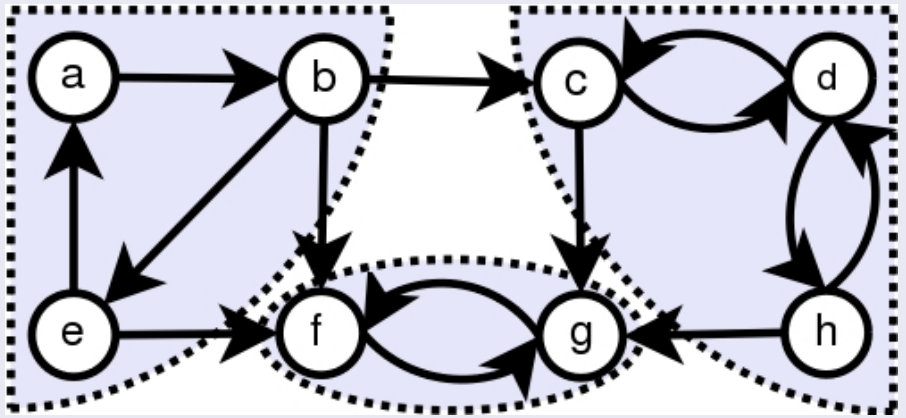
Pervasive Systems

Ioannis Chatzigiannakis

Sapienza University of Rome
Department of Computer, Control, and Management Engineering (DIAG)

Lecture 11:
Broadcast, Convergecast & Distributed Data Structures

Strongly Connected General Network



- The graph is strongly connected.
- Each process has UID – is not aware of the UID of the other processes.

Algorithm Flood

Every process maintains a record of the maximum UID it has seen so far (initially its own). At each round, each process propagates this maximum on all of its outgoing edges. After $diam(G)$ rounds, all nodes will know the maximum UID in the network.

- Processes are not aware of the total number of processes (n).
- Processes are aware of the network diameter — $\delta = diam(G)$

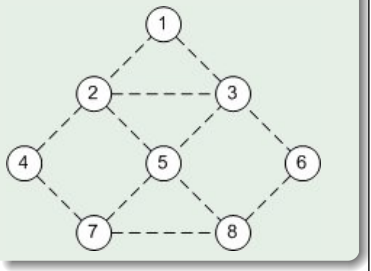
Pseudo-code for Flood

```
#DEFINE UID = <...>;
#DEFINE δ = <...>;
void main() {
   int max_id = UID;
   for (int i = 0 ; i < δ; i++ ) {
      sendMessage(max_id);
      while (int new_msg = readMessage()) {
         if (new_msg > max_id)
            max_id = new_msg;
      }
   }
}
```

Example of Execution for Flood Algorithm

- Let a synchronous distributed system of $n = 8$ processes..
  - General network where $\delta = 3$
  - Processes are index $1 \ldots 8$

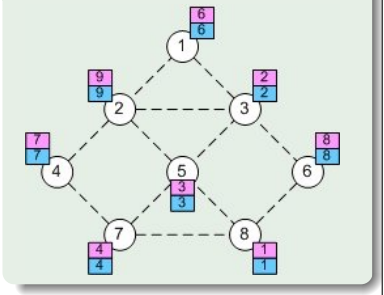
General Network



Example of Execution for Flood Algorithm

- Let a synchronous distributed system of $n = 8$ processes..
  - General network where $\delta = 3$
  - Processes are index $1 \ldots 8$
- The processes have UID.
  - Not aware of the UID of the other processes.

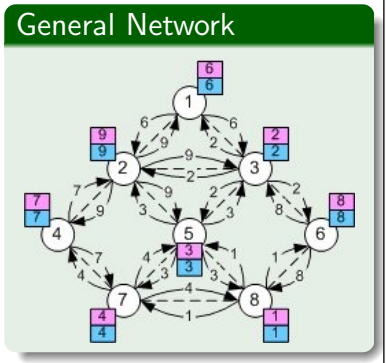
General Network



Example of Execution for Flood Algorithm

- Let a synchronous distributed system of $n = 8$ processes..
  - General network where $\delta = 3$
  - Processes are index $1 \ldots 8$
- The processes have UID.
  - Not aware of the UID of the other processes.
- First Round

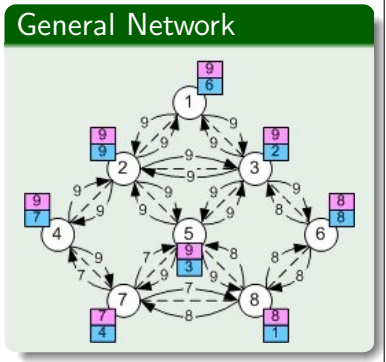
General Network



Example of Execution for Flood Algorithm

- Let a synchronous distributed system of $n = 8$ processes..
  - General network where $\delta = 3$
  - Processes are index $1 \ldots 8$
- The processes have UID.
  - Not aware of the UID of the other processes.
- First Round
- Second Round

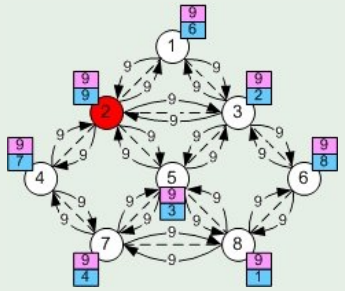
General Network



Example of Execution for Flood Algorithm

- Let a synchronous distributed system of $n = 8$ processes..
  - General network where $\delta = 3$
  - Processes are index $1 \ldots 8$
- The processes have UID.
  - Not aware of the UID of the other processes.
- First Round
- Second Round
- Final Round

General Network



Properties of Flood Algorithm

Let n processes and m channels, where the process with the highest UID is i_{max}.

- Process i_{max} is identified at the end of round δ.
- Time complexity is $\mathcal{O}(diam(G))$.
- Message complexity $\mathcal{O}(diam(G) \cdot m)$.

Proof of Correctness

Theorem (3.3)

In the Flood algorithm, process i_{max} is identified at the end of round δ.

Proof.

The key to the proof is the fact that after r rounds, the maximum UID has reached every process that is within distance r of i_{max}. In view of the definition of the diameter of the graph, this implies that every process has the maximum UID by the end of δ rounds. □

Directed spanning tree

A directed spanning tree of a directed graph $G = (V, E)$ is a rooted tree that consists entirely of directed edges in E, all edges directed from parents to children in the tree, and that contains every vertex of G.

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.

- Every strongly connected digraph has a breadth-first directed spanning tree.
- Constructing a Breadth-First directed spanning tree is useful for efficient collection of information.

SynchBFS Algorithm

At any point during execution, there is some set of processes that is "marked", initially just i_0. Process i_0 sends out a search message at round 1, to all of its outgoing neighbors. At any round, if an unmarked process receives a search message, it marks itself and chooses one of the processes from which the search has arrived as its parent. At the first round after a process gets marked, it sends a search message to all of its outgoing neighbors.

- Processes are not aware of the total number of processes (n)
- All processes have UIDs.

Example of Execution for SyncBFS Algorithm

Initial Network

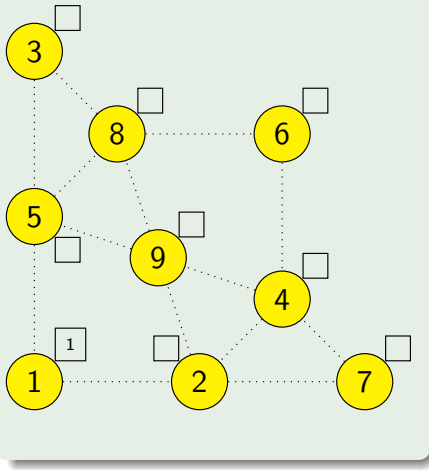
The network contains 9 processes, 14 channels

Process **1** initiates the execution.

Process **1** is marked.

All other processes are not marked.

Initial State

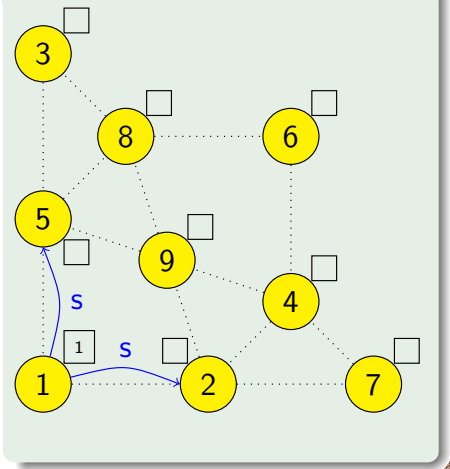


Example of Execution for SyncBFS Algorithm

1^{st} Round – 1^{st} Step

Process **1** sends search to its neighbors.

1^{st} Round – 1^{st} Step



Example of Execution for SyncBFS Algorithm

1^{st} Round – 1^{st} Step

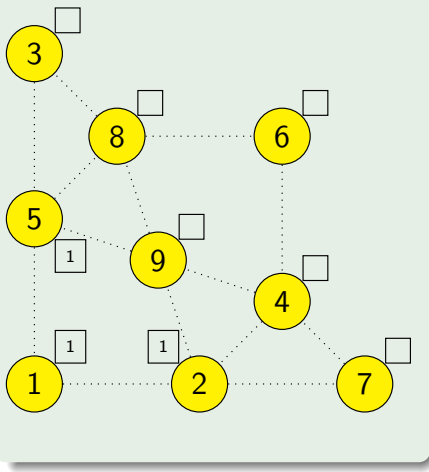
Process **1** sends search to its neighbors.

1^{st} Round – 2^{nd} Step

Processes **2, 5** are marked.

Processes **2, 5** select **1** as parent process.

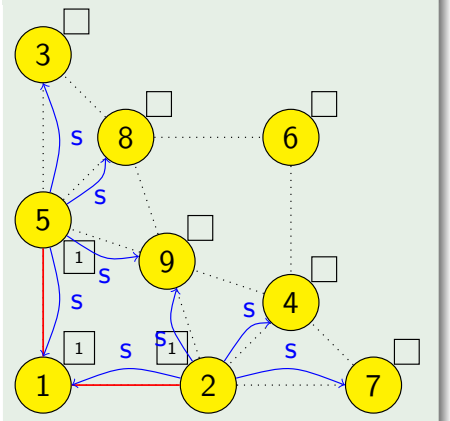
1^{st} Round – 2^{nd} Step



Example of Execution for SyncBFS Algorithm

2^{nd} Round – 1^{st} Step Processes **2, 5** send search to all neighbors.

2^{nd} Round – 1^{st} Step



Example of Execution for SyncBFS Algorithm

2^{nd} Round – 1^{st} Step Processes **2, 5** send search to all neighbors.

2^{nd} Round – 2^{nd} Step

Process **1** ignores all search messages received.

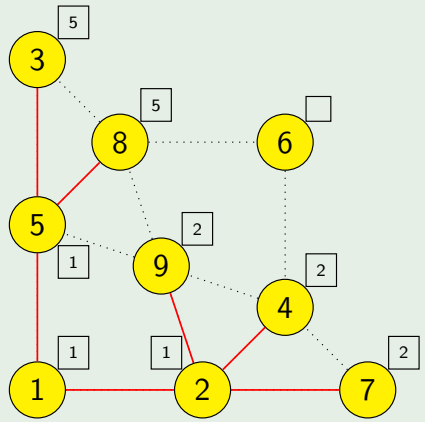
Processes **3, 4, 7, 8, 9** are marked.

Processes **3, 8** set process **5** as parent process.

Processes **4, 7** set process **2** as parent process.

Process **9** chooses (randomly) process **2** as parent process.

2^{nd} Round – 2^{nd} Step

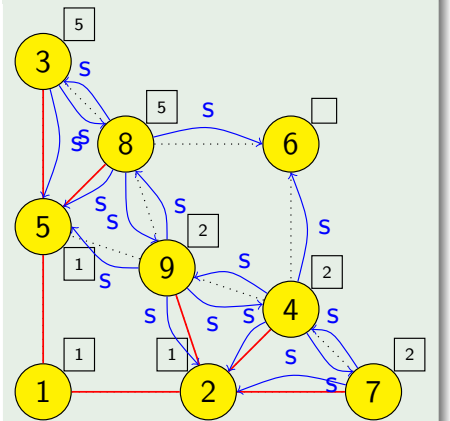


Example of Execution for SyncBFS Algorithm

3^{rd} Round – 1^{st} Step

Processes **3, 4, 7, 8, 9** send search to all neighbors.

3^{rd} Round – 1^{st} Step



Example of Execution for SyncBFS Algorithm

3^{rd} Round – 1^{st} Step

Processes **3, 4, 7, 8, 9** send search to all neighbors.

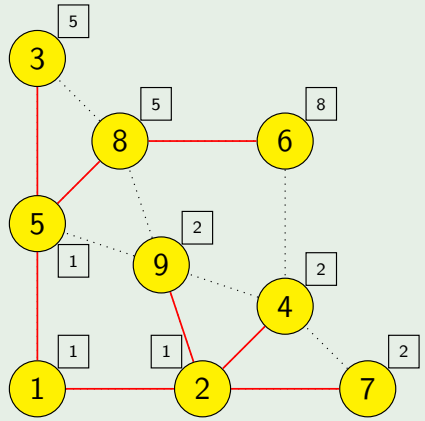
3^{rd} Round – 2^{nd} Step

Processes **2, 3, 4, 5, 7, 8, 9** ignore the search messages received.

Process **6** is marked.

Process **6** chooses (randomly) process **8** as parent process.

3^{rd} Round – 2^{nd} Step

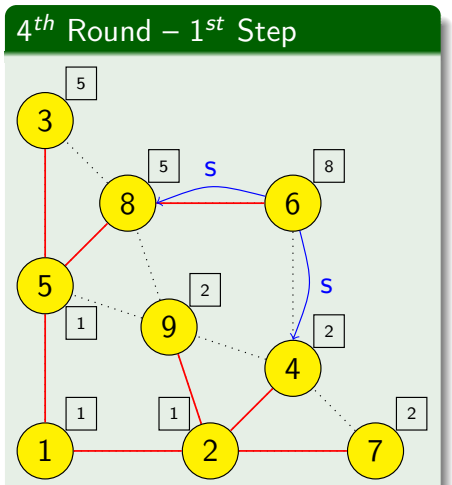


Example of Execution for SyncBFS Algorithm

4^{th} Round – 1^{st} Step

Process **6** sends search to all neighbors.

4^{th} Round – 1^{st} Step



Example of Execution for SyncBFS Algorithm

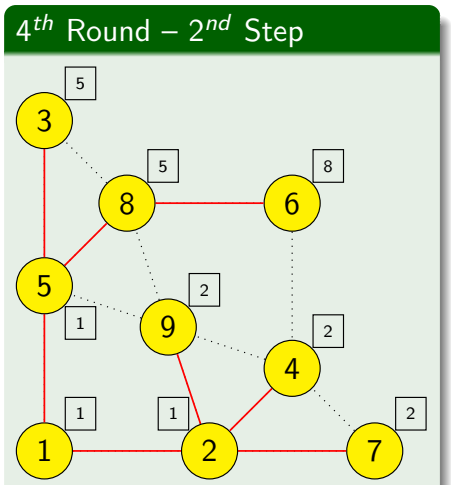
4^{th} Round – 1^{st} Step

Process **6** sends search to all neighbors.

4^{th} Round – 2^{nd} Step

Processes **4, 8** ignore the search messages received.

4^{th} Round – 2^{nd} Step



Example of Execution for SyncBFS Algorithm

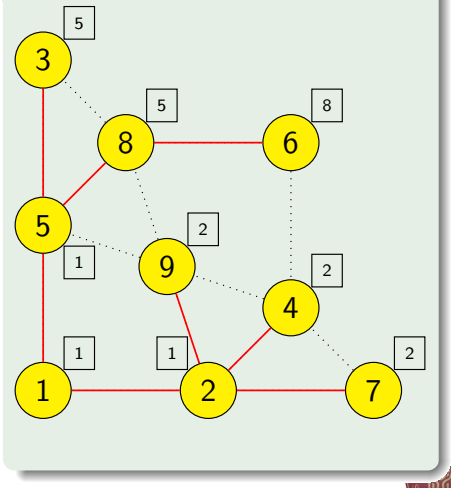
Final Step

Breadth-first directed spanning tree is constructed.

Total number of rounds: 4

Total number of messages: 28

Final State



Properties of SynchBFS Algorithm

- Algorithm SynchBFS constructs a breadth-first directed spanning tree.
- The structure is not stored in some "centralized" process.
- The tree pointers are "distributed" across the network.
- The time complexity is $\mathcal{O}(diam(G))$
  - In practice it is the maximum distance from u_0
  - In the example, the diameter is 4 – maximum distance from u_0 is 3.
- Message complexity: $\mathcal{O}(m)$

Improving Message Complexity

We can reduce the total number of messages exchanged by the algorithm as follows:

- The processes can identify the channel from which they received a search message.
- The processes do not send search towards those channels.

In the example, messages are reduced to 10 (i.e., 18 less).

Message Broadcast

The algorithm can easily be augmented to implement message broadcast.

- A process has a message m that it wants to communicate to all of the processes in the network.
- It initiates an execution of SynchBFS with itself as the root.
- Piggybacks message m on the search message in round 1.
- Other processes continue to piggyback m on all their search messages as well.
- Since the tree eventually spans all the nodes, message m is eventually delivered to all the processes.

Convergecast

Message convergecast is the inverse of a broadcast in a message-passing system (see Flooding) – instead of a message propagating down from a single root to all nodes, data is collected from outlying nodes through a direct spanning tree to the root. Typically some function is applied to the incoming data at each node to summarize it, with the goal being that eventually the root obtains this function of all the data in the entire system. (Examples would be counting all the nodes or taking an average of input values at all the nodes.)

Child Pointers

In order to use SyncBFS for message broadcast it is required that each process learn not only who its parent in the tree is, but also who all of its children are.

- If bidirectional communication is allowed between all pairs of neighbors, i.e., the network is undirected, this is simple.
- Each unmarked process, upon receiving the first search message, it sends a message parent to the process from which the message was received.

When SynchBFS' terminates, all processes are aware of their "children" processes.

The modified algorithm SynchBFS' requires $diam(G) + 2$ rounds and uses $m + n - 1$ messages.

Termination

How can the source process i_0 tell when the construction of the tree has completed ?

- The diameter of the network is know known, neither the total number of processes n.

If each search message is answered with either a parent or non-parent message, then after any process has received responses from all of its search messages, it knows who all its children are and knows that they have all been marked.

Termination

How can the source process i_0 tell when the construction of the tree has completed ?

Starting from the leaves of the BFS tree, notification of completion can be "fanned in" to the source:

- each process can send notification of completion to its parent in the tree as soon as
  1. it has received responses for all its search messages (so that it knows who its children are and knows that they have been marked)
  2. it has received notification of completion from all its children.

This type of procedure is called a convergecast.

Example of Execution for SyncBFS$_c$ Algorithm

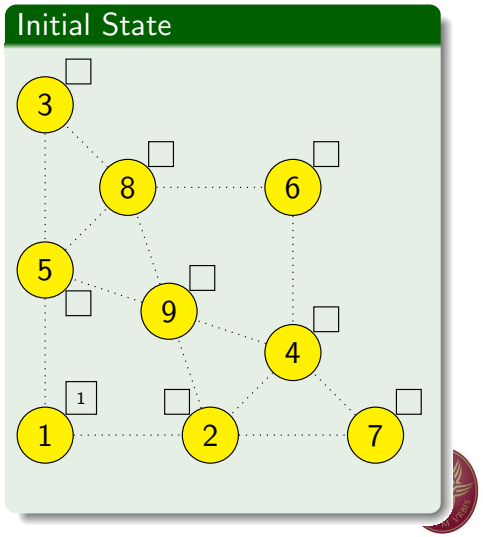
Initial Network

The network contains 9 processes, 14 channels

Process **1** initiates the execution.

Process **1** is marked.

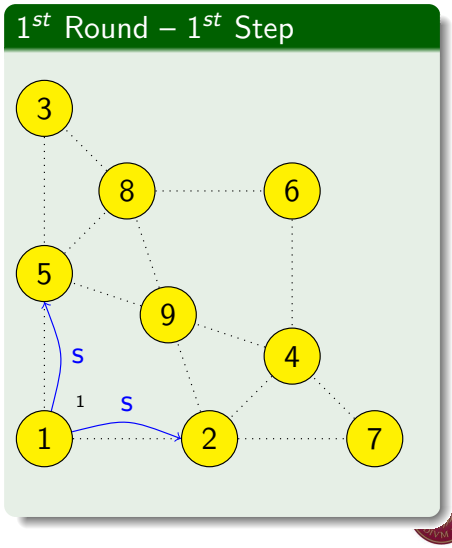
All other processes are not marked.



Example of Execution for SyncBFS$_c$ Algorithm

1st Round – 1st Step

Process **1** sends search to its neighbors.



Example of Execution for SyncBFS$_c$ Algorithm

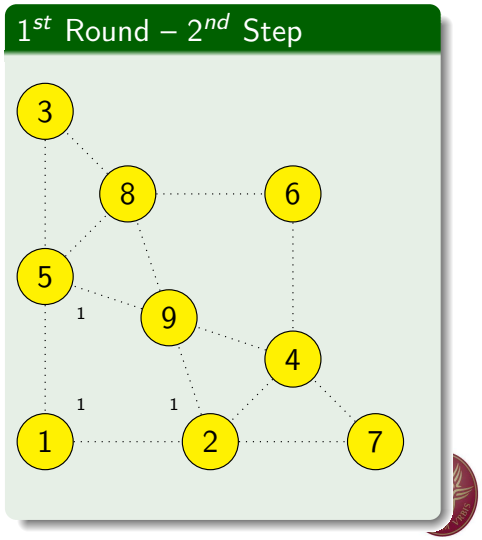
1st Round – 1st Step

Process **1** sends search to its neighbors.

1st Round – 2nd Step

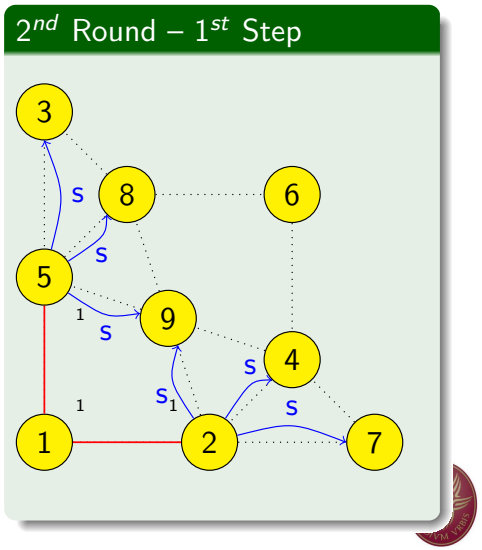
Processes **2, 5** are marked.

Processes **2, 5** select **1** as parent process.



Example of Execution for SyncBFS$_c$ Algorithm

2nd Round – 1st Step Processes **2, 5** send search to all neighbors.



Example of Execution for SyncBFS$_c$ Algorithm

2nd Round – 1st Step Processes **2, 5** send search to all neighbors.

2nd Round – 2nd Step

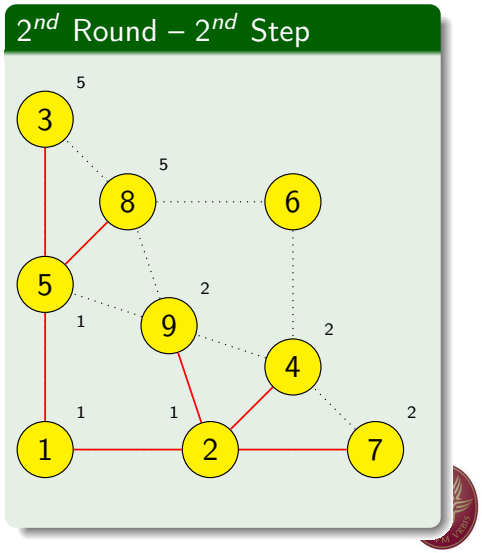
Process **1** ignores all search messages received.

Processes **3, 4, 7, 8, 9** are marked.

Processes **3, 8** set process **5** as parent process.

Processes **4, 7** set process **2** as parent process.

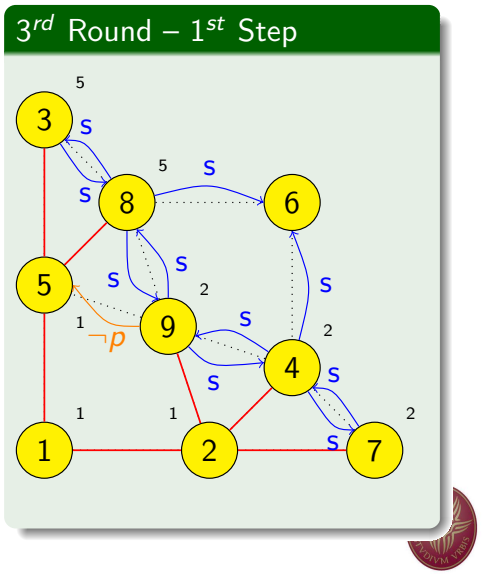
Process **9** chooses (randomly) process **2** as parent process.



Example of Execution for SyncBFS$_c$ Algorithm

3rd Round – 1st Step

Processes **3, 4, 7, 8, 9** send search to all neighbors.



Example of Execution for SyncBFS$_c$ Algorithm

3rd Round – 1st Step

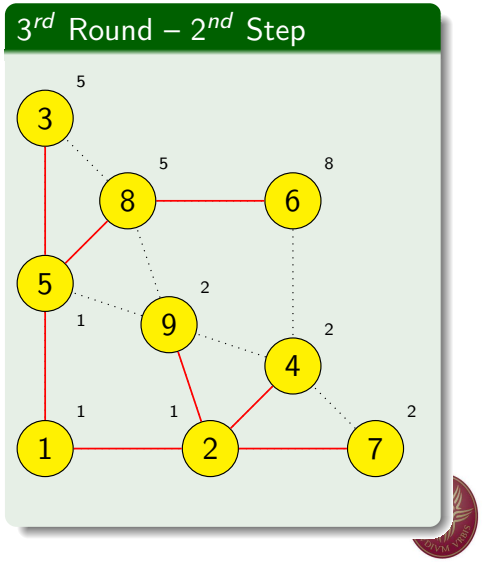
Processes **3, 4, 7, 8, 9** send search to all neighbors.

3rd Round – 2nd Step

Processes **2, 3, 4, 5, 7, 8, 9** ignore the search messages received.

Process **6** is marked.

Process **6** chooses (randomly) process **8** as parent process.

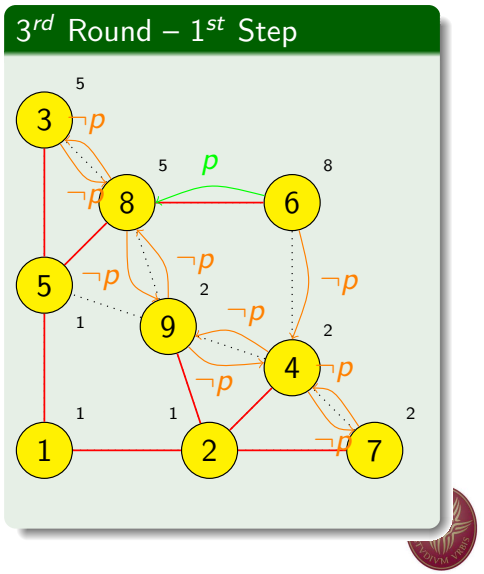


Example of Execution for SyncBFS$_c$ Algorithm

4th Round – 1st Step

Process **3** sends non-parent to **8**
Process **8** sends non-parent to **3**
Process **8** sends non-parent to **9**
Process **9** sends non-parent to **4**
Process **4** sends non-parent to **7**
Process **7** sends non-parent to **4**
Process **6** sends non-parent to **4**

Process **6** "" **8**



Example of Execution for SyncBFS$_c$ Algorithm

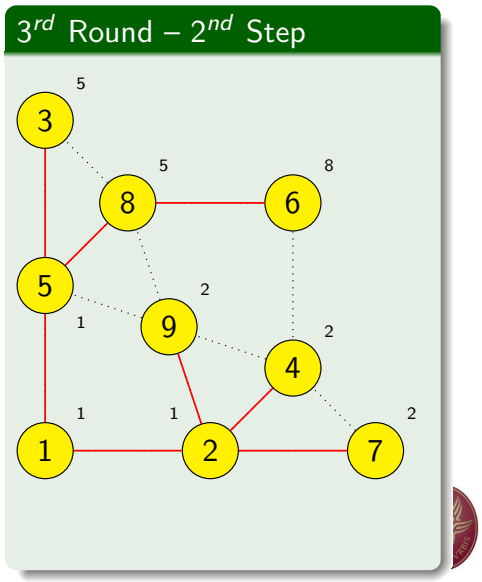
4th Round – 1st Step

Process **3** sends non-parent to **8**
Process **8** sends non-parent to **3**
Process **8** sends non-parent to **9**
Process **9** sends non-parent to **4**
Process **4** sends non-parent to **7**
Process **7** sends non-parent to **4**
Process **6** sends non-parent to **4**

Process **6** "" **8**

4th Round – 2nd Step

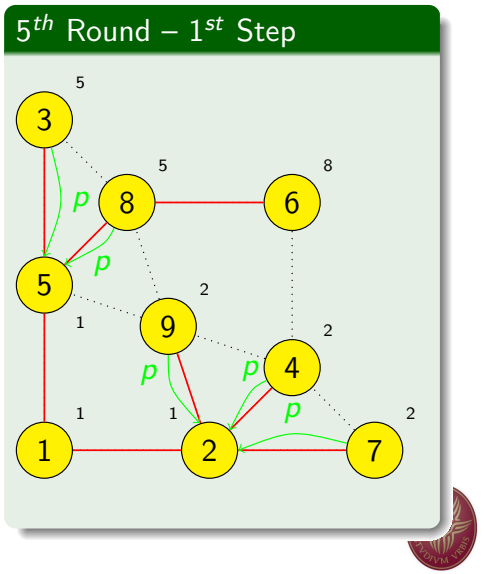
Process **8** detects the completion of the sub-tree of **6**



Example of Execution for SyncBFS$_c$ Algorithm

5th Round – 1st Step Processes **3, 8** send parent message to **5**

Processes **4, 7, 9** send parent message to **2**



Example of Execution for SyncBFS$_c$ Algorithm

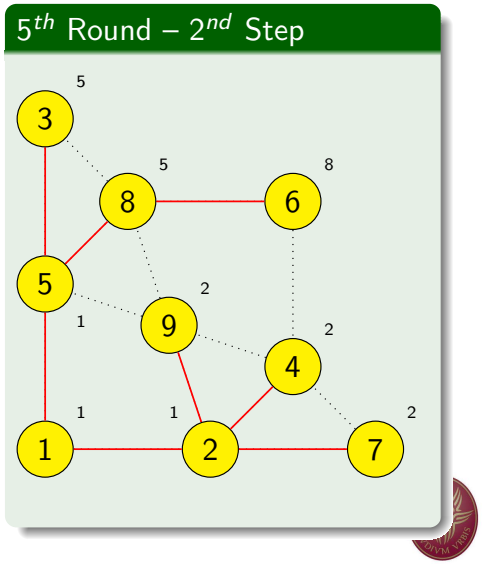
5th Round – 1st Step Processes **3, 8** send parent message to **5**

Processes **4, 7, 9** send parent message to **2**

5th Round – 2nd Step

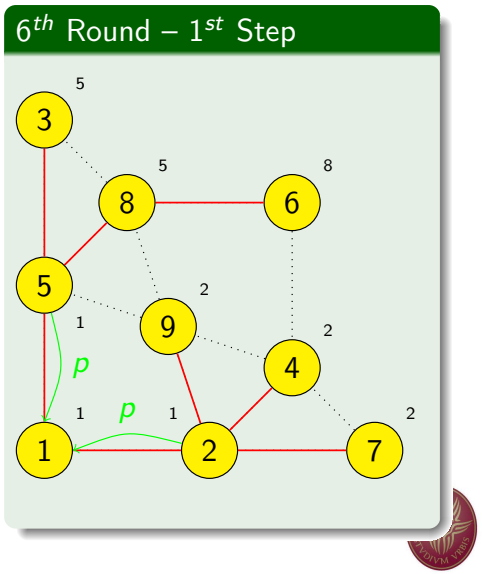
Process **5** detects the completion of the sub-trees of **3, 8**

Process **2** detects the completion of the sub-trees of **4, 7, 9**



Example of Execution for SyncBFS$_c$ Algorithm

6th Round – 1st Step Processes **2, 5** send parent message to **1**



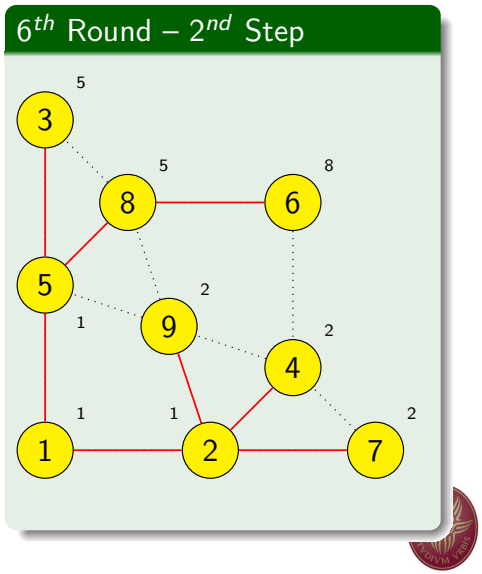
Example of Execution for SyncBFS$_c$ Algorithm

6th Round – 1st Step Processes **2, 5** send parent message to **1**

6th Round – 2nd Step

Process **1** detects the completion of the sub-trees of **2, 5**

Process **1** terminates



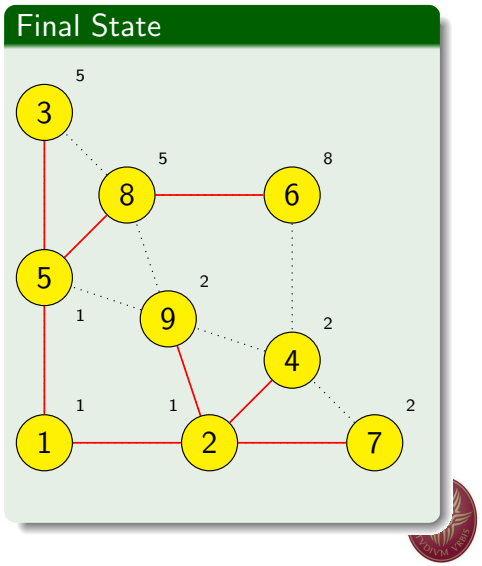
Example of Execution for SyncBFS$_c$ Algorithm

Final Step

Breadth-first directed spanning tree is constructed.

Total number of rounds: 6

Total number of messages: 36



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.

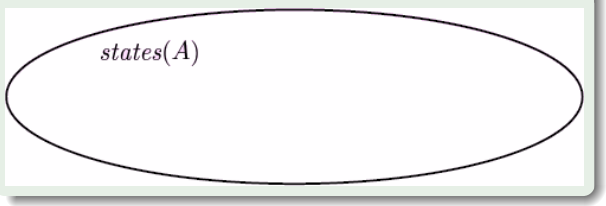
Definition

- 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 state in $\mathcal{L}$
- Then we identify a potential function (convergence function).

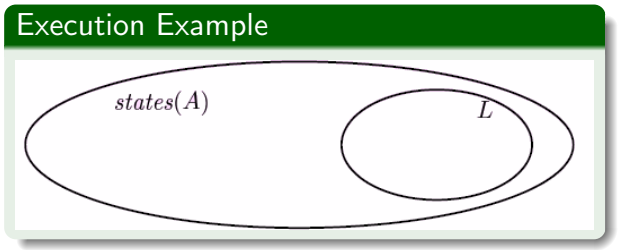
Execution Example



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 state in $\mathcal{L}$
- Then we identify a potential function (convergence function).

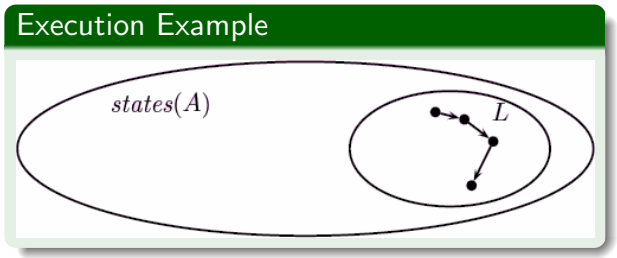
Execution Example



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 state in $\mathcal{L}$
- Then we identify a potential function (convergence function).

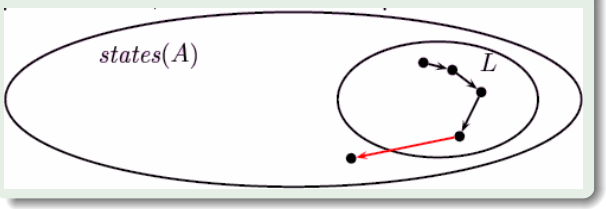
Execution Example



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 state in $\mathcal{L}$
- Then we identify a potential function (convergence function).

Execution Example

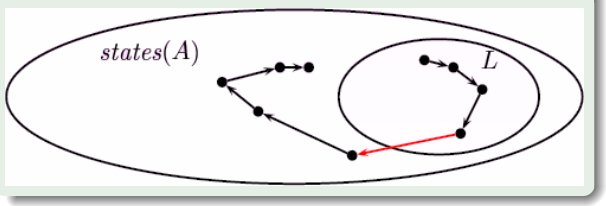


General Synchronous Networks
Synchronous Networks with Failures
Stabilization

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 state in $\mathcal{L}$
- Then we identify a potential function (convergence function).

Execution Example



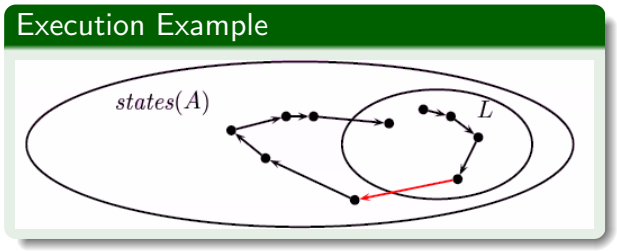
Ioannis Chatzigiannakis
Pervasive Systems
Lecture 11 21 / 29

General Synchronous Networks
Synchronous Networks with Failures
Stabilization

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 state in $\mathcal{L}$
- Then we identify a potential function (convergence function).

Execution Example



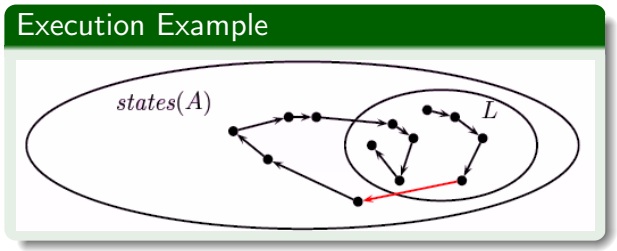
Ioannis Chatzigiannakis
Pervasive Systems
Lecture 11 21 / 29

General Synchronous Networks
Synchronous Networks with Failures
Stabilization

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 state in $\mathcal{L}$
- Then we identify a potential function (convergence function).

Execution Example



Ioannis Chatzigiannakis
Pervasive Systems
Lecture 11 21 / 29

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.

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.

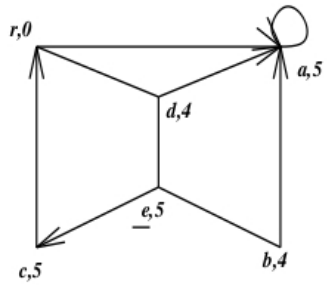
Main Idea

- 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

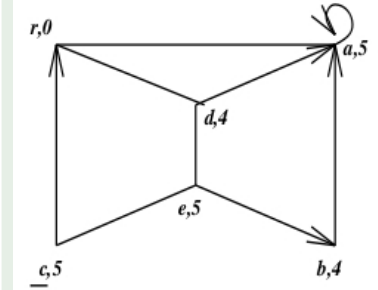
- Processes maintain a variable parent set to $\varnothing$ and height their hop-distance from the controlling process, set to $\varnothing$.
- The Controlling process sets height to 0 and broadcasts the search message with a counter set to 0.
- Processes receiving the search message set height to the 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.



General Synchronous Networks
Synchronous Networks with Failures
Self-Stabilizing Data Structures

Self-Stabilizing Tree Construction

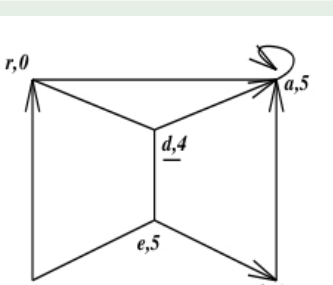
- Processes maintain a variable parent set to $\varnothing$ and height their hop-distance from the controlling process, set to $\varnothing$.
- The Controlling process sets height to 0 and broadcasts the search message with a counter set to 0.
- Processes receiving the search message set height to the 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.



General Synchronous Networks
Synchronous Networks with Failures
Self-Stabilizing Data Structures

Self-Stabilizing Tree Construction

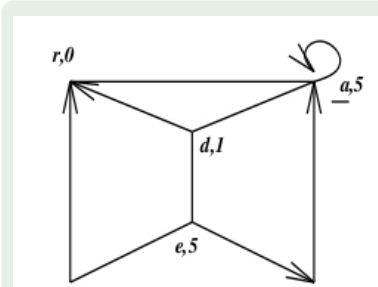
- Processes maintain a variable parent set to $\varnothing$ and height their hop-distance from the controlling process, set to $\varnothing$.
- The Controlling process sets height to 0 and broadcasts the search message with a counter set to 0.
- Processes receiving the search message set height to the 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.



General Synchronous Networks
Synchronous Networks with Failures
Self-Stabilizing Data Structures

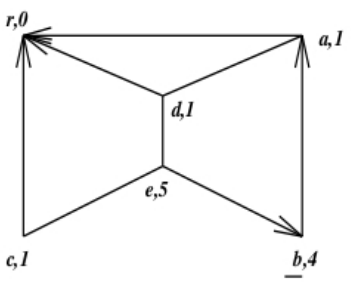
Self-Stabilizing Tree Construction

- Processes maintain a variable parent set to $\varnothing$ and height their hop-distance from the controlling process, set to $\varnothing$.
- 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

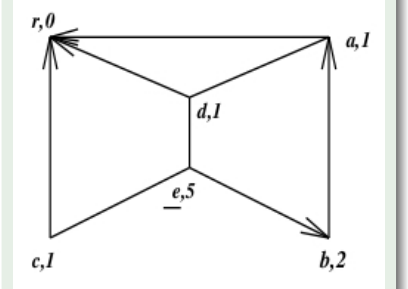
- Processes maintain a variable parent set to $\varnothing$ and height their hop-distance from the controlling process, set to $\varnothing$.
- The Controlling process sets height to 0 and broadcasts the search message with a counter set to 0.
- Processes receiving the search message set height to the 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.



General Synchronous Networks
Synchronous Networks with Failures
Self-Stabilizing Data Structures

Self-Stabilizing Tree Construction

- Processes maintain a variable parent set to $\varnothing$ and height their hop-distance from the controlling process, set to $\varnothing$.
- 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.

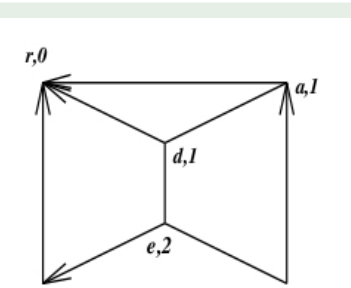


Ioannis Chatzigiannakis
Pervasive Systems
Lecture 11 28 / 29

General Synchronous Networks
Synchronous Networks with Failures
Self-Stabilizing Data Structures

Self-Stabilizing Tree Construction

- Processes maintain a variable parent set to $\varnothing$ and height their hop-distance from the controlling process, set to $\varnothing$.
- 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.

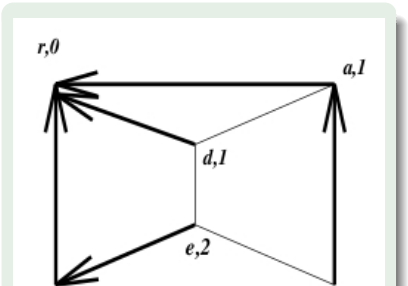


Ioannis Chatzigiannakis
Pervasive Systems
Lecture 11 28 / 29

General Synchronous Networks
Synchronous Networks with Failures
Self-Stabilizing Data Structures

Self-Stabilizing Tree Construction

- Processes maintain a variable parent set to $\varnothing$ and height their hop-distance from the controlling process, set to $\varnothing$.
- 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.



Ioannis Chatzigiannakis
Pervasive Systems
Lecture 11 28 / 29

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.