SELF-STABILIZING ALGORITHMS

SATU VIRTANEN

satu.virtanen@hut.fi

T-79.192 Special Course in Theoretical Computer Science

Seminar presentation, December 4th, 2002

OUTLINE

1. Definitions and terminology
2. Properties and complexity measures
3. Some examples
4. Proof techniques
5. Pseudo-stabilization
6. Superstabilization

SELF-STABILIZATION

- a fault-tolerance concept
- automatic recovery after a (transient) fault
- may give wrong results for a finite time after initiation
- e.g. routing algorithms; crashes need to be tolerated in message delivery protocols but practically no “regular” algorithm holds up well
- introduced by Dijkstra in the 70s (e.g. [Dij74])
- brought forward by Lamport in the early 80s

DEFINITION OF SELF-STABILIZING ALGORITHMS

Consider a distributed system $\mathcal{S}$ described by a set of configurations $\mathcal{C}$ and a transition relation $\mathcal{T} : \mathcal{C} \to \mathcal{C}$.

A system $\mathcal{S} = (\mathcal{C}, \mathcal{T})$ *stabilizes* to a specification $\mathcal{P}$ if there exists a set $\mathcal{L} \subseteq \mathcal{C}$ of *legitimate* configurations such that

(i) Every execution of $\mathcal{S}$ starting in a configuration in $\mathcal{L}$ satisfies $\mathcal{P}$

(ii) Every execution contains a configuration of $\mathcal{L}$

(i) $\triangleq$ Correctness

(ii) $\triangleq$ Convergence

[Tel00]

Legitimate configurations

Starting from an arbitrary initial state, the system performs some steps that must eventually reach a legal configuration. Thereafter the execution stays within legal configurations until a fault occurs.

Example: Distributed mutual exclusion

Legal configurations are those in which only one processor at a time may use the critical resource.

Defining postconditions

Often the goal of the algorithm is to achieve a post-condition ψ.
$\Longrightarrow$ $\mathcal{P}$ = every configuration satisfies ψ

A system $\mathcal{S} = (\mathcal{C}, \mathcal{T})$ is said to stabilize to ψ if a set $\mathcal{L} \subseteq \mathcal{C}$ exists such that

(i) $\mathcal{L}$ is closed under $\mathcal{T}$ and $\gamma \in \mathcal{L} \Rightarrow \psi(\gamma)$

(ii) every execution reaches a configuration $\gamma \in \mathcal{L}$

[Tel00]

Example: Distributed mutual exclusion

In Dijkstra's algorithm for a ring, eventually only one node v_i at a time will be able to change its state variable x_i.

A special node executes forever

$$\texttt{if}\ (x_1 = x_n)\ :\ x_1 := (x_1 + 1) \mod (n + 1)$$

while the others execute

$$\texttt{if}\ (x_i \neq x_{i-1})\ :\ x_i := x_{i-1}$$

In a legal configuration, all x_i are equal. (Fair execution is assumed.) This reaches a legal configuration in $\mathcal{O}(n^2)$ rounds.

Properties of self-stabilizing algorithms

(i) Fault tolerance: full and automatic protection against transient process failures (i.e., data corruption)

(ii) No need for initialization: the system may start in an arbitrary configuration

(iii) Dynamic topology allowed: topology-dependent computation will converge into a new result after a topology change

[Tel00]

Disadvantages of self-stabilizing algorithms

(i) Initial inconsistency: the algorithm does not necessarily provide accurate results while starting or recovering from a fault/change.

(ii) Non-terminating computation: the processes have no way of detecting when to stop computing (as new faults may occur).

(iii) High complexity: usually the computational complexity is worse and also the performance inferior to rigid algorithms.

[Tel00]

Measuring complexity of distributed systems

Computational complexity is measured in asynchronous rounds of computation steps:

> The first **round** of an execution $\mathcal{E}$ is the shortest *prefix* $\mathcal{E}'$ of E in which each processor of the system $\mathcal{S}$ executes at least one step of its code.

As each processor usually has an infinite loop to execute, another proper measure would be the number of asynchronous cycles:

> The first **cycle** of an execution $\mathcal{E}$ is the shortest prefix $\mathcal{E}'$ in which each processor executes at least one complete iteration of its loop.

[Dol00]

Example: Constructing a spanning tree

- a distributed BFS variant for a graph $G = (V, E)$
- one node must be selected as the root [a]
- all nodes $v_i \in V$ know their neighborhood $\Gamma(v_i) \subseteq V$

[a] distributed leader selection is addressed later in this talk

Example: Constructing a spanning tree

- each node attempts to compute its distance from the root, reporting the outcome of each iteration to its neighbors
- communication is done by accessing registers of the neighboring nodes: v_i communicates with $v_j \in \Gamma(v_i)$ by writing to register *register*$_{ij}$ and reading from *register*$_{ji}$
- each register consists of two fields: *register*$_{ij}$ = $\langle$*dist*, *parent*$\rangle$
- the *parent* field of *register*$_{ij}$ is set to *true* if v_j is the parent of v_i in the BFS tree
- the *dist* field holds the estimated distance to the root v_0, a value bounded from above by $|V|$

Example: Constructing a spanning tree

The root node v_0 executes the following loop:

$\forall v_m \in \Gamma(v_0)$: write $register_{0m} := \langle 0, 0 \rangle$

Other nodes v_i, $i = 1, \ldots, n$ execute the following:

$\forall u_m \in \Gamma(v_i)$: $local_{mi} :=$ read ($register_{mi}$)
$found := false$
$dist := 1 + \min \{ local_{mi}.dist \mid 1 \le m \le \deg(v_i) \}$
$\forall u_m \in \Gamma(v_i)$:
  if ($\neg found \wedge (local_{mi}.dist = dist - 1)$)
    write $register_{im} := \langle true, dist \rangle$
    $found := true$
  else write $register_{im} := \langle false, dist \rangle$

Processor identifiers

In a uniform self-stabilizing algorithm, all processes are equal. Thus the underlying network may be anonymous.

Often some processor identities are necessary for a solution to exist (e.g. token ring for non-prime ring sizes). If there are no unique process identifiers available, randomization may be helpful [Dol00].

However, the identification data may also corrupt!
$\Longrightarrow$ Uniform algorithms are safer.

Example: Constructing a spanning tree

The BFS spanning tree can be read from the parent fields of the registers after stabilization. A register configuration is legal is it encodes a BFS tree.

Let $\Delta = \max \{ \deg(v_i) \mid v_i \in V \}$. For every $k > 0$ and any configuration that results in $\Delta + 4k\Delta$ rounds,

- ➜ if some distance register holds a value smaller that the true distance, smallest such value is at least k, and
- ➜ every node within distance k from the root stores the exact distance in its respective register field.

Leader selection

Existence of unique identifiers $\mathcal{N}.id$ is assumed. The algorithm selects the node with maximal identifier to be the leader.

Each processor $i \in \{1, \ldots, n\}$, holds two integer values: $\mathcal{N}.max$, which is an estimate of the maximum ID, and $\mathcal{N}.dist$ which is the estimated distance of the processor with $\mathcal{N}.id = max$.

The algorithm is described by a set of rules $\langle$label$\rangle : \mathcal{G}_i \mapsto \mathcal{A}_i$, where $\mathcal{G}_i$ is a *guard* and $\mathcal{A}$ is the set of actions that will take place in node $\mathcal{N}_i$ when the guard is satisfied.

[PD02]

Leader election: follow steps

These rules aim to locate the maximal ID value by setting the $\mathcal{N}_i.max$ to the maximum known within the neighboring nodes $\mathcal{N}_j,\ j \in \Gamma(i)$ and increasing the distance estimate by one.

$$\begin{aligned}\langle f_1\rangle \quad : \quad & ((\exists j \in \Gamma(i) \mid j.max > i.max) \wedge (j.dist + 1 \le n-1)) \\ & \mapsto \{ i.max = j.max,\ i.dist = j.dist + 1 \}\end{aligned}$$

$$\begin{aligned}\langle f_2\rangle \quad : \quad & ((\exists j \in \Gamma(i) \mid j.max \le i.max) \wedge \\ & (\exists k \in \Gamma(i) \mid (k.max = i.max) \wedge (k.dist < i.dist - 1))) \\ & \mapsto \{ i.dist = k.dist + 1 \}\end{aligned}$$

Leader election: termination condition

Denote the leader by $\mathcal{N}_\ell$. When property $\mathcal{P}_\mathcal{G}$ is true, no guard is enabled for any node. Otherwise at least one node has an enabled guard. None of the nodes can make infinite moves in presence of a finite number of faults.

$$\begin{aligned}\mathcal{P}_1 \quad &\equiv \quad ((\ell.max = \ell.id) \wedge (\ell.dist = 0)) \\ \mathcal{P}_2 \quad &\equiv \quad ((\forall j \neq \ell,\ ((j.max = \ell.id) \wedge (\exists k \in \Gamma(j) \mid j.dist = k.dist + 1))) \\ \mathcal{P}_3 \quad &\equiv \quad (\forall j \neq \ell,\ \nexists k \in \Gamma(j) \mid (k.dist < j.dist - 1))\end{aligned}$$

$$\mathcal{P}_\mathcal{G} \equiv \mathcal{P}_1 \wedge \mathcal{P}_2 \wedge \mathcal{P}_3$$

Simulations indicate $O(n)$ time complexity.

Leader election: correction steps

The correction steps fix erroneous states such that the leader not knowing that it is the leader.

$$\begin{aligned}\langle c_1\rangle \quad &: \quad ((i.dist = 0) \wedge (i.max \neq i.id)) \mapsto \{ i.max = i.id \} \\ \langle c_2\rangle \quad &: \quad ((i.dist \neq 0) \wedge (i.max = i.id)) \mapsto \{ i.dist = 0 \} \\ \langle c_3\rangle \quad &: \quad (i.dist > n-1) \mapsto \{ i.max = i.id,\ i.dist = 0 \} \\ \langle c_4\rangle \quad &: \quad (i.max < i.id) \mapsto \{ i.max = i.id,\ i.dist = 0 \} \\ \langle c_5\rangle \quad &: \quad ((i.dist > 0) \wedge \\ & \qquad (\nexists j \in \Gamma(i) \mid ((j.max = i.max) \wedge (i.dist = j.dist + 1)))) \\ & \qquad \mapsto \{ i.max = i.id,\ i.dist = 0 \}\end{aligned}$$

Proof techniques

- Potential function — proving a function over the configuration set to be monotonically in/decreasing; after reaching a certain threshold, the system is in a legal configuration. E.g. self-stabilizing matching.
- Convergence stairs — proving that the algorithms converges to fulfill a sequence of $k > 1$ refinements $\mathcal{R}_1, \mathcal{R}_2, \ldots, \mathcal{R}_k$ where $\mathcal{R}_{i+1}$ is a refinement of $\mathcal{R}_i$ such that $\mathcal{R}_{i+1} \Rightarrow \mathcal{R}_i$. Start by proving $\mathcal{R}_1$. E.g. self-stabilizing leader election.
- Scheduler-Luck game — The goal of the Scheduler is to prevent a randomized algorithm from achieving its task and the opponent, Luck, gets to intervene with the random function being used when she wants. E.g. upper bounds on time complexity.

Fair composition

Building a composition from self-stabilizing algorithms

$$\mathcal{A}_1,\ \mathcal{A}_2,\ \mathcal{A}_3,\ \ldots,\ \mathcal{A}_k$$

such that the result computed by $\mathcal{A}_i$ is used as input to $\mathcal{A}_{i+1}$.

The stabilization of $\mathcal{A}_i$ results in an arbitrary but legal starting configuration for $\mathcal{A}_{i+1}$. Finally $\mathcal{A}_k$ will stabilize to legal behavior after a chain of stabilizations.

The algorithms are executed by fair scheduling: each processor executes a step of each algorithm infinitely often.

[Dol00]

Superstabilization

A superstabilizing algorithm, previously holding up a legal configuration, quickly reaches a new legal configuration "gracefully" after a experiencing a crash within the system.

- Passage predicate: a strong predicate that holds when a topology change occurs in a legal configuration; weaker than legitimacy
- Superstabilization complexity: time of re-stabilization after a single fault
- Adjustment measure: the number of nodes that contribute to the recovery process

Example of a passage predicate: within a token ring system, there exists *at most one token at a time*.

[Dol00]

Pseudo-stabilization

A weaker requirement: each execution has a *suffix* that satisfies the system specification.

All self-stabilizing algorithms satisfy this, but not vice versa:

$$\begin{aligned}
\mathcal{S} &= (\mathcal{C},\mathcal{T})\\
\mathcal{C} &= \{a,b\}\\
\mathcal{T} &= \{a\mapsto a,\ a\mapsto b,\ b\mapsto b\}\\
\mathcal{P} &= \text{all configurations are the same}
\end{aligned}$$

Suffixes of executions?

Definition of the set of legitimate configurations $\mathcal{L}$?

Executions satisfying $\mathcal{P}$ do not necessarily contain a legitimate configuration!

Superstabilizing graph coloring with $\Delta + 1$ colors

$C_{\mathtt{neigh}} := \emptyset,\ C_{\mathtt{greater}} := \emptyset$
$\forall u_m \in \Gamma(v_i)\ :$
  $copy_m := \mathtt{read}(reg_m)$
  $C_{\mathtt{neigh}} := C_{\mathtt{neigh}} \cup \{copy_m.color\}$
  $\mathtt{if}\ (copy_m.id > reg_i.id)$
    $C_{\mathtt{greater}} := C_{\mathtt{greater}} \cup \{copy_m.color\}$
$\mathtt{if}\ ((reg_i.color = \bot) \vee (reg_i.color \in C_{\mathtt{greater}}))$
  $\mathtt{write}\ reg_i.color := random(C_{\mathtt{all}} \setminus C_{\mathtt{neigh}})$
$\mathtt{if}\ (recover_{ij} \wedge (j > i))\ :\ \mathtt{write}\ reg_i.color := \bot$

$$\begin{aligned}
C_{\mathtt{all}} &= \text{all } \Delta+1 \text{ available colors}\\
C_{\mathtt{neigh}} &= \text{colors of all neighboring nodes}\\
C_{\mathtt{greater}} &= \text{colors of those neighbors that have a greater identifier}
\end{aligned}$$

Properties of superstabilizing coloring

- Passage predicate: colors of neighboring nodes are different in any execution starting in a safe configuration when only one edge may fail or recover between safe configurations
- Complexity: stabilizing complexity $O(n)$, superstabilizing complexity $O(1)$
- Adjustment measure: only one node takes action upon an edge addition or removal

More algorithms

Self-stabilizing distributed queuing [HT01]: Each process knows whether it is the last one. When new processes join the queue, they find the end of the queue by following "pointers".

Maximum flow [GGP94]: The flow calculation adjusts to topology changes. The algorithm is similar to *preflow-push*.

Symbolic model checking for self-stabilizing algorithms [TNPK01] has also been investigated.

References

[Dij74] Edsger W. Dijkstra. Self stabilizing systems in spite of distributed control. *Communications of the Association of the Computing Machinery*, 17:643–644, 1974.

[Dol00] Shlomi Dolev. *Self-Stabilization*. The MIT Press, Cambridge, Massachusetts, 2000.

[GGP94] Sukumar Ghosh, Arobinda Gupta, and Sriram V. Pemmaraju. A self-stabilizing algorithm for the maximum flow problem. Technical Report TR94-10, The University of Iowa, 1994.

[HT01] Maurice Herlihy and Srikanta Tirthapura. Self stabilizing distributed queuing. *Lecture Notes in Computer Science*, 2180:209, 2001.

[PD02] Vinayaka Pandit and D. M. Dhamdhere. Self-stabilixing maxima finding on general graphs. Technical report, IBM Research Division, April 2002.

[Tel00] Gerard Tel. *Introduction to Distributed Algorithms*. Cambridge University Press, second edition, 2000. Chapter 17: Stabilization.

[TNPK01] Tatsuhiro Tsuchiya, Shin'ichi Nagano, Rohayu Bt Paidi, and Tohru Kikuno. Symbolic model checking for self-stabilizing algorithms. *IEEE Transactions on Parallel and Distributed Systems*, 12(1), January 2001.