Self-Stabilizing Asynchronous Phase Synchronization in General Graphs*

Chi-Hung Tzeng
National Tsing Hua University
Hsinchu, Taiwan
clark@cs.nthu.edu.tw

Jehn-Ruey Jiang †
National Central University
Chungli, Taiwan
jrjiang@csie.ncu.edu.tw

Shing-Tsaan Huang
National Central University
Chungli, Taiwan
sthuang@csie.ncu.edu.tw

Abstract

The phase synchronization problem requires each node to infinitely transfer from one phase to the next one under the restriction that at most two consecutive phases can appear among all nodes. In this paper, we propose a self-stabilizing algorithm under the parallel execution model to solve this problem for semi-uniform systems of general graph topologies. The proposed algorithm is memory-efficient; its space complexity per node is $O(\log \Delta + \log K)$ bits, where Δ is the maximum degree of the system and $K > 1$ is the number of phases.

Keywords: Distributed system, Fault tolerance, Phase Synchronization, Self-Stabilization, Spanning tree.

*This research was supported in part by the National Science Council of the Republic of China under the Contract NSC 92-2213-E-008-029

†The correspondence author

1 Introduction

This paper proposes a self-stabilizing phase synchronization algorithm for asynchronous systems of general graph topologies. A system may be disordered due to unexpected transient faults. We can make the system resilient to such faults by the concept of *self-stabilization*, introduced by Dijkstra [3]. A system is said to be self-stabilizing if it has the following two properties: (1) *Convergence*: Starting from any initial configuration (possibly illegal), the system can converge to a legal one in finite time. (2) *Closure*: Once the system is in a legal configuration, it remains so henceforth. When a self-stabilizing system encounters transient faults, it can be thought as in an arbitrary initial configuration. With the convergence property, it can reach a legal configuration; with the closure property, it can then function correctly henceforth.

The proposed algorithm makes each node go through a cyclic sequence of K phases: phase 0, phase 1,..., phase $K-1$, phase 0, phase 1, ... The phases of all nodes must satisfy the following criterion:

Criterion 1 (Phase Synchronization)

- No node can proceed to phase $k+1 \pmod K$ until all nodes are in phase k.
- When all nodes are in phase k, each node eventually proceeds to phase $k+1 \pmod K$.

The above phase synchronization criterion follows that discussed in [12]. Since the nodes in the system make moves in *asynchronous mode*, each time when we observe the phases of the nodes, the phase may not be identical, but should be no more than one apart. However, in illegitimate states caused either by transient faults or by arbitrary initialization, the phases of the nodes may be more than one apart.

The phase synchronization algorithm builds a synchronous environment over asynchronous one. Thus, applications developing for a synchronous environment can be executed on an asynchronous environments.

There are many self-stabilizing phase synchronization algorithms proposed in the literature [1, 2, 4, 6–9, 11, 12]. The algorithms in [1, 8, 9, 11, 12] are for the asynchronous environment; the others, the synchronous environment. Since we focus on the asynchronous environment in this paper, we only introduce the former algorithms below. The algorithm in [12] is designed for uniform complete graphs. (If all nodes have identical behavior, the system is said to be *uniform*.) It demands a node to proceed to a proper phase by examining all others' phases. The algorithm in [11] is for non-uniform rings. It uses the concept of token circulation: a node with a token can proceed to the next phase. The algorithm in [1] devotes to rooted tree networks and classifies nodes into the root node, internal nodes and leaf nodes. The root initiates a new phase whenever it detects the end of the last phase, whereas any other node just copies that phase. The algorithm in [8] is for uniform rings of odd size. It also uses token circulation to carry out the synchronizer: a node receiving a token copies the sender's phase and increments the token's counter by one. When the counter value is equal to the number of nodes in the system, the token owner resets the counter, then proceeds to the next phase and sends out the token. The algorithm in [9] is for uniform rings of any size. It views a ring as a set of segments whose heads can move from one node to another and make the number of segments decrease to one. Therefore, it works by allowing only the head to change its phase.

The proposed algorithm is semi-uniform; i.e., all system nodes, except a special node, have identical behavior. The basic idea of the algorithm is to utilize token circulation to construct a spanning tree and then to achieve phase synchronization. After the construction of the spanning tree, the tree root can initiate a phase and then sends a token containing the phase number to all

its children. On receiving the token, a node just follows the phase and then again forwards the token to all its children. The token bounces at leaf nodes; that is, on receiving the token, leaf nodes just send it back to parents. Furthermore, a node sends the bounced token to its parent if all its children have done so. In this manner, the token circulates in the root-to-leaf and then the leaf-to-root directions. The root can initiate a new phase when it receives bounced tokens from all its children. The new phase then proceeds properly, and so do all phases. It is noted that the proposed algorithm is not just a combination of a tree construction algorithm, such as those in [5], and a phase synchronization algorithm for the tree network, such as that in [1]. Instead, we use the the token circulation concept to achieve spanning tree construction and phase synchronization simultaneously. That is, by the time the tree is constructed, the system immediately meets the criterion of phase synchronization.

The proposed algorithm has the advantage of memory efficiency; its space complexity per node is $O(\log \Delta + \log K)$ bits, where Δ is the maximum degree of the system and $K > 1$ is the number of phases. Another advantage of the algorithm is that it operates correctly in the parallel model, which is more general than the serial model adopted by the algorithm in [8].

The rest of the paper is organized as follows. Section 2 presents the system model and some terms used throughout this paper. Section 3 shows the proposed algorithm and its correctness proofs. Finally, section 5 concludes this paper.

2 The System Model

We model the system by a connected, undirected, n-node graph $G = (V, E)$ where V is the set of nodes and E is the set of edges representing the links between a pair of nodes. Two nodes i and j are said to be neighbors if $(i, j) \in E$. Each node keeps a set of variables, to which it can write its own state and from which it can read the neighbors' states. Throughout this paper, we use the notation $VAR.i$ to denote the variable VAR maintained by node i.

The behavior of a node is defined by a set of rules in the form "$guard \rightarrow action$", where $guard$ is a boolean formula while $action$ is a set of program statements instructing how to update the values of the variables. Once a node evaluates the guard part of one rule true, we say that the node is *privileged* and the rule is *enabled*. The privileged node can execute the action part of the enabled rule; we say that it executes a rule. In this paper, we assume that the system is *semi-uniform*; namely, each node except the special node r has the same set of rules.

We use the term $configuration$ to refer to a vector of all nodes' states for representing the system status. Given a configuration c and its successor c', the transition from c to c' is called a *computation step*, denoted by $c \rightarrow c'$. During $c \rightarrow c'$, one or more privileged nodes in the configuration c concurrently execute rules and each of them executes exactly one rule. After executing the rules, the system enters the configuration c' and the next computation step starts. In this paper, we assume a system running under the *parallel model*. That is, we assume a *daemon* selecting an arbitrary non-empty subset of privileged nodes to execute rules during every computation step.

The computation of the system can be expressed by a series of configurations $(c_0, c_1, \ldots)$, where c_0 is an arbitrary initial configuration and each $c_k \rightarrow c_{k+1}$ is a computation step. We use $c_k \rightsquigarrow c_{k+m}$ to denote m consecutive computation steps, where $m > 0$ and $k \geq 0$. Given a configuration, its successor may not be unique, depending on how the daemon selects privileged nodes. A self-stabilizing system must guarantee that it eventually reaches a legal configuration c_ℓ from any possible initial configuration c_0; that is, $c_0 \rightsquigarrow c_\ell$, where ℓ is a finite integer. This requirement is called *convergence*. Another requirement of self-stabilization is called *closure*: Given a legal configuration, its successor is also legal.

For the sake of simplicity, we use *round* instead of computation step to explain how the system converges to a legal configuration. Starting from a configuration c_k, a round is the least consecutive

Variables:
P: Parent pointer
$D \in \{F, B\}$
$C \in \{0, 1, 2\}$

For the root node r, $P.r = r$ and $D.r = F$.
$\mathbb{R}0$: $(\forall j \in Child.r : D.j = B \wedge C.j = C.r) \rightarrow C.r = C.r + 1$;

For $i \neq r$:
$\mathbb{R}1$: $(D.i = F) \wedge (\forall j \in Child.i : D.j = B \wedge C.j = C.i) \rightarrow D.i = B$;
$\mathbb{R}2$: $(P.i \neq nil) \wedge (D.i = B) \wedge (D.P.i = F) \wedge (C.i \neq C.P.i) \rightarrow D.i = F$; $C.i = C.P.i$;

Figure 1: The token circulation for a static tree rooted at r

computation steps $c_k \rightsquigarrow c_{k+m}$ such that every privileged node in c_k has executed one or more rules when the system is in c_{k+m}. The first round starts from c_0, and its ending configuration is the beginning of the second round, ..., and so on. By this definition, the time complexity is the number of rounds converging to the first legal configuration in the worst case.

3 The Algorithm

In this section, we develop a phase synchronization algorithm for semi-uniform systems under the parallel execution model. Our idea is to construct a spanning tree rooted at the special node r. The node r is responsible for initiating a new phase when it detects the end of the last phase. Any other node simply copies the phase of its parent; thus the new phase is propagated in a top-to-down manner and eventually all nodes proceed to the new phase.

To realize the above idea, we define a conceptual object called token circulating along tree edges only. (An edge is an tree edge if one of the endpoints is the other's parent.) There are two types of tokens: *forward tokens* and *backward tokens*. Forward tokens travel the tree from the root to the leaf nodes, while backward tokens travel reversely. During traveling, forward tokens help (1) propagate the current phase, and (2) construct the spanning tree. On the other hand, backward tokens help the root node know when to initiate a new phase, but they don't have actual effects on tree construction.

The proposed algorithm is developed on the basis of token circulation mechanism adapted from [10], which is originally designed for a static tree rooted at r. Fig. 1 shows the three rules of the token circulation mechanism, in which $P.i$ is the parent of node i and $Child.i = \{j | P.j = i\}$ stands for the set of i's children nodes. In addition to the pointer variable P, every node keeps two scalar variables D and C. The variable D stands for the token's direction and its value is either B (Backward) or F (Forward). The variable C stands for the node's color and its value is 0, 1, or 2. Throughout this paper, the arithmetic operations on C are assumed to be under modulo 3 and such predicates as $(\forall j \in Child.r : D.j = B \wedge C.j = C.r)$ are assumed to be true if $Child.r = \emptyset$. A token is assumed to have the same color with its owner. A non-r node i is said to own a forward token if $(D.i = B) \wedge (D.P.i = F) \wedge (C.i \neq C.P.i)$, and it is said to receive a backward token from its child j if $(D.i = F) \wedge (D.j = B) \wedge (C.i = C.j)$. For the root node r, the definition of receiving a backward token is the same as that of a non-r node. However, the node r is assumed to have a forward token once it receives backward tokens from all of its children.

Consider the following *perfect state*: $\forall i \neq r : (C.i = C.r) \wedge (D.i = B) \wedge (D.r = F)$, in which only r has a forward token. From the perfect state, tokens circulate the tree as follows: By executing $\mathbb{R}0$, the root node r changes its color to $C.r + 1$, and propagates a forward token with this color to each of its children. When a non-r node receives a forward token, it executes $\mathbb{R}2$ to copy the parent's color and passes one forward token with the new color to each of its children. When a leaf

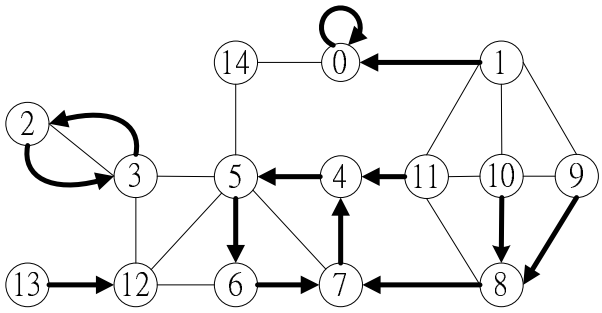

Figure 2: An example of connected components

node receives a forward token, it will execute $\mathbb{R}2$ and then $\mathbb{R}1$. The forward token thus becomes a backward token and travels back to the root. When a non-r node receives backward tokens from all of its children, it merges those tokens into one and passes the backward token to its parent by executing $\mathbb{R}1$. Once the node r receives backward tokens from all of its children, a period of token circulation is assumed to be finished and the system enters another perfect state. Afterwards, the root node will initiate a new token circulation. Please note that during the time of the token circulation, the colors of nodes are either $C.r$ or $C.r - 1$. Therefore, the color variable C can be viewed as a kind of phase variable and Fig. 1 is actually a 3-phase synchronizer.

Because the network topology in this paper is a general graph instead of a tree, we define that the root node r always points to itself and that any other node points to its neighbor or nil. By this setting, the system has three kinds of connected components: *R-tree*, *O-tree* and *Nil-tree*. The R-tree is the tree rooted at the node r; an O-tree contains a cycle and branches pointing to the cycle; a Nil-tree is a tree rooted at a node pointing to nil. A Nil-tree of single node is especially called an *isolated node*. We show an example in Fig. 2, in which r is labeled 0 and the arrows represent parent pointers. The set {0, 1}* (the symbol * stands for the inclusion of the attached arrow edges) is the R-tree; {2, 3}* and {4, 5, 6, 7, 8, 9, 10, 11}* are two O-trees in which {8, 9, 10}* and {11}* are branches; {12, 13}* is a Nil-tree; node 14 is an isolated node. Note that the labels in Fig. 2 are only used for illustration since our algorithms don't rely on node IDs.

As shown in [10], when we apply the token circulation mechanism in Fig. 1 to a tree network, such as the R-tree, the system eventually reaches the perfect state from any arbitrary initial state and then tokens circulate the system infinitely often. However, when we apply the mechanism to an O-tree or a Nil-tree, there will be no token eventually. This is because there is no root node r generating and propagating tokens in O-trees/Nil-trees. In terms of phases, nodes in the R-tree keep changing their phases, whereas no node in O-trees/Nil-trees can change its phase. As will be shown later, this asymmetric property is useful to determine whether a node is in the R-tree.

Below, we start to develop the rules for constructing a spanning tree. The basic idea is to break O-trees to be Nil-trees and then to be isolated nodes, and isolated nodes then join the R-tree. Our solution requires a node to know whether its neighbor has a forward token or not, so the range of the variable D is extended to be $\{FT, F\}$ for the root node r and to be $\{FT, F, B\}$ for every non-r node. When $D.i = FT$ holds, it means that i owns a token of the direction "Forward". Before passing a forward token, a node first sets $D = FT$ and renews its color. Afterwards, the node sets $D = F$ and the token is sent out. Due to this setting, $\mathbb{R}0$ is divided into two rules (a) and (b):

(a) $(D.r = F) \wedge (\forall j \in Child.r : D.j = B \wedge C.j = C.r) \rightarrow D.r = FT; C.r = C.r + 1;$
(b) $(D.r = FT) \rightarrow D.r = F;$

Since the variable P of a non-r node i may point to nil, rule $\mathbb{R}1$ becomes rule (c) by adding the condition $(P.i \neq nil)$ into the guard part. On the other hand, $\mathbb{R}2$ becomes two rules (d) and (e) as $\mathbb{R}0$ does.

(c) $(P.i \neq nil) \wedge (D.i = F) \wedge (\forall j \in Child.i : D.j = B \wedge C.j = C.i) \rightarrow D.i = B$;
(d) $(P.i \neq nil) \wedge (D.i = B) \wedge (D.P.i = F) \wedge (C.i \neq C.P.i) \rightarrow D.i = FT; C.i = C.P.i$;
(e) $(D.i = FT) \rightarrow D.i = F$;

Now, we explain how to use forward tokens to break O-trees. As we mentioned above, during the token circulation with color $C.r$ in the R-tree, the colors of nodes should be either $C.r$ or $C.r - 1$. Also recall the asymmetric property that there is always a token in the R-tree, whereas there will be no token in O-trees eventually. Thus, if a node i detects that one of its neighbors j owns a forward token of color $C.i + 2$, then i is aware that j is in the R-tree and itself is in an O-tree/Nil-tree. For such a case, node i should set $P.i = nil$ to break the O-tree/Nil-tree. Let $N.i$ denote the set of i's neighbors. We thus have the following rule:

(f) $(P.i \neq nil) \wedge (\exists j \in N.i : D.j = FT \wedge C.j = C.i + 2) \rightarrow P.i = nil$;

By rule (f), an O-tree is broken to be a Nil-tree. The next thing is to break the Nil-tree to be isolated nodes. This task is easy to achieve because a Nil-tree can collapse in a top-to-down manner without the help of tokens:

(g) $(P.i \neq nil) \wedge (P.P.i = nil) \rightarrow P.i = nil$;

The last step is to make isolated nodes join the R-tree with the help of forward tokens:

(h) $(P.i = nil) \wedge (Child.i = \emptyset) \wedge (\exists j \in N.i : D.j = FT \wedge P.j \neq nil) \rightarrow P.i = j$;

Below, we discuss the issues caused by an adversary daemon. Let j be a node not in the R-tree such that j and r are neighbors. When j evaluates the guard of rule (f) true, r must also evaluate the guard of rule (b) true at the same time. If r takes a move earlier than j does, j's privilege vanishes. An adversary daemon can make this always happen to prevent j from executing rule (f). Therefore, we must modify (b) to demand node r to wait until j takes a move.

(b*) $(D.r = FT) \wedge (\forall j \in N.i : C.j \neq C.r + 1) \rightarrow D.r = F$;

Similarly, rule (e) should be modified to be (e*):

(e*) $(D.i = FT) \wedge (\forall j \in N.i : C.j \neq C.i + 1) \rightarrow D.i = F$;

A Nil-tree root node i with $(D.i = FT)$ should reset $D.i$ unconditionally. Thus rule (e*) is modified to be rule (e**):

(e**) $(D.i = FT) \wedge ((\forall j \in N.i : C.j \neq C.i + 1) \vee (P.i = nil)) \rightarrow D.i = F$;

The last issue is to guarantee no disturbance in token circulation even when isolated nodes join the R-tree. Therefore, rule (h) should be further modified: When i sets $P.i = j$, it also sets $D.i = B$ and $C.i = C.j$, as if i has already received a token of color $C.j$ in this period of token circulation.

(h*) $(P.i = nil) \wedge (Child.i = \emptyset) \wedge (\exists j \in N.i : D.j = FT \wedge P.j \neq nil) \rightarrow P.i = j; D.i = B; C.i = C.j$;

The above rules are sufficient to build a spanning tree as well as a 3-phase synchronizer by the variable C. To extend the rules to be a K-phase synchronizer, each node maintain another variable $H \in \{0, 1, ..., K-1\}, K > 1$ to keep track of its phase. We add $H.r = H.r + 1 \mod K$ to the action part of rule (a) and add $H.i = H.P.i$ to that of rules (d) and (h*). The guard parts of all the rules remain unchanged. That is, a node updates its phase variable H whenever it changes its color. For this reason, the system satisfies criterion 1 as soon as the spanning tree is constructed. In Fig. 3, we provide a state transition diagram for a non-r node i in legal configurations to help readers better understand how the algorithm runs. In the diagram, the text attached to an arrow indicates the execution of the nodes. For example, the text "$i, \{R3, R4\}$" stands for node i executing R3 and then R4.

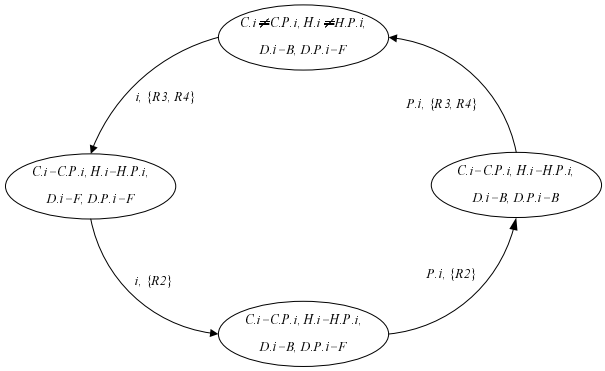

Figure 3: The state transition diagram for a non-r node i in legal configurations.

Variables:
P: parent pointer
$C \in \{0, 1, 2\}$ // for color
$D \in \{FT, F, B\}$ // for direction
$H \in \{0, 1, .., K-1\}$ // for phase

For the root node r: $P.r = r$ and $D.r \in \{FT, F\}$
R0: $(D.r = F) \wedge (\forall j \in Child.r : D.j = B \wedge C.j = C.r) \rightarrow D.r = FT; C.r = C.r + 1; H.r = H.r + 1;$
R1: $(D.r = FT) \wedge (\forall j \in N.r : C.j \neq C.r + 1) \rightarrow D.r = F;$

For $i \neq r$:
R2: $(P.i \neq nil) \wedge (D.i = F) \wedge (\forall j \in Child.i : D.j = B \wedge C.j = C.i) \rightarrow D.i = B;$
R3: $(P.i \neq nil) \wedge (D.i = B) \wedge (D.P.i = F) \wedge (C.i \neq C.P.i) \rightarrow D.i = FT; C.i = C.P.i; H.i = H.P.i;$
R4: $(D.i = FT) \wedge ((\forall j \in N.i : C.j \neq C.i + 1) \vee (P.i = nil)) \rightarrow D.i = F;$
R5: $(P.i \neq nil) \wedge ((\exists j \in N.i : D.j = FT \wedge C.j = C.i + 2) \vee (P.P.i = nil)) \rightarrow P.i = nil;$
R6: $(P.i = nil) \wedge (Child.i = \emptyset) \wedge (\exists j \in N.i : D.j = FT \wedge P.j \neq nil) \rightarrow P.i = j; D.i = B; C.i = C.j; H.i = H.j;$

Figure 4: The proposed algorithm.

All the rules mentioned above constitute our algorithm, which is listed in Fig. 4. The root node r has two rules (a) and (b*), corresponding to R0 and R1 respectively. For non-r nodes, the rules (f) and (g) are combined into one rule R5, so it has five rules: R2 to R6. We assume that each rule has a priority and a rule with a smaller number has a higher priority. As readers can check, the memory usage is quite low and is independent of system size. Each node keeps a pointer variable P, a phase variable H, and two scalar variables of totally 6 (resp., 9) states for the node r (resp., for a non-r node.) Let Δ denote the maximum degree of the graph; the variable P requires $O(\log \Delta)$ bits. Combining the the number bits for the variables H, D, and C, the space complexity per node is $O(\log \Delta + \log K)$.

4 Correctness and time complexity analysis

In this subsection, we show that the system stabilizes in $O(n^2)$ rounds, regardless of any arbitrary initial configuration. Let $n = |V|$ denote the number of nodes in the system; we first define the legal configuration as below.

Definition 1. *(Legal Configuration)*
A configuration is legal if it satisfies the following three conditions:
(1) $\forall i \neq r : (C.i = C.r) \wedge (D.i = B)$.
(2) $D.r = F$.

(3) the number of nodes in the R-tree is n.
Furthermore, any configuration that arises from the one satisfying (1), (2) and (3) by the completion of one or more moves is also a legal configuration.

Before showing that the system eventually reaches a legal configuration, we must guarantee that at least one node is privileged for any arbitrary configuration. In other words, the system is never deadlocked.

Lemma 1. *For any configuration, at least one node is privileged.*

Proof. We prove this lemma by contradiction. Assume that no node is privileged. According to the value of the variable D, there are two cases to be considered:
Case (1) every node has either $D = B$ or $D = F$:

Let $h(i) \geq 1$ be the height of a node i in the R-tree. We first use induction on $h(i)$ to show that every non-r node i in the R-tree has $D.i = B$. (Basis) Since a leaf node i ($h(i) = 1$) cannot execute R2, we have $D.i = B$. (Hypothesis) $D.j = B$ for any non-r node j with $h(j) < \lambda$. (Induction) Let i be a non-r node with $h(i) = \lambda$ and j be a child of i. We have $D.j = B$ by the hypothesis. Assume that $D.i = F$. If $C.j \neq C.i$ for some node j, then j can execute R3. If $C.j = C.i$ for any node j, node i can execute R2. Since no rule is enabled, the case of $D.i = F$ does not occur and we have $D.i = B$, as desired.

For the root node r, $D.r = F$ holds because the range of $D.r$ doesn't contain B. Since r cannot execute R0, it has a child j such that $C.j \neq C.r$, by which the node j can execute R3 due to $(P.j \neq nil) \wedge (D.j = B) \wedge (D.P.j = F) \wedge (C.j \neq C.P.j)$. Contradiction occurs.
Case (2) some node i has $D.i = FT$:

Because node i cannot execute R4, we have $P.i \neq nil$ and i has a neighbor j with $C.j = C.i + 1$. We have two sub-cases to consider: (i) $P.j \neq nil$: j can execute R5 because $(D.i = FT) \wedge (C.i = C.j - 1 = C.j + 2)$. (ii) $P.j = nil$: Since j cannot execute R6, we have $Child.j \neq \emptyset$. Node j's child k can execute R5 because $(P.k \neq nil) \wedge (P.P.k = P.j = nil)$. We get a contradiction for each sub-case. □

Below, we begin to prove the convergence property. We first define tokens as follows:

Definition 2. *(Forward Token)*
The root node r is said to own a forward token iff
$(D.r = F \wedge (\forall j \in Child.r : D.j = B \wedge C.r = C.j)) \vee (D.r = FT)$,
whereas a non-r node is said to own a forward token iff
$(P.i \neq nil) \wedge ((D.i = B \wedge D.P.i = F \wedge C.i \neq C.P.i) \vee D.i = FT)$.

Definition 3. *(Backward Token)*
A non-leaf node i is said to receive a backward token from its child j iff
$(D.i = F) \wedge (D.j = B) \wedge (C.i = C.j)$,
whereas a non-isolated leaf node i is said to have a backward token iff
$(D.i = F)$.

Our ultimate goal is to show that eventually there is a fixed spanning tree. To do this, we first prove some properties of tokens in lemmas 2, 3, 4, and 5. By these lemmas, we can infer the property of tokens in O-trees, Nil-trees, and the R-tree, as shown in lemmas 6, 7 and 8, respectively. Finally, lemmas 9 and 10 show how the system converge to the legal configuration.

Lemma 2. *In fixed components(R-tree, O-trees, or Nil-trees), a non-r node does not receive contiguous forward and contiguous backward tokens.*

Proof. We first show that a non-r node i never receives contiguous forward tokens. By definition, when node i owns a forward token, either $D.i = B$ or $D.i = FT$ holds and it can execute R3 and then R4 (in case of $D.i = B$), or simply execute R4 (in case of $D.i = FT$) to pass the token to its children. After the token passing, its status is $D.i = F$. Before owning a forward token again, node i has to execute R2 so that the value of $D.i$ becomes B from F. Since the guard of R2 implies the possession of (at least) a backward token, it means that node i must own (at least) a backward token before getting a forward token again. That is, node i never receives contiguous forward tokens.

Based on the same strategy, we can also prove that node i never receives contiguous backward tokens, so this proof is skipped. □

Lemma 3. *Once a forward token meets a backward token, one of them disappears.*

Proof. Consider two nodes i and j, $P.j = i$ and $P.i \neq nil$, such that j can execute R2 and i can execute R0 (if $i = r$) or R3 (if $i \neq r$). By definition, node j has a backward token, whereas node i has a forward token. We prove this lemma by checking how many tokens are left after the node pass the tokens.

Because the tokens meet by node i or node j or both executing the rules, we have the following three cases to consider:

Case (1) Only j passes the backward token by executing R2:

For this case, node j is of $D.j = B$ so the backward token disappears. On the other hand, node i still holds the forward token.

Case (2) Only i passes the forward token by executing R0 and R1 (or R3 and R4):

For this case, node i is of $D.i = F$ so the forward token disappears. On the other hand, node j still holds the backward token.

Case (3) Both i and j pass tokens:

After j executes R2 and i executes R0 and then R1 (or R3 and then R4), we have $D.j = B$, $D.i = F$ and $C.i = C.P.i$. According to the relation between $C.i$ and $C.j$, we have two sub-cases to consider: (i) $C.i = C.j$: Node i has $(D.j = B) \wedge (D.i = F) \wedge (C.i = C.j)$, so it has a backward token coming from j. On the other hand, node j does not own the forward token. (ii) $C.i \neq C.j$: In this case, node j has $(D.j = B) \wedge (D.P.j = F) \wedge (C.j \neq C.P.j)$ so it has a forward token. On the other hand, node i does not receive the backward token coming from j.

Because either the forward token or the backward token disappears in each case, this lemma holds. □

For normal token circulation, tokens bounce between the root node r and leaf nodes. That is, a backward token should become a forward token when it arrives the node r, whereas a forward token should become a backward token when it arrives a leaf node. However, in the beginning a token may change its direction at an internal node because of the unpredictable initial configuration. And we say that an internal node i performs an *illegal forward (resp., backward) token reverse* if it receives a forward (resp., backward) token but sends out a backward (resp., forward) token. In terms of rules, if node i perform an illegal forward token reverse, it executes R3, R4, and R2 consecutively. (Note that node i can execute R2 right after executing R4, but in normal situations it cannot do so.) Similarly, if node i perform an illegal backward token reverse, it executes R2, R3 and R4 consecutively.

Lemma 4. *Eventually no internal node can perform illegal forward or backward token reverse.*

Proof. To prove this lemma, we show that an internal node i can perform at most once illegal reverse of a forward token and of a backward token respectively.

Consider the case that node i performing an illegal forward token reverse. By definition, it executes R3, R4 and R2 consecutively. After executing the three rules, node i has $(D.i = B) \wedge (\forall j \in$

$Child.i : C.j = C.i \wedge D.j = B$). Let j be a child of i. The next time i receives a forward token, node i has $C.i \neq C.P.i$. After i executes R3 and R4 to pass the forward token, the condition $C.j \neq C.i$ holds so it cannot execute R2 immediately. That is, node i cannot reverse the forward token.

Now, consider the case that node i performing an illegal backward token reverse. Similarly, it means that node i executes R2, R3 and R4 consecutively. After executing the three rules, node i has $(D.i = F) \wedge (D.P.i = F) \wedge (C.i = C.P.i)$. The next time i owns a backward token and executes R2, it cannot execute R3 immediately because $C.i = C.P.i$ holds. That is, node i cannot reverse the backward token. □

According the proof of lemma 4, an illegal token reverse never occurs at a node having executed R2, R3, and R4, or never occurs at a node having received a token. By the fact that a token transfer from a node to the next one in $O(1)$ rounds, we have the following lemma:

Lemma 5. *After $O(n^2)$ rounds, no internal node can perform illegal forward or backward token reverse.*

Proof. We call a node an illegal bouncing point if it is an internal node that can perform an illegal token reverse. Similarly, we call a node a bouncing point if it is the root node r, a leaf node, or an illegal bouncing point. To prove this lemma, we show that no illegal bouncing point exists in $O(n^2)$ rounds. For the sake of simplicity, we assume that no token disappears.

Because tokens travel along tree edges only and they swing between bouncing points, in $O(n)$ rounds a token reaches a bouncing point. If that bouncing point is an internal node, according to lemma 4, the node no longer serve as a bouncing point. That is, every $O(n)$ rounds a token eliminates an illegal bouncing point, or arrives either the root node or a leaf node. Because there may be $O(n)$ illegal bouncing points in the initial configuration, it takes $O(n) \times O(n) = O(n^2)$ rounds to get rid of all of them, as desired. □

In the following three lemmas, we show the behavior of tokens in O-trees, Nil-trees, and the R-tree, respectively.

Lemma 6. *After $O(n^2)$ rounds, there will be no token in O-trees.*

Proof. To prove this lemma, we show that, for any O-tree, tokens in the branches go into the cycle in $O(n)$ rounds and then disappear in $O(n^2)$ rounds. With the help of lemma 5, we assume that no illegal token reverse would occur.

First, focus on the tokens in the O-tree branches. In such components, a forward token becomes a backward when it arrives a leaf node and the backward token either goes into the O-tree cycle or disappears. The time complexity for this is $O(n)$ rounds, including $O(n)$ rounds for a forward token to become a backward token and another $O(n)$ rounds for the backward token to go into the cycle.

Now, consider the tokens in the O-tree cycle. According to lemma 3, the number of tokens decreases when two tokens of different directions meet; hence eventually the tokens in the cycle are of the same direction, either forward or backward. The time complexity for this is $O(n) \times O(n) = O(n^2)$ rounds because there may be $O(n)$ tokens in the cycle and two tokens of different types meet in $O(n)$ rounds. Afterwards, these survival tokens disappear in $O(n)$ rounds, since a node never continuously receive tokens of the same direction, according to lemma 2. In summary, all the tokens in the O-tree cycle disappear in $O(n^2)$ rounds, as desired. □

Lemma 7. *After $O(n)$ rounds, there will be no token in Nil-trees.*

Proof. By definition, an isolated has no token. Therefore, we prove this lemma by showing that any Nil-tree becomes a set of isolated nodes in $O(n)$ rounds.

According to R5, a child of a Nil-tree root can point to nil, so the Nil-tree's height decreases by one every $O(1)$ rounds. Combining the fact that the tree height is $O(n)$, the Nil-tree becomes a set of isolated nodes in $O(n)$ rounds. □

Below, we show that the R-tree eventually reaches the *perfect state*; viz. $D.r = F$ and any non-r node i in the R-tree has $(C.i = C.r) \wedge (D.i = B)$.

Lemma 8. *After $O(n^2)$ rounds, the R-tree reaches the perfect state.*

Proof. To prove this lemma, we first show that in $O(n^2)$ rounds only one token exists in each tree path from a leaf node to the root node r. Afterwards, the R-tree enters the perfect state in $O(n)$ rounds. Similar to lemma 6, we assume that no illegal token reverse would occur.

Consider a tree path from a leaf node to the root node r. By the proof of lemma 1, there is at least one token in this path. Let the number of tokens in this path be $O(n)$. Because the tokens bounce between the root node and the leaf node, they meet one another in $O(n)$ rounds. By lemma 3, it means that the number of tokens decreases by one every $O(n)$ rounds, or, equivalently, decreases to one in $O(n) \times O(n) = O(n^2)$ rounds. If the last survival token is forward, it reaches the leaf node in $O(n)$ rounds and becomes a backward token traveling back to the root node.

Now we consider the configuration in which there is exactly one backward token in any tree path from a leaf node to the root node. For a non-r node i receiving all of the backward tokens from its children, it executes R2 to pass the merged backward token to its parent. After the execution, node i has $D.i = B \wedge C.j = C.i \wedge D.j = B$, where $j \in Child.i$. Since every $O(1)$ rounds a backward token moves from the node of height k to the node of height $k+1$, the root node receives backward tokens from all the children in $O(n)$ rounds. By that time, the root node is of $D.r = F$ and any non-r node i in the R-tree is of $D.i = B \wedge C.i = C.P.i = C.r$. That is, the R-tree is in the perfect state.

According to the above proof, the R-tree enters the perfect state in $O(n^2)$ rounds. □

Lemma 9. *Once the R-tree reaches the perfect state and there is no token in O-trees/Nil-trees, the number of nodes in the R-tree is monotonically increasing.*

Proof. To prove this lemma, we show that a node i in the R-tree does not execute R5 to depart from the R-tree. Let j be a neighbor of i. Our attempt is to show that $C.j \neq C.i - 2$ holds when $D.j = FT$ holds. This node j must be in the R-tree; otherwise $D.j = FT$ cannot hold since there is no token in O-trees/Nil-trees. Below, we consider a token circulation in the R-tree, observe how the color C changes, and prove the desired property: $C.j \neq C.j - 2$.

Let's consider a token circulation starting from the perfect state, in which every node has the same color $C.r = \alpha - 1$. During this token circulation, the root node r executes exactly two rules R0 and R1, and any other R-tree node executes exactly three rules R2, R3, and R4. Because a node changes its color to be α only when it executes R0 or R3, and because these two rules set $D = FT$ as well, the condition $C.j = \alpha$ must hold when $D.j = FT$ holds. For node i, its color is either $C.i = \alpha$ or $C.i = \alpha - 1$. It is easy to check that $C.j \neq C.i - 2$, as desired. □

Lemma 10. *Once the R-tree reaches the perfect state and there is no token in O-trees/Nil-trees, the R-tree spans all the nodes in $O(n^2)$ rounds.*

Proof. Let i and j be two adjacent nodes such that j is in the R-tree, while i is not. We can prove this lemma by showing that node i joins the R-tree in $O(n)$ rounds.

Consider the token circulations in the R-tree. During a token circulation propagating color 0, node j is of $D.j = FT$ and $C.j = 0$ at some time by executing R0 (if $j = r$) or R3 (if $j \neq r$). Similarly, during the token circulations propagating color 1 and color 2, node j is of $(D.j = FT) \wedge (C.j = 1)$ and $(D.j = FT) \wedge (C.j = 2)$ at some point, respectively. Because node

i has no token and thus cannot change $C.i$, the condition $(D.j = FT) \wedge (C.j = C.i + 2)$ holds at some time within three consecutive token circulations. Since each token circulation finishes in $O(n)$ rounds, this condition holds within $3 * O(n)$ rounds. The next step is to prove that node i points to node j within $O(1)$ rounds when this condition holds.

By the components where i locates, there are three cases to consider:
Case (1) i is an isolated node:
Node i executes R6 to set $P.i = j$ to join the R-tree.
Case (2) i is in an O-tree:
Node i executes R5 to set $P.i = nil$ and becomes a Nil-tree root. Then its children, if any, execute R5 so node i becomes an isolated node. The remaining proof of this case is similar to that of Case (1).
Case (3) i is in a Nil-tree containing more than one node:
The proof of this case is similar to that of Case (2).

The actions in all the three cases take $O(1)$ rounds, so node i becomes a part of the R-tree in $O(1)$ rounds when $(D.j = FT) \wedge (C.j = C.i + 2)$ holds. Because this condition holds in $O(n)$ rounds, the number of nodes in the R-tree increases by one every $O(n)$ rounds, until the R-tree spans all the nodes. Thus the overall time complexity is $O(n) \times O(n) = O(n^2)$ rounds. □

Theorem 1. *The system enters legal configurations in $O(n^2)$ rounds. (convergence)*

Proof. This is a direct consequence of lemmas 5, 6, 7, 8, 9 and 10. □

Theorem 2. *Once the system is in a legal configuration, it remains so henceforth. (closure)*

Proof. Note that criteria (1) and (2) in definition 1 are the conditions of perfect states. Therefore, this theorem is a direct consequence of lemma 9. □

5 Conclusion

We propose a self-stabilizing algorithm for the phase synchronization problem for asynchronous system of general graph topologies. The algorithm runs under the parallel model and constructs a spanning tree rooted at the unique special node r that is responsible for initiating a new phase. To the best of our knowledge, it is the first such algorithm for general graphs. Its another advantage is the low space complexity: $O(\log \Delta + \log K)$ bits per node, where Δ is the maximum degree of the graph and $K > 1$ is the number of phases. Moreover, it can be refined to be a spanning tree construction algorithm and a 3-phase synchronizer by removing the phase variable H.

In our algorithm, we assume a semi-uniform system. This assumption is for constructing a spanning tree in a deterministic way, but it may be unnecessary for a phase synchronizer. Therefore, it is an open problem of how to develop a deterministic, memory-efficient phase synchronizer for uniform systems.

References

[1] L. O. Alima, J. Beauquier, A. K. Datta, and S. Tixeuil. Self-stabilization with global rooted synchronizers. In *Proceedings of the 18th International Conference on Distributed Computing Systems*, pages 102–109, 1998.

[2] A. Arora, S. Dolev, and M. G. Gouda. Maintaining digital clocks in step. *Parallel Processing Letters*, 1:11–18, 1991.

[3] E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644, 1974.

[4] S. Dolev. Possible and impossible self-stabilizing digital clock synchronization in general graphs. *Real-Time Systems*, 12(1):95–107, 1997.

[5] F. C. Gärtner. A Survey of Self-Stabilizing Spanning-Tree Construction Algorithms. Technical report, Swiss Federal Institution of Technology, 2003.

[6] M. G. Gouda and T. Herman. Stabilizing unison. *Information Processing Letters*, 35:171–175, 1990.

[7] S. T. Huang and T. J. Liu. Self-stabilizing 2^m-clock for unidirectional rings of odd size. *Distributed Computing*, 12:41–46, 1999.

[8] S. T. Huang and T. J. Liu. Phase synchronization on asynchronous uniform rings with odd size. *IEEE Transactions on Parallel and Distributed System*, 12(6):638–652, 2001.

[9] S. T. Huang, T. J. Liu, and S. S. Hung. Asynchronous phase synchronization in uniform unidirectional rings. *IEEE Transactions on Parallel and Distributed System*, 15(4):378–384, 2004.

[10] H. S. M. Kruijer. Self-stabilization(in spite of distributed control) in tree-structured systems. *Information Processing Letters*, 8(2):91–95, 1979.

[11] S. Kulkarni and A. Arora. Fine-grain multitolerant barrier synchronization. Technical report, Technical Report OSU-CISRC TR34, Ohio State University, 1997.

[12] S. Kulkarni and A. Arora. Multitolerant barrier synchronization. *Information Processing Letters*, 64(1):29–36, 1997.