdots$	0	0

Self-Stabilizing Mutual Exclusion Algorithm

Each process u holds a variable $x_u \in \{0, 1, \ldots, K-1\}$. Process u_1 gains access to execute its CS if $x_1 = x_n$. Each other process u gains access to execute its CS if $x_u \neq x_{u-1}$. The process that has access to the common resource may changes its state and release the resource by setting:

$$x_u = \begin{cases} x_{u-1} & \text{if } u \neq 1 \\ (x_n + 1) \mod K & \text{if } u = 1 \end{cases}$$

Example of Execution – Intermediate States

u	1	2	3	4	$\cdots$	$n-1$	n
x_u	1	1	1	1	$\cdots$	1	0

Self-Stabilizing Mutual Exclusion Algorithm

Each process u holds a variable $x_u \in \{0, 1, \ldots, K-1\}$. Process u_1 gains access to execute its CS if $x_1 = x_n$. Each other process u gains access to execute its CS if $x_u \neq x_{u-1}$. The process that has access to the common resource may changes its state and release the resource by setting:

$$x_u = \begin{cases} x_{u-1} & \text{if } u \neq 1 \\ (x_n + 1) \mod K & \text{if } u = 1 \end{cases}$$

Example of Execution – Intermediate States

u	1	2	3	4	$\cdots$	$n-1$	n
x_u	1	1	1	1	$\cdots$	1	1

Self-Stabilizing Mutual Exclusion Algorithm

Each process u holds a variable $x_u \in \{0, 1, \ldots, K-1\}$. Process u_1 gains access to execute its CS if $x_1 = x_n$. Each other process u gains access to execute its CS if $x_u \neq x_{u-1}$. The process that has access to the common resource may changes its state and release the resource by setting:

$$x_u = \begin{cases} x_{u-1} & \text{if } u \neq 1 \\ (x_n + 1) \mod K & \text{if } u = 1 \end{cases}$$

Example of Execution – Intermediate States

u	1	2	3	4	$\cdots$	$n-1$	n
x_u	2	1	1	1	$\cdots$	1	1

Self-Stabilizing Mutual Exclusion Algorithm

Each process u holds a variable $x_u \in \{0, 1, \ldots, K-1\}$. Process u_1 gains access to execute its CS if $x_1 = x_n$. Each other process u gains access to execute its CS if $x_u \neq x_{u-1}$. The process that has access to the common resource may changes its state and release the resource by setting:

$$x_u = \begin{cases} x_{u-1} & \text{if } u \neq 1 \\ (x_n + 1) \mod K & \text{if } u = 1 \end{cases}$$

Example of Execution – Intermediate States

u	1	2	3	4	$\cdots$	$n-1$	n
x_u	2	2	(1)	1	$\cdots$	1	1

Process 1

```
while (true) {
  if (myX == prevX) {
    execCS(); // execute Critical Section
    myX = (myX+1) % (n+1);
  }
  sendReceive(myX, prevX);
}
```

Process u ($u \neq 1$)

```
while (true) {
  if (myX != prevX) {
    execCS(); // execute Critical Section
    myX = prevX;
  }
  sendReceive(myX, prevX);
}
```

- It works when we start it with

$$x_1 = x_2 = x_3 = \ldots = x_n = 0$$

- One processor may change state at a time.
- What if errors occur?
- Assigns each processor with an arbitrary state (in the range of its state space) and then assume that no further errors occur.
- For example $\{3, 4, 4, 1, 0\}$.
- Processors 2, 4 and 5 have the privilege !
- Will the system ever recover ?

$\{0,0,0,0,0\}$
$\{1,0,0,0,0\}$
$\{1,1,0,0,0\}$
$\{1,1,1,0,0\}$
$\{1,1,1,1,0\}$
$\{1,1,1,1,1\}$
$\{2,1,1,1,1\}$
$\{2,2,1,1,1\}$
$\{2,2,2,1,1\}$
$\{2,2,2,2,1\}$
$\{2,2,2,2,2\}$
$\ldots$

Process 1 changes state infinitely often.

- Assume not – i.e., let s be the fixed state of process 1.
- Then process 2 eventually copies s from process 1.
- Then process 3 eventually copies s from process 2.
- $\ldots$
- Then process n eventually copies s from process $n-1$.
- Then process 1 changes state. !

Process 1 changes state in the order $4, 5, 0, 1, 2, 3, 4, 5, 0, \ldots$

- Process 1 after at most n steps will be the only process with $x_1 = 0$. Then x_1 will traverse the network assuring that only 1 process has the privilege.

Algorithm's Properties

- At least one process has the privilege.
  - For sure 1 if no other one has the privilege.
- In each step, the number of processes with the privilege to use the resource does not increase.
  - The process that has the privilege will lose it at the end of the round.
  - Only the next process will benefit from such a round.
- $\mathcal{L} = \{\kappa : \text{ only one process has the privilege}\}$
- If the execution is at a state within $\mathcal{L}$, then we have a correct execution and the privilege is cycling the network (correctness)
- $f = \sum_{x \in V} (n - x)$
  Where $V = \{x : x \geq 1 \text{ and has the privilege}\}$
- f is reducing at every step of u if $u \neq 1$.

Algorithm's Properties

- At most $\frac{n(n-1)}{2}$ steps occur before process 1 gets the privilege.
- The initial state (i.e., immediately after faults stop) may have at most n distinct states.
- In any initial state at least one state is missing:
  In $\{4, 4, 1, 0, 2\}$, state 3 and 5 are missing.
- Once process 1 reaches the missing state, e.g., 5, all the processors must copy 5, before process 1 reads 5 from process n and changes state to 0.
- The value will traverse the ring, and before the next step of 1 at most one process will have access
  - $x_2 = x_1 = \ldots = x_n = x_1 = K$
- The system always recovers.
- The number of steps required to converge is $O\left(n^2\right)$.

Breadth-First directed spanning tree

A directed spanning tree of G with root i is breadth-first provided that each node at distance d from i in G appears at depth d in the tree.

- A self-stabilizing algorithm must guarantee
  - In each unstable state, at least one process is active.
  - In each stable state, no process is active, i.e., the system has reached a deadlock.
  - For all initial states and all possible executions, the system guarantees convergence to a stable state in finite number of steps.

StabBFS Algorithm

Each process u maintains a variable p_u for storing its parent in the tree and variable d_u for its height from u_0 (based on the current state), initially if $u \neq u_0 : p_u = \infty, d_u = \infty$ otherwise $u = u_0 : p_u = u_0, d_u = 0$. In each round, u transmits d_u to its neighbors. Checks values received and if it listens a message from v where $d_v < d_u$, it sets $d_u = d_v + 1$ and $p_u = v$.

- Process u_0 is the root of the tree – this is known to the processes.
- Let n the size of the network.
- Let $d(u)$ the distance of u_0 from u in G.

Definitions

- For height of u it holds that $0 \leq d(u) \leq n-1$.
- In an unstable state, each process apart from u_0 may have any height $0 \ldots n-1$.
- In an unstable state, each process apart from u_0 may assume any other process as its parent in the tree (except from u_0).
- For each process we set the state S_u as follows

$$S_u = \{v : v = nbrs_u \wedge d_u = min_{i \in nbrs_u}\{d_i\}\}$$

- S_u includes all the neighbors of u with minimum height – it may include more than one process but it cannot be empty.
- All processes in S_u have the same height, $d(S_u)$.

Stable State

- We define as stable state each state where the following global predicate is true

$$\forall u \neq u_0 : d_u = d(S_u) + 1 \wedge p_u \in S_u$$

- The term $p_u \in S_u$ denotes that the parent variable of each process u points to a neighboring node of u.

Lemma

For each connected symmetric graph, the above stable state defines a Breadth-First directed spanning tree rooted at u_0.

Stable State

- The root of the tree u_0 has fixed height 0.
- Thus, in a stable state, all neighboring nodes of u_0 must have height 1.
- Therefore, all neighboring nodes of these nodes must have height 2 ...
  - and their parent variable points to one of the nodes with height 1.
- Following this argument for all the nodes of the network, it is clear that the parent and height variables will consisute a directed spanning tree rooted at u_0.
- The goal of the algorithm is to converge to such a stable state.

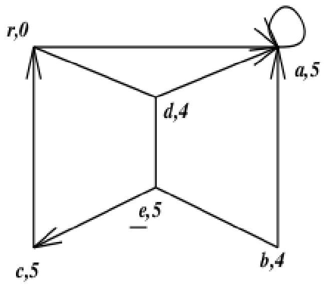
Main Idea

- When the system reaches an unstable state, at least one node will identify this and become active in order to start taking corrective actions.
- The algorithm enforces a uniform rule for all processes apart from the root.
- The rule involves two parts:
  1. Evaluate a local predicate based on the height of the node and the height of its neighbors.
  2. Change the parent node so that the local state becomes stable.

$$u \neq u_0 \wedge d(S_u) \neq n-1 \wedge \{d_u \neq d(S_u) + 1 \vee p_u \notin S_u\}$$
$$\Longrightarrow d_u = d(S_u) + 1; p_u = v, v \in S_u$$

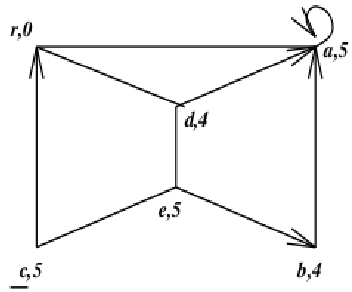
Self-Stabilizing Tree Construction

- Processes maintain a variable parent set to ∅ and height their hop-distance from the controlling process, set to ∅.
- The Controlling process sets height to 0 and broadcasts the search message with a counter set to 0.
- Processes receiving the search message set height to the value of the counter +1.
- Periodically processes broadcast their height and parent.
- Processes change parent if they discover a neighbor closer to the controlling process.



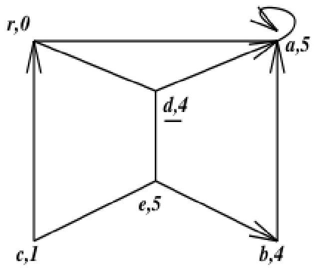
Self-Stabilizing Tree Construction

- Processes maintain a variable parent set to ∅ and height their hop-distance from the controlling process, set to ∅.
- The Controlling process sets height to 0 and broadcasts the search message with a counter set to 0.
- Processes receiving the search message set height to the value of the counter +1.
- Periodically processes broadcast their height and parent.
- Processes change parent if they discover a neighbor closer to the controlling process.



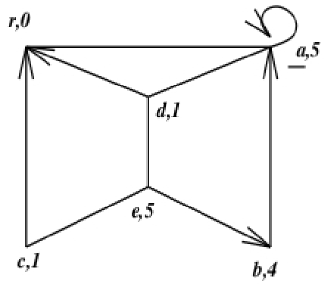
Self-Stabilizing Tree Construction

- Processes maintain a variable parent set to ∅ and height their hop-distance from the controlling process, set to ∅.
- The Controlling process sets height to 0 and broadcasts the search message with a counter set to 0.
- Processes receiving the search message set height to the value of the counter +1.
- Periodically processes broadcast their height and parent.
- Processes change parent if they discover a neighbor closer to the controlling process.



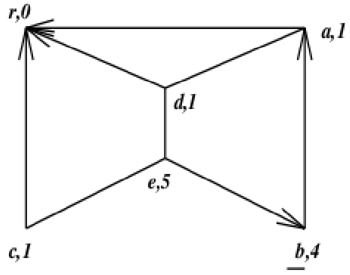
Self-Stabilizing Tree Construction

- Processes maintain a variable parent set to ∅ and height their hop-distance from the controlling process, set to ∅.
- The Controlling process sets height to 0 and broadcasts the search message with a counter set to 0.
- Processes receiving the search message set height to the value of the counter +1.
- Periodically processes broadcast their height and parent.
- Processes change parent if they discover a neighbor closer to the controlling process.



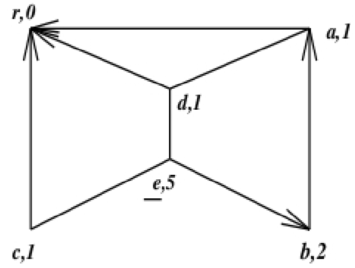
Self-Stabilizing Tree Construction

- Processes maintain a variable parent set to $\emptyset$ and height their hop-distance from the controlling process, set to $\emptyset$.
- The Controlling process sets height to 0 and broadcasts the search message with a counter set to 0.
- Processes receiving the search message set height to the value of the counter $+1$.
- Periodically processes broadcast their height and parent.
- Processes change parent if they discover a neighbor closer to the controlling process.



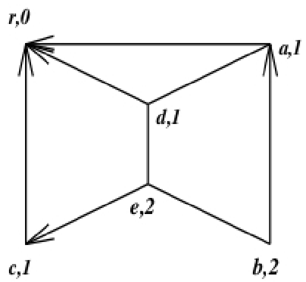
Self-Stabilizing Tree Construction

- Processes maintain a variable parent set to $\emptyset$ and height their hop-distance from the controlling process, set to $\emptyset$.
- The Controlling process sets height to 0 and broadcasts the search message with a counter set to 0.
- Processes receiving the search message set height to the value of the counter $+1$.
- Periodically processes broadcast their height and parent.
- Processes change parent if they discover a neighbor closer to the controlling process.



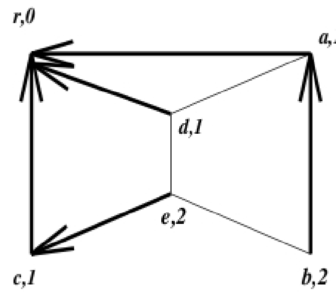
Self-Stabilizing Tree Construction

- Processes maintain a variable parent set to $\emptyset$ and height their hop-distance from the controlling process, set to $\emptyset$.
- The Controlling process sets height to 0 and broadcasts the search message with a counter set to 0.
- Processes receiving the search message set height to the value of the counter $+1$.
- Periodically processes broadcast their height and parent.
- Processes change parent if they discover a neighbor closer to the controlling process.



Self-Stabilizing Tree Construction

- Processes maintain a variable parent set to $\emptyset$ and height their hop-distance from the controlling process, set to $\emptyset$.
- The Controlling process sets height to 0 and broadcasts the search message with a counter set to 0.
- Processes receiving the search message set height to the value of the counter $+1$.
- Periodically processes broadcast their height and parent.
- Processes change parent if they discover a neighbor closer to the controlling process.



Proving Correctness

- Our goal is to prove that the three properties hold
  - In each unstable state, at least one process is taking a corrective action.
  - In each stable state, no process is active.
  - For all initial states and all possible executions, the algorithm guarantees convergence to a stable state in finite number of rounds.

Lemma

In a stable state, no process is active

- Holds due to the rule.

Proving Correctness

Lemma

In each unstable state at least one process is active, that is, in each unstable state it is guaranteed that some process will execute a corrective action.

- We prove the lemma by contradiction.
- Let an unstable state where no process is active.
- Then a process $u \neq u_0$ exists for which $d_u \neq d(S_u) + 1$ or $p_u \notin S_u$ or both.
- Then S_u must have height $n-1$ otherwise u would be active due to the rule.
- Let assume that all neighboring processes of u_0 (that have height 0)
  - These are the processes with height 1

Proving Correctness

- Then let assume all neighboring processes of these processes
  - These are the processes with height 2
- Continuing in the same way, we examine all the process of the network
  - In the wost case, process v may have height $n-1$
  - ... the network is a chain/line of length $n-1$.
- Even in this case, S_u is strictly smaller than $n-1$.
- Thus, when no process is active, we cannot identify any process u that holds the initial assumption.
- We have proved that the lemma holds.

Proving Correctness

Lemma

Regardless of the initial state, and regardless of the way processes are activated, the algorithm will always reach a stable state in finite number of steps.

- Since the number of states is finite, it is enough to show that starting from any initially unstable state, the system cannot re-enter the same initial state.
- Let x and y two identical states and $x \neq y$
  - State x is the state reached after x actions, starting from an initially unstable state.

Proving Correctness

- We assume that in x, process u (and maybe other nodes as well) is active
  - Thus u will take the $x+1$-th action
- We examine the possible actions that process u may execute
  1. u reduces its height by $k \geq 1$
  2. u increases its height by $k \geq 1$
- In both cases we follow the same arguments.
- Let's examine the 1st case.
- The has to be a process $v \in S_u$ neighboring u such that $d_u - k - 1$, that forced u to take an action.
- To be able to reach state $y(=$ state $x)$, $d(S_u)$ must increase by k.

Proving Correctness

- Thus at least one neighbor of u, let i, will increase its height, d_i by k.
- For this to happen there must be a process $j \in S_i$ with height $d_j = d_i + k - 1$ that forces i to take an action.
- Let assume a j such that $j \in S_i$ and $d(S_i) = d_i + k - 1$ and let d'_i is the new value of d_i ($d'_i = d_i + k$).
- However, now, the height of i differs from the height it had at state x (and thus in state y where we wish to reach)
- Thus, a neighboring node of i must re-instate it to the previous height (that is re-change $d(S_i)$)

Proving Correctness

- Repeating the same argument, there is always a node that needs to change its height so that it fixes the heights of those nodes that differ from state y.
- Therefore, we cannot reach the same state.