Computer Networks

EDA387/DIT663

Fault-tolerant Algorithms for Computer Networks

Self-Stabilizing Algorithms for Model Conversions (*Ch. 4*)

Compiler

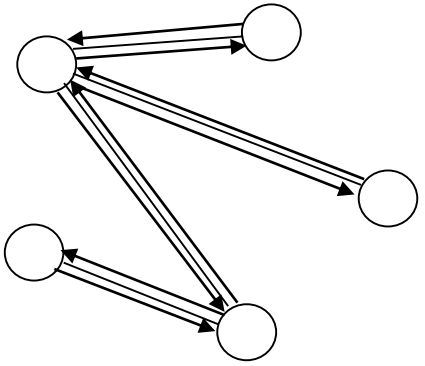
- All the above facts are our motivation for designing a compiler:

- An algorithm designed to stabilize in the presence of distributed daemon must stabilize in a system with central daemon

Converting

- How does the compiler work ?

1. Compose a spanning tree using the Spanning Tree Construction algorithm



2. Construct an Euler tour on the tree to create a virtual ring for the Mutual Exclusion algorithm
3. A processor that enters the critical section reads the state of its neighbors changes state and writes, then it exits the critical section

Al for T (read/write) = Al for T (daemon) ◦ Mutual Exclusion

Chapter 4: roadmap

4.3 Self-Stabilizing Ranking: Converting an Id-based System to a Special-processor System

4.4 Update: Converting a Special Processor to an Id-based Dynamic System

Converting an Id-based System to a Special-processor System

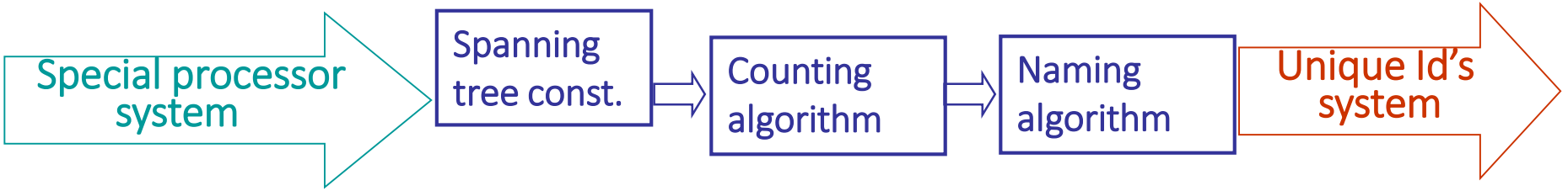
- Our goal is to design a compiler that converts self-stabilizing algorithm for a unique *ID* system to work in special processor system.

- The *ranking* (compiler) task is to assign each of the *n* processors in the system with a unique identifier in the range 1 to n.

Converting an Id-based System to a Special-processor System

We will form the self stabilizing *ranking* algorithm by running 3 self-stabilizing algorithms one after the other:

1) Self-Stabilizing spanning tree construction (section 2.5)
2) Self-Stabilizing counting algorithm
3) Self-Stabilizing naming algorithm



Self-Stabilizing Counting Algorithm

- Assuming rooted spanning tree system in which every processor knows its parent and children's
- P_i has a variable $count_i$ that hold the number of processor in sub-tree where P_i is a root
- The correctness proofs by induction on the height of a processor

Self-Stabilizing Counting Algorithm

```
01 Root: do forever
02         sum:= 0
03                 forall p_j ∈ children(i) do
04                         lr_ji := read(r_ji)
05                         sum := sum + lr_ji.count
06                 od
07                 count_i = sum + 1
08         od
09 Other: do forever
10                 sum:= 0
11                 forall p_j ∈ children(i) do
12                         lr_ji := read(r_ji)
13                         sum := sum + lr_ji.count
14                 od
15                 count_i = sum + 1
16                 write r_i,parent.count := count_i
17         od
```

Calculate $count_i$: sum the values of r_{ji} registers of his child's and 1 (himself)

Write local count value to communication register

Self-Stabilizing Naming Algorithm

- The *naming* algorithm uses the value of the *count* fields from *counting* algorithm.
- Algorithm assign unique identifiers to the processors.
- The identifier of a processor is stored in the ID_i variable.
- Proof of the stabilization by induction on the distance of the processors from the root

Self-Stabilizing Naming Algorithm

```
01 Root: do forever
02                  ID_i := 1
03                  sum := 0
04                  forall p_j ∈ children(i) do
05                          lr_ji := read(r_ji)
05                          write r_ij.identifier := Id + 1 + sum
06                          sum := sum + lr_ji.count
08                  od
09          od
10 Other: do forever
11                  sum:= 0
12                  lr_parent,i := read(r_parent,I)
13                  ID_i := lr_parent,I.identifier
14                  forall p_j ∈ children(i) do
15                          lr_ji := read(r_ji)
16                          write r_ij.identifier := Id_i + 1 + sum
17                          sum := sum + lr_ji.count
18                  od
19          od
```

Counting Algorithm for non-rooted tree

```
01 do forever
02        forall P_j ∈ N(i) do lr_ji := read (r_ji)
03        sum_i := 0
04        forall P_j ∈ N(i) do
05                  sum_j := 0
06                  forall P_k ∈ N(i) do
07                            if P_j != P_k then
08                                      sum_j := sum_j + lr_ki.count
09                  od
10                  count_i[j] := sum_j + 1
11                  sum_i := sum_i + sum_j
12                  write r_ij.count := count_i[j]
13        od
14        count_i = sum_i + 1
15 od
```

Each processor P_i has a variable $count_i[j]$ for every neighbor P_j

The value of $count_i[j]$ is the number of processors in subtree of T to which P_i belongs and P_j doesn't.

The correctness proof is by induction on the height of the registers r_{ij}

Chapter 4: roadmap

4.3 Self-Stabilizing Ranking: Converting an Id-based System to a Special-processor System

4.4 Update: Converting a Special Processor to an Id-based Dynamic System

Update - Converting a Special Processor to an Id-based Dynamic System

- The task of the update algorithm in id-based system is to inform each processor of the other processors that are in its connected component.
- As a result every processor in the connected component knows the maximal identifier in the system, and a single leader is elected.
- The update algorithm is a self-stabilizing leader election algorithm within O(d) cycles (pulses).

The motivation for the restriction that the update must work in id-based system can be viewed by examining Dijkstra self stabilizing ME algorithm for a ring of processors…

Dijkstra proof

- Dijkstra proved that without a special processor, it is impossible to achieve ME in a self-stabilizing manner
- The impossibility proof is for composite number of identical processors connected in a ring activated by a central daemon

ME requires that processor P_i should execute the critical section if and only if it is the only processor that can change its state (by reading its neighbors' states) at that execution point

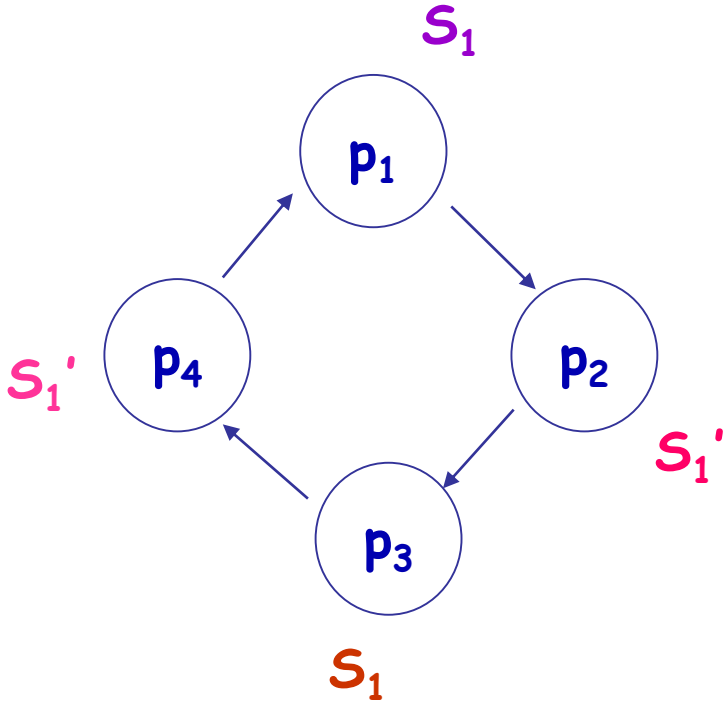
Dijkstra proof

- Consider P_1,P_3 starting in the same state S_0, P_2, P_4 starting in the same state S_0'

And execution order is:

P_1 ,P_3 ,P_2 ,P_4 …

- Then, p_1 and p_3 will be at S_1, p_2 and p_4 will be at S_1'.
- We can see that symmetry in state is preserved



Conclusions from Dijkstra's proof

- Whenever P_1 has permission to execute critical so does P_3. (Like that for P_2 and P_4).
- With no central daemon the impossibility result of self-stabilizing ME algorithm holds also for a ring of prime number of identical processors.

  We start in a configuration that all the processors are in the same state, and the contents of all the registers are identical. An Execution that preserve that symmetry forever is one that every processor reads all neighbors registers before any writing is done.

The restriction for designing the update algorithm in an **id-based** (not identical) system is thus clear.

The Update algorithm outlines

- **The algorithm constructs n directed BFS trees. One for every processor.**

1. Each processor P_i holds a BFS tree rooted at P_i.
2. When a node P_j with distance k+1 from P_i has more than one neighbor at distance k from P_i, P_j is connected to the neighbor (parent) with the maximal ID.
3. Each P_i reads from his δ neighbors their set of tuples <j,x> where j represents a processor id, and x is the distance of j from the processor that holds this tuple.
4. P_i tuples are computed from these neighbors' sets, adapting for each $j \neq i$ the tuple <j,x> with the smallest x and adding 1 to x. P_i also adds the tuple <i,0> indicating the distance from itself. At the end of the iteration, P_i removes the floating (false) tuples.

Update algorithm

01 **do forever**
02 $Readset_i := \emptyset$
03 **forall** $p_j \in N(i)$ **do**
04 $Readset_i := Readset_i \cup$ **read**$(Processors_j)$
05 $Readset_i := Readset_i$ \\ <i,*>
06 $Readset_i := Readset_i$ ++ \\<*,1>
07 $Readset_i := Readset_i \cup$ {<i,0>}
08 **forall** $p_j \in processors(Readset_i)$ **do**
09 $Readset_i := Readset_i$\\ *NotMin*Dist(P_j, $Readset_i$)
10 **write** $Processors_i := ConPrefix(Readset_i)$
11**od**

Floating Tuples

- Floating tuple in ReadSet is a tuple with a processor's id that does not exist.
- Let y be the minimal missing distance value in $ReadSet_i$.
- We remove every tuple with distance greater than y in $ReadSet_i$.

  ConPrefix($ReadSet_i$) removes these tuples out.

Correctness

To prove the correctness, we define *safe configuration,* such that for every p_i it holds:

1. $Processors_i$ includes n tuples, a tuple <j,y> for every processor p_j in the system, where y is the distance of p_j from p_i.
2. $ReadSet_i$ tuples will rewrite the same contents to $Processors_i$

Tuple Addition (Lemma 4.6)

In every arbitrary execution following the k^{th} cycle, for each pair of processors p_i and p_j that are at distance $\ell < \min(k, d+1)$ it holds:

Invariant 1 (inclusion): A tuple <j,l> appears in $Processors_i$.

Invariant 2 (constant value): If a tuple <x,y>, such that $y \leq \ell$ appears in $Processors_i$, there exists a processor p_x at distance y from p_i.

Tuple Addition – cont.

The proof is by induction on k, which is the number of cycles in the execution.

Base case:

- k=1, by adding <i,0> to $ReadSet_i$ (line 7), incrementing the distance of all other tuples to be greater than 0 (line 6), Invariant 1 (inclusion) holds.
- After these increments, only tuple <i,0> is left with distance 0. Thus Invariant 2 (constant value) holds.

Induction assumption :

Invariants 1 and 2 hold after k cycles.

Induction Step:

- By the induction assumption, in the start of the $k+1^{th}$ cycle the processors variables are correct up to distance ℓ-1.
- Cycle k+1 reads all the correct tuples with distance ℓ-1, so it computes all the tuples of distance ℓ correctly.

Correctness (Lemma 4.7)

In every arbitrary execution following d+2 cycles,
it holds for every tuple <x,y> in every $processors_i$ variable that a processor x exists in the system.

Correctness - cont.

The proof depend on Tuple addition (Lemma 4.6)

- According to Lemma 4.6, it holds that following d+1 cycles every tuple with distance smaller than d is not a floating tuple.

- If a floating tuple <x,y> exists after cycle d+1 in a $processors_i$ variable, then y > d. Cycle d+2 then increments y, thus y > d+1.

- Since no tuple of distance d+1 exists, the function ConPrefix removes it.

Stabilization (Corollary 4.1)

In any execution, any configuration that follows the first d+3 cycles is a *safe configuration*

Proof:

- In accordance with Lemma 4.6, it holds that, in every configuration that follows the first d+1 cycles, every tuple with distance smaller than *d* is not a floating tuple, and also $\forall p_j$ and P_i at distance $\ell \leq d$, it holds that a tuple $<j, \ell> \in Processors_i$.

- In accordance with Lemma 4.7, in every configuration that follows d+2 cycles, no tuples of grater distance than d exists in $Processors_i$ variables.

- Therefore, during the d+3 cycle, a *safe configuration* is reached.

Self-Stabilizing Convergecast for Topology Update

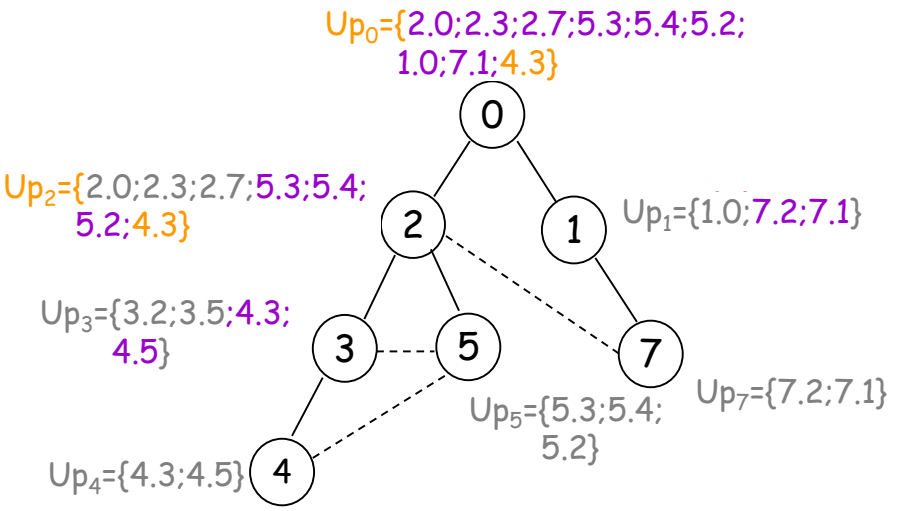
In the topology-update algorithm, the information that is convergecast is the local topology (the descendant's neighbors identity).

We assume that every processor knows its parent and children in the BFS tree of the leader

(can be done in O(d) cycles with update).

Self-Stabilizing Convergecast for Topology Update

- In the convergecast every processor P_i uses the variable up_i to write to his parent his local topology and also (if P_i is not a leaf) his children's topology.
- The stabilization of the convergecast is based on the correct information in the leaves and direction, in which information is collected - from leaves up.



Self Stabilizing Broadcast for Topology Update

In order to inform every processor the tree topology collected by the leader, we use a self-stabilizing broadcast mechanism:

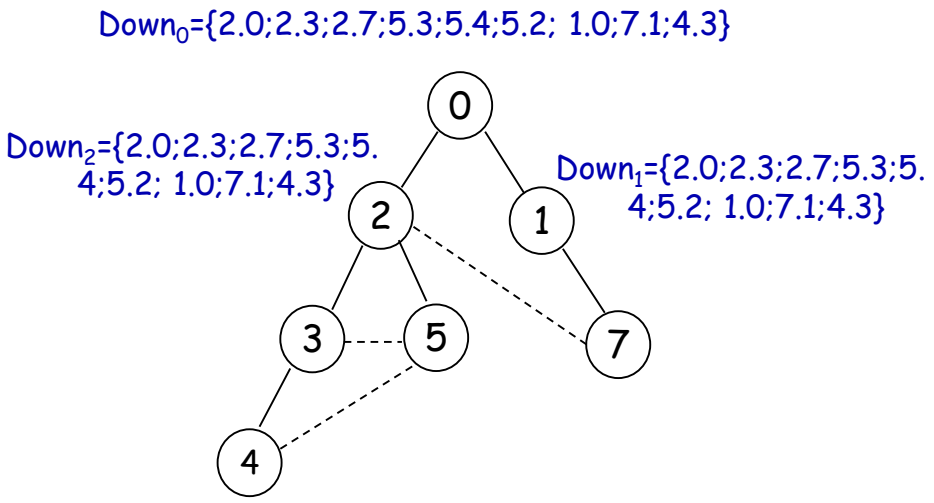
Root (x):

$Down_x = Up_x$

Everyone else (i):

$Down_i = Down_j \mid j=parent(i)$

This is done in O(d).



Adaptive self-stabilizing Algorithm

In dynamic systems, the parameters of the system such as diameter and number of processors are not fixed

- A self-stabilizing algorithm is time-adaptive if the number of cycles until convergence is proportional to the system's parameters.
- A self-stabilizing algorithm is memory-adaptive if the amount of memory used during safe configuration is proportional to the system's parameters.
- A silent self-stabilizing algorithm is communication-adaptive if the number of bits that are communicated is proportional to the system's parameters.

The update algorithm stabilizes within O(d) cycles therefore it's time-adaptive. It reads/writes O(n) tuples in the communication register and is therefore memory-adaptive and communication-adaptive.

Summary

We have looked into some common algorithmic and proof techniques in self-stabilization

Review Questions

1. Design a self-stabilizing mutual exclusion algorithm for a system with processors $p_1, p_2, \ldots, p_n$ that are connected in a line. The leftmost processor p_1 is the special processor. Every processor p_i, $2 \leq i \leq n - 1$, communicates with its left neighbor $p_i - 1$ and its right neighbor $p_i + 1$. Similarly, p_1 communicates with p_2 and p_n with p_{n-1}.
2. Define a safe configuration for the self-stabilizing synchronous consensus algorithm of figure 2.7.