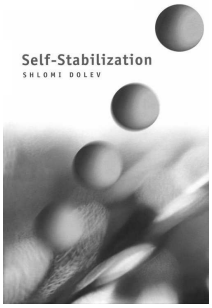
Chapter 2
Self-Stabilization



Self-Stabilization
Shlomi Dolev
MIT Press , 2000

Draft of May 2003,
Shlomi Dolev, All Rights Reserved ©

Chapter 2: roadmap

2.1 Definitions of Computational Model
2.2 Self-Stabilization Requirements
2.3 Complexity Measures
2.4 Randomized Self-Stabilization
2.5 Example: Spanning-Tree Construction
2.6 Example: Mutual Exclusion
2.7 Fair Composition of Self-Stabilization Algorithms
2.8 Recomputation of Floating Output
2.9 Proof Techniques
2.10 Pseudo-Self-Stabilization

What is a Distributed System?

- Communication networks
- Multiprocessor computers
- Multitasking single processor

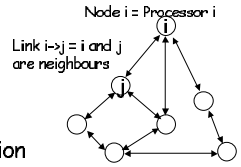
A Distributed System is modeled by a set of n state machines called processors that communicate with each other

The Distributed System Model

Denote :

- P_i - the i'th processor
- neighbor of P_i - a processor that can connect with it

How to Represent the Model?

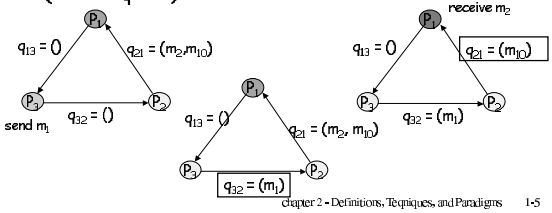


Ways of communication

- message passing - fits communication networks and all the rest
- shared memory - fits geographically close systems

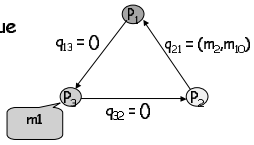
Asynchronous Distributed Systems - Message passing

- A communication link which is unidirectional from P_i to P_j transfers message from P_i to P_j
- For a unidirectional link we will use the abstract q_{ij} (a FIFO queue)



Asynchronous Distributed Systems - Message passing

- System configuration (configuration) : Description of a distributed system at a particular time.
- A configuration will be denoted by $c = (s_1, s_2, \ldots, s_n, q_{1,2}, q_{1,3}, \ldots, q_{i,j}, q_{n,n-1})$, where s_i = State of P_i $q_{i,j}$ ($i \neq j$) the message queue

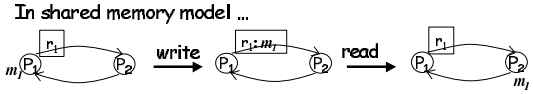


Asynchronous Distributed Systems – Shared Memory

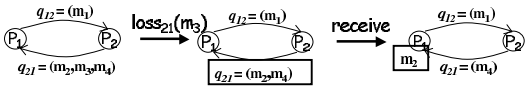
- Processors communicate by the use of shared communication registers
- The configuration will be denoted by
  $c = (s_1,s_2,...,s_n,r_1,r_2,...,r_m)$ where
  s_i = State of P_i
  r_i= Content of communication register i

The distributed System – A Computation Step

In shared memory model ...



And in message passing model ...



The Interleaving Model

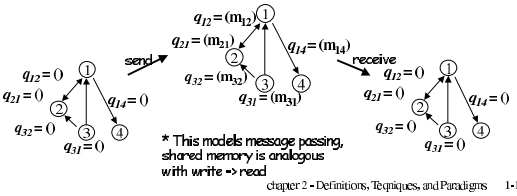
- Scheduling of events in a distributed system influence the transition made by the processors
- The interleaving model – at each given time only a single processor executes a computation step
- Every state transition of a process is due to communication-step execution
- A step will be denoted by a
- For c_1 and c_2 configurations, c_2 is reached from c_1 by a single step 'a' of a processor, will be denoted by: $c_1 \xrightarrow{a} c_2$

The distributed System – more definitions

- Step a is applicable to configuration c iff $\exists\, c' : c \xrightarrow{a} c'$.
  - In the message passing system - queue $q_{i,j}$ contains m, and in the c_{k+1} m is removed from $q_{i,j}$ and c_k & c_{k+1} are identical in all the rest
- An execution $E = (c_1,a_1,c_2,a_2,...)$, an alternating sequence such that $c_{i-1} \xrightarrow{a} c_i$ $(i>1)$
- A fair execution - every step that is applicable infinitely often is executed infinitely often
  - In the message passing model - a message which is sent infinitely often will be received infinitely often – number of loss is finite

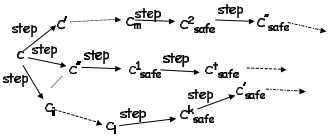
Synchronous Distributed Systems

- A global clock pulse (pulse) triggers a simultaneous step of every processor in the system
- Fits multiprocessor systems in which the processors are located close to one another
- The execution of a synchronous system E = (c1,c2,...) is totally defined by c1, the first configuration in E



Legal Behavior

- A desired legal behavior is a set of legal executions denoted LE



A self-stabilizing system can be started in any arbitrary configuration and will eventually exhibit a desired "legal" behavior

Time complexity

- The first asynchronous round (round) in an execution E is the shortest prefix E' of E such that each processor executes at least one step in E', E=E'E".
- The number of rounds = time complexity
- A Self-Stabilizing algorithm is usually a do forever loop
- The number of steps required to execute a single iteration of such a loop is $O(\Delta)$, where Δ is an upper bound on the number of neighbors of P_i

Time complexity ...

- Asynchronous cycle (cycle) the first cycle in an execution E is the shortest prefix E' of E such that each processor executes at least one complete iteration of it's do forever loop in E', E=E'E".
- Note : each cycle spans $O(\Delta)$ rounds
- The time complexity of synchronous algorithm is the number of pulses in the execution

Space complexity

- The space complexity of an algorithm is the total number of (local and shared) memory bits used to implement the algorithm

Randomized Self-Stabilization

Assumptions and definitions

- Processor activity is managed by a scheduler
- The scheduler's assumption - at most one step is executed in every given time
- The scheduler is regarded as an adversary
- The scheduler is assumed to have unlimited resources and chooses the next activated processor *on-line*
- A scheduler S is fair if, for any configuration c with probability 1, an execution starting from c in which processors activated by S is fair

Randomized Self-Stabilization - Assumptions and definitions ..

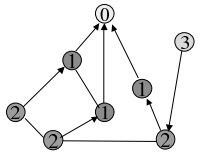
- An algorithm is randomized self-stabilizing for task LE if, starting with any system configuration and considering any fair scheduler, the algorithm reaches a safe configuration within an expected number of rounds that is bounded by some constant k

Randomized algorithms are often used to break symmetry in a system of totally identical processors

Chapter 2: roadmap

2.1 Definitions of Computational Model
2.2 Self-Stabilization Requirements
2.3 Complexity Measures
2.4 Randomized Self-Stabilization
2.5 Example: Spanning-Tree Construction
2.6 Example: Mutual Exclusion
2.7 Fair Composition of Self-Stabilization Algorithms
2.8 Recomputation of Floating Output
2.9 Proof Techniques
2.10 Pseudo-Self-Stabilization

Spanning-Tree Construction



The root writes 0 to all it's neighbors

The rest - each processor chooses the minimal distance of it's neighbors, adds one and updates it's neighbors

Spanning-Tree Algorithm for Pi

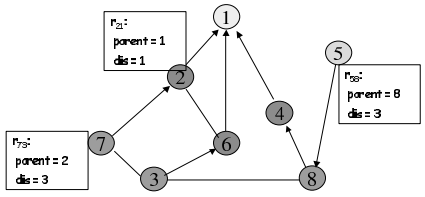
```
01 Root: do forever
02          for m := 1 to δ do write r_im := ⟨0,0⟩
03      od                                  δ = # of processor's neighbors
04 Other: do forever
05          for m := 1 to δ do write lr_mi := read(r_mi)
06          FirstFound := false
07          dist := 1 + min{lr_mi.dis | 1 ≤ m ≤ δ }
08          for m := 1 to δ                  lr_ji the last value of r_ji
09          do                               read by P_i
10                if not FirstFound and lr_mi.dis = dist -1
11                      write r_im := ⟨dist,1⟩
12                      FirstFound := true
13                else
14                      write r_im := ⟨0,dist⟩
15          od
16      od
```


Spanning-Tree, System and code

- Demonstrates the use of our definitions and requirements
- The system –
  - We will use the shared memory model for this example
  - The system consists n processors
  - A processor P_i communicates with its neighbor P_j by writing in the communication register r_{ij} and reading from r_{ji}

Spanning-Tree, System and code

- The output tree is encoded by means of the registers



Spanning-Tree Application

RUN

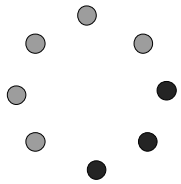
Spanning-Tree, is Self-Stabilizing

- The legal task ST - every configuration encodes a BFS tree of the communication graph
- Definitions :
  - A floating distance in configuration c is a value in $r_{ij}.dis$ that is smaller than the distance of P_i from the root
  - The smallest floating distance in configuration c is the smallest value among the floating distance
  - Δ is the maximum number of links adjacent to a processor

Spanning-Tree, is Self-Stabilizing

- For every k > 0 and for every configuration that follows $\Delta + 4k\Delta$ rounds, it holds that: (Lemma 2.1)
  - If there exists a floating distance, then the value of the smallest floating distance is at least k
  - The value in the registers of every processor that is within distance k from the root is equal to its distance from the root
- Note that once a value in the register of every processor is equal to it's distance from the root, a processor Pi chooses its parent to be the parent in the first BFS tree, this implies that :
- The algorithm presented is Self-Stabilizing for ST

Mutual Exclusion



The root changes it's state if equal to it's neighbor

The rest – each processor copies it's neighbor's state if it is different

Dijkstra's Algorithm

01 P_1: **do** forever

02 **if** $x_1=x_n$ **then**

03 $x_1:=(x_1+1) \bmod (n+1)$

04 $P_i (i \neq 1)$: **do** forever

05 **if** $x_i \neq x_{i-1}$ **then**

06 $x_i:=x_{i-1}$

Mutual-Exclusion Application

RUN

Dijsktra's alg. is Self-Stabilizing

- A configuration of the system is a vector of n integer values (the processors in the system)
- The task ME :
  - exactly one processor can change its state in any configuration,
  - every processor can change its state in infinitely many configurations in every sequence in ME
- A safe configuration in ME and Dijkstra's algorithm is a configuration in which all the x variables have the same value

Dijsktra's alg. is Self-Stabilizing

- A configuration in which all x variables are equal, is a safe configuration for ME (Lemma 2.2)
- For every configuration there exists at least one integer j such that for every i $x_i \neq j$ (Lemma 2.3)
- For every configuration c, in every fair execution that starts in c, P_1 changes the value of x_1 at least once in every n rounds (Lemma 2.4)
- For every possible configuration c, every fair execution that starts in c reaches a safe configuration with relation to ME within $O(n^2)$ rounds (Theorem 2.1)

Fair Composition -Some definitions

- The idea – composing self-stabilizing algorithms $AL_1, \ldots, AL_k$ so that the stabilized behavior of $AL_1, AL_2, \ldots, AL_i$ is used by AL_{i+1}
- AL_{i+1} cannot detect whether the algorithms have stabilized, but it is executed as if they have done so

The technique is described for k=2 :

- Two simple algorithms server & client are combined to obtain a more complex algorithm
- The server algorithm ensures that some properties will hold to be used by the client algorithm

Fair Composition -more definitions

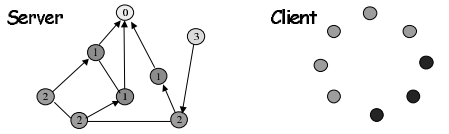
- Assume the server algorithm AL_1 is for a task defined by a set of legal execution T_1, and the client algorithm AL_2 for T_2
- Let A_i be the state set of P_i in AL_1 and $S_i = A_i \times B_i$ the state set of P_i in AL_2 ,where whenever P_i executes AL_2 , it modifies only the B_i components of $A_i \times B_i$
- For a configuration $c \in S_1 \times \ldots \times S_n$, define the A-projection of c as the configuration $(ap_1, \ldots, ap_n) \in A_1 \times \ldots \times A_n$
- The A-projection of an execution - consist of the A-projection of every configuration of the execution

Fair Composition -more definitions ...

- AL_2 is self-stabilizing for task T_2 given task T_1 if any fair execution of AL_2 that has an A-projection in T_1 has a suffix in T_2
- AL is a fair composition of AL_1 and AL_2 if, in AL, every processor execute steps of AL_1 and AL_2 alternately
- Assume AL_2 is self-stabilizing for task T_2 given task T_1. If AL_1 is self-stabilizing for T_1, then the fair composition of AL_1 and AL_2 is self-stabilizing for T_2 (Theorem 2.2)

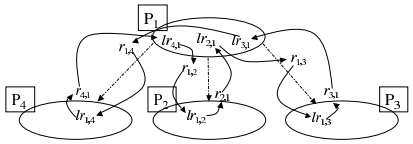
Example : Mutual exclusion for general communication graphs

- Will demonstrate the power of fair composition method
- What will we do ? Compose the spanning-tree construction algorithm with the mutual exclusion algorithm



Modified version of the mutual exclusion algorithm

- Designed to stabilize in a system in which a rooted spanning tree exists and in which only read/write atomicity is assumed
- Euler tour defines a virtual ring



Mutual exclusion for tree structure (for P_i)

```
01 Root: do forever
02                  lr1,i := read (r1,i)
03                  if lrδi = ri,1 then
04                              (* critical section*)
05                              write ri,2 := (lr1,i + 1 mod (4n -5))
06                  for m := 2 to δ do
07                              lrm,i = read(rm,i )
08                              write ri,m+1 := lrm,i
09                  od
10          od
11 Other: do forever
12                  lr1,i := read (r1,i)
13                  if lr1,i ≠ ri,2
14                              (* critical section*)
15                              write ri,2 := lr1,i
16                  for m := 2 to δ do
17                                          lrm,i := read (rm,i)
18                                          write ri,m+1 := lrm,i
19                  od
20          od
```

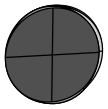

Mutual exclusion for communication graphs

- Mutual-exclusion can be applied to a spanning tree of the system using the ring defined by the Euler tour
- Note - a parent field in the spanning-tree defines the parent of each processor in the tree
- When the server reaches the stabilizing state, the mutual-exclusion algorithm is in an arbitrary state, from there converges to reach the safe configuration

Chapter 2: roadmap

2.1 Definitions of Computational Model
2.2 Self-Stabilization Requirements
2.3 Complexity Measures
2.4 Randomized Self-Stabilization
2.5 Example: Spanning-Tree Construction
2.6 Example: Mutual Exclusion
2.7 Fair Composition of Self-Stabilization Algorithms
2.8 Recomputation of Floating Output
2.9 Proof Techniques
2.10 Pseudo-Self-Stabilization

Recomputation of Floating Output -The Idea



- floating output variables

What do we need to prove ?

- From every possible configuration, an execution of *AL* will eventually begin from a predefined initial state
- Any two executions of *AL* that begin from the initial state have identical output

Recomputation of Floating Output Example: Synchronous Consensus

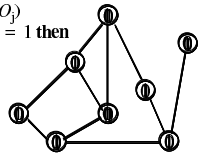
- Demonstrates the core idea behind the recomputation of the floating output
- Assumptions
  - We will ignore synchronization issues
  - We will assume that the system is a message-passing synchronous system
- Define –
  The synchronous consensus task - a set of (synchronous) executions SC in which the output variable of every processor is 1 iff there exists an input value with value 1

Non-stabilizing synchronous consensus algorithm

This algorithm is not a self stabilizing algorithm

```
01 initialization
02      pulse := 0
03      O_i = I_i
04 while pulse_i ≤ D do
05      upon a pulse
06          pulse_i := pulse_i + 1
07          send (O_i)
08          forall P_j ∈ N(i) do receive (O_j)
09          if O_i = 0 and ∃ P_j ∈ N(i) | O_j = 1 then
10              O_i := 1
```

$\forall i : I_i = 0 ;$



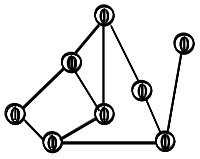
Self-stabilizing synchronous consensus algorithm

```
01 upon a pulse
02      if (pulse_i mod (D +1)) = 0 then
03          begin
04              O_i = o_i (*floating output is assigned by the result*)
05              o_i = I_i (*initializing a new computation*)
06          end
07      send (⟨o_i , pulse_i⟩)
08      forall P_j ∈ N(i) do receive (⟨o_j , pulse_j⟩)
09      if o_i = 0 and ∃ P_j ∈ N(i) | o_j = 1 then
10          o_i := 1
11      pulse_i := max {{pulse_i} ∪ {pulse_j | P_j ∈ N(i)}} +1
```


Self-stabilizing synchronous consensus algorithm

- The output variable O_i of every processor P_i is used as a floating output, which is eventuallly fixed and correct

$\forall i : I_i = 0 ;$



Chapter 2: roadmap

2.1 Definitions of Computational Model
2.2 Self-Stabilization Requirements
2.3 Complexity Measures
2.4 Randomized Self-Stabilization
2.5 Example: Spanning-Tree Construction
2.6 Example: Mutual Exclusion
2.7 Fair Composition of Self-Stabilization Algorithms
2.8 Recomputation of Floating Output
2.9 Proof Techniques
2.10 Pseudo-Self-Stabilization

Variant Function

$c \xrightarrow{step} c^1 \xrightarrow{step} c^2 \xrightarrow{step} c^3 \xrightarrow{steps} c^i_{safe}$

$|VF(c)| \geq |VF(c^1)| \geq |VF(c^2)| \geq |VF(c^3)| \geq ... \geq |VF(c^i)| \geq ... \geq bound$

- Used for proving convergence
- Can be used to estimate the number of steps required to reach a safe configuration

Variant Function - Example: self stabilizing Maximal Matching

Every processor P_i tries to find a matching neighbor P_j

Program for P_i :

```
01 do forever
02     if pointer_i = null and (∃ P_j ∈ N(i) | pointer_j = i ) then
03         pointer_i = j
04     if pointer_i = null and (∀ P_j ∈ N(i) | pointer_j ≠ i ) and
05                         (∃ P_j ∈ N(i) | pointer_j = null ) then
06         pointer_i = j
07     if pointer_i = j and pointer_j = k and k ≠ i then
08         pointer_i = null
09 od
```


Maximal Matching Application

RUN

Variant Function - Example: self stabilizing Maximal Matching

- The algorithm should reach a configuration in which $pointer_i = j$ implies that $pointer_j = i$
- We will assume the existence of a central daemon
- The set of legal executions MM for the maximal matching task includes every execution in which the values of the pointers of all the processors are fixed and form a maximal matching

Variant Function - Example: self stabilizing Maximal Matching- definitions

Program for P_i :

```
01 do forever
02     if pointer_i = null and (∃ P_j ∈ N(i) | pointer_j = i ) then
03         [ pointer_i = j ]   matched
04 free if pointer_i = null and (∀ P_j ∈ N(i) | pointer_j ≠ i ) and
05                         (∃ P_i ∈ N(i) | pointer_j = null ) then
06         [ pointer_i = j ]   waiting
07     if pointer_i = j and pointer_j = k and k ≠ i then   chaining
08         pointer_i = null
09 od
```

single

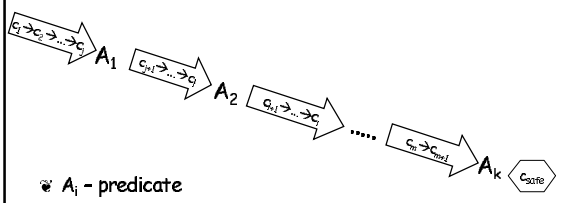
Variant Function - Example: self stabilizing Maximal Matching- proving correctness

- The variant function VF(c) returns a vector (m+s,w,f,c)
- Values of VF are compared lexicographically
- VF(c) = (n,0,0,0) ⇔ c is a safe configuration with relation to MM and to our algorithm
- Once a system reaches a safe configuration, no processor changes the value of its pointer

Variant Function - Example: self stabilizing Maximal Matching- proving correctness

- In every non-safe configuration, there exists at least one processor that can change the value of its pointer
- Every change of a pointer-value increases the value of VF
- The number of such pointer-value changes is bounded by the number of all possible vector values , $O(n^3)$

Convergence Stairs



- A_i - predicate
- for every $1 \le i < k$, A_{i+1} is a refinement of A_i

Convergence Stairs - Example: Leader election in a General Communication Network

Program for Pi, each processor reads it's neighbors leader and chooses the candidate with the lowest value :

```
01 do forever
02     ⟨candidate,distance⟩ = ⟨ ID(i), 0 ⟩
03     forall Pj ∈ N(i) do
04         begin
05             ⟨leader_i[j],dis_i[j]⟩ = ⟨ leader_j, dis_j ⟩
06             if (dis_i[j] < N) and ((leader_i[j] < candidate) or
07                ((leader_i[j] = candidate) and (dis_i[j] < distance))) then
08                     ⟨candidate,distance⟩ = ⟨ leader_i[j],dis_i[j] + 1⟩
09         end
10     write ⟨leader_i ,dis_i⟩ := ⟨candidate,distance⟩
11 od
```


Convergence Stairs - Example: Leader election in a General Communication Networks

- We assume that every processor has a unique identifier in the range 1 to N
- The leader election task is to inform every processor of the identifier of a single processor in the system, this single processor is the leader
- Floating identifier - an identifier that appears in the initial configuration, when no processor in the system with this identifier appears in the system

Convergence Stairs - Example: Leader election, proving correctness

- We will use 2 convergence stairs :
  - A_1 - no floating identifiers exists
  - A_2 (for a safe configuration) every processor chooses the minimal identifier of a processor in the system as the identifier of the leader
- To show that A_1 holds, we argue that, if a floating identifier exists, then during $O(\Delta)$ rounds, the minimal distance of a floating identifier increases

Convergence Stairs - Example: Leader election, proving correctness ...

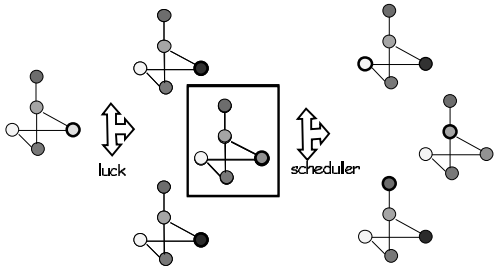
- Every fair execution that starts from any arbitrary configuration reaches the safe configuration
- Notice : if A_1 wasn't true, we couldn't prove the correctness

Scheduler-Luck Game

- Purpose : proving the upper bounds of the time complexity of randomized distributed algorithms by probability calculations
- Using the sl-game method . It tries to avoid considering every possible outcome of the randomized function used by the randomized algorithm

The sl-game

- Given a randomized algorithm AL, we define a game between 2 players, scheduler and luck



The sl-game(2)

- Luck's strategy is used to show that the algorithm stabilizes
- Some definitions :
  - cp = $\Pi_{i=1}^{f} p_i$
    f is # of lucks intervenes
    p_i - probability of the ith intervention
  - luck has a (cp,r)-strategy to win the game
- If luck has a (cp,r)-startegy,then AL reaches a safe configuration within, at most, r/cp expected number of rounds (Theorem 2.4)

SL-Game, Example: Self Stabilizing Leader election in Complete Graphs

The next algorithm works in complete graph systems, communication via shared memory

Each processor becomes the leader with probability 0.5 if it has the potential

Program for P_i:

```
01 do forever
02     forall P_j ∈ N(i) do
03         il_i[j] = read( leader_j )
04     if (leader_j = 0 and {∀ j ≠ i | l_i[j] = 0}) or
05         (leader_i = 1 and {∃ j ≠ i | l_i[j] = 1}) then
06             write leader_i := random({0,1})
07     end
```


Random Leader election in Complete Graphs

RUN

SL-Game, Example: Self Stabilizing Leader election in Complete Graphs

- Task LE - the set of executions in which there exists a single fixed leader throughout the execution
- A configuration is safe if it satisfies the following:
  - for exactly one processor, say P_i, $leader_i = 1$ and $\forall\, j \neq i$ $l_i[j] = 0$
  - for every other processor $P_j \neq P_i$ $leader_j = 0$ and $l_j[i] = 1$
- In any fair execution E that starts with a safe configuration, P_i is a single leader and thus $E \in LE$

SL-Game, Example: Self Stabilizing Leader election in Complete Graphs - proof

- The algorithm stabilizes within $2^{O(n)}$ expected number of rounds (Lemma 2.6)
  - using theorem 2.4 we will show that the number of round is expected to be $2n2^n$
  - we present an $(1/2^n, 2n)$-strategy for luck to win the sl-game
  - Luck's strategy is as follows: whenever some processor P_i tosses a coin, luck intervenes; if for all $j \neq i$, $leader_j = 0$, then luck fixes the coin toss to be 1; otherwise, it fixes the coin toss to be 0

SL-Game, Example: Self Stabilizing Leader election in Complete Graphs - proof...

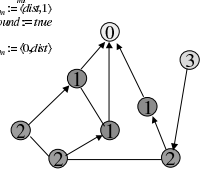
- The algorithm is not self-stabilizing under fine-atomicity (Lemma 2.7)
  By presenting a winning strategy for the scheduler that ensures that the system never stabilizes

Neighborhood Resemblance

- Purpose : proving memory lower bounds
- silent self-stabilizing algorithm - self-stabilizing algorithm in which the communication between the processors is fixed from some point of the execution
- The technique can be applied to silent self- stabilizing algorithms
- The idea : using a configuration c_0 (safe configuration) and construct a non-safe configuration c_1 in which every processor has the same neighborhood and therefore cannot distinguish c_0 from c_1

Reminder ... Spanning-Tree Algorithm for Pi

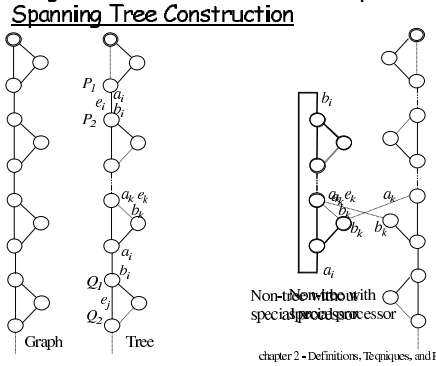
```
01 Root: do forever
02              for m := 1 to δ do write r_m := ⟨0,0⟩
03          od
04 Other: do forever
05              for m := 1 to δ do write lr_m := read(r_m)
06              FirstFound := false
07              dist := 1 + min{lr_m.dis | 1 ≤ m ≤ δ}
08              for m := 1 to δ
09              do
10                  if not FirstFound and lr_m.dis = dist - 1
11                      write r_m := ⟨dist,1⟩
12                      FirstFound := true
13                  else
14                      write r_m := ⟨0,dist⟩
15              od
16          od
```



Neighborhood Resemblance - Example: Spanning-Tree Construction

- We will show that implementing the distance field requires $\Omega(\log d)$ bits in every communication register, where d is the diameter of the system
- Note - the Spanning-Tree Construction is indeed a silent self-stabilizing algorithm

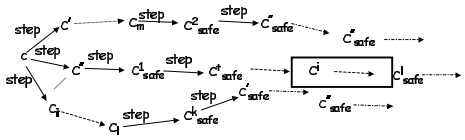
Neighborhood Resemblance, Example: Spanning Tree Construction



Chapter 2: roadmap

2.1 Definitions of Computational Model
2.2 Self-Stabilization Requirements
2.3 Complexity Measures
2.4 Randomized Self-Stabilization
2.5 Example: Spanning-Tree Construction
2.6 Example: Mutual Exclusion
2.7 Fair Composition of Self-Stabilization Algorithms
2.8 Recomputation of Floating Output
2.9 Proof Techniques
2.10 Pseudo-Self-Stabilization

What is Pseudo-Self-Stabilization ?



- The algorithm exhibits a legal behavior; but may deviate from this legal behavior a finite number of times

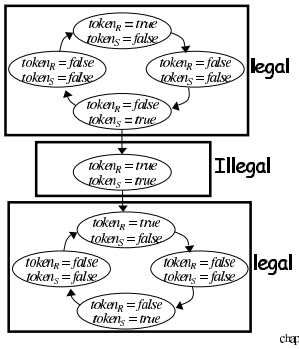
An Abstract Task

- An abstract task - variables and restrictions on their values
- The token passing abstract task AT for a system of 2 processors: Sender (S) and Receiver (R). S and R have boolean variable $token_S$ and $token_R$
- Given E = $(c_1,a_1,c_2,a_2,...)$ one may consider only the values of $token_S$ and $token_R$ in every configuration c_i to check whether the token-passing task is achieved

Pseudo-self-Stabilization

- Denote :
  - c_i|tkns the value of the boolean variables ($token_S$, $token_R$) in c_i
  - E|tkns as (c_1|tkns, c_2|tkns, c_3|tkns, ...)
- We can define AT by E|tkns as follows: there is no c_i|tkns for which $token_S$=$token_R$=true
- It is impossible to define a safe configuration in terms of c_i|tkns, since we ignore the state variables of R/S

Pseudo-Self-Stabilization



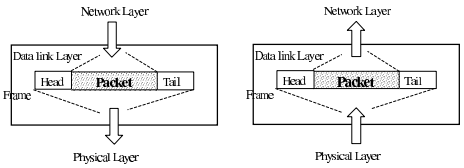
- The system is not in a safe configuration, and there is no time bound for reaching a safe configuration
- Only after a message is lost does the system reach the safe configuration

Pseudo-Self-Stabilization, The Alternating Bit Algorithm

- A data link algorithm used for message transfer over a communication link
- Messages can be lost, since the common communication link is unreliable
- The algorithm uses retransmission of messages to cope with message loss
- frame distinguishs the higher level messages, from the messages that are actually sent (between S and R)

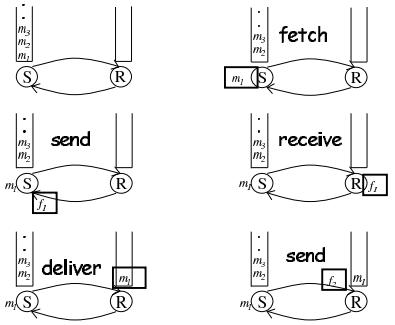
Pseudo-Self-Stabilization, The Data Link Algorithm

- The task of delivering a message is sophisticated, and may cause message corruption or even loss
- The layers involved



Pseudo-Self-Stabilization - The Data Link Algorithm

The flow of a message:



Pseudo-Self-Stabilization, Back to The Alternating Bit Algorithm

- **The abstract task of the algorithm:**
  S has an infinite queue of input messages (im_1,im_2,...) that should be transferred to the receiver in the same order without duplications, reordering or omissions.
  R has an output queue of messages (om_1,om_2,...).
  The sequence of messages in the output queue should always be the prefix of the sequence of messages in the input queue

The alternating bit algorithm - Sender

```
01   initialization
02   begin
03      i := 1
04      bit_s := 0
05      send(⟨bit_s,im_i⟩) (*im_i is fetched*)
06   end (*end initialization*)
07   upon a timeout
08            send(⟨bit_s,im_i⟩)
09   upon frame arrival
10   begin
11            receive(FrameBit)
12            if FrameBit = bit_s then (*acknowledge arrives*)
13               begin
14                     bit_s := (bit_s + 1) mod 2
15                     i := i + 1
16               end
17               send(⟨bit_s,im_i⟩) (*im_i is fetched*)
18   end
```


The alternating bit algorithm - Receiver

```
01   initialization
02   begin
03      j := 1
04      bit_r := 1
05   end (*end initialization*)
06   upon frame arrival
07   begin
08            receive(⟨FrameBit , msg⟩)
09            if FrameBit ≠ bit_r then (*a new message arrived*)
10               begin
11                     bit_r := FrameBit
12                     j := j + 1
13                     om_j := msg (*om_j is delivered*)
14               end
15               send(bit_r)
16   end
```

Pseudo-Self-Stabilization, The Alternating Bit Algorithm

- Denote L = $bit_s, q_{s,r}, bit_r, q_{r,s}$, the value of the of this label sequence is in [0*