6.852: Distributed Algorithms
Fall, 2009

Class 24
Today’s plan

• Self-stabilization
• Self-stabilizing algorithms:
  – Breadth-first spanning tree
  – Mutual exclusion
• Composing self-stabilizing algorithms
• Making non-self-stabilizing algorithms self-stabilizing
• Reading:
  – [Dolev, Chapter 2]
• Next time:
  – Partially synchronous distributed algorithms
  – Clock synchronization
  – Reading:
    • Chapters 23-25
    • [Attiya, Welch] Section 6.3, Chapter 13
Self-stabilization

• A useful fault-tolerance property for distributed algorithms.
• Algorithm can start in any state---arbitrarily corrupted.
• From there, if it runs normally (usually, without any further failures), it eventually gravitates back to correct behavior.

• [Dijkstra 73: Self-Stabilizing Systems in Spite of Distributed Control]
  – Dijkstra’s most important contribution to distributed computing theory.
  – [Lamport talk, PODC 83] Reintroduced the paper, explained its importance, popularized it.
  – Became (still is) a major research direction.
  – Award renamed the Dijkstra Prize.

• [Dolev book, 00] summarizes main ideas of the field.
Today…

• Basic ideas, from [Dolev, Chapter 2]

• Rest of the book describes:
  – Many more self-stabilizing algorithms.
  – General techniques for designing them.
  – Converting non-SS algorithms to SS algorithms.
  – Transformations between models, preserving SS.
  – SS in presence of ongoing failures.
  – Efficient SS.
  – Etc.
Self-Stabilization: Definitions
Self-stabilization

• [Dolev] considers:
  – Message-passing models, with FIFO reliable channels.
  – Shared-memory models, with read/write registers.
  – Asynchronous and synchronous models.

• To simplify, avoids internal process actions---combines these with sends, receives, or register access steps.

• Sometimes considers message losses ("loss" steps).

• Many models, must continually specify which is used.

• Defines executions:
  – Like ours, but needn’t start in initial state.
  – Same as our "execution fragments".

• Fair executions:
  – Described informally.
  – Our task-based definition is fine.
Legal execution fragments

- Given a distributed algorithm A, define a set $L$ of legal execution fragments of A.
- $L$ can include both safety and liveness conditions.
- **Example:** Mutual exclusion problem
  - $L$ might be the set of all fragments $\alpha$ satisfying:
    - Mutual exclusion:
      - No two processes are in the critical region, in any state in $\alpha$.
    - Progress:
      - If in some state of $\alpha$, someone is in T and no one is in C, then sometime thereafter, someone $\rightarrow$ C.
      - If in some state of $\alpha$, someone is in E, then sometime thereafter, someone $\rightarrow$ R.
Self-stabilization: Definition

- A global state $s$ of algorithm $A$ is safe with respect to legal set $L$, provided that every fair execution fragment of $A$ that starts with $s$ is in $L$.
- Algorithm $A$ is self-stabilizing for legal set $L$ if every fair execution fragment $\alpha$ of $A$ contains a state $s$ that is safe with respect to $L$.
  - Implies that the suffix of $\alpha$ starting with $s$ is in $L$.
  - Also, any other fair execution fragment starting with $s$ is in $L$.

- Weaker definition: Algorithm $A$ is self-stabilizing for legal set $L$ if every fair execution fragment $\alpha$ has a suffix in $L$. 

\[
\begin{align*}
\text{\alpha} & \quad \text{s} \\
\text{In L}
\end{align*}
\]
Stronger vs. weaker definition of self-stabilization

- **Stronger definition:** Algorithm $A$ is self-stabilizing for legal set $L$ if every fair execution fragment of $A$ contains a state $s$ that is safe with respect to $L$.
- **Weaker definition:** Algorithm $A$ is self-stabilizing for legal set $L$ if every fair execution fragment has a suffix in $L$.

- [Dolev] generally uses the stronger definition; so will we.
- But occasionally, he appears to be using the weaker definition; we’ll warn when this arises.

- **Q:** Equivalent definitions? Not in general. LTTR.
Non-termination

• Self-stabilizing algorithms for nontrivial problems don’t terminate.

• E.g., consider message-passing algorithm A:
  – Suppose A is self-stabilizing for legal set L, and A has a terminating global state s.
    • All processes quiescent, all channels empty.
  – Consider a fair execution fragment $\alpha$ starting with s.
  – $\alpha$ contains no steps---just global state s.
  – Since A is self-stabilizing with respect to L, $\alpha$ must contain a safe state.
  – So s must be a safe state.
  – Then the suffix of $\alpha$ starting with s is in L; that is, just s itself is in L.
  – So L represents a trivial problem---doing nothing satisfies it.

• Similar argument for shared-memory algorithms.
Self-Stabilizing Algorithm 1: Self-Stabilizing Breadth-First Spanning Tree Construction
Breadth-first spanning tree

- Shared-memory model
- Connected, undirected graph $G = (V,E)$.
- Processes $P_1, \ldots, P_n$, $P_1$ a designated root.
- Permanent knowledge (built into all states of the processes):
  - $P_1$ always knows it’s the root.
  - Everyone always knows who their neighbors are.

- Neighboring processes in $G$ share registers in both directions:
  - $r_{ij}$ written by $P_i$, read by $P_j$.

- **Output:** A breadth-first spanning tree, recorded in the $r_{ij}$ registers:
  - $r_{ij}.\text{parent} = 1$ if $j$ is $i$’s parent, 0 otherwise.
  - $r_{ij}.\text{dist} = \text{distance from root to } i \text{ in the BFS tree} = \text{smallest number of hops on any path from 1 to } i \text{ in } G$.
  - Values in registers should remain constant from some point onward.
In terms of legal sets...

- Define execution fragment $\alpha$ to be legal if:
  - The registers have correct BFS output values, in all states in $\alpha$.
  - Registers never change.

- $L = $ set of legal execution fragments.

- Safe state $s$:
  - Global state from which all extensions have registers with correct, unchanging BFS output values.

- SS definition says:
  - Any fair execution fragment $\alpha$, starting from any state, contains some safe state $s$.
  - That is, one from which all extensions have registers with correct, unchanging BFS output values.
  - Implies that any fair execution fragment $\alpha$ has a suffix in which the register contents represent a fixed BFS tree.
BFS Algorithm strategy

• The system can start in any state, with
  – Any values (of the allowed types) in registers,
  – Any values in local process variables.
• Processes can’t assume that their own states and output registers are initially correct.

• Repeatedly recalculate states and outputs based on inputs from neighbors.
• In case of tie, use some default rule for selecting parent.

• Prove correctness, stabilization time, using induction on distance from root.
Root process $P_1$

do forever
  for every neighbor $m$ do
    write $r_{1m} := (0,0)$

- Keep writing $(0,0)$ everywhere.
- Access registers in fixed, round-robin order.
Non-root process $P_i$

• Maintains local variables $lr_{ji}$ to hold latest observed values of incoming registers $r_{ji}$.

• First loop:
  – Read all the $r_{ji}$, copy them into $lr_{ji}$.

• Use this local info to calculate new best distance $\text{dist}$, choose a parent that yields this distance.
  – Use default rule, e.g., smallest index, so always break ties the same way.
  – Needed to ensure stabilization to a fixed tree.

• Second loop:
  – Write $\text{dist}$ to all outgoing registers.
  – Notify new parent.
Non-root process $P_i$

- do forever
  - for every neighbor $m$ do
    - $lr_{mi} := \text{read}(r_{mi})$
    - $\text{dist} := \min\{lr_{mi}.\text{dist}\} + 1$
    - $\text{found} := \text{false}$
    - for every neighbor $m$ do
      - if not $\text{found}$ and $\text{dist} = lr_{mi}.\text{dist} + 1$ then
        - write $r_{im} := (1,\text{dist})$
        - $\text{found} := \text{true}$
      - else
        - write $r_{im} := (0,\text{dist})$

- Note:
  - $P_i$ doesn’t take min of its own $\text{dist}$ and neighbors’ $\text{dists}$.
  - Unlike non-SS relaxation algorithms.
  - Ignores its own $\text{dist}$, recalculates solely from neighbors’ $\text{dists}$.
  - Because its own value could be erroneous.
Correctness

- Prove this stabilizes to a particular “default” BFS tree.
- Define the default tree to be the unique BFS tree where ties in choosing parent are resolved using the rule:
  - Choose the smallest index yielding the shortest distance.
- Prove that, from any starting global state, the algorithm eventually reaches and retains the default BFS tree.
- More precisely, show it reaches a safe state, from which any execution fragment retains the default BFS tree.

- Show this happens within bounded time: \( O(\text{diam} \ \Delta \ l) \), where
  - \( \text{diam} \) is diameter of \( G \) (max distance from \( P_1 \) to anyone is enough).
  - \( \Delta \) is maximum node degree
  - \( l \) is upper bound on local step time
  - The constant in the big-O is about 4.
Correctness

• Uses a lemma marking progress through distances 0, 1, 2,..., diam, as for basic AsynchBFS.

• **New complication:** Erroneous, too-small distance estimates.

• Define a **floating distance** in a global state to be a value of some $r_{ij}\cdot\text{dist}$ that is strictly less than the actual distance from $P_1$ to $P_i$.
  – Can’t be correct.

• **Lemma:** For every $k \geq 0$, within time $(4k+1)\Delta t$, we reach a configuration such that:
  1. For any $i$ with $\text{dist}(P_1,P_i) \leq k$, every $r_{ij}\cdot\text{dist}$ is correct.
  2. There is no floating distance $< k$.

• Moreover, these properties persist after this configuration.
Proof of lemma

• Lemma: For every \( k \geq 0 \), within time \((4k+1)\Delta l\), we reach a configuration such that:
  1. For any \( i \) with \( \text{dist}(P_1, P_i) \leq k \), every \( r_{ij}.\text{dist} \) is correct.
  2. There is no floating distance \(< k \).

• Proof: Induction on \( k \).
  – \( k = 0 \): \( P_1 \) writes \((0,0)\) everywhere within time \( \Delta l \).
  – Assume for \( k \), prove for \( k+1 \):
    • Property 1:
      – Consider \( P_i \) at distance \( k+1 \) from \( P_1 \).
      – In one more interval of length \( 4\Delta l \), \( P_i \) has a chance to update its local dist and outgoing register values.
      – By inductive hypothesis, these updates are based entirely on:
        » Correct distance values from nodes with distance \( \leq k \) from \( P_1 \), and
        » Possibly some floating values, but these must be \( \geq k \).
      – So \( P_i \) will calculate a correct distance value.
    • Property 2:
      – For anyone to calculate a floating distance \(< k+1 \), it must see a floating distance \(< k \).
      – Can’t, by inductive hypothesis.
Proof, cont’d

• We have proved:
  – **Lemma:** For every k \( \geq 0 \), within time \((4k+1)\Delta l\), we reach a configuration such that:
    1. For any \( i \) with \( \text{dist}(P_1,P_i) \leq k \), every \( r_{ij}.\text{dist} \) is correct.
    2. There is no floating distance \(< k\).

• So within time \((4 \text{ diam } +1)\Delta l\), all the \( r_{ij}.\text{dist} \) values become correct.

• Persistence is easy to show.

• Once all the \( r_{ij}.\text{dist} \) values are correct, everyone will use the default rule and always obtain the default BFS tree.

• Ongoing failures:
  – If arbitrary failures occur from time to time, not too frequently, the algorithm gravitates back to correct behavior in between failures.
  – Recovery time depends on size (diameter) of the network.
Self-Stabilizing Algorithm 2: Self-Stabilizing Mutual Exclusion
Self-stabilizing mutual exclusion

- [Dijkstra 73]
- Ring of processes, each with output variable $x_i$.
- Large granularity: In one atomic step, process $P_i$ can read both neighbors’ variables, compute its next value, and write it to variable $x_i$.

$P_1$:
  do forever:
    if $x_1 = x_n$ then $x_1 := x_1 + 1 \mod (n+1)$

$P_i, i \neq 1$:
  do forever:
    if $x_i \neq x_{i-1}$ then $x_i := x_{i-1}$

- $P_1$ tries to make its variable one more than its predecessor’s (mod $n+1$).
- Each other process tries to make its variable equal to its predecessor’s

That’s $(n+1)$, not $n$. 
Mutual exclusion

• In what sense does this “solve mutual exclusion”?
• Definition: “Pi is enabled” (or “Pi can change its state”) in a configuration, if the variables are set so Pi can take a step and change the value of its variable xi.

• Legal execution fragment α:
  – In any state in α, exactly one process is enabled.
  – For each i, α contains infinitely many states in which Pi is enabled.

• Use this to solve mutual exclusion:
  – Say Pi interacts with requesting user Ui.
  – Pi grants Ui the critical section when:
    • Ui has requested it, and
    • Pi is enabled.
  – When Ui returns the resource, Pi actually does its step, changing xi.
  – Guarantees mutual exclusion, progress.
  – Also lockout-freedom.
Lemma 1

- **Legal $\alpha$:**
  - In any state in $\alpha$, exactly one process is enabled.
  - For each $i$, $\alpha$ contains infinitely many states in which $P_i$ is enabled.

- **Lemma 1:** A configuration in which all the $x$ variables have the same value is safe.
- This means that, from such a configuration, any fair execution fragment is legal.
- **Proof:** Only $P_1$ can change its state, then $P_2$, then $P_3$, ..., and so on around the ring (forever).

- Remains to show: Starting from any state, the algorithm eventually reaches a configuration in which all the $x$ values are the same.
- This uses some more lemmas.
Lemma 2

- **Lemma 2:** In every configuration, at least one of the potential x values, \{0,\ldots,n\}, does not appear in any \(x_i\).
- **Proof:** Obviously. There are \(n+1\) values and only \(n\) variables.
Lemma 3

- **Lemma 3**: In any fair execution fragment (from any configuration c), $P_1$ changes $x_1$ at least once every $nl$ time.

- **Proof**:
  - Assume not---$P_1$ goes longer than $nl$ without changing $x_1$ from some value $v$.
  - Then by time $l$, $P_2$ sets $x_2$ to $v$,
  - By time $2l$, $P_3$ sets $x_3$ to $v$,
  - ... 
  - By $(n-1)l$, $P_n$ sets $x_n$ to $v$.
  - All these values remain $= v$, as long as $x_1$ doesn’t change.
  - But then by time $nl$, $P_1$ sees $x_n = x_1 = v$, and increments $x_1$. 
Lemma 4

• **Lemma 4:** In any fair execution fragment $\alpha$, a configuration in which all the $x$ values are the same (and so, a safe configuration) occurs within time $(n^2 + n)l$.

• **Proof:**
  – Let $c =$ initial configuration of $\alpha$.
  – Let $v =$ some value that doesn’t appear in any $x_i$, in $c$.
  – Then $v$ doesn’t appear anywhere, in $\alpha$, unless/until $P_1$ sets $x_1 := v$.
  – Within time $nl$, $P_1$ changes $x_1$, incrementing it by 1, mod $(n+1)$.
  – Within another $nl$, $P_1$ increments $x_1$ again.
  – …
  – Within $n^2l$, $P_1$ increments $x_1$ to $v$.
  – At that point, there are still no other $v$’s anywhere else.
  – Then this $v$ propagates all the way around the ring.
  – $P_1$ doesn’t change $x_1$ until $v$ reaches $x_n$.
  – Yields all $x_i = v$, within time $(n^2 + n)l$. 

Putting the pieces together

• Legal execution fragment $\alpha$:
  – In any state in $\alpha$, exactly one process is enabled.
  – For each $i$, $\alpha$ contains infinitely many states in which $P_i$ is enabled.

• $L = \text{set of legal fragments}$.

• Theorem: Dijkstra’s algorithm is self-stabilizing with respect to legal set $L$.
  • In the sense of reaching a safe state.

• Remark:
  – This uses $n+1$ values for the $x_i$ variables.
  – A curiosity:
    • This also works with $n$ values, or even $n-1$.
    • But not with $n-2$ [Dolev, p. 20].
Reducing the atomicity

- Dijkstra’s algorithm reads $x_{i-1}$, computes, and writes $x_i$, all atomically.
- Now adapt this for usual model, in which only individual read/write steps are atomic.

- Consider Dijkstra’s algorithm on a 2n-process ring, with processes $Q_j$, variables $y_j$, $j = 1, 2, \ldots, 2n$.
  - Needs $2n+1$ values for the variables.

- Emulate this in the usual n-process ring, with processes $P_i$, variables $x_i$:
  - $P_i$ emulates both $Q_{2i-1}$ and $Q_{2i}$.
  - $y_{2i-1}$ is a local variable of $P_i$.
  - $y_{2i}$ corresponds to $x_i$. 
Reducing the atomicity

- Consider Dijkstra’s algorithm on a 2n-process ring, with processes $Q_j$, variables $y_j$. $j = 1, 2, \ldots, 2n$.
- Emulate this in an n-process ring, with processes $P_i$, variables $x_i$.
  - $P_i$ emulates both $Q_{2i-1}$ and $Q_{2i}$.
  - $y_{2i-1}$ is a local variable of $P_i$.
  - $y_{2i}$ corresponds to $x_i$.

To emulate a step of $Q_{2i-1}$, $P_i$ reads from $x_{i-1}$, writes to its local variable $y_{2i-1}$.
To emulate a step of $Q_{2i}$, $P_i$ reads from its local variable $y_{2i-1}$, writes to $x_i$.
Since in each case one variable is internal, can emulate each step with just one ordinary read or write to shared memory.
Composing Self-Stabilizing Algorithms
Composing self-stabilizing algorithms

• Consider several algorithms, where
  – $A_1$ is self-stabilizing for legal set $L_1$,
  – $A_2$ is SS for legal set $L_2$, “assuming $A_1$ stabilizes for $L_1$”
  – $A_3$ is SS for legal set $L_3$, “assuming $A_1$ stabilizes for $L_1$ and $A_2$ stabilizes for $L_2$”
  – etc.

• Then we should be able to run all the algorithms together, and the combination should be self-stabilizing for $L_1 \cap L_2 \cap L_3 \cap \ldots$

• Need composition theorems.
• Details depend on which model we consider.
• E.g., consider two shared memory algorithms, $A_1$ and $A_2$. 
Composing SS algorithms

• Consider read/write shared memory algorithms, $A_1$ and $A_2$, where:
  – All of $A_1$’s shared registers are written only by $A_1$ processes.
    • No inputs arrive in $A_1$’s registers.
  – All of $A_2$’s shared registers are written only by $A_1$ and $A_2$ processes.
    • No other inputs arrive in $A_2$’s registers.
  – Registers shared between $A_1$ and $A_2$ are written only by $A_1$ processes, not by $A_2$ processes.
  – One-way information flow, from $A_1$ and $A_2$.
  – $A_1$ makes sense in isolation, but $A_2$ depends on $A_1$ for some inputs.

• Definition: $A_2$ is self-stabilizing for $L_2$ with respect to $A_1$ and $L_1$ provided that: If $\alpha$ is any fair execution fragment of the combination of $A_1$ and $A_2$ whose projection on $A_1$ is in $L_1$, then $\alpha$ has a suffix in $L_2$.

• Theorem: If $A_1$ is SS for $L_1$ and $A_2$ is SS for $L_2$ with respect to $A_1$ and $L_1$, then the combination of $A_1$ and $A_2$ is SS for $L_2$. 
Weaker definition of SS

- At this point, [Dolev] seems to be using the weaker definition for self-stabilization:
- Instead of:
  - Algorithm A is self-stabilizing for legal set L if every fair execution fragment $\alpha$ of A contains a state s that is safe with respect to L.
- Now using:
  - Algorithm A is self-stabilizing for legal set L if every fair execution fragment $\alpha$ has a suffix in L.
- So we’ll switch here.
Composing SS algorithms

- **Def:** A₂ is self-stabilizing for L₂ with respect to A₁ and L₁ provided that any fair execution fragment of the combination of A₁ and A₂ whose projection on A₁ is in L₁, has a suffix in L₂.

- **Theorem:** If A₁ is SS for L₁ and A₂ is SS for L₂ with respect to A₁ and L₁, then the combination of A₁ and A₂ is SS for L₂.

- **Proof:**
  - Let α be any fair exec fragment of the combination of A₁ and A₂.
  - We must show that α has a suffix in L₂ (weaker definition of SS).
  - Projection of α on A₁ is a fair execution fragment of A₁.
  - Since A₁ is SS for L₁, this projection has a suffix in L₁.
  - Therefore, α has a suffix α’ whose projection on A₁ is in L₁.
  - Since A₂ is self-stabilizing with respect to A₁, α’ has a suffix α” in L₂.
  - So α has a suffix in L₂, as needed.

- **Total stabilization time is the sum of the stabilization times of A₁ and A₂.**
Applying the composition theorem

- Theorem supports modular construction of SS algorithms.

- Example: SS mutual exclusion in an arbitrary rooted undirected graph
  - $A_1$:
    - Constructs rooted spanning tree, using the SS BFS algorithm.
    - The $r_{ij}$ registers contain all the tree info (parent and distance).
  - $A_2$:
    - Takes $A_1$’s $r_{ij}$ registers as input.
    - Solves mutual exclusion using a Dijkstra-like algorithm, which runs on the stable tree in the $r_{ij}$ registers.
  - Q: But Dijkstra’s algorithm uses a ring---how can we run it on a tree?
  - A: Thread the ring through the nodes of the tree, e.g.:
Mutual exclusion in a rooted tree

• Use the read/write version of the Dijkstra ring algorithm, with local and shared variables.

• Each process $P_i$ emulates several processes of Dijkstra algorithm.

• Bookkeeping needed, see [Dolev, p. 24-27].

• Initially, both the tree and the mutex algorithm behave badly.

• After a while ($O(\text{diam} \Delta I)$ time), the tree stabilizes (since the BFS algorithm is SS), but the mutex algorithm continues to behave badly.

• After another while ($O(n^2 I)$ time), the mutex algorithm also stabilizes (since it’s SS given that the tree is stable).

• Total time is the sum of the stabilization times of the two algorithms: $O(\text{diam} \Delta I) + O(n^2 I) = O(n^2 I)$. 
Self-Stabilizing Emulations
Self-stabilizing emulations
[Dolev, Chapter 4]

• Design a SS algorithm $A_2$ to solve a problem $L_2$, using a model that is more powerful than the “real” one.

• Design an algorithm $A_1$ using the real model, that “stabilizes to emulate” the powerful model

• Combine $A_1$ and $A_2$ to get a SS algorithm for $L_2$ using the real model.
Self-stabilizing emulations

- **Example 1 [Dolev, Section 4.1]:** Centralized scheduler
  - Rooted undirected graph of processes.
  - Powerful model: Process can read several variables, change state, write several variables, all atomically.
  - Basic model: Just read/write steps.
  - Emulation algorithm $A_1$:
    - Uses Dijkstra-style mutex algorithm over BFS spanning tree algorithm
    - Process performs steps of $A_2$ only when it has the critical section (global lock).
    - Performs all steps that are performed atomically in the powerful model, before exiting the critical section.
  - Stabilizes to emulate the more powerful model.
  - Initially, both emulation $A_1$ and algorithm $A_2$ behave badly.
  - After a while, emulation begins behaving correctly, yielding mutual exclusion.
  - After another while, $A_2$ stabilizes for $L_2$. 
Self-stabilizing emulations

• Example 2 [Nolte]: Virtual Node layer for mobile networks
  – Mobile ad hoc network: Collection of processes running on mobile nodes, communicating via local broadcast.
  – Powerful model: Also includes stationary Virtual Nodes at fixed geographical locations (e.g., grid points).
  – Basic model: Just the mobile nodes.
  – Emulation algorithm $A_1$:
    • Mobile nodes in the vicinity of a Virtual Node’s location cooperate to emulate the VN.
    • Uses Replicated State Machine strategy, coordinated by a leader.
  – Application algorithm $A_2$ running over the VN layer:
    • Geocast, or point-to-point routing, or motion coordination,…
  – Initially, both the emulation $A_1$ and the application algorithm $A_2$ behave badly.
  – Then the emulation begins behaving correctly, yielding a VN Layer.
  – Then the application stabilizes.
Making Non-Self-Stabilizing Algorithms Self-Stabilizing
Making non-self-stabilizing algorithms self-stabilizing

• [Dolev, Section 2.8]: Recomputation of floating outputs.
  – Method of converting some non-SS distributed algorithms to SS algorithms.

• What kinds of algorithms?
  – Algorithm A, computes a distributed function based on distributed inputs.
  – Assumes processes’ inputs are in special, individual input variables, \( I_i \),
    whose values never change (e.g., contain fixed information about local
    network topology).
  – Outputs placed in special, individual output variables \( O_i \).

• Main idea: Execute A repeatedly, from its initial state, with the fixed
  inputs, with two kinds of output variables:
  – Temporary output variables \( o_i \).
  – Floating output variables \( FO_i \).

• Use the temporary variables \( o_i \) the same way A uses \( O_i \).
• Write to the floating variables \( FO_i \) only at the end of function computation.
• When restarting A, reset all variables except the floating outputs \( FO_i \).
• Eventually, the floating outputs should stop changing.
Example: Consensus

- Start with a simple synchronous, non-fault-tolerant, non-self-stabilizing network consensus algorithm A, and make it self-stabilizing.
- Undirected graph $G = (V, E)$, known upper bound $D$ on diameter.
- **Non-SS consensus algorithm A:**
  - Everyone starts with Boolean input in $I_i$.
  - After $D$ rounds, everyone agrees, and decision value = 1 iff someone’s input = 1.
  - At intermediate rounds, process $i$ keeps current consensus proposal in $O_i$.
  - At each round, send $O_i$ to neighbors, resets $O_i$ to “or” of its current value and received values.
  - Stop after $D$ rounds.
- A works fine, in synchronous model, if it executes once, from initial states.
Example: Consensus

• To make this self-stabilizing:
  – Run algorithm A repeatedly, with the FO\textsubscript{i} as floating outputs.
  – While running A, use o\textsubscript{i} instead of O\textsubscript{i}.
  – Copy o\textsubscript{i} to FO\textsubscript{i} at the end of each execution of A.

• This is not quite right…
  – Assumes round numbers are synchronized.
  – Algorithm begins in an arbitrary global state, so round numbers can be off.
Example: Consensus

- Run algorithm A repeatedly, with the FO$_i$ as floating outputs.
- While running A, use o$_i$ instead of O$_i$.
- Copy o$_i$ to FO$_i$ at the end of each execution of A.

- Must also synchronize round numbers 1, 2, ..., D.
  - Needs a little subprotocol.
  - Each process, at each round, sets its round number to max of its own and all those of its neighbors.
  - When reach D, start over at 1.

- Eventually, rounds become synchronized throughout the network.

- Thereafter, the next full execution of A succeeds, produces correct outputs in the FO$_i$ variables.

- Thereafter, the FO$_i$ will never change.
Extensions

• Can make this into a fairly general transformation, for synchronous algorithms.

• Using synchronizers, can extend to some asynchronous algorithms.
Making non-SS algorithms SS: Monitoring and Resetting [Section 5.2]

- AKA Checking and Correction.
- Assumes message-passing model.
- Basic idea:
  - Continually monitor the consistency of the underlying algorithm.
  - Repair the algorithm when inconsistency is detected.
- For example:
  - Use SS leader election service to choose a leader (if there isn’t already a distinguished process).
  - Leader, repeatedly:
    - Conducts global snapshots,
    - Checks consistency,
    - Sends out corrections if necessary.
- Local monitoring and resetting [Varghese thesis, 92]
  - For some algorithms, can check and restore local consistency predicates.
  - E.g., BFS: Can check that local distance is one more than parent’s distance, recalculate dist and parent if not.
Other stuff in the book

• Discussion of practical motivations.
• Proof methods for showing SS.
• Stabilizing to an abstract specification.
• Model conversions, for SS algorithms:
  – Shared memory → message-passing
  – Synchronous → asynchronous
• SS in presence of ongoing failures.
  – Stopping, Byzantine, message loss.
• Efficient “local” SS algorithms.
• More examples.
Next time…

- Partially synchronous distributed algorithms
- Reading:
  - Chapters 23-25
  - [Attiya, Welch], Section 6.3, Chapter 13