Snap-Stabilizing k -Wave Synchronizer

Doina Bein¹, Ajoy K. Datta², Mehmet H. Karaata³, and Safaa Zaman⁴

¹ School of Computer Science, University of Nevada Las Vegas, NV
siona@cs.unlv.edu

² School of Computer Science, University of Nevada Las Vegas, NV
datta@cs.unlv.edu

³ Department of Computer Engineering, Kuwait University
karaata@cairo.eng.kuniv.edu.kw

⁴ Department of Computer Engineering, Kuwait University

Abstract. We propose a *snap-stabilizing k -wave* algorithm for routed trees, called *kWA*, which implements k consecutive waves, $k > 1$. A *snap-stabilizing* algorithm, starting from an arbitrary system configuration, always behaves according to its specification. The maximum number of states per process is $2k + 1$, better than the previous result of at most $3k$ states per process. A *snap-stabilizing* algorithm guarantees that, starting from an arbitrary system configuration, always behaves according to its specification.

Keywords: Distributed computing, fault tolerance, multi wave algorithm, routed tree, snap stabilization, synchronizer.

1 Introduction

Solutions to a large number of fundamental problems in distributed computing require the execution of a constant number of consecutive distinct waves with corresponding feedbacks [1] (e.g. broadcasting of information, distributed infimum computation, leader election, mutual exclusion, spanning tree construction problems, clock synchronization, etc.). Consecutive distinct broadcasts with the corresponding feedbacks are implemented by the *multi-wave* algorithms, also referred to as the *asynchronous clock synchronizers*.

In the literature, *PIF* (propagation of information with feedback) algorithms are also referred to as *echo algorithms* [2]. Such wave algorithms are essential components of some distributed algorithms: leader election [3], snapshot [4], global reset [4, 5, 6, 7], synchronization [6, 8, 9].

A *self-stabilizing* system automatically recovers to normal behavior in case of transient faults, without a centralized control. Regardless of the current configuration, the system reaches a legal state in finite number of steps and the system state remains legal thereafter [10], being able to withstand *transient failures* (a fault that perturbs the state of the system but not the program). Dolev provides a detailed survey on self-stabilization in [11].

If a certain stabilizing algorithm always behaves according to its specification, then is said to be *snap-stabilizing*, therefore it has optimal stabilization time. The notion of snap-stabilization was introduced in 1999 in [12]. The first snap-stabilizing *PIF* algorithm for arbitrary networks is proposed in [13]. A snap-stabilizing implementation of the *PIF* with cleaning called the *PFC scheme* (*propagation of information with feedback and cleaning*) is proposed by Bui et. al in [12], in which are three states in every wave. Therefore, a straight forward solution for a k -wave snap-stabilizing algorithm will have $3k$ states per process.

One of the main advantages of the *PIF* scheme to be snap-stabilizing is that the arbitrary initial state has limited or no effect on the pace of the broadcast propagation, so it implements the propagation with the optimal delay. Using the snap-stabilizing *PIF* algorithm of Bui et. al [12], distinct consecutive waves can be implemented by means of a counter per each process incremented upon receipt of each broadcast message. However, notice that the resulting algorithm is no longer stabilizing due to the need to initialize the counters. Therefore, the adaptation of the snap-stabilizing wave algorithm of Bui et. al to multiple consecutive distinct waves with feedback is not straight forward. The primary challenge stems from the introduction of new states representing the new phases while maintaining the snap-stabilization property.

Contributions. In this paper, we propose a *snap-stabilizing k -wave algorithm* (kWA), $k > 1$, for routed trees. The algorithm runs forever and the delay to start a kWA cycle is $O(h)$ rounds, which is optimal, where h is the height of the tree (the delay represents the time taken by the algorithm to start a kWA cycle beginning in an arbitrary initial state). A *round* refers to a minimum execution sequence in which each enabled action is performed at least once. The maximum number of states per process is $2k + 1$, which is better than the previous implementations of $3k$ states per process and the same delay. A round refers to a minimum execution sequence in which each enabled action is taken at least once. The algorithm is optimal with respect to its time complexity and can be generalized to general networks using any of the self-stabilizing spanning tree construction algorithms proposed in [5, 6, 14].

Outline of the paper. Section 2 describes the distributed model used and snap-stabilization. Section 3 describes the specification of propagation of information with feedback (*PIF*). The k -wave snap-stabilizing algorithm is presented in the Section 4. Because of lack of space, the complete correctness proof of kWA algorithm is omitted, only few important remarks are given. The paper ends with concluding remarks and discussions on future work in Section 5.

2 Stabilization in Distributed Systems

A tree can be represented by an undirected graph $T = (V, E)$ with vertex (node) set V and edge set E , where $|V| = n$, and $|E| = n - 1$. We assume that each vertex of T is a process with a unique ID. We denote the set of leaf processes in

T by L and the set of internal nodes by I . For each process i , $N.i$, $D.i$, and $p.i$ denote the set of its neighbors, the set of its children and its parent, respectively.

Each process maintains a set of local variables whose values can only be updated by the process itself after inspecting them and the local variables of its neighbors.

The distributed program of some process can be expressed as a finite set of *guarded commands or guarded actions* [15]:

$$* [G[1] \rightarrow A[1] \square G[2] \rightarrow A[2] \square \dots \square G[k] \rightarrow A[k]],$$

In the *guarded action* $G[] \rightarrow A[]$, the *guard* $G[]$ is a Boolean function of the variables of the process and its neighbors, and *action* $A[]$ can read the variables of the process and its neighbors, and update only the process variables. If $G[]$ is *true*, then the guarded action is called *enabled*. A process that has at least one guard enabled is called *enabled* too. $*[S]$ corresponds to the repeated execution of S while there exists enabled guards. $\square$ is called the *nondeterminism* symbol, and means that one of the guarded actions separated by those symbols is selected nondeterministically in each iteration and executed.

We assume the presence of a *distributed daemon or scheduler*: if more than one process is enabled, then a subset of enabled processes is selected, and exactly one enabled guard of each selected process is executed atomically and concurrently. In addition, we assume that the scheduler is *weakly fair*, i.e., any action remaining enabled is eventually executed. A self-stabilizing algorithm implementing this can be adapted from [16, 17]. Therefore, an implementation of this is omitted.

The *state (configuration)* of a process is the set of variables used by the process. The *system state* is the cartesian product of the states of the processes in the system. The set of all the possible system states is denoted by $\mathcal{C}$. An execution is a maximal sequence of states $e = c_1, c_2, \dots$ such that $\forall c_i \in \mathcal{C} : c_{i-1} \mapsto c_i$ by executions some guarded action, for each $i > 0$. Given P a predicate defined over $\mathcal{C}$, the set of all states that satisfy P , $\mathcal{L}_P$, is called the set of all *legitimate states with respect to P* , or simply, the set of all *legitimate states*. Let P be a predicate defined over $\mathcal{C}$. The notation $c \vdash P$ means that an element $c \in \mathcal{C}$ satisfies the predicate P defined on the set $\mathcal{C}$. Given a task $\mathcal{T}$, the specification of $\mathcal{T}$ is a predicate called $\mathcal{SP}_{\mathcal{T}}$, and the distributed protocol to accomplish the task $\mathcal{T}$ is denoted by $\mathcal{P}_{\mathcal{T}}$. The set of all possible computations of $\mathcal{P}_{\mathcal{T}}$ in the system is denoted by $\mathcal{E}$.

Definition 1 (Snap-stabilization). *Let $\mathcal{T}$ be a task, and $\mathcal{SP}_{\mathcal{T}}$ the specification of $\mathcal{T}$. The program $\mathcal{P}_{\mathcal{T}}$ is snap-stabilizing for $\mathcal{SP}_{\mathcal{T}}$ on $\mathcal{E}$ if and only if $\forall e \in \mathcal{E} :: e \vdash \mathcal{SP}_{\mathcal{T}}$.*

3 Propagation of Information with Feedback (*PIF*)

Wave algorithms are often implemented using the *propagation of information with feedback (PIF) algorithms* [2, 18]. A snap-stabilizing implementation of the *PIF* with cleaning called the *PFC scheme (propagation of information with*

feedback and cleaning) is proposed by Bui et. al in [12]. The value C , called the *cleaning state*, denotes the initial state of a process before it participates in a *PIF*-cycle, or the ending state after it participates in a *PIF* cycle.

Each process i maintains a variable S_i called the S -value of i . The root can use only two state values: B (broadcast) and C (cleaning). When executing its B -action and C -action, the root changes its state from C to B and B to C , respectively. The leaf processes use only F (feedback) and C states: the F -action changes the state from C to F , and the C -action changes the state from F to C . An internal process can be in C , B , and F states: after executing a B -action, F -action, respectively C -action, the S -value changes from C to B , from B to F and F to C , respectively.

According to the *PFC* cycle specification, a normal broadcast phase is followed by a feedback phase which is initiated by the leaf processes. After initiating the feedback phase, in the next round, the leaf processes can initiate the cleaning phase by changing their states from F to C . So, the feedback and the cleaning phases run concurrently. Eventually, the feedback phase reaches the descendants of the root. The root now executes its C -action, i.e., changes its state from B to C , and then waits until all its descendants are in cleaning phase. Then it changes its state from C to B . This marks the end of the current *PIF* cycle and the start of the next *PIF* cycle. Since this solution requires three phases (*broadcast*, *feedback*, and *cleaning*) in the *PIF* cycle, the method is called the *propagation of information with feedback and cleaning (PFC)* and the corresponding cycle the *PFC cycle*.

We say that a process p is *B-done*, *F-done*, or *C-done* to indicate that p has executed its corresponding actions (B , F , or C , respectively).

4 k -Wave Snap-Stabilizing Algorithm (kWA)

We present the k -wave algorithm, $k > 1$, to implement k consecutive distinct waves scheme on tree structured networks.

A kWA cycle consists of k consecutive *PIF* cycles where each *PIF* has its own broadcast and feedback phase with a specific task. Starting from an arbitrary configuration where no message has yet been broadcasted, the root (r) initiates the *first broadcast phase* by sending a *first broadcast message* to its children. The internal processes forward the broadcast message to their children upon receipt of the first broadcast message. Once the first broadcast phase reaches the leaf processes, they notify their parents of the termination of the broadcast phase by initiating the *first feedback phase* by sending *first feedback messages*. After receiving first feedback messages from all its children, a process sends a first feedback message to its parent. When all the neighbors of the root are in the first feedback phase, the second wave will be initiated by the root through the *second broadcast phase*, which acts similarly. Following the k feedback phase, a *clear phase* is initiated.

These k waves can overlap in a manner similar to those of the *PFC* cycle. Unlike the *PFC* cycle, in a kWA cycle the feedback phase of each wave and the

cleaning phase do not execute concurrently. Instead, upon termination of the l feedback phase, the $l + 1$ wave is initiated, $1 \leq l < k$.

The state value C denotes the initial state of any process before it participates in a kWA cycle. The state values B^l , respectively F^l denotes the broadcast, respectively feedback state of a process, $1 \leq l < k$. The root uses $k + 1$ state values: $C, B^l, 1 \leq l < k$. An internal process can have $2k + 1$ different state values C, B^l and $F^l, 1 \leq l < k$. A leaf process uses only $k + 1$ state values, $C, F^l, 1 \leq l < k$.

The root executes its rB^1 -action, rB^l -action, $1 < l < k$, and rC -action by changing its state from C to B^1 , B^{l-1} to B^l , and B^k to C , respectively. An internal process executes its iB^1 -action, iF^1 -action, iB^l -action, iF^l -action, $1 < l \leq k$, and iC -action, by changing its S -value from C to B^1 , from B^1 to F^1 , from F^{l-1} to B^l , from B^l to F^l , and from F^k to C respectively. A leaf process executes its lF^1 -action, lF^l -action, $1 < l \leq k$, and lC -action respectively by changing its state from C to F^1 , from F^{l-1} to F^l , and from F^k to C respectively.

Based on the above description, we define the kWA cycle in terms of the following kWA -actions: B^1 -action, F^1 -action, B^2 -action, F^2 -action, ..., B^k -action, F^k -action, C -action. The B^l -action, F^l -action, with $1 \leq l \leq k$, refer to the actions executed during l -th broadcast, l -th feedback phase respectively. The C -action is executed in order to terminate the current kWA cycle.

Definition 2 (kWA cycle). A finite computation $e \in \mathcal{E}$ is called a kWA cycle, denoted by $e \vdash kWA$ -cycle, if and only if the following conditions hold:

- L1** At least one process r , called the initiator, sends a message or completes the kWA cycle (rB^1).
- L2** If an internal process i ($i \in I$) receives a first/second/.../ k -th broadcast message from its parent $p.i$, then i eventually sends this message to all its descendants (iB^1, iB^l).
- L3** If a leaf process i ($i \in L$) receives a first/second/.../ k -th broadcast message from its parent $p.i$, then i eventually sends a first/second/.../ k -th feedback message to $p.i$ (lF^1, lF^l).
- L4** If an internal process i ($i \in I$) receives the first/second/... k -th feedback message from all of its descendants, then i eventually sends this message to its parent $p.i$ (iF^1, iF^l).
- L5** If the initiator is in l -th broadcast phase and receives l -th feedback message from all of its descendants, then it sends the $(l + 1)$ -th broadcast message to all of its descendants (rB^l).
- L6** If the root r receives the k -th feedback message from all of its descendants, then r eventually terminates the current kWA cycle (rC).
- S** The root r cannot terminate the kWA cycle (rC) more than once.

Conditions [L1] to [L6] are called the liveness properties, and condition [S] is called the safety property. By Condition [L1], we know that there exists at least one initiator in a kWA cycle.

The algorithm implementing the above mechanism is given in Figure 1. The value of l is $1 < l \leq k$.

Variables

- $S_i(i = r) \in \{B^1, B^2, \dots, B^k, C\}$ for the root
 $S_i(i \in I) \in \{B^1, F^1, B^2, F^2, \dots, B^k, F^k, C\}$ for internal processes
 $S_i(i \in L) \in \{F^1, F^2, \dots, F^k, C\}$ for the leaf processes

Actions

$$\begin{array}{l}
 \{\text{Program for root process } i\} \\
 \text{rB1} \quad * \left[S_i = C \wedge \forall_{j \in N.i} S_j = C \right. \quad \longrightarrow \quad S_i := B^1; \\
 \text{rBl} \quad \square S_i = B^{l-1} \wedge \forall_{j \in D.i} S_j = F^l \quad \longrightarrow \quad S_i := B^l; \\
 \text{rC} \quad \square (S_i = B^k \wedge (\forall_{j \in N.i} S_j = F^k) \vee \\
 \quad (S_i = B^1 \wedge \exists_{j \in D.i} S_j \notin \{B^1, F^1, C\})) \vee \\
 \quad (1 < l < k \wedge S_i = B^l \wedge \exists_{j \in D.i} S_j \notin \{B^{l-1}, F^{l-1}, B^l, F^l\}) \quad \longrightarrow \quad S_i := C; \Big] \\
 \\
 \{\text{Program for internal process } i\} \\
 \text{iB1} \quad * \left[S_i = C \wedge S_{p.i} = B^1 \wedge \forall_{j \in D.i} S_j = C \right. \quad \longrightarrow \quad S_i := B^1; \\
 \text{iF1} \quad \square S_i = B^1 \wedge S_{p.i} = B^1 \wedge \forall_{j \in D.i} S_j = F^1 \quad \longrightarrow \quad S_i := F^1; \\
 \text{iBl} \quad \square S_i = F^{l-1} \wedge S_{p.i} = B^l \wedge \forall_{j \in D.i} S_j = F^{l-1} \quad \longrightarrow \quad S_i := B^l; \\
 \text{iFl} \quad \square S_i = B^l \wedge S_{p.i} = B^l \wedge \forall_{j \in D.i} S_j = F^l \quad \longrightarrow \quad S_i := F^l; \\
 \text{iC} \quad \square (S_i = F^k \wedge S_{p.i} = F^k \wedge \forall_{j \in D.i} S_j \in \{F^k, C\}) \vee \\
 \quad (1 < l \leq k \wedge S_i = B^{l-1} \wedge (S_{p.i} \notin \{B^{l-1}, F^{l-1}, B^l\} \vee \\
 \quad \exists_{j \in D.i} S_j \notin \{B^{l-1}, F^{l-1}, F^l\})) \vee \\
 \quad (1 < l \leq k \wedge S_i = F^{l-1} \wedge (S_{p.i} \notin \{B^{l-1}, F^{l-1}, B^l\} \vee \\
 \quad \exists_{j \in D.i} S_j \neq F^{l-1})) \quad \longrightarrow \quad S_i := C; \Big] \\
 \\
 \{\text{Program for leaf process } i\} \\
 \text{iF1} \quad * \left[S_i = C \wedge S_{p.i} = B^1 \right. \quad \longrightarrow \quad S_i := F^1; \\
 \text{iFl} \quad \square S_i = F^{l-1} \wedge S_{p.i} = B^l \quad \longrightarrow \quad S_i := F^l; \\
 \text{lC} \quad \square (S_i = F^k \wedge S_{p.i} = F^k) \vee \\
 \quad (1 < l \leq k \wedge S_i = F^{l-1} \wedge S_{p.i} \notin \{B^{l-1}, F^{l-1}, B^l\}) \vee \\
 \quad (S_i = F^k \wedge S_{p.i} \notin \{B^k, F^k, C\}) \quad \longrightarrow \quad S_i := C; \Big]
 \end{array}$$

Fig. 1. The Snap-Stabilizing k -Wave Algorithm (Algorithm kWA)

Let $Init$ be the set of processes that initiate kWA cycles. We will call a kWA cycle as a *normal kWA cycle* if $Init = \{r\}$ and the first action of r is a rB^1 -action.

Definition 3 (kWA scheme). We define computation e as a kWA scheme, denoted by $e \vdash \mathcal{SP}_{kWA}$, as an infinite sequence of kWA cycles, $e = e_0, e_1, \dots$ such that $\forall i \geq 1$, e_i is a complete kWA cycle.

The following proposition follows from the definition of the kWA cycle.

Proposition 1. *In a normal kWA cycle, every process executes a B^1 -action, a F^1 -action, ..., a B^k -action, and a F^k -action at most once.*

From Definition 2, it is obvious that once p executes one of the kWA -actions, p will not execute that action anymore in the same kWA cycle.

In any configuration, either the root is enabled to execute and the other processes have no enabled actions, or the root has no enabled actions and the other processes are enabled to execute. The following proposition follows from Definition 2 and Proposition 1.

Proposition 2. *In any configuration of the kWA cycle, either one of the following conditions holds:*

(i) *The root r is enabled to start the l^{th} wave, $1 \leq l \leq k$, by executing its B^l -action.*

(ii) *The root r is surrounded by a maximal set of processes all of which are enabled to execute a kWA -action corresponding to either l^{th} wave, $1 \leq l \leq k$, such that each process between r and a process in this set (including r) is B^l -done.*

Definition 4 (Snap-Stabilizing kWA). *A kWA algorithm is snap-stabilizing if and only if the root executes a kWA -action infinitely often, and the kWA cycle satisfies SP_{kWA} .*

Theorem 1. *Algorithm kWA is snap-stabilizing.*

The optimal delay of starting a kWA cycle of any implementation of the kWA cycle is shown to be $O(h)$ rounds in [6]. We can show that the delay to start the first kWA cycle using Algorithm kWA is bounded by $O(h)$ rounds. Therefore, the algorithm kWA is optimal with respect to its delay to start a kPW cycle.

5 Conclusions

In this paper, we propose a *snap-stabilizing k -wave* algorithm for routed trees, called kWA , with $k > 1$. The algorithm ensures that a cycle may start after $O(h)$ rounds, which is optimal, where h is the height of the tree. The maximum number of states per process is $2k + 1$, which is better than the previous implementations of $3k$ states per process and the same delay. We intend to apply the concept of kWA scheme in devising optimal snap-stabilizing algorithms for some fundamental problems in distributed computing.

References

1. G. Tel. Total algorithms. In: Vogt F.H. (ed) *Concurrency 88*, Springer-Verlag LNCS 335, pages 277–291, 1988.
2. E.J. Chang. Echo algorithms: depth parallel operations on general graphs. *IEEE Transactions on Software*, SE-8:391–401, 1982.

3. S. Dolev, A. Israeli, and S. Moran. Uniform dynamic self-stabilizing leader election. *IEEE Transactions on Parallel and Distributed Systems*, 8(4):424–440, 1997.
4. G. Varghese. Self-stabilization by counter flushing. *SIAM Journal on Computing*, 30(2):486–510, 2000.
5. A. Arora and M.G. Gouda. Distributed reset. *IEEE Transactions on Computers*, 43:1026–1038, 1994.
6. B. Awerbuch, S. Kutten, Y. Mansour, B. Patt-Shamir, and G. Varghese. Time optimal self-stabilizing synchronization (extended abstract). In *Proceedings of the Twenty-Fifth Annual ACM Symposium on the Theory of Computing*, pages 652–661, San Diego, California, 16–18 May 1993.
7. B. Awerbuch, B. Patt-Shamir, and G. Varghese. Self-stabilization by local checking and correction. In *Proceedings of the 32th Annual IEEE Symposium on Foundations of Computer Science*, pages 268–277, 1991.
8. L.O. Alima, J. Beauquier, A.K. Datta, and S. Tixeuil. Self-stabilization with global rooted synchronizers. In *ICDCS98 Proceedings of the 18th International Conference on Distributed Computing Systems*, pages 102–109, 1998.
9. B. Awerbuch and G. Varghese. Distributed program checking: a paradigm for building self-stabilizing distributed protocols. In *FOCS91 Proceedings of the 31st Annual IEEE Symposium on Foundations of Computer Science*, pages 258–267, 1991.
10. E.W. Dijkstra. Self-stabilizing systems in spite of distributed control. In *EWD 391, In Selected Writings on Computing: A Personal Perspective*, pages 41–46, 1973.
11. S. Dolev. *Self-Stabilization*. MIT Press, Cambridge, MA, 2000. Ben-Gurion University of the Negev, Israel.
12. A. Bui, A.K. Datta, F. Petit, and V. Villain. State-optimal snap-stabilizing PIF in tree networks. In *Proceedings of the Third Workshop on Self-Stabilizing Systems (published in association with ICDCS99 The 19th IEEE International Conference on Distributed Computing Systems)*, pages 78–85. IEEE Computer Society, 1999.
13. A. Cournier, A.K. Datta, F. Petit, and V. Villain. Snap-stabilizing PIF algorithm in arbitrary networks. In *ICDCS02 The 22nd IEEE International Conference on Distributed Computing Systems*, pages 199–206, 2002.
14. S.T. Huang and N.S. Chen. A self-stabilizing algorithm for constructing breadth-first trees. *Information Processing Letters*, 41:109–117, 1992.
15. E.W. Dijkstra. Guarded commands, nondeterminacy, and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975.
16. M.G. Gouda and F. Haddix. The alternator. In *Proceedings of the Third Workshop on Self-Stabilizing Systems (published in association with ICDCS99 The 19th IEEE International Conference on Distributed Computing Systems)*, pages 48–53. IEEE Computer Society, 1999.
17. M. Mizuno and M. Nesterenko. A transformation of self-stabilizing serial model programs for asynchronous parallel computing environments. *Information Processing Letters*, 66(6):285–290, 30 June 1998.
18. A. Segall. Distributed network protocols. *IEEE Transactions on Information Theory*, IT-29(1):23–35, January 1983.