Ted Herman
Sébastien Tixeuil (Eds.)

LNCS 3764

Self-Stabilizing Systems

7th International Symposium, SSS 2005
Barcelona, Spain, October 2005
Proceedings

Springer

TP311-53
S774
2005

Ted Herman Sébastien Tixeuil (Eds.)

Self-Stabilizing Systems

7th International Symposium, SSS 2005
Barcelona, Spain, October 26-27, 2005
Proceedings

Springer

Volume Editors

Ted Herman
University of Iowa
Department of Computer Science
Iowa City, IA 52242, USA
E-mail: herman@cs.uiowa.edu

Sébastien Tixeuil
Université Paris-Sud
LRI
Bâtiment 490, 91405 Orsay Cedex, France
E-mail: tixeuil@lri.fr

Library of Congress Control Number: 2005934787

CR Subject Classification (1998): C.2.4, C.2, C.3, F.1, F.2.2, K.6

ISSN	0302-9743
ISBN-10	3-540-29814-2 Springer Berlin Heidelberg New York
ISBN-13	978-3-540-29814-4 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 11577324 06/3142 5 4 3 2 1 0

¥396.48元

Preface

Self-stabilization is an established principle of modern distributed system design. The advantages of systems that self-recover from transient failures, temporary security attacks, and spontaneous reconfiguration are obvious. Less obvious is how the ambitious goal of recovering from the most general case of a transient fault, namely that of an arbitrary initial state, can lead to a simpler system design than dealing with particular cases of failures. In the area of mathematical problem-solving, Pólya gave the term "the inventors paradox" to such situations, where generalizing the problem may simplify the solution. The dramatic growth of distributed systems, peer-to-peer distribution networks, and large grid computing environments confronts designers with serious difficulties of complexity and has motivated the call for systems that self-recover, self-tune, and self-manage. The principles of self-stabilization can be useful for these goals of autonomous system behavior.

The Symposium on Self-Stabilizing Systems (SSS) is the main forum for research in the area of self-stabilization. Previous Workshops on Self-Stabilizing Systems (WSS) were held in 1989, 1995, 1997, 1999, and 2001. The previous Symposium on Self-Stabilizing Systems (SSS) took place in 2003. Thirty-three papers were submitted to SSS 2005 by authors from Europe (16), North America (8), Asia (4), and elsewhere (5). From the submissions, the program committee selected 15 for inclusion in these proceedings. In addition to the presentation of these papers, the symposium event included a poster session with brief presentations of recent work on self-stabilization.

The technical contributions to the symposium this year showed that the area has matured deeply since its first mathematical definition more than thirty years ago. Although there remains a core of four "classical" self-stabilization papers (that close gaps and open problems), the main part of the proceedings is dedicated to either extensions of self-stabilization (six contributions, dealing with snap-stabilization, code stabilization, self-stabilization with either dynamic, faulty or Byzantine components) or to applications of self-stabilization (five contributions, related to operating systems, security, or mobile and *ad hoc* networks).

The symposium of 2005 was one of the events of MANWEEK 2005, which also included the International Conference on Management of Multimedia Networks and Services (MMNS 2005), the International Workshop on IP Operations and Management (IPOM 2005), and the IEEE/IFIP International Workshop on Autonomic Grid Networking and Management (AGNM 2005). The site for the symposium and the other conferences was the Universitat Politècnica de Catalunya, in Barcelona. The SSS 2005 sessions were held on October 26 and 27.

We thank the organizers of MANWEEK 2005, especially Joan Serrat of the Universitat Politècnica de Catalunya, for making local arrangements.

August 2005

Ted Herman
Sébastien Tixeuil

Organization

Steering Committee

Anish Arora, The Ohio State University
Ajoy K. Datta, University of Nevada at Las Vegas
Shlomi Dolev, Ben-Gurion University of the Negev
Sukumar Ghosh, University of Iowa
Mohamed G. Gouda, University of Texas at Austin
Ted Herman, University of Iowa
Shing-Tsaan Huang, National Central University, Taiwan
Vincent Villain, Université de Picardie

Program Committee

Jorge Cobb, University of Texas at Dallas
Pascal Felber, Université de Neuchâtel
Roy Friedman, Technion
Felix Gärtner, RWTH Aachen
Maria Gradinariu, IRISA / INRIA Rennes
Ted Herman (Chair), University of Iowa
Jaap-Henk Hoepman, Radboud University Nijmegen
Hirotsugu Kakugawa, Hiroshima University
Mikhail Nesterenko, Kent State University
Marina Papatriantafilou, Chalmers University
Manish Parashar, Rutgers University
Franck Petit, Université de Picardie
Srikanta Tirthapura, Iowa State University
Sébastien Tixeuil, Université Paris-Sud

Additional Reviewers

Doina Bein
Ken Calvert
Bertrand Ducourthial
Stéphane Devismes
Sukumar Ghosh
Lisa Higham
Ronen Kat
Sandeep Kulkarni
Toshimitsu Masuzawa
Yoshihiro Nakaminami
Sriram Pemmaraju
Nir Tzachar
Vincent Villain
Guangsen Zhang
Christian Boulinier
Thomas Clouser
Ajoy Datta
Shlomi Dolev
Mohamed Gouda
Shing-Tsaan Huang
Yoshiaki Katayama
Mikel Larrea
Vincent Matossian
Rajesh Patel
Michel de Rougemont
Oliver Theel
Antonino Virgillito
Anat Bremler-Bar
Praveen Danturi
Murat Demirbas
Martin Gairing
Yinnon Haviv
Sayaka Kamei
Boris Koldehofe
Xiaolin Li
Stephane Messika
Phillipe Raipin Parvedy
Laurent Rosaz
Philippas Tsigas
Chen Zhang

Table of Contents

Snap-Stabilizing Optimal Binary Search Tree
Doina Bein, Ajoy K. Datta, Vincent Villain 1

Synchronous vs. Asynchronous Unison
Christian Boulinier, Franck Petit, Vincent Villain 18

A Snap-Stabilizing DFS with a Lower Space Requirement
Alain Cournier, Stéphane Devismes, Vincent Villain 33

Self-stabilization of Byzantine Protocols
Ariel Daliot, Danny Dolev 48

Self-stabilization with r-Operators Revisited
Sylvie Delaët, Bertrand Ducourthial, Sébastien Tixeuil 68

Self-stabilization Preserving Compiler
Shlomi Dolev, Yinnon Haviv, Mooly Sagiv 81

Self-stabilizing Mobile Node Location Management and Message Routing
Shlomi Dolev, Limor Lahiani, Nancy Lynch, Tina Nolte 96

Memory Management for Self-stabilizing Operating Systems
Shlomi Dolev, Reuven Yagel 113

Code Stabilization
Felix C. Freiling, Sukumar Ghosh 128

Stabilizing Certificate Dispersal
Mohamed G. Gouda, Eunjin (EJ) Jung 140

On the Possibility and the Impossibility of Message-Driven Self-stabilizing Failure Detection
Martin Hutle, Josef Widder 153

Approximation of Self-stabilizing Vertex Cover Less Than 2
Jun Kiniwa 171

Self-stabilization in Spite of Frequent Changes of Networks: Case Study of Mutual Exclusion on Dynamic Rings
Toshimitsu Masuzawa, Hirotsugu Kakugawa 183

Towards Automatic Convergence Verification of Self-stabilizing Algorithms
Jens Oehlerking, Abhishek Dhama, Oliver Theel 198

About the Self-stabilization of a Virtual Topology for Self-organization in Ad Hoc Networks
Fabrice Theoleyre, Fabrice Valois 214

Author Index 229

Snap-Stabilizing Optimal Binary Search Tree

Doina Bein[1], Ajoy K. Datta[1], and Vincent Villain[2]

[1] School of Computer Science, University of Nevada, Las Vegas
{siona, datta}@cs.unlv.edu
[2] LaRIA,Université de Picardie Jules Verne, France
villain@laria.u-picardie.fr

Abstract. We present the first snap-stabilizing distributed *binary search tree* (BST) algorithm. A *snap-stabilizing* algorithm guarantees that the system always behaves according to its specification provided some processor initiated the protocol. The maximum number of items that can be stored at any time at any processor is constant (independent of the size (n) of the network). Under this space constraint, we show a lower bound of $\Omega(n)$ on the time complexity for the BST problem. We then prove that starting from an arbitrary configuration where the nodes have distinct internal values drawn from an arbitrary set, our algorithm arranges them in a BST order in $O(n)$ rounds. Therefore, our solution is asymptotically optimal in time and takes $O(n)$ rounds. A processor i requires $O(\log s_i)$ bits of space where s_i is the size of the subtree rooted at i. So, the root uses $O(\log n)$ bits. The proposed algorithm uses a *heap* algorithm as a preprocessing step. This is also the first snap-stabilizing distributed solution to the heap problem. The heap construction spends $O(h)$ (where h is the height of the tree) rounds. Its space requirement is constant (independent of n). We then exploit the heap in the next phase of the protocol. The root collects values in decreasing order and delivers them to each node in the tree in $O(n)$ rounds following a pipelined delivery order of sorted values in decreasing order.

Keywords: Binary search tree, heap, self-stabilization, snap-stabilization.

1 Introduction

Given a binary tree where every node holds one key (value) drawn from an arbitrary set of real values, we design a snap-stabilizing distributed algorithm to arrange the values in the tree to obtain a binary search tree. A self-stabilizing [5,6] system, regardless of the initial states of the processors and initial messages in the links, is guaranteed to converge to the intended behavior in finite time. A *snap-stabilizing* [2,4] algorithm guarantees that it always behaves according to its specification. In other words, a snap-stabilizing algorithm is also a self-stabilizing algorithm which stabilizes in 0 steps.

The BST construction works as follows. First, the values in the tree are re-arranged as a heap (we implement a MaxHeap but a MinHeap is equally possible). Based on the heap arrangement, the root collects values in decreasing order and delivers them to each node in the tree (a sequential, pipelined delivery of sorted values in decreasing order). The tree structure is not modified by our algorithm.

Related Work: A heap construction that supports insert and delete operations in arbitrary states over a variant of the standard binary heap [3] with the maximum capacity of K items is proposed in [8]. It takes $O(m \log K)$ heap operations to stabilize (m is the initial number of items in the heap). The space complexity per node i is $O(h_i)$ where h_i is the height of the subtree T_i in the binary heap rooted at node i. Stabilizing search 2-3 trees are investigated in [9]. The stabilization time is $O(n \log n)$ rounds where n is the number of nodes in the initial state and the space complexity per node i is $O(d_i)$ where d_i is the distance from the root to node i.

Contributions: This paper has two major contributions. It includes the first snap-stabilizing binary search tree (BST) and the first snap-stabilizing heap algorithm. Being snap-stabilizing gives our algorithms a unique feature — they *always* behave as expected by their specifications. It should be noted that a self-stabilizing algorithm is guaranteed to satisfy the desired specification *only* in a finite time. In the context of the BST problem, in a self-stabilizing BST solution, if the root initiates a BST computation, it is not guaranteed that the tree will become a BST when the computation terminates. If the computation is repeated (a bounded but unknown number of times), the self-stabilizing algorithm guarantees that eventually, the tree will become a BST. The proposed snap-stabilizing solution achieves a much better solution than the above. It ensures that when a BST computation initiated by the root terminates, the tree is a BST. Thus, we do not need to repeat the computation unless the application program demands repeated sorting of the values in the tree.

A key feature of our solution is that the maximum number of items that can be stored at any time at any processor is constant (independent of the size (n) of the network). Under this space constraint, our solution is asymptotically optimal in time and takes $O(n)$ rounds. A processor i requires $O(\log s_i)$ bits where s_i is the size of the subtree rooted at i. So, the root uses $O(\log n)$ bits. The proposed algorithm uses a snap-stabilizing heap algorithm as a preprocessing step. This is also the first snap-stabilizing distributed solution to the heap problem. The cost of the heap construction is $O(h)$ rounds and constant (independent of n) space.

Outline of the Paper: In Section 2, we present the computational model, snap-stabilization, and the specification of the BST problem. We present the solution (the detail code of the algorithm) in Section 3. Due to lack of space, the detail code of the predicates and macros are omitted. They are available in the technical report [1]. We give a sketch of the correctness proof in Section 4, while the detail proof is available in [1]. We finish the paper with some concluding remarks in Section 5.

2 Preliminaries

Distributed System: We consider an asynchronous binary tree network of n processors with distinct ID's. The root is denoted by r. We will use nodes and processors interchangeably. The processors communicate using bi-directional links. We assume the local shared memory model of communication. The program of every processor consists of a set of *shared variables* and a finite set of actions. A processor can only write to its own variables, and read its own variables and variables owned by the neighboring processors. Each action is of the following form: $<$ *label* $>$ $<$ *guard* $>$ $\longrightarrow$ $<$ *statement* $>$. The guard of an action in the program of any process p is a boolean expression involving the variables of p and its neighbors. The statement of an action of p updates one or more variables of p. An action can be executed only if its guard evaluates to true. We assume that the actions are atomically executed, meaning, the evaluation of a guard and the execution of the corresponding statement of an action, if executed, are done in one atomic step.

The *state* of a processor is defined by the value of its variables. The *state* of a system is the product of the states of all processors. We will refer to the state of a processor and system as a (*local*) *state* and (*global*) *configuration*, respectively. A processor p is said to be *enabled* in a configuration γ if there exists at least an action A such that the guard of A is true in γ. We consider that any processor p executed a *disabling action* in the computation step $\gamma_i \mapsto \gamma_{i+1}$ if p was enabled in γ_i and not enabled in γ_{i+1}, but did not execute any action between these two configurations. (The disabling action represents the following situation: At least one neighbor of p changed its state between γ_i and γ_{i+1}, and this change effectively made the guard of all actions of p false.) Similarly, an action A is said to be enabled (in γ) at p if the guard of A is true at p (in γ). We assume an *unfair and distributed daemon.* The *unfairness* means that a processor p may never be chosen by the daemon to execute an action even if it is continuously enabled unless it is the only enabled processor.

A *computation step* is a transition between two configurations where the transition contains at least one action and at most one action per processor. The *distributed* daemon implies that during a computation step, if one or more processors are enabled, then the daemon chooses at least one (possibly more) of these enabled processors to execute an action.

In order to compute the time complexity measure, we use the definition of *round* [7]. This definition captures the execution rate of the slowest processor in any computation. Given a computation e, the *first round* of e (let us call it e') is the minimal prefix of e containing the execution of one action (an action of the protocol or the disable action) of every continuously enabled processor from the first configuration. Let e'' be the suffix of e, i.e., $e = e'e''$. Then *second round* of e is the first round of e'', and so on.

Snap-Stabilization: We assume that in a normal execution, at least one processor (called, the *initiator*) initiates the protocol upon an external (w.r.t. the protocol) request by executing a special type of action, called an *initialization action*.

Definition 1 (Snap-Stabilization). *Let P be a protocol designed to solve a task T. P is called* snap-stabilizing *if and only if, starting from any configuration, any execution E of P always satisfies the specification of T.*

Specification 21 (BST Problem). *A protocol P is considered as a BST algorithm, if and only if the following conditions are true: (i) Any computation initiated by the root terminates in finite time. (ii) When the computation terminates, the values in the tree satisfy the BST property.*

Remark 1. To prove that a BST algorithm is snap-stabilizing, we have to show that every execution of the protocol satisfies the following two properties: (i) starting from any configuration, the root eventually executes an initialization action. (ii) Any execution, starting from this action, satisfies Specification 21.

The time needed to reach the configuration where the initialization action is enabled is called the *delay* of the protocol.

3 Binary Search Tree Algorithm

In this section, we describe the data structures used, followed by a detailed explanation of how the algorithm works when the initiator (the root process) starts the algorithm until the values are arranged in the tree such that it becomes a BST. We divide the algorithm code in two parts: module *Heap* (Subsection 3.1) and module *Sort* (Subsection 3.2).

A node i holds four *constants*. The constants are not changed by the BST algorithm. The constants are: the value $V.i$ that needs to be sorted in the tree, the parent ID $p.i$, the left child ID $left.i$, and the right child ID $right.i$. If i does not have any of the above three neighbors, the corresponding constant's value is represented as $\perp$. For example, for the root node r, $p.r = \perp$, and for the leaf nodes, $left.i = right.i = \perp$. We denote the set of neighbors and set of children of i by $N.i$ and $D.i$, respectively. We assume that the tree has n nodes and has a height of h. Let T_i be the subtree rooted at i. Then s_i and h_i denote the number of nodes and height, respectively, of T_i.

Our BST construction is transparent to the changes (addition or removal of notes) in the tree structure. If such changes occur, then the algorithm will incorporate the changes "on the fly" by nodes either entering an abnormal situation with respect to their new neighbors, or by completing the current cycle and restarting a new cycle with added/deleted values. We assume that after the add/remove operations/queries are executed, our algorithm will be initiated by the root and a new BST tree will be constructed in $O(n)$ rounds. This makes the lower bound of $\Omega(n)$ under the constraints considered in this work higher than that of the usual functions (e.g., find, insert, and delete) for a non-stabilizing BST.

The basic idea of the algorithm is as follows: The algorithm runs in two phases. The root initiates the BST computation by starting a heapify process (shown as Module *Heap* in the algorithm) to create a maxheap of the tree. Then the root initiates the second phase (shown as *Sort* module). During this phase, the values are placed in the nodes in the BST order, placing the highest value first, the second highest value next, and so on. As the maxheap has been created in the previous phase, the root holds the maximum value of the tree. This highest value is sent to the rightmost node (say, i) of the tree. The destination of the second highest value (say, *second*) is dependent on if i is a leaf or an internal node. If i is a leaf node, then *second* is sent to the parent of i (say, j). Then the third highest value (say, *third*) will be sent to the left child of j (if present) or to the parent of j. If i is an internal node, then *second* goes to the left child of i. Thus, values are placed in the tree following a right-parent-left order.

The algorithm will be similar if we have constructed a minheap instead of the maxheap. In that case, in the second phase, the values will be placed following a left-parent-right order. From now on, heap will imply maxheap. If a node i satisfies the maxheap property with respect to its parent and children, we say i is in heap order or in HO in short.

Some of the variables used by a node i are described below. The rest of the variables will be defined in the informal explanations in the next two subsections. The sorted value $SV.i$ will contain the final sorted value at the end of the algorithm. $tSV.i$ is used to store a temporary sorted value. The heap value $HV.i$ is the result of the first phase (*Heap* module). The module *Sort* needs to maintain the size of the subtrees rooted at each node. This size variable $s.i$ for node i is computed in *Heap* and used in *Sort*.

A node may use at most seven states (see Figure 1 below). Module *Heap* uses six states: C (cleaning state), B (ready to start the heapify process), M, M^{left}, M^{right} (the states corresponding to if the maximum heap value HV is based on its own heap value, the maximum heap value of its left child, the maximum heap value of its right child, respectively), P (the *Heap* phase finished at this node, and the *Sort* phase is ready to start at this node). Module *Sort* uses C, P, and T (the algorithm is terminated).

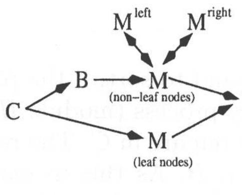


Fig. 1. The seven states used by the algorithm

A configuration in which the root is in state C is called a *clean configuration.* Starting from such a configuration, all other nodes in the tree will eventually reach C state. If all nodes are in C state, then the corresponding configuration is termed as a *normal starting configuration.* Any configuration reachable from a normal starting configuration by executing the algorithm guards is called a *normal configuration.* All other configurations are considered to be *abnormal.*

Some abnormal configurations can be locally detected by the processors. This local detection is implemented using the *abnormal* predicates in Algorithms 3.1 and 3.2. These predicates are used as guards of correction actions in order to avoid possible deadlocks and to speed up the protocol. Unfortunately, some problems of abnormal configurations cannot be locally detected. For example, the initial configuration may contain some sorted values (in tSV) that do not match any V values. The correction actions can remove the locally detectable problems in $O(h)$ rounds even before the root executes its initialization action. The other problems are eventually removed during the suffix of the protocol starting from the initialization action of the root.

Starting from an abnormal configuration, an execution not necessarily will bring the system to a normal starting configuration, but to a normal configuration. When a node has an abnormal predicate enabled, it will change its state to C, and all the nodes in its subtree will enter C state, but not necessarily its parent (e.g. if the parent state is B).

Starting from a normal configuration where the root is able to execute the initialization action with no delay, the tree will become a BST in $O(n)$ rounds. In general, the worst delay is $O(n)$ rounds because the worst initial configuration is the one where no node has any of the abnormal predicates enabled, but there is a node with an incorrect tSV value (that does not match any V values). Thus, the abnormal configurations do not increase the asymptotic time bound. So, starting from any configuration, the tree will become a BST in $O(n)$ rounds.

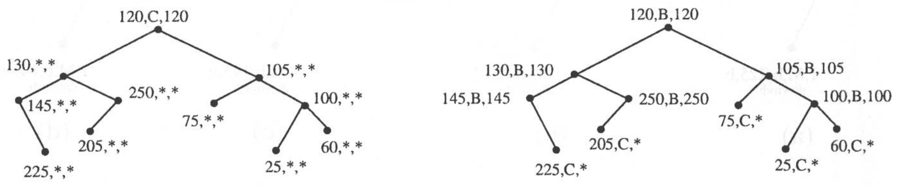
The interface between the two layers (application and BST) at a node i is implemented by two variables: input value to the sorting protocol $V.i$ and the final or output sorted value $SV.i$. However, every time the BST protocol runs, we do not want to disturb the application layer by writing (or overwriting) the value of $SV.i$ unless the value has changed. So, when the BST protocol terminates, i's sorted value is first placed in $tSV.i$. Then $tSV.i$ and $SV.i$ are compared. The value of $tSV.i$ is copied into $SV.i$ only if the values are different (see Actions $rP3$, $iP3$, and $lP1$&3 of module *Sort*).

3.1 Constructing the Heap

Upon receiving an external command to sort, if the root is enabled to start the BST protocol, it starts the heapify process (module *Heap*). The root is enabled to initiate if it is in C and its children are in C. The root broadcasts the heapify command by changing its state to B. As this message (wave) goes down the tree, all internal nodes change their state from C to B. When this broadcast wave reaches the leaf nodes, they change their state from C to M to initiate the heapify process (or wave). During this upward wave, the nodes compute two things: the heap value (the maximum value in their subtrees) and the size of their subtrees. When this wave reaches the root, the root changes its state to M and the heap is created. The root then initiates another top-down wave by changing its state from M to P. The next phase, i.e., the BST construction phase starts from the P state. We now describe the heap construction in more detail by referring Algorithm 3.1.

1. *(Start building a Heap)* If the root is in C, its children will change to C in at most one round. Either Action aCm or aCb is enabled, and since it is the only enabled action, it is eventually executed in at most one round. When its children change to C, the root changes its state from C to B and sets HV to its internal value V (Action CB). An internal node changes its state from C to B when its parent is in B and its children are in C. An internal node also initializes its heap value HV with its input (or initial) value V (Action CB).

Figure 2(a) shows the clean configuration for a 11-node binary tree. After B wave is executed top-down, the tree state is shown in Figure 2(b). We show only the node's internal value V, state S, and heap value HV. Symbol * means that the value is not important.



(a) Clean configuration (b) B wave is executed top-down

Fig. 2. Initial stage of constructing the heap

2. *(Calculating Heap and s.i Values)* A leaf node i changes its state from C to M and executes macro $init(i)$ (Action CM). In the macro $init(i)$, the node i sets the size of its subtree, $s.i$ to 1 and sets the heap values of its left (lHV) and right (rHV) subtrees to $\perp$ (indicating a non-existent value).

When a parent of a leaf node detects that all its children are in state M (Action BM^* is enabled), it executes $init(i)$, change from B to M, and executes macro $set\_HVs(i)$. If the (parent) node holds a value smaller than any of the heap values of its children, it chooses as its heap value the larger heap value (lHV or rHV) among its children and pushes its own heap value (HV) toward the child that was holding the larger heap value. This heapification process goes up the tree until it reaches the root.

Predicate $update\_HVs(i)$ is *true* when due to the heapification process at the parent of i, i's heap value became smaller than the values of its children. So, $HV.i$ needs to be swapped with that of one of its children. Predicate $h\_order(i)$ is true if i satisfies the heap property with respect to its children.
For a non-leaf node i that is about to execute the macro $set\_HVs(i)$, we consider three cases.

Case 1). $HV.i$ is larger than the heap values of its children. So, heap order is maintained at i. Then the macro $set\_HVs(i)$ does not change the variables $S.i$ (remains M) and $HV.i$.