A Self-Stabilizing Synchronization Protocol For Arbitrary Digraphs

(*A Self-Stabilizing Distributed Clock Synchronization Protocol For Arbitrary Digraphs*)

Mahyar R. Malekpour
Langley Research Center, Hampton, Virginia
mahyar.r.malekpour@nasa.gov

Abstract— This paper presents a self-stabilizing distributed clock synchronization protocol in the absence of faults in the system. It is focused on the distributed clock synchronization of an arbitrary, non-partitioned digraph ranging from fully connected to 1-connected networks of nodes while allowing for differences in the network elements. This protocol does not rely on assumptions about the initial state of the system, other than the presence of at least one node, and no central clock or a centrally generated signal, pulse, or message is used. Nodes are anonymous, i.e., they do not have unique identities. There is no theoretical limit on the maximum number of participating nodes. The only constraint on the behavior of the node is that the interactions with other nodes are restricted to defined links and interfaces. This protocol deterministically converges within a time bound that is a linear function of the self-stabilization period. We present an outline of a deductive proof of the correctness of the protocol. A bounded model of the protocol was mechanically verified for a variety of topologies. Results of the mechanical proof of the correctness of the protocol are provided. The model checking results have verified the correctness of the protocol as they apply to the networks with unidirectional and bidirectional links. In addition, the results confirm the claims of determinism and linear convergence. As a result, we conjecture that the protocol solves the general case of this problem. We also present several variations of the protocol and discuss that this synchronization protocol is indeed an emergent system.

Keywords: self-stabilizing, arbitrary, digraph, emergent system, distributed, clock, synchronization

I. INTRODUCTION

How can a distributed system solve a problem that is inherently global by executing a set of rules locally? For millennia people have witnessed in awe flocks of birds fly in unison, hundreds of frogs croak in harmony and thousands of fireflies flash in synchrony and wondered how such collective (or mass) synchrony comes about. In Southeast Asia, a large number of *Malaccae* fireflies routinely flash on and off in synchrony. Do these insects follow a leader or do they have an inherit sense of rhythm? This question was asked by George Hudson in 1918 [1][2]. The synchronization phenomenon, whether a natural occurrence or artificially induced, still fascinates us today and has become one of the most interesting scientific problems of our time. In a recent survey, Arenas [3] reports on the advances in the understanding of synchronization phenomena by oscillating elements in a complex network topology. He concludes that: "Synchronization processes are ubiquitous in nature and play a very important role in many different contexts as biology, ecology, climatology, sociology, technology, or even in arts." But, how does collective synchrony emerge from chaos? The answer to this question has intrigued mankind, in particular, some of the greatest minds of the twentieth century, including Albert Einstein, Richard Feynman, Norbert Wiener, Brian Josephson, Edward Lorenz, and Arthur Winfree. Many questions still persist today. Is synchrony inevitable? If so, how exactly does it happen? When and under what circumstances is it possible or impossible to achieve? What are the ramifications of either case? What are the theoretical and practical implications of either case?

Besides being an intellectual curiosity and a theoretical problem in computer science and engineering, synchronization has practical significance as a fundamental service for higher-level algorithms that solve other problems. For example, in safety-critical TDMA (Time Division Multiple Access) architectures [4][5][6][7], synchronization is the most crucial element of these systems.

Clock synchronization algorithms are essential for managing the use of resources and controlling communication in a distributed system. We define **synchronization** of a distributed system as the process of **achieving** and **maintaining** coordination among independent local clocks by exchanging local time information. We define **bounded-synchrony** as the exchange of local time information by the nodes in unison but within a given bound. True synchrony, as operating and exchanging messages in perfect unison, is only possible under strict assumptions and ideal conditions. Bounded-synchrony on the other hand is a more general term that encompasses imperfections in the network. Here, we use the term **synchrony** to mean bounded-synchrony.

Charlie Peskin [8] posed the self-organization idea around 1975 while working on cardiac pacemakers and, at about the same time, Edsger Dijkstra [9] presented the self-stabilization problem in a distributed system. These two scientists asked whether it would be possible for a set of oscillators or machines to self-organize and self-stabilize their collective behavior in spite of unknown initial conditions and distributed control.

A distributed system is defined to be self-stabilizing if, from an arbitrary state, it is guaranteed to reach a legitimate state in a finite amount of time and remain in a legitimate

state. A legitimate state is a state where all parts in the system are in synchrony. The self-stabilizing distributed-system clock synchronization problem is to develop an algorithm (i.e., a protocol) to *achieve* and *maintain* synchrony of local clocks in a distributed system after experiencing system-wide disruptions in the presence of network element imperfections. Hereafter in this paper, we use the term synchronization to mean self-stabilizing clock synchronization in distributed systems.

There is a vast literature on synchronization phenomenon exhibited by humans, animals, and even inanimate objects. There are also many proposed solutions for synchronization of a large number of entities based on models inspired by nature or abstract ideas. There exist many solutions for special cases and restricted conditions. Other solutions require embedding a directed spanning tree or rewiring the network in order to achieve synchrony [10][11][12][13]. In [14], a solution for the general case is presented, but a closer examination reveals that it only addresses maintaining synchronization (stability of stable in-phase synchronization) and not how to achieve it. In computer science and computer engineering terminology, stability is referred to as the closure property. The **convergence** and **closure** properties address *achieving* and *maintaining* network synchrony, respectively (see Section III.E for a formal definition of these parameters). There are many solutions that deal with the closure property [15][16][17] and either do not address convergence or provide an ad hoc solution [18] for initialization and integration, separately. Typically, the assumed topology is a regular[1] graph such as a fully connected graph or a ring. These topologies do not necessarily correspond to practical applications or biological, social, or technical networks. Furthermore, the existing models and solutions do not always achieve synchrony and, therefore, do not solve the general case of the distributed synchronization problem. Even when the solutions achieve synchrony, the time to achieve synchrony is very large for many of the solutions.

Another key factor in a proposed solution is whether or not it deals with faults. A **fault** is a defect or flaw in a system component resulting in an incorrect state [6][19][20]. Large-scale distributed systems have become an integral part of safety-critical computing applications, necessitating system designs that incorporate complex fault-tolerant resource management functions to provide globally coordinated operations with ultra-reliability. As a result, robust clock synchronization has become a required fundamental component of fault-tolerant safety-critical distributed systems. The requirement to handle faults adds a new dimension to the complexity of the synchronization of fault-tolerant distributed networks. Ultra-reliable distributed systems are designed to deal with variety of faults that reflect the desired degree of reliability of the system. We define the **fault spectrum** as a range of faults that span from no faulty nodes at one extreme end to arbitrary (Byzantine) faulty nodes at the other extreme end.

A fundamental property of a robust distributed system is the capability of tolerating and potentially recovering from failures that are not predictable in advance. In [15][21] various ideas for overcoming failures in a robust distributed system are addressed that include tolerating Byzantine faults. There are many algorithms that address permanent faults [16], where the issue of transient failures is either ignored or inadequately addressed. There are many efficient Byzantine clock synchronization algorithms proposed that are based on assumptions on initial synchrony of the nodes [16][17] or existence of a common pulse at the nodes, e.g., the first protocol in [22]. There are many clock synchronization algorithms that are based on randomization and, therefore, are non-deterministic, e.g., the second protocol in [22]. In [23] a counterexample is presented to a clock synchronization algorithm [24] that is based on the existence of a common pulse at the nodes.

Two Byzantine-fault-tolerant self-stabilizing protocols for distributed systems were reported in [25] and [26]. Instances of these protocols are demonstrated via mechanical verification to self-stabilize from any state, in the presence of at most one permanent Byzantine faulty node, and deterministically converge in linear time with respect to the synchronization period [27]. These protocols synchronize a fully connected network of two or more nodes in the absence of faults. These protocols, however, do not solve the general case of the problem in the presence of multiple Byzantine faults.

A thorough understanding of the synchronization of a distributed system has proven to be elusive for decades. The main challenges associated with distributed synchronization are the complexity of developing a solution and proving the correctness of the solution. It is possible to have a solution that is hard to prove or refute. Such a solution, however, is not likely to be accepted or used in practical systems. The proposed solutions must restore synchrony and coordinated operations after experiencing system-wide disruptions in the presence of network element imperfections and, for ultra-reliable distributed system, in the presence of various faults. In addition, a proposed solution must be proven to be correct. If a mathematical proof is deemed difficult, at a minimum, the proposed solution must be shown to be correct using available formal methods. Furthermore, addressing network element imperfections is necessary to make a solution applicable to realizable systems.

In this paper, we present a solution for an arbitrary, non-partitioned network (digraph) in the absence of faults. The networks range from fully connected to 1-connected networks of nodes, while allowing for differences in the network elements. Some networks of interest include grid, ring, fully connected, bipartite, and star (hub). We do not require any particular information flow nor imposes changes to the network in order to achieve synchrony. However, we focus on one extreme of the fault spectrum and only consider

[1] A regular graph is a graph where each vertex has the same number of neighbors, i.e., every vertex has the same degree or valency. A regular graph with vertices of degree k is called a k-regular graph or regular graph of degree k.

distributed systems in the absence of faults. The assumption of an absence of faults is equivalent to the assumption that all faults are detectable. This departure from the Byzantine extreme of the fault spectrum is in part because of the niche use and the extra cost associated with the Byzantine faults. Also, using authentication and error detection techniques, it is possible to substantially reduce the effects of variety of faults in the system. Furthermore, the classical definition of a self-stabilizing algorithm assumes generally that there are no faults in the system. In other words, we wanted to search for a general solution in the absence of faults before attempting to solve the problem in the presence of various faults [28].

In Section II of this paper, we provide a system overview. We present the protocol description in section III. In Section IV we present results of mechanical proof of the protocol via model checking. In Section V we discuss variations of the protocol including the general case of the protocol that encompasses dynamic node count and dynamic topology. Finally, we present concluding remarks in Section VI.

II. System Overview

We consider a system of pulse-coupled entities (e.g., oscillators, pacemaker cells) pulsating periodically at regular time intervals. These entities are said to be coupled through some physical means (wire or fiber cables, chemical process, or wirelessly through air or vacuum) that allows them to influence each other. We model the system as a set of nodes that represent the pulse-coupled entities and a set of communication links that represent their interconnectivity.

The underlying topology considered is an arbitrary, non-partitioned digraph ranging from fully connected to 1-connected network of $K \geq 1$ nodes that exchange messages through a set of communication links. Nodes are anonymous, i.e., they do not have unique identities. All nodes are assumed to be good, i.e., actively participate in the synchronization process and correctly execute the protocol. The communication links are assumed to connect a set of source nodes to a set of destination nodes with a source node being different than a destination node. All communication links are assumed to be good, i.e., reliably transfer data from their source nodes to their destination nodes. The nodes communicate with each other by exchanging broadcast messages. Broadcast of a message by a node is realized by transmitting the message, at the same time, to all nodes that are directly connected to it. The communication network does not guarantee any relative order of arrival of a broadcast message at the receiving nodes, that is, a consistent delivery order of a set of messages does not necessarily reflect the temporal or causal order of the message transmissions [4]. There is neither a central system clock nor an externally generated global pulse or message at the network level. The communication links and nodes can behave arbitrarily provided that eventually the system adheres to the protocol assumptions (Section III.E).

A. Drift Rate (ρ)

Each node is driven by an independent, free-running local physical oscillator (i.e., the phase is not controlled in any way) and a logical-time clock (i.e., a counter), denoted *LocalTimer*, which locally keeps track of the passage of time and is driven by the local physical oscillator. An **oscillator tick**, also called a **clock tick** or a **system tick**, is a discrete value and the basic unit of time in the network [6].

An ideal oscillator has zero drift rate with respect to real-time, perfectly marking the passage of time. Real oscillators are characterized by non-zero drift rates with respect to real-time. The oscillators of the nodes are assumed to have a known bounded drift rate, ρ, which is a small constant with respect to real-time, where ρ is a unitless non-negative real value and is constrained to $0 \leq \rho << 1$. The maximum drift of the fastest *LocalTimer* over a time interval of t is given by $(1+\rho)t$. The maximum drift of the slowest *LocalTimer* over a time interval of t is given by $(1/(1+\rho))t$. Therefore, the **maximum relative drift** of the fastest and slowest nodes with respect to each other over a time interval of t is given by $\delta(t) = ((1+\rho) - 1/(1+\rho))t$.

B. The Logical Clock (LocalTimer)

The *LocalTimer* is driven by the local physical oscillator, takes on discrete values, and locally keeps track of the passage of time. As shown in Figure 1, the *LocalTimer* is a monotonic linear function increasing from an initial value to a maximum value. If uninterrupted, i.e., when the node does not receive any messages from other nodes, the *LocalTimer* periodically takes on all integer values from its initial value, 0, to its maximum value, P, linearly increasing within each period, i.e., the *LocalTimer* is bounded by $0 \leq LocalTimer \leq P$.

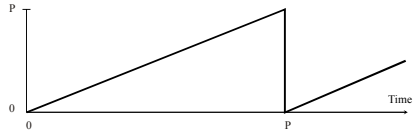

Figure 1. The *LocalTimer*.

C. Communication Delay (D, d, and γ)

The communication delay between adjacent nodes is expressed in terms of the minimum event-response delay, D, and network imprecision, d. These parameters have units of real time clock ticks and are described with the help of Figure 2. As depicted in this figure, a message transmitted by a node at real time t_0 is expected to arrive at its directly connect adjacent nodes, be processed, and subsequent messages generated within the time interval of $[t_0+D, t_0+D+d]$.

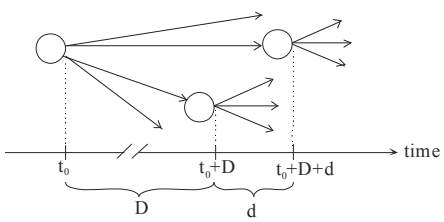

Figure 2. Event-response delay, D, and network imprecision, d.

Communication between independently clocked nodes is inherently imprecise. The network imprecision, d, is the maximum time difference among all receivers of a message from a transmitting node with respect to real time. The imprecision is due to the drift of the oscillators with respect to real time, jitter, discretization error, temperature effects and differences in the lengths of the physical communication media. These two parameters are assumed to be bounded such that $D \geq 1$ and $d \geq 0$ and both have units of real time clock ticks. The communication latency, denoted γ is defined as $\gamma = (D+d)$ and has units of real time clock ticks. The communication delay between any two adjacent nodes is bounded by $[D, \gamma]$.

D. Topology (T)

A communication link, or simply link, is an edge in the graph representing a direct physical connection between two nodes. A path is a logical connection between two nodes consisting of one or more links. A path-length is the number of links connecting any two nodes. The general topology, T, considered is a strongly connected directed graph (digraph) consisting of K nodes, where each node is connected to the graph by at least one link, there is a path from any node to any other node, and the links are either unidirectional or bidirectional. Furthermore, we assume there is no direct link from a node to itself, i.e., no self-loop, and there are no multiple links directly connecting any two nodes in any one direction.

In this paper, we use the terms network and graph interchangeably. The following terms are used in the subsequent sections.

- Two nodes are said to be **adjacent** to each other or neighbors if they are connected to each other via a direct communication link.
- L, an integer value, number of links, denotes the largest **loop** in the graph, i.e., the maximum value of the longest path-lengths from a node back to itself visiting the nodes along the path only once (except for the first node which is also the last node).
- W, an integer value, number of links, signifies the **width** or diameter of the graph, i.e., the maximum value of the shortest path connecting any two nodes.

For digraphs of size $K > 1$, L and W are bounded by $2 \leq L \leq K$ and $1 \leq W \leq K - 1$.

III. Protocol Description

In this section we provide a description of the protocol and provide an intuitive depiction of its behavior. The system has two synchronization states: **synchronized** and **unsynchronized**. The system is in the unsynchronized state when it starts up, i.e., at power-on. The system is in the synchronized state when the nodes are within an expected bounded precision. The system transitions from the unsynchronized state to the synchronized state after the execution of a synchronization protocol. Therefore, the clock synchronization protocol is expected to enable the system to transition to the synchronized state and remain there. When a system reaches and operates in the synchronized state, it is said to be synchronous or in synchrony. Due to the inherent drift in the local times, a synchronization protocol must be re-executed at regular intervals to ensure that the local times are kept synchronized. The rate of resynchronization is constrained by physical parameters of the design (e.g., oscillator drift rates) as well as precision and accuracy goals. The protocol presented in this paper addresses achieving and maintaining the precision goal of the system. Achieving the clock *accuracy* goal is beyond the scope of this paper and is addressed separately as described in [25]. Therefore, the clock synchronization protocol enables the system to achieve and maintain synchrony among distributed local logical clocks, i.e., *LocalTimers* (not local physical oscillators).

The clocks need to be periodically synchronized due to their inherent drift with respect to each other. In order to achieve synchronization, the nodes communicate by exchanging **Sync** messages. A node is said to **time out** when its *LocalTimer* reaches its maximum value. Upon time out, the node generates a new *Sync* message and broadcasts it to others. A node is said to be **interrupted** when it accepts an incoming *Sync* message before its *LocalTimer* reaches the maximum value, i.e., before it times out. Upon interrupt and except for a predefined window (Section 3.1), the node relays the incoming *Sync* message by broadcasting it to others. The periodic time synchronization after achieving the initial synchrony is referred to as the **resynchronization process** whereby all nodes reengage in the synchronization process. The resynchronization process begins when the first node times out and transmits a *Sync* message and ends after the last node transmits a *Sync* message. For $\rho << 1$, the fastest node cannot time out again before the slowest node transmits a *Sync* message (see [29] for more discussion on ρ). A *Sync* message is transmitted either as a result of a resynchronization timeout, or when a node receives *Sync* message(s) indicative of other nodes engaging in the resynchronization process. The messages to be delivered to the destination nodes are deposited on communication links.

The following definitions and terms are used in the description and operation of the protocol presented in this paper. All protocol parameters have discrete values with the time-based terms having units of real time clock ticks. The discretization is for practical purposes in implementing and model checking of the protocol. Although, the network level measurements are real values, locally and at the node level, all parameters are discrete.

- The **resynchronization period**, denoted P, has units of real time clock ticks and is defined as the upper bound on the time interval between any two consecutive resets of the *LocalTimer* by a node.
- **Drift per *t***, denoted $\delta(t)$, has units of real time clock ticks and is defined as the maximum amount of drift between any two nodes for the duration of t, $\delta(t) \geq 0$. In particular:
  - Drift per D, denoted $\delta(D)$, for the duration of one D, $\delta(D) \geq 0$.

- Drift per γ, denoted $\delta(\gamma)$, for the duration of one γ, $\delta(\gamma) \geq 0$.
- Drift per P, denoted $\delta(P)$, for the duration of one period P, $\delta(P) \geq 0$.

- The **graph threshold**, T_S, is based on a specified graph topology and has units of real time clock ticks.
- The guaranteed precision or simply **precision** of the network, denoted π, $0 \leq \pi < P$, has units of real time clock ticks and is defined as the guaranteed achievable precision among all nodes.
- The **convergence time**, denoted C, has units of real time clock ticks and is defined as the bound on the maximum time it takes for the network to converge, i.e., to achieve synchrony.
- **Precision between *LocalTimers*** of any two adjacent nodes N_i and N_j denoted by Δ_{ij} and has units of real time clock ticks.
- The **initial synchrony** is a state of the network and the earliest time when the precision among all nodes, upon convergence, is within π. The initial synchrony occurs at time C_{Init}.
- The **initial precision** among *LocalTimers* of all nodes is denoted by Δ_{Init}, has units of real time clock ticks and, for all $t \geq C_{Init}$, is defined as a measure of the precision of the network immediately after a resynchronization process.
- The **initial guaranteed precision** among *LocalTimers* of all nodes is denoted by $\Delta_{InitGuaranteed}$, has units of real time clock ticks and, for all $t \geq C$, is defined as a measure of the precision of the network immediately after a resynchronization process.

A. How Does The Protocol Work?

In this section we provide an intuitive description of the protocol behavior. A node periodically undergoes a resynchronization process either when its *LocalTimer* times out or when it receives a *Sync* message. If it times out, it broadcasts a *Sync* message and so initiates a new round of a resynchronization process. However, since we are assuming that there are no faults (F) present, i.e., $F = 0$, when a node receives a *Sync* message, except during a predefined window, it accepts the *Sync* message and undergoes the resynchronization process where it resets its *LocalTimer* and relays the *Sync* message to others. This process continues until all nodes participate in the resynchronization process and converge to a guaranteed precision. The predefined window where the node ignores all incoming *Sync* messages, referred to as **ignore window**, provides a means for the protocol to stop the endless cycle of resynchronization processes triggered by the follow up *Sync* messages.

B. The Graph Threshold (T_S)

When a node receives a *Sync* message, except during a predefined ignore window, it accepts the *Sync* message and undergoes the resynchronization process where it resets its *LocalTimer* and relays the *Sync* message to others. We bound the ignore window to [D, T_S). The lower bound is due to the minimum event-response delay, D, and the upper bound, referred to as the graph threshold, T_S, is a function of a specified graph topology.

C. Sync Message And Its Validity

In order to achieve synchrony, the nodes communicate by exchanging *Sync* message[2]. When the system is in synchrony, the protocol overhead is at most one message per resynchronization period P. Assuming physical-layer error detections are dealt with separately, the reception of a *Sync* message is indicative of its validity in the value domain. The protocol performs as intended when the timing requirements of the messages from every node are satisfied. However, in the absence of faults, the reception of a *Sync* message is indicative of its validity in the value and time domains. A valid *Sync* message is discarded after it is relayed to the synchronizer and has been kept for one local clock tick.

D. The Monitor, Synchronizer, And Protocol Functions

A node consists of a **synchronizer** and a set of **monitors**. To assess the behavior of other nodes, a node employs as many monitors as the number of nodes it is directly connected to with one monitor for each source of incoming messages. A node neither uses nor monitors its own messages. A monitor keeps track of the activities of its corresponding source node. Specifically, a monitor reads, evaluates, validates, and stores the last valid message it receives from that node. Upon conveying the valid message to the local synchronizer, a monitor disposes of the valid message after it has been kept for one local clock tick.

```
ValidateMessage():
if (incoming message = Sync) then
{Message is valid,
  Store it.}

ConsumeMessage():
if (stored message timer ≥ 1 tick) then
{Message is expired,
  Clear it.}

ValidSync():
if (number of stored messages > 0) then
{   return true,
  else
      return false.}
```

Figure 3. The protocol functions.

The functions *ValidateMessage()* and *ConsumeMessage()*, Figure 3, are used by the monitors. The function *ValidSync()* is used by the synchronizer.

[2] Since only one message type is used for the operation of this protocol, a single bit suffices.

E. Protocol Assumptions

1. $K \geq 1$.
2. All nodes correctly execute the protocol.
3. All links correctly transmit data from their sources to their destinations.
4. T is a non-partitioned, strongly connected digraph.
5. $0 \leq \rho << 1$.
6. A message sent by a node will be received and processed by its adjacent nodes within γ, $\gamma = (D + d)$.
7. The initial values of the variables of a node are within their corresponding data-type range, although possibly with arbitrary values. (In an implementation, it is expected that some local mechanism exists to enforce type consistency for all variables.)

F. The Distributed Self-Stabilizing Problem

To simplify the presentation of this protocol, it is assumed that all time references are with respect to an initial real time t_0, where $t_0 = 0$ and for all $t \geq t_0$ the system operates within the *protocol assumptions*. The maximum difference in the value of *LocalTimer* for all pairs of nodes at time t, $\Delta_{Net}(t)$, is determined by the following equation that accounts for the variations in the values of the *LocalTimer* across all nodes.

$r = \lceil (W + 1)(\gamma + \delta(\gamma)) \rceil$,
$LocalTimer_{min}(x) = min\ (N_i.LocalTimer(x))$, and
$LocalTimer_{max}(x) = max\ (N_i.LocalTimer(x))$, for all i.
$\Delta_{Net}(t) = min\ ((LocalTimer_{max}(t) - LocalTimer_{min}(t)),$
$(LocalTimer_{max}(t - r) - LocalTimer_{min}(t - r)))$.

The following symbols were defined earlier and are listed here for reference:

- C denotes a bound on the maximum convergence time.
- $\Delta_{Net}(t)$, for real time t, is the maximum difference of values of the *LocalTimers* of any two nodes (i.e., the relative clock skew) for $t \geq t_0$.
- π, the synchronization precision, is the guaranteed upper bound on $\Delta_{Net}(t)$, for all $t \geq C$.

There exist C and π such that the following self-stabilization properties hold.

1. **Convergence:** $\Delta_{Net}(C) \leq \pi$, $0 \leq \pi < P$
2. **Closure:** For all $t \geq C$, $\Delta_{Net}(t) \leq \pi$
3. **Congruence:** For all nodes N_i, for all $t \geq C$, $(N_i.LocalTimer(t) = \gamma)$ implies $\Delta_{Net}(t) \leq \pi$.
4. **Liveness:** For all $t \geq C$, *LocalTimer* of every node sequentially takes on at least all integer values in $[\gamma, P-\pi]$.

G. The Self-Stabilizing Distributed Clock Synchronization Protocol For Arbitrary Digraphs

The protocol is presented in Figure 4 and consists of a synchronizer and a set of monitors which execute once every local clock tick. The protocol parameters are obtained analytically.

The following is a list of protocol parameters when all links are bidirectional.

$T_S \geq (L+2)(\gamma + \delta(\gamma))$
$P \geq 3T_S$, for $\rho = 0$
$P \geq 3(T_S + \delta(T_S))$, for $L = K$ and $\rho > 0$
$P \geq \max\ ((2K + 1)(\gamma + \delta(\gamma)), 3(T_S + \delta(T_S)))$, for $L = f(T)$ and $\rho > 0$

The following is a list of protocol parameters for digraphs, i.e., when at least one link is unidirectional.

$T_S \geq (K+2)(\gamma + \delta(\gamma))$
$P \geq K(T_S + \delta(T_S))$

Regardless of the types of links in the network, the following is a list of protocol measures.

$C_{Init} = 2P + K(\gamma + \delta(\gamma))$
$\Delta_{Init} \leq (K - 1)(\gamma + \delta(\gamma))$
$C = C_{Init} + \lceil \Delta_{Init} / \gamma \rceil P$
$Wd \leq \Delta_{InitGuaranteed} \leq W(\gamma + \delta(\gamma))$, for all $t \geq C$
$\pi = \Delta_{InitGuaranteed} + \delta(P) \geq 0$, for all $t \geq C$, and $0 \leq \pi < P$

A trivial solution is when $P = 0$. Since $P > T_S$ and the *LocalTimer* is reset after reaching P (worst-case wraparound), a trivial solution is not possible.

Monitor:
case (message from the corresponding node)
{*Sync*:
    ValidateMessage()
Other:
    Do nothing.
} // case
ConsumeMessage()

Synchronizer:
E0: if (*LocalTimer* < 0)
    LocalTimer := 0,
E1: elseif (*ValidSync()* and (*LocalTimer* < *D*))
    LocalTimer := γ, // *interrupted*
E2: elseif ((*ValidSync()* and (*LocalTimer* ≥ T_S))
    LocalTimer := γ, // *interrupted*
    Transmit *Sync*,
E3: elseif (*LocalTimer* ≥ *P*) // *timed out*
    LocalTimer := 0,
    Transmit *Sync*,
E4: else
    LocalTimer := *LocalTimer* + 1.

Figure 4. The self-stabilizing clock synchronization protocol for arbitrary digraphs.

IV. Proof Of The Protocol

There are two general formal methods approaches for the verification of the correctness of a protocol: **theorem proving** and **model checking**. Proof via theorem proving requires a deductive proof of the protocol. Proof via model checking is

based on specific scenarios and generally limited to a subset of the problem space. A deductive proof of the protocol is the subject of a subsequent report. In the mean time, we chose the model checking approach for its ease, feasibility, and quick examination of a subset of the problem space while attempting a more comprehensive proof via theorem proving.

What follows in this section is the model checking results of the proof of correctness of the protocol. In particular, model checking effort encompasses the verification of correctness of a bounded model of the protocol by confirming that a candidate system self-stabilizes from any state. This effort, furthermore, includes the verification of claims of determinism and linear convergence of the bounded model of the protocol with respect to the synchronization period.

The main theorems are enumerated here and address the following questions. Assuming a *Sync* message does not get ignored and P is sufficiently large, is it possible for a message to circulate within the network without dying out? In other words, will *E2*[3] get executed indefinitely? Is it possible for a node to transmit *Sync* messages without ever timing out? In other words, will *E3* ever get executed? Also, will *E4* ever get executed?

Theorem *Convergence – For all $t \geq C$, the network converges to a state where the guaranteed network precision is π, i.e., $\Delta_{Net}(t) \leq \pi$.*

Theorem *Closure – For all $t \geq C$, a synchronized network where all nodes have converged to $\Delta_{Net}(t) \leq \pi$, shall remain within the synchronization precision π.*

Theorem *Congruence – For all nodes N_i and for all $t \geq C$, (N_i.LocalTimer(t) = γ) implies $\Delta_{Net}(t) \leq \pi$.*

Lemma *InitGuaranteedPrecision – For all $t \geq C$, the initial guaranteed precision of the network is $Wd \leq \Delta_{InitGuaranteed} \leq W(\gamma+\delta(\gamma))$, where $\Delta_{InitGuaranteed} \leq Wd$, for $\rho = 0$, and $\Delta_{InitGuaranteed} \leq W(\gamma+\delta(\gamma))$, for $\rho > 0$.*

Theorem *Liveness – For all $t \geq C$, LocalTimer of every node sequentially takes on at least all integer values in $[\gamma, P-\pi]$.*

Lemma *ConvergenceTime – For $\rho \geq 0$, the convergence time is $C = C_{Init} + \lceil \Delta_{Init}/\gamma \rceil P$.*

Since in the protocol we do not limit K, model checking of all possible connected graphs for all K, even for idealized scenarios ($d = 0$, $\rho = 0$), is simply impossible. Model checking of all possible topologies for a given K is also a daunting task. Given the limited resources available and to circumvent state space explosion, we had to limit the network size. Nevertheless, to verify our claims of the correctness of the protocol, we have model checked all possible graphs for small K, $K \leq 5$. Additionally, we were able to model check some topologies for larger K, $K = 20$. Table 1 is a list of the model checked networks with their sizes and corresponding number of topologies while bounding the drift to $\rho \leq 0.2$. Each row corresponds to a given K and two types of topologies considered with the number of model checked graphs of the possible total combinations for the corresponding topology type in its column.

Table 1. Model checked networks.

K	**Topology** (all links bidirectional)	**Topology** (digraphs)
2	1 of 1	1 of 1
3	2 of 2	5 of 5
4	6 of 6	83 of 83
5	21 of 21	Single Directed Ring, 2 Variations of Doubly Connected Directed Ring
6	112 of 112	-
7	Linear*	Linear*
7	Star*	Star*
7	Fully Connected*	Fully Connected*
7 (3×4)	Fully Connected Bipartite*	Fully Connected Bipartite*
7	Grid	-
7	Full Grid	-
9 (3×3)	Grid	-
20	Star*	Star*

* For *Linear* and *Star* topologies and for the network to be strongly connected (to be precise,1-connected), the links are by necessity bidirectional. For *Fully Connected (Complete)* and *Fully Connected Bipartite* topologies the links are by definition bidirectional.

Details of the model checking efforts are reported in [29]. Thus far, the model checking results have verified the correctness of the protocol as they apply to the networks with unidirectional and bidirectional links as described earlier (Section II.D). In addition, the results so far confirm the claims of determinism and linear convergence. As a result, we conjecture that the protocol solves the general case of this problem for all $K \geq 1$.

V. DISCUSSIONS

From the expression for Δ_{Init}, the synchronization time C and precision π are functions of the network topology and the drift rate, specifically, the graph's width and the amount of drift the network experiences. In other words, $C = f(W, \delta(P))$ and $\pi = f(W, \delta(P))$. From the expressions for Δ_{Init} and $\Delta_{InitGuaranteed}$ it follows that for networks with small W values, $\Delta_{InitGuaranteed}$ occurs instantaneously, but for networks with large W values $\Delta_{InitGuaranteed}$ is a gradual process. The general equation for Δ_{Init} applies to the ideal ($\rho = 0$, $d = 0$) and semi-ideal ($\rho = 0$, $d \geq 0$) scenarios. For these scenarios, $\Delta_{Init} \leq W\gamma$.

Although the initial (coarse) synchrony, Δ_{Init}, occurs within C_{Init}, the initial guaranteed precision, $\Delta_{InitGuaranteed}$, takes place after a number of periods and after achieving the initial synchrony. The general equation for π applies to the ideal and semi-ideal scenarios. Since $\Delta_{InitGuaranteed} = f(W, \delta(P))$, for large values of $\delta(P)$, $\Delta_{InitGuaranteed} = \Delta_{Init}$ and no improvement on Δ_{Init} is achievable. However, since typically $0 \leq \rho << 1$, for small values of $\delta(P)$, $\Delta_{InitGuaranteed} < \Delta_{Init}$ and improvement on

[3] Labels *E0* - *E4* refer to different parts of the synchronizer.

$\Delta_{InitGuaranteed}$ is possible. In particular, for the ideal and semi-ideal scenarios, subsequent resynchronization processes beyond the initial synchrony result in tighter precision. Specifically, for $C = C_{Init} + \lceil \Delta_{Init}/\gamma \rceil P$, for the ideal scenario, the result is $\Delta_{InitGuaranteed} = 0$ and $\pi = 0$, while for the semi-ideal scenario, $\Delta_{InitGuaranteed} = Wd$ and $\pi = Wd$. Therefore, $\Delta_{InitGuaranteed}$ is 0, Wd, and $W(\gamma + \delta(\gamma))$, for the ideal, semi-ideal, and realizable systems ($\rho \geq 0$, $d \geq 0$), respectively.

So far we have studied the system in the absence and presence of ρ. In [29] we discuss whether or not ρ should be bounded and determine its theoretical upper bound. Recall that $\pi = f(W, \delta(\gamma))$ and $C = f(W, \delta(\gamma))$. Therefore, depending on the values of W and $\delta(\gamma)$, the precision of the network and the convergence time may be quite large. So, is it possible to achieve faster synchrony? Is it possible to achieve a desired precision? From the expression for π it follows that for networks with small W values, synchronization occurs instantaneously with optimal precision while for networks with large W values, synchronization is a gradual process and with larger precision. For instance, for a fully connected graph, $W = 1$, $\pi = d+\delta(\gamma)$ is at its minimum with minimal dependence on the drift, and the convergence time is at its minimum value of $C = C_{Init}$, whereas for a linear graph, $W = K$-1, π is at its maximum and more dependent on the drift, and the convergence time is at its maximum value of C. Indeed, for the worst case where drift is very high, no improvement on Δ_{Init} is possible no matter how much time passes. So, to achieve a desired precision, we must reduce either W or $\delta(P)$, or both.

To reduce W, we have to add new links to the graph, but where to add the new links and how many links to add? The idea of adding a few random links and rewiring links with a certain probability to provide shortcuts between different segments of a graph has been studied by Watts and Strogatz [30] and others [31][32][33][34][35][36]. As Arenas [3] concluded from these studies, "In general, the addition of shortcuts to regular lattices improves synchronization." and "The basic observation is that the network synchronizes when the coupling strength is increased." These studies have shown the effects of adding new links, but they do not specify how many links and where to add them in order to expedite synchronization. However, thus far in our report we have established that $\pi = f(T, \rho)$ and, so, $\pi = f(W, \delta(P))$. Therefore, to achieve the tightest precision, i.e., $\pi = d+\delta(\gamma)$, we need to add new links to the graph such that we successively halve the graph width W and, hence, double the precision. This implies that the number of links (or edges) to be added, E, is given by $E \geq \lceil \log_2 \Delta_{Init} \rceil$.

To reduce the drift, more accurate oscillators are needed, but the more accurate the oscillators, the higher the cost. What if the graph cannot or should not be modified by adding new links? Also, there are no perfect oscillators. So, what if we cannot improve upon the drift beyond a practical limit? Is there another way to achieve synchrony faster and with more accurate precision? The following section addresses these issues and examines variations of this protocol.

A. Variations Of The Synchronization Protocol

In this section we present several variations of the synchronization protocol. But first we provide an intuitive explanation. One of the key elements of the presented protocol is the proper setting of the *LocalTimer* upon receiving a *Sync* message. In the protocol we set the *LocalTimer* to γ. The rationale is that when a node times out, it resets its *LocalTimer*, i.e., *LocalTimer* = 0, and after one γ, the transmitting and receiving nodes would naturally be in relative synchrony of at most d clock ticks from each other. If we set the *LocalTimer* to D, the protocol behaves similarly but with a lower precision. In fact, as we'll see in the following section, setting the *LocalTimer* to any value less than γ produces lower precision than setting it to γ. We will not consider setting the *LocalTimer* to D a variation of the protocol.

Setting the *LocalTimer* to other values would not produce the desired effect. On the other hand, if a node gets interrupted, the receiving nodes have no knowledge of the transmitting node's *LocalTimer* value (which could be either 0 or γ). Once again, in the protocol we chose to set the *LocalTimer* to γ upon interrupt and we verified that it achieves the desired goal. However, upon interrupt, the *LocalTimer* could be assigned to other values, but what value should be chosen? An arbitrary value is not going to produce the desired synchrony, but if the value of the transmitting node's *LocalTimer* is forwarded, then the *LocalTimer* of the receiving node could be set to that value (offset by γ) and once again the two nodes will be in relative synchrony. In the following sections, we analyze these variations. We believe that transmitting any value other than the transmitting node's *LocalTimer* value does not produce the desired effect.

1) Variation #1, Reset

In this variation *LocalTimer* is reset, i.e., *LocalTimer* = 0, upon receiving a *Sync* message (*E1* and *E2*). Thus far, the model checking results have verified the correctness of this variation of the protocol. This variation of the protocol also synchronizes the network for $\rho \geq 0$ and $d \geq 0$ with the same Δ_{Init}, i.e., $\Delta_{Init} \leq (K - 1)(\gamma+\delta(\gamma))$. Also, when $\rho = 0$ and $d = 0$, unlike the original protocol where $\Delta_{InitGuaranteed} = 0$, $\Delta_{InitGuaranteed} = W\gamma$. Setting the *LocalTimer* to other values between 0 and γ would produce similar results as the original protocol and this variation of it with $0 < \Delta_{InitGuaranteed} < W\gamma$.

In this version, since $\Delta_{InitGuaranteed} = W\gamma$, even in the absence of drift, the system's behavior resembles a ripple effect where the nodes remain at most one γ apart from each other with the leading node as the center and originator of the ripple.

From variation #1 and the original protocol, one could conclude that upon receiving a *Sync* message, setting the *LocalTimer* from 0 to γ results in improvement of the initial guaranteed precision. An interesting question is whether setting the *LocalTimer* to a greater value than γ would improve upon the performance even further. As argued in the opening of this section, the next logical value beyond γ would be *LocalTimer* of the transmitting node. The following variation of the protocol is based on this idea.

2) Variation #2, Jump Ahead

In this variation, the current value of the *LocalTimer* is transmitted along with the *Sync* message and, so, upon receiving a *Sync* message *LocalTimer* is set to the incoming value plus γ to compensate for the worst case message delay. If the sum reaches or exceeds P, the *LocalTimer* is reset to zero (*E1* and *E2*). Thus far, partial model checking has also confirmed the correctness of this variation of the protocol. This variation introduces more overhead due to the transmission of *LocalTimer* value but synchronizes the network for $\rho \geq 0$ and $d \geq 0$ with the same initial precision. In other words, $\Delta_{Init} \leq (K\text{-}1)(\gamma + \delta(\gamma))$. However, this variation produces tighter initial guaranteed precision for the same convergence time, i.e., $\Delta_{InitGuaranteed} = (1+d)\delta(P)$ and $C = C_{Init} + \lceil \Delta_{Init} / \gamma \rceil P$.

This variation of the protocol has two drawbacks. The first drawback is that it requires greater number of exchanges of *Sync* messages during the convergence process. The excess transmission of the *Sync* messages is due to the burst of relays of *Sync* messages prior to the convergence. Note that since after receiving a *Sync* message the *LocalTimer* of a node gets incremented, all messages will eventually die out when the *LocalTimer* of a node reaches or exceeds its maximum value of P. Recall that in the original protocol, by setting the *LocalTimer* to γ, the node immediately enters the ignore window. In this variation, however, depending on the initial value of the *LocalTimer* of a node, a message may not get ignored until eventually the *LocalTimer* of a node reaches or exceeds its maximum value of P and then enters the ignore window.

The second drawback is that due to an interrupt, the slowest nodes may never get set to a γ during a resynchronization process even when the system is in synchrony. As a result (Theorem *Congruence*), for $t \geq C$, the nodes are in synchrony when $N_i.LocalTimer(t) = W\gamma$. In the original protocol, for all $t \geq C$, *LocalTimer* of every node sequentially takes on at least all integer values in $[\gamma, P - \pi]$. However, for this variation the minimum range of values is $[W\gamma, P - \pi]$.

B. Dynamic Digraphs

We have elaborated thus far in previous sections that the general form of the distributed synchronization problem, S, is defined by the following septuple.

$$S = (K, T, D, d, \rho, P, F)$$

In other words, the distributed synchronization problem is a function of the number of nodes (K), network topology (T), event-response delay (D), communication imprecision (d), oscillator drift rate (ρ), synchronization period (P), and number of faults (F), respectively.

However, so far, we have considered topologies with static nodes and links. This restriction helped to reduce the complexity of the problem to a more manageable size. We now define the most general form of the distributed synchronization problem, S', by the following septuple.

$$S' = (K(t), T(t), D, d, \rho, P, F)$$

Where, $K(t)$ represents the **dynamic node count** and $T(t)$ represents the **dynamic topology** for a given $K(t)$.

In a dynamic node count, the number of nodes comprising the network can change at any time. Since the nodes are anonymous and do not have unique identifiers, the presented protocol and its variations are readily applicable to this scenario. The dynamic topology allows for topologies with any combination of unidirectional and bidirectional links as described in Section II.D, whether they are static or dynamic. In other words, for a given $K(t)$, the number of links can change at any time.

In a dynamic digraph, once synchrony is achieved, the system maintains its synchrony provided that the new nodes enter the network from a reset state where they are clear of all residual effects. We have model checked a number of topologies with static nodes and various combinations of static unidirectional and bidirectional links and, thus far, the model checking results have verified the correctness of the protocol. We conjecture that the presented protocols are applicable to the general case.

VI. Conclusions

How can a distributed system solve a problem that is inherently global by executing a set of rules locally? In this paper, we have attempted to answer this question by providing a solution that synchronizes an arbitrary digraph, ranging from fully connected to 1-connected networks of nodes, under variety of conditions ranging from ideal to non-ideal circumstances. These networks include grid, ring, fully connected, bipartite, and star (hub) formation, to name a few, while allowing differences in the network elements. In our proposed solution, there is no central control or a centrally generated signal, pulse, or message. Nodes are anonymous, i.e., they do not have unique identities. We discussed the complexity of the problem and defined the parameters constituting the distributed synchronization problem.

We provided an intuitive description of the behavior of the protocol. We also provided an outline of a deductive proof of the protocol followed by the model checking results that have verified the correctness of the protocol as they apply to the networks with unidirectional and bidirectional links. In addition, the model checking results so far have confirmed the claims of determinism and linear convergence. We also provided variations of the protocol and presented model checking results of those variations. We also discussed generalization of the protocol to include dynamic node count and dynamic topology. Details of the deductive proof and details of the model checking efforts of this protocol, and its variations, are the subject of subsequent reports. We elaborated on the effects of the oscillator drift rate on the convergence time and network precision and discussed whether or not it should have an upper bound.

The proposed self-stabilizing protocol is expected to have many practical applications as well as many theoretical implications. Embedded systems, power grid, distributed process control, synchronization, computer networks, the Internet, Internet applications, security, safety, automotive,

aircraft, distributed air traffic management systems, swarm systems, wired and wireless telecommunications, graph theoretic problems, leader election, time division multiple access (TDMA), and the SPIDER (Scalable Processor-Independent Design for Enhanced Reliability) project [6][7] at NASA-LaRC are a few examples.

There does not seem to be a consensus on the definition of either emergent behavior or emergent systems. However, in the context of self-organization systems Goldstein defines emergence as: "the arising of novel and coherent structures, patterns and properties during the process of self-organization in complex systems" [37]. Emergent systems tend to display a collective behavior that is greater than the sum of their parts. An emergent behavior or emergent property surfaces in systems as a result of the interactions at an elemental level. The family of clock synchronization protocols presented in this paper is an emergent system. In these protocols all nodes operate asynchronously while the system operates synchronously. Finally, we believe this protocol can be used as a basis for modeling and studying mass synchrony as exhibited in biological and social systems.

ACKNOWLEDGMENT

The author would like to thank the reviewers for their helpful comments.

REFERENCES

[1] George H. Hudson, Science 48, pp. 573-575, 1918.

[2] Strogatz, S.H.: "SYNC, How Order Emerges From Chaos in the Universe, Nature, and Daily Life," ISBN 978-0-7868-8721-7, 2003.

[3] Arenas, A.; Diaz-Guilera, A.; Kurths, J.; Moreno, Y.; Zhou, C.: "Synchronization in complex networks," PACS: 05.45.Xt, 89.75.Fb, 89.75.Hc, December 2008.

[4] Kopetz, H: "Real-Time Systems, Design Principles for Distributed Embedded Applications," Kluwar Academic Publishers, ISBN 0-7923-9894-7, 1997.

[5] Miner, P.S. ; Malekpour, M.R.; Torres, W.: "A Conceptual Design For a Reliable Optical Bus (ROBUS)", Presented at the 21st Digital Avionics Systems Conference (DASC), Irvine, California, October 27-31, 2002.

[6] Torres-Pomales, W.; Malekpour, M.R.; Miner, P.S.: "ROBUS-2: A Fault-Tolerant Broadcast Communication System," NASA/TM-2005-213540, March 2005.

[7] Torres-Pomales, W.; Malekpour, M.R.; Miner, P.S.: "Design of the Protocol Processor for the ROBUS-2 Communication System," NASA/TM-2005-213934, pp. 252, November 2005.

[8] Peskin C.: "Mathematical Aspects of Heart Physiology", 1975. http://www.math.nyu.edu/faculty/peskin/heartnotes/CLN-Peskin1975-7.pdf

[9] Dijkstra, E.W.: "Self stabilizing systems in spite of distributed control," Commun. ACM 17, pp. 643-644, 1974.

[10] Nishikawa, T.; Motter, A.E.: "Maximum performance at minimum cost in network synchronization," Physica D 224 (2006) 77–89.

[11] Nishikawa, T.; Motter, A.E.: "Synchronization is optimal in nondiagonalizable networks," Phys. Rev. E 73 (2006) 065106.

[12] Gleiser, P.M.; Zanette, D.H.: "Synchronization and structure in an adaptive oscillator network," Europ. Phys. J. B 53 (2006) 233–238.

[13] Brede, M.: "Locals vs. global synchronization in networks of non-identical kuramoto oscillators," Europ. Phys. J. B 62 (2008) 87–94.

[14] Earl, M.G.; Strogatz, S.H.: "Synchronization in Oscillator Networks With Delayed Coupling: A Stability Criterion," The American Physical Society, 2003.

[15] Lamport, L; Melliar-Smith, P.M.: "Synchronizing clocks in the presence of faults," J. ACM, vol. 32, no. 1, pp. 52-78, 1985.

[16] Srikanth, T.K.; Toueg, S.: "Optimal clock synchronization," Journal of the ACM, 34(3), pp. 626–645, July 1987.

[17] Welch, J.L.; Lynch, N.: "A New Fault-Tolerant Algorithm for Clock Synchronization," Information and Computation volume 77, number 1, pp.1-36, April 1988.

[18] Davies, D.; Wakerly, J.F.: "Synchronization and matching in redundant systems," IEEE Transactions on Computers, 27(6), pp. 531-539, June 1978.

[19] Girault, A.; Rutten, E.: "Modeling Fault-tolerant Distributed Systems for Discrete Controller Synthesis," Electronic Notes in Theoretical Computer Science, vol. 133, pp. 81-100, 2005.

[20] Butler, R.: "A primer on architectural level fault tolerance," NASA/TM-2008-215108, February 2008.

[21] Lamport, L.; Shostak, R.; Pease, M.: "The Byzantine General Problem," ACM Transactions on Programming Languages and Systems, 4(3), pp. 382-401, July 1982.

[22] Dolev, S.; Welch, J.L.: "Self-Stabilizing Clock Synchronization in the Presence of Byzantine Faults," Journal of the ACM, Vol.51, No. 5, pp. 780-799, September 2004.

[23] Malekpour, M.R.; Siminiceanu, R.: "Comments on the 'Byzantine Self-Stabilizing Pulse Synchronization' Protocol: Counterexamples." NASA/TM-2006-213951, February 2006.

[24] Daliot, A.; Dolev, D.; Parnas, H.: "Linear Time Byzantine Self-Stabilizing Clock Synchronization," Proceedings of 7th International Conference on Principles of Distributed Systems (OPODIS-2003), La Martinique, France, December 2003.

[25] Malekpour, M.R.: "A Byzantine-Fault Tolerant Self-Stabilizing Protocol for Distributed Clock Synchronization Systems." Eighth International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS06), November 2006.

[26] Malekpour, M.R.: "A Self-Stabilizing Byzantine-Fault-Tolerant Clock Synchronization Protocol," NASA/TM-2009-215758, June 2009.

[27] Malekpour, M.R.: "Verification of a Byzantine-Fault-Tolerant Self-Stabilizing Protocol for Clock Synchronization." IEEE Aerospace Conference, March 2008.

[28] Malekpour, M.R.: "A Self-Stabilizing Distributed Clock Synchronization Protocol For Arbitrary Digraphs," NASA/TM-NASA/TM-2011-217054, February 2011.

[29] Malekpour, M.R.: "Model Checking A Self-Stabilizing Distributed Clock Synchronization Protocol For Arbitrary Digraphs," NASA/TM-NASA/TM-2011-217152, May 2011.

[30] Watts, D.J.; and Strogatz, S.H.: "Collective dynamics of 'small-world' networks," Nature (London) 393, 440, 1998.

[31] Gade, P.M.; Hu, C.K.: "Synchronous chaos in coupled map lattices with small-world interactions," Phys. Rev. E 62 (2000) 6409–6413, 2000.

[32] Barahona, M.; Pecora, L.M.: "Synchronization in Small-World Systems," Phys. Rev. Lett. 89 (2002) 054101, 2002.

[33] Hong H.; Choi, M.Y.; Kim, B.J.: "Synchronization on small-world networks," Phys. Rev. E 65 (2002) 026139, 2002.

[34] Hong H.; Kim, B.J.; Choi, M.Y.; Park, H.: "Factors that predict better synchronizability on complex networks," Phys. Rev. E 69 (2004) 067105, 2004.

[35] Li, C.; Chen, G.: "Phase synchronization in small-world networks of chaotic oscillators," Physica A 341 (2004) 73–79, 2004.

[36] Gomez-Gardenes, J.; Moreno, Y.; Arenas, A.: "Paths to Synchronization on Complex Networks," Phys. Rev. Lett. 98 (2007), 034101, 2007.

[37] Goldstein, j: "Emergence as a Construct: History and Issues," Emergence 11, pp. 49-72, 1999.