# A Self-Stabilizing Hybrid Fault-Tolerant Synchronization Protocol

**Mahyar R. Malekpour**
**NASA Langley Research Center, Hampton, VA 23681-2199**
**757-864-1513**
**Mahyar.R.Malekpour@nasa.gov**

*Abstract* — **This paper presents a strategy for solving the Byzantine general problem for self-stabilizing a fully connected network from an arbitrary state and in the presence of any number of faults with various severities including any number of arbitrary (Byzantine) faulty nodes. The strategy consists of two parts: first, converting Byzantine faults into symmetric faults, and second, using a proven symmetric-fault tolerant algorithm to solve the general case of the problem. A protocol (algorithm) is also present that tolerates symmetric faults, provided that there are more good nodes than faulty ones. The solution applies to realizable systems, while allowing for differences in the network elements, provided that the number of arbitrary faults is not more than a third of the network size. The only constraint on the behavior of a node is that the interactions with other nodes are restricted to defined links and interfaces. The solution does not rely on assumptions about the initial state of the system and no central clock nor centrally generated signal, pulse, or message is used. Nodes are anonymous, i.e., they do not have unique identities. A mechanical verification of a proposed protocol is also present. A bounded model of the protocol is verified using the Symbolic Model Verifier (SMV). The model checking effort is focused on verifying correctness of the bounded model of the protocol as well as confirming claims of determinism and linear convergence with respect to the self-stabilization period.**

## TABLE OF CONTENTS

**1. INTRODUCTION.....1**
**2. SYSTEM OVERVIEW .....2**
**3. PROTOCOL DESCRIPTION .....4**
**4. VERIFICATION OF THE CORRECTNESS OF THE PROTOCOL VIA MODEL CHECKING .....8**
**5. CONCLUSIONS .....9**
**REFERENCES .....9**
**BIOGRAPHY .....10**

## 1. INTRODUCTION

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. Since physical oscillators are inherently imperfect, local clocks of nodes of a distributed system, driven by these oscillators, do not keep perfect time and can drift with respect to real time and one another. Thus, the local clocks of the nodes must periodically be resynchronized. As a result, a fault-tolerant system needs a clock synchronization algorithm that tolerates imprecise local clocks and faulty behavior by some processes. In this paper we present a strategy for synchronizing distributed systems in the presence of various faults, including any number of arbitrary (Byzantine) faults.

We define **synchronization** of a distributed system as the process of **achieving** and **maintaining** a bounded skew among independent local clocks by exchanging local time information. A distributed system is defined to be self-stabilizing if, from an arbitrary initial state, it is guaranteed to reach a legitimate state in a finite amount of time and remain in a legitimate state. For clock synchronization, 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 it experiences system-wide disruptions in the presence of network element imperfections. 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 [1]-[4], synchronization is the most crucial element of these systems. Hereafter in this paper, we use the term *synchronization* to mean self-stabilizing clock synchronization in distributed systems.

There is a vast literature on the synchronization phenomena 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. In the context of synchronization, the **convergence** and **closure** properties address *achieving* and *maintaining* network synchrony, respectively (see Section 3.3 for a formal definition of these parameters). There are many solutions that deal with the closure property [5]-[7] which either do not address convergence or provide an ad hoc solution [8] for initialization and integration, separately. Typically, the assumed topology is a regular graph[1] such as a fully

[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*.

978-1-4799-5380-6/15/$31.00 ©2015 IEEE

connected graph or a ring. Although these topologies do not necessarily correspond to practical applications or biological, social, or technical networks, nevertheless, they provide a base case to solve the distributed synchronization problem. Furthermore, the existing models and solutions do not always achieve synchrony and, therefore, do not solve the general case of the distributed synchronization problem. Furthermore, 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 [3], [9]. The requirement to handle faults adds a new dimension to the complexity of the synchronization of fault-tolerant distributed systems. A fundamental property of a robust distributed system is the capability of tolerating and potentially recovering from failures that are not predictable in advance. See [5] and [10] for various ideas for overcoming failures in a robust distributed system that include tolerating Byzantine faults. There are many algorithms that address permanent faults [6], where the issue of transient failures is either ignored or inadequately treated. There are many efficient Byzantine clock synchronization algorithms proposed that are based on assumptions on initial synchrony of the nodes [6], [7] or existence of a common pulse at the nodes, e.g., the first protocol in [11]. There are also many clock synchronization algorithms that are based on randomization and, therefore, are non-deterministic, e.g., the second protocol in [11].

The main challenge associated with distributed synchronization is the complexity of developing a correct and verifiable solution. It is possible to have a candidate 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. In the absence of a paper-and-pencil proof, the use of fully automated formal methods techniques is a viable alternative. In [12] a counterexample is presented to a clock synchronization algorithm given in [13] that is based on the existence of a common pulse at the nodes. Furthermore, addressing network element imperfections, such as oscillator drift with respect to real time and differences in the lengths of the physical communication media, is necessary to make a solution applicable to realizable systems.

Two Byzantine-fault-tolerant self-stabilizing protocols for distributed systems were reported in [14] and [15]. Instances of these protocols were demonstrated via mechanical verification to self-stabilize from any state, in the presence of at most one permanent Byzantine faulty node, and to deterministically converge in linear time with respect to the synchronization period [16]. These protocols, however, do not solve the general case of the problem in the presence of multiple Byzantine faults [15].

Drawing from our prior experience, in this paper we present a strategy for solving the Byzantine general problem. Our solution self-stabilizes a fully connected network from an arbitrary initial state and in the presence of any number of arbitrary (Byzantine) faulty nodes, for realizable systems, while allowing for differences in the network elements, provided that the number of arbitrary faults is not more than a third of the network size [5], [10], [11]. The main problem in the self-stabilization problem is a lack of a symmetric view of the system across all good (non-faulty) nodes (processors). What if this issue is somehow resolved? Can the system self-stabilize in the presence of symmetric faults? A fault is symmetric when all good nodes observe consistent error manifestations, but do not know that it is bad [2]. Thus, the crux of the solution presented in this paper is to 1) first convert any message to a symmetric message and, 2) use a verified protocol that is based on a message symmetry assumption to solve the synchronization problem.

There are a number of ways of achieving message symmetry across the system. The Oral Message algorithm of Lamport et al. [10] that solves the Byzantine Agreement (BA) problem [17], for instance, can be used to transform a message, including an asymmetric message, to a symmetric message, whereby the good nodes collectively either accept or reject it symmetrically (an agreement) within a time bound. Other methods include using variety of engineering practices, for example, using self-checking pair at the node level [18], [19] or central guardian at the system level [20], [21].

In this paper, we present a protocol (algorithm) that tolerates symmetric faults, provided that there are more good nodes than faulty ones. We also present the model checking results of a bounded model of the protocol that was used to validate the correctness of the protocol as it applies to fully connected networks and confirmed the claims of determinism and linear convergence. Our solution applies equally well to any method that can guarantee message symmetry across all receiving good nodes.

This paper is organized as follows. In Section 2 we provide a system overview. We present the protocol and its description in Section 3. In Section 4 we present the model checking efforts toward verification of correctness of a bounded model of the protocol and the results of that effort. Finally, we present concluding remarks in Section 5 and enumerate some possible applications.

## 2. SYSTEM OVERVIEW

We considered a system of pulse-coupled entities (e.g., oscillators, pacemaker cells) pulsating 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 modeled the system as a graph with a set of nodes (vertices) that represent the pulse-coupled entities and a set of communication links (edges) that represent their interconnectivity.

The underlying topology considered is a fully connected network of $K^2 \geq 1$ nodes that exchange messages through a set of communication links. Nodes are anonymous, i.e., they do not have unique identities. The system consists of a set of good nodes and a set of faulty nodes. A good node is assumed to actively participate in the synchronization process and correctly execute the protocol. A faulty node is either benign (detectably bad), symmetric, or arbitrary (Byzantine). We define a faulty node from the perspective of a source node (sender). A maximum of $F$ faulty nodes are assumed to be present in the system, where $F \geq 0$. The minimum number of good nodes in the system, $G$, is defined by $G = K - F$ nodes. We denote the maximum number of detectably bad nodes by $F_D$, symmetrically bad nodes by $F_S$, arbitrarily (Byzantine) bad nodes by $F_A$, and the maximum number of bad nodes by $F = F_D + F_S + F_A$. 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, furthermore, we assume no physical self-loop link from the node back to itself. We attribute a faulty link behavior to its source node. Therefore, 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 [1]. 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 the system eventually adheres to the protocol assumptions (Section 3.3).

### *2.1. Drift Rate*

Each node is driven by an independent, free-running local physical oscillator (i.e., the phase is not controlled in any way) and two clocks (i.e., counters), denoted *StateTimer* and *LocalTimer*, which locally keep track of the passage of time and are driven by the local physical oscillator. An **oscillator tick**, also called a **clock tick**, is a discrete event and the basic unit of time in the network [3].

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, $\rho$, where $\rho$ is a constant, unitless, non-negative real value and is constrained to $0 \leq \rho << 1$. The maximum drift of the fastest clock of a good node over a time interval of $t$ is given by $(1 + \rho)t$. The maximum drift of the slowest clock of a good node over a time interval of $t$ is given by $(1/(1 + \rho))t$. Thus, the relative drift of the fastest and slowest good nodes is $(1 + \rho)t - (1/(1 + \rho))t$.

In simulation and model checking, typically time is modeled to reflect real time with a certain accuracy, and the drift of a node is measured with respect to that model of time. In a distributed system, addressing clock accuracy is orthogonal to achieving and maintaining **synchrony** which is a measure of the relative precision of the good nodes. Thus, in the context of a correctness proof of a distributed protocol, only the relative drift of the good nodes is considered.

### *2.2. The Clocks*

Each node has two primary clocks, *StateTimer* and *LocalTimer*, which locally keep track of the passage of time and are driven by the node's local physical oscillator. The *StateTimer* is used for operations local to the node as they relate to achieving and maintaining synchrony among the good nodes. The *LocalTimer* is used to properly filter out inherent deviation in the *StateTimer* during the *resynchronization process* (to be defined shortly) by providing a jitter-free clock to the higher level protocols. The *LocalTimer* is also used in assessing the state of the system from an external perspective. Activities of the *StateTimer* and *LocalTimer* of a node during steady state are depicted in Figure 1.

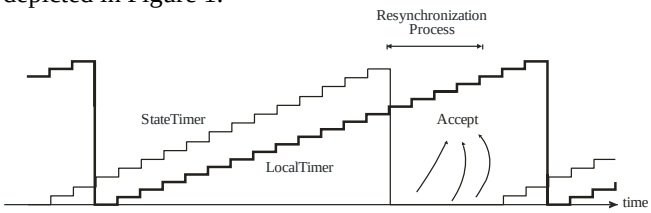


Figure 1. Activities of a good node during steady state.

The *StateTimer* takes on discrete values and is a monotonic function increasing from an initial value to a maximum value. The synchronization period during steady state, denoted $P_{ST}$, is defined as the largest time interval between any two consecutive resets of the *StateTimer* by a good node. As shown in Figure 1, if uninterrupted, the *StateTimer* periodically takes on all discrete values from its initial value, 0, to its maximum value, $P_{ST}$, linearly increasing within each period and is bounded by $0 \leq StateTimer \leq P_{ST}$.

The *LocalTimer* is also driven by the local physical oscillator, takes on discrete values, and locally keeps track of the passage of time. The *LocalTimer* is a monotonic linear function increasing from an initial value to a maximum value. As shown in Figure 1, if uninterrupted,

[2] Since we use $N_i$ to address a node, we use $K$ here instead of $n$ as is traditionally used in the literature.

the *LocalTimer* periodically takes on all discrete values from its initial value, 0, to its maximum value, $P_{LT}$, linearly increasing within each period and is bounded by $0 \le LocalTimer \le P_{LT}$.

These logical clocks need to be periodically synchronized due to the inherent drift in their local physical oscillators. In order to achieve synchronization, the nodes communicate by exchanging **Sync** messages. The periodic synchronization during steady state is referred to as the **resynchronization process** which starts when the first good node begins to transmit a burst of consecutive *Sync* messages and ends after the last occurrence of consequent *accept event* at a good node. An **accept event** occurs when a good node receives a sufficient number of *Sync* messages from as many good nodes. The sufficiency of *Sync* messages is a function of the type and number of faults being tolerated. An upper bound on the duration of the resynchronization process will be determined later in this paper.

The *LocalTimer* is intended to be used by higher level protocols, and it must be managed properly to provide the desired monotonically increasing value between adjustments and despite inherent deviation in the *StateTimer*. The *LocalTimer* is incremented once every local clock tick and is reset either when it reaches its maximum allowed value, $P_{LT}$, or when the *StateTimer* of the node has reached *ResetLocalTimerAt*, where *ResetLocalTimerAt* is constrained by the following inequality:

$$\lceil \pi_{init} \rceil \le ResetLocalTimerAt \le P_{ST} - \lceil \pi \rceil \qquad (1)$$

Where $\lceil\ \rceil$ is the ceiling function, $\pi_{init}$ is the initial network precision after a resynchronization process, and $\pi$ is the upper bound on the guaranteed precision, i.e., the guaranteed upper bound on the maximum separation between the *LocalTimers* of any two good nodes. Furthermore, the initial precision, $\pi_{init}$, is the maximum difference between *StateTimers* of any two good nodes upon completion of the resynchronization process. The *ResetLocalTimerAt* can be given any value in the range specified in inequality (1). However, the value must be the same at all good nodes. In this inequality, the lower bound indicates when all good nodes have reset their *StateTimers* and the upper bound indicates when the first good node might time out and begin the next round of resynchronization process. We choose the earliest such value, $ResetLocalTimerAt = \lceil \pi_{init} \rceil$, to reset the *LocalTimer* of all good nodes. Any value greater than $\lceil \pi_{init} \rceil$ will prolong the convergence time. The **convergence time**, denoted *C*, is defined as the bound on the maximum time it takes the network to achieve the guaranteed precision $\pi$.

### *2.3. Communication Delay*

The communication delay between directly connected (adjacent) nodes is expressed in terms of the minimum event-response delay, *D*, and network imprecision, *d*. These parameters are described with the help of Figure 2.

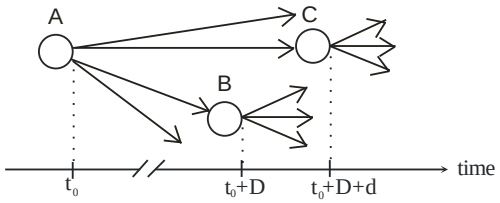


Figure 2. Network parameters, *D* and *d*.

As depicted in this figure, a message transmitted by node A at real time $t_0$ is expected to arrive at its directly connected adjacent nodes (B, C, …), be processed, and subsequent messages to be generated by those nodes within the time interval $[t_0 + D,\ t_0 + D + d]$. 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 many factors including, but not limited 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, $0 < D \le D_{max}$, $0 \le d \le d_{max}$, and both have units of real-time clock ticks. The communication delay, denoted $\gamma$, is expressed in terms of *D* and *d*, is defined as $\gamma = D + d$, and has units of real-time clock ticks. Therefore, the communication delay between any two directly connected adjacent nodes is bounded by $[D,\ \gamma]$. In other words, we assume synchronous communication.

Although from an external perspective, the value of *D* and *d*, and hence $\gamma$, are real numbers, locally and at the node level, they are treated as discrete values. In other words, and in the rest of the paper, from the local perspective of a node, $D = \lceil D \rceil$, $d = \lceil d \rceil$, and $\gamma = D + d$.

## 3. PROTOCOL DESCRIPTION

In this section we provide an intuitive description of the protocol behavior followed by a detailed description. In order to achieve synchronization, the nodes communicate by exchanging **Sync** messages. Nodes periodically undergo a new round of the resynchronization process. When a node's *StateTimer* times out, it initiates a new round of the resynchronization process by broadcasting a continual burst of (once per $\gamma$) *Sync* messages to all other nodes that are directly connected to it. During this process, the *StateTimer* is at its maximum and remains constant, i.e., the node neither increments nor resets its *StateTimer*. This process continues until all good nodes participate in the resynchronization process and converge to the guaranteed precision $\pi$. A good node uses its own message. When a good node receives a sufficient number of *Sync* messages from as many good nodes, an accept event occurs. The sufficiency of *Sync* messages is a function of the type and number of faults being tolerated. When an *accept event* occurs, the node ends its continual broadcast and concludes the resynchronization process by resetting its *StateTimer*.

Note that, as depicted in Figure 1, consecutive accept events may occur during a resynchronization process.

Since, due to drift, the duration of the resynchronization process is potentially more than $\gamma$, and during this process, the *StateTimer* holds a constant value (its maximum), it is not a reliable clock source for higher level protocols. Thus, the *LocalTimer* is introduced. To provide the desired monotonically increasing clock source to higher-level protocols, the *LocalTimer* gets reset when the *StateTimer* has reached a predefined value greater than or equal to the guaranteed initial network precision, $\pi_{init}$.

### *3.1. The Sync Message And Its Validity*

In order to achieve and maintain desired synchrony, the nodes communicate by exchanging *Sync* messages[3]. Assuming physical-layer error detection is dealt with separately, the reception of a *Sync* message is indicative of its validity in the value domain. Upon starting a new round of the resynchronization process, the node continually sends out *Sync* messages, once per $\gamma$, to other nodes that are connected to it. Therefore, we say that a *Sync* message has a **life-span**, and we limit the life-span of the *Sync* message at the receiving nodes to be $\gamma$. A *Sync* message from a given source is **valid** if it arrives at or after one $D$ of an immediately preceding *Sync* message from that source, that is, the *valid* message in the value domain, i.e., *valid Sync* messages are rate-constrained. Again assuming physical-layer error detection is dealt with separately, the reception of a *Sync* message is indicative of its validity in the value and time domains. Since a good node uses its own message, but we assume no physical self-loop link from the node back to itself, the message becomes valid only after $\gamma$, giving the own message the longest (worst case) transmission delay time. A valid *Sync* message becomes invalid after its life-span expires. However, while valid, the *Sync* message may be used multiple times and result in multiple accept events.

### *3.2. 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 that are directly connected to it, with one monitor for each source of incoming message. A node uses but does not monitor, its own message. The message is kept within the node and, therefore, no physical self-loop link back to itself. 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. A valid *Sync* message is then conveyed to the local synchronizer. The assessment results of the monitored nodes are then utilized by the synchronizer in the synchronization process. A monitor disposes of the valid message after its life-span expires.

The function *ValidateMessage()* (shown in Figure 3) is used by the monitors to determine whether a received *Sync* message meets the minimum timing requirement, and thus is valid in both value and time domains, and whether a stored valid *Sync* message has reached its lifespan and expired. The function *Accept()*, used by the synchronizer, examines availability of sufficient valid *Sync* messages. The sufficiency of available, valid messages, denoted by $T_A$, is a function of the type and number of faults to be tolerated. $T_A = F_D + 1$ and $T_A = F_S + 1$. For tolerating benign and symmetric faults, respectively, the following relations must hold: $T_A = F_D + F_S + 1$ for tolerating $F_D + F_S$ simultaneous faults. When a sufficient number of messages have been received, the *Accept()* function returns a Boolean value of true.

**ValidateMessage():**

if (*incoming message* = *Sync*) and (*MessageTimer* $\geq D$)
    *MessageValid* = true, // *store it,*
    *MessageTimer* = 0,

elseif (*MessageTimer* $\geq$ *MessageLifeSpan*)
    *MessageValid* = false, // *it expired*

elseif (*MessageTimer* < *MessageLifeSpan*)
    *MessageTimer* = *MessageTimer* + 1.

**Accept():**

if (*number of stored Sync messages* $\geq T_A$)
    *return true,*
else
    *return false.*

Figure 3. The protocol functions.

### *3.3. Protocol Assumptions*

1. The topology is a fully connected graph[4].
2. The number of nodes constituting the network is $K$, where $K \geq 2F_S + F_D + 1$, and $F_S$ is the maximum number of symmetrically bad nodes.
3. Nodes either correctly execute the protocol and are good, symmetrically bad, $F_S$ or detectably bad, $F_D$.
4. Links are bidirectional and correctly transmit data from their sources to their destinations.
5. The bound on the oscillator drift rate is $\rho$, where $0 \leq \rho << 1$.
6. A message sent by a node will be received and processed by its directly-connected, adjacent nodes within $\gamma$, where $\gamma = (D + d)$.

Physical-layer error detection is dealt with separately, thus, the reception of a *Sync* message is indicative of its validity in the value and time domains.

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

[4] We leave generalization of the topology for future work.

*3.4. The Self-Stabilizing Distributed Clock Synchronization Problem*

To simplify the presentation of the solution, 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 good nodes.

$r = \lceil \pi (1 + \rho) \rceil$ is a time interval encompassing $\pi$,
$LocalTimer_{min}(t) = min\ (N_i.LocalTimer(t))$, for all $i$, and
$LocalTimer_{max}(t) = max\ (N_i.LocalTimer(t))$, 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 convenience:

- $P_{LT}$ 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 and $P_{LT} > 0$.
- $\Delta_{Net}(t)$, for real time $t$, is the maximum difference of values of the *LocalTimers* of any two nodes (i.e., the relative clock skew) for $t \geq t_0$.
- $\pi$, the synchronization precision, is the guaranteed upper bound on $\Delta_{Net}(t)$ for all $t \geq C$, $0 \leq \pi << P_{LT}$.
- $C$, the convergence time, is defined as the bound on the maximum time for the network to achieve the guaranteed precision $\pi$.

To prove that a protocol is self-stabilizing, it has to be shown that there exist $C$ and $\pi$ such that the following self-stabilization properties hold.

1. **Convergence:** $\Delta_{Net}(C) \leq \pi$, $0 \leq \pi << P_{LT}$
2. **Closure:** For all $t \geq C$, $\Delta_{Net}(t) \leq \pi$
3. **Congruence:** For all nodes $N_i$, for all $t \geq C$, $(N_i.LocalTimer(t) = \lceil \pi \rceil) \rightarrow \Delta_{Net}(t) \leq \pi$.
4. **Liveness:** For all $t \geq C$, *LocalTimer* of every node sequentially takes on at least all discrete values in $[0, P_{ST} - \pi - \gamma]$, see Figures 1 and 6.a.

*3.5. What Self-Stabilization Properties Mean*

The *convergence* and *closure* properties address achieving and maintaining network synchrony, respectively. As formally defined in the previous section, given sufficient time, $C$, the convergence property examines whether or not the system has reached a point where all nodes are within the specified precision. The closure property, on the other hand, examines whether or not the system starting within the specified precision will remain within that precision thereafter. The convergence and closure properties provide an external view of the system, whereby the external viewer can examine whether or not the system has self-stabilized.

In safety-critical TDMA architectures, synchronization is the most crucial element of these systems. More precisely, TDMA-type applications are based on the fundamental assumption of the existence of initial synchrony. The protocol presented in this paper is meant to provide this fundamental requirement of TDMA-type applications to higher-level protocols. However, one of the challenges in employing multiple protocols in distributed system has been the integration of these protocols operating at different levels of application. Previously, the integration of a lower-level protocol with higher-levels either has not been addressed or had simply been overlooked. The *congruence* property addresses this essential requirement. Unlike the convergence and closure properties that provide system view from the perspective of an external viewer, the congruence property provides a local view from the perspective of a node by providing the necessary and sufficient conditions for the node to locally determine whether or not the system has converged. The congruence property, therefore, is essential in the integration of this underlying self-stabilization protocol with higher-level protocols in the system.

The *liveness* property examines whether or not a node takes on all possible discrete values within an expected range. In other words, the system is "alive" and the good nodes execute the protocol properly, and time advances within each node.

*3.6. The Self-Stabilizing, Symmetric-Fault Tolerant Synchronization Protocol*

In this section, we present the self-stabilizing, symmetric-fault tolerant synchronization protocol that is based on message symmetry assumption. We mentioned earlier that in order to achieve and maintain synchrony, the nodes communicate by exchanging *Sync* messages. Assuming physical-layer error detection is dealt with separately, the reception of a *Sync* message is indicative of its validity in the value domain. Upon start of a new round of a resynchronization process, the node continually sends out *Sync* messages, once per $\gamma$, to other nodes that are connected to it. Consequently, the life-span of a *Sync* message at the receiving nodes is set to be $\gamma$. Also, we mentioned earlier that for tolerating symmetric faults, sufficiency for the *Accept()* function is determined by $T_A = F_D + F_S + 1$.

The protocol, executed by all good nodes, is presented in Figure 4 and consists of a synchronizer and a set of monitors which execute once every local clock tick. Four concurrent *if* statements collectively describe the synchronizer. These statements are labeled ST (*StateTimer*), LT (*LocalTimer*), TS (Transmit *Sync*), and TT (*TransmitTimer*). The function *ValidateMessage()* describes the monitor.

**Synchronizer:**

ST1: if (*StateTimer* < 0) or (*Accept()*)
*StateTimer* := 0, // *reset*

ST2: elseif (*StateTimer* < $P_{ST}$)
*StateTimer* := *StateTimer* + 1.

LT1: if (*LocalTimer* < 0) or
(*LocalTimer* ≥ $P_{LT}$) or
(*StateTimer* = $\lceil \pi_{init} \rceil$)
*LocalTimer* := 0, // reset

LT2: else
*LocalTimer* := *LocalTimer* + 1.

TT1: if (*TransmitTimer* < 0) or
((*TransmitTimer* ≥ $\gamma$) and
(*StateTimer* ≥ $P_{ST}$))
*TransmitTimer* := 0,

TT1: elseif (*TransmitTimer* < $\gamma$)
*TransmitTimer* := *TransmitTimer* + 1.

TS1: if (*StateTimer* ≥ $P_{ST}$) and // *timed out*
(*TransmitTimer* ≥ $\gamma$) and
(not *Accept()*)
Transmit *Sync*.

**Monitor:**
*ValidateMessage()*.

Figure 4. The Symmetric-Fault Tolerant Protocol.

The following is a list of pertinent protocol measures.

$K \geq 2F_S + 1$, where $F_S$ is the maximum number of simultaneous symmetrically faulty nodes

$\delta(P_{ST})$ denotes the maximum drift for the duration of $P_{ST}$, $\delta(P_{ST}) \geq 0$

$0 \leq \rho << 1$

$0 < D \leq \gamma << P_{ST} < P_{LT}$

$0 \leq StateTimer \leq P_{ST}$

$0 \leq LocalTimer \leq P_{LT}$

$\pi_{init} = d + \gamma + \delta(d+\gamma)$

$\pi = \pi_{init} + 2\delta(P_{ST}) \geq 0$, for all $t \geq C$, and so, $0 \leq \pi << P_{ST}$

$t_{rp} = \pi + 2\gamma + \pi_{init}$, where, $t_{rp}$ denotes duration of the resynchronization process during steady state.

$P_{LT} \geq P_{ST} + t_{rp} = P_{ST} + \pi + 2\gamma + \pi_{init}$

$C = P_{LT} + ResetLocalTimerAt + 2\gamma$

Since $0 < \gamma << P_{ST} < P_{LT}$, and the *LocalTimer* is reset after reaching $P_{LT}$ (worst-case wraparound), a trivial solution is not possible.

Appendix A provides an example to give the reader a quick review and help in understanding of the behavior of the protocol.

*3.7. Determining Protocol Parameters*

We refer to $\rho$, $d$, $D$, $K$, $T$, and $P_{ST}$ as the *fundamental protocol parameters* and the remaining as the *derived parameters*. In this section, we show how the derived protocol parameters are computed.

***$\pi_{init}$*** – The initial precision, $\pi_{init}$, is the maximum difference between *StateTimer*s of any two good nodes during steady sate, for all $t \geq C$, and upon completion of a resynchronization process. Thus, as depicted in Figure 5, $\pi_{init} = d + \gamma + \delta(d+\gamma)$. In this figure, transmitted *Sync* messages are shown using '↑', received *Sync* messages using '↓', and the accept events are marked by '●' on the time axis.

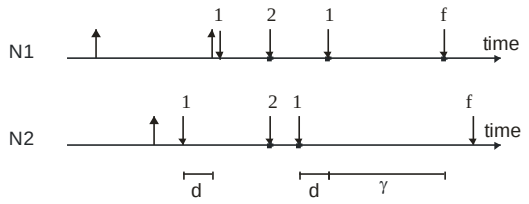


Figure 5. Network precision.

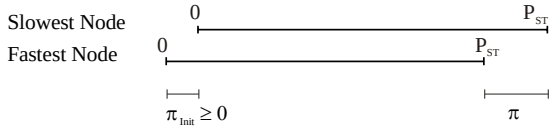


Figure 6.a. Network precision.

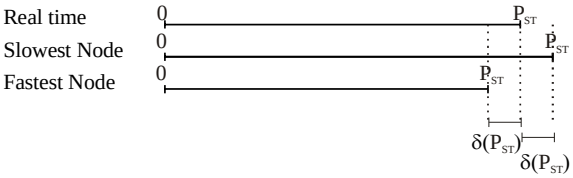


Figure 6.b. Network precision.

***$\pi$*** – From the definition of the network precision, $\pi$, it follows that, for all $t \geq C$, $\pi$ is the sum of initial precision and the maximum drift among the good nodes after $P_{ST}$ from the completion of the resynchronization process. Thus, as depicted in Figure 6, even when the nodes start in perfect synchrony, $\pi = 2\delta(P_{ST})$ and since in the worst case they start within $\pi_{init}$, therefore, for the worst case, $\pi = \pi_{init} + 2\delta(P_{ST})$.

***$t_{rp}$*** – From the definition of the resynchronization process, it follows that, during steady state it takes $\pi$ ticks for all good nodes to time out and begin transmitting *Sync* messages. It takes $\gamma$ ticks for the transmitted messages, to reach other good nodes and result in subsequent accept events at all good nodes. Since a Sync message has a life-span of one $\gamma$, subsequent accept events occur within the next $\gamma$. At the end of the resynchronization process, the good nodes are within $\pi_{init}$ ticks of each other. Thus, $t_{rp} = \pi + 2\gamma + \pi_{init}$.

***$P_{LT}$*** – The value of $P_{LT}$ is derived from the behavior of the network during steady state and it is a measure of the worst case scenario between two consecutive resets of the *LocalTimer* of a good node. Thus,

$P_{LT} \geq P_{ST} + t_{rp} = P_{ST} + \pi + 2\gamma + \pi_{init}$.

***C*** – The convergence time, *C*, is measured from $t_0$. Its value is the sum of one $\gamma$, due to randomness in the initial value of the *MessageTimer* in the good nodes, plus the worst case scenario for the good nodes undergoing a resynchronization process, i.e., $P_{ST}$, and finally converge to the predicted precision $\pi$. Therefore,

$C = \gamma + P_{ST} + t_{rp} + ResetLocalTimerAt$, and so,

$C = \gamma + P_{ST} + t_{rp} + \pi_{init}$.

## 4. VERIFICATION OF THE CORRECTNESS OF THE PROTOCOL VIA MODEL CHECKING

In this section we present a mechanical verification of the protocol using the model checking approach for its ease, feasibility, and quick examination of the problem space. The details of the model checking effort are similar to [22], where the models of the system components and their data structures are fully described and similar abstractions are employed with respect to the size of the model and real-time delays. We do not restate the details of model checking effort here. Instead, we focus on the model checking results. Similar to [22], the Symbolic Model Verifier (SMV) was used in the modeling of this protocol on a PC with 4GB of memory running Linux [23]. SMV's language description and modeling capability provide relatively easy translation from the pseudo-code. SMV semantics are synchronous composition, where all assignments are executed in parallel and synchronously. Thus, a single step of the resulting model corresponds to a step in each of the components.

The protocol described in this paper is fairly subtle and must necessarily cope with many kinds of timing behaviors. Model checking has been used to explore and verify distributed algorithms but also faces certain difficulties [16], [24]-[26]. One of the foremost challenges is a realistic representation of time as a continuous variable. As we elaborated earlier in this paper, although the network level measurements are real values, locally and at the node level, all parameters are discrete. The discretization is used for practical purposes in implementing and model checking the protocol. Since continuous time modeling is impracticable, we used the same abstractions as in [22] for discrete time.

### *4.1. Propositions*

Computational tree logic (CTL), a temporal logic, is used to express properties of a system. In CTL formulas are composed of **path quantifiers**, *E* and *A*, and **temporal operators**, *X*, *F*, *G*, and *U* [27]. In this section the claims of *convergence*, *closure*, *congruence*, and *liveness* properties as well as the claims of maximum convergence time and determinism of the protocol are examined. Although in the description of the protocol convergence and closure properties are stated separately, they are examined via one CTL proposition. This proposition also expresses the claims of determinism and linear convergence. Validation of this general CTL proposition requires examination of a number of underlying propositions. In particular, since $\Delta_{LocalTimer}(t)$ is defined in terms of the *LocalTimer* of the nodes, examination of the properties that describe proper behavior of the *LocalTimer* take precedence. The variable *ElapsedTime* is used in these properties and is defined here.

*ElapsedTime = (GlobalClock ≥ ConvergenceTime) ;*

The *GlobalClock* is a measure of elapsed time from the beginning of the operation with respect to the real time, i.e., external view. The *ElapsedTime* is indicative of the *GlobalClock* reaching its target maximum value of *ConvergenceTime*.

**Proposition *SystemLiveness*:** This property addresses the liveness property of the system by examining whether or not time advances and the amount of time elapsed, *ElapsedTime*, has advanced beyond the predicted convergence time, *ConvergenceTime*.

```
AF (ElapsedTime)
```

**Proposition *ConvergenceAndClosure*:** This proposition encompasses the criteria for the convergence and the closure properties as well as the claims of maximum convergence time and determinism. The proposition specifies whether or not the system will converge to the predicted precision after the elapse of convergence time, *ElapsedTime*, and whether or not it will remain within that precision thereafter. This and subsequent properties are expected to hold.

The proper value of the *AllWithinPrecision* is determined by measuring the difference between the maximum and minimum values of the *LocalTimers* of all nodes for the current tick and in conjunction with the result from the previous $r = \lceil \pi (1 + \rho) \rceil$ ticks. The expected difference of *LocalTimers* is the predicted precision bound.

```
-- Determinism Property
AF (ElapsedTime) ∧
-- Convergence Property
AG (ElapsedTime → AllWithinPrecision) ∧
-- Closure Property
AG ((ElapsedTime ∧ AllWithinPrecision) →
        AX (ElapsedTime ∧ AllWithinPrecision))
```

To eliminate trivial results and false positives, the following proposition is examined, and the expected result is a value of false. This property specifies that after the elapse of convergence time, *ElapsedTime*, whether or not the system will not converge or if it converges, whether or not it drifts apart beyond the expected precision bound.

$$AF\ (ElapsedTime) \wedge$$
$$AG\ (ElapsedTime \rightarrow AllWithinPrecision) \wedge$$
$$AG\ ((ElapsedTime \wedge AllWithinPrecision) \rightarrow EX\ (\neg AllWithinPrecision))$$

**Proposition *Congruence*:** This property specifies the criteria for the congruence property of the protocol. Unlike the convergence and closure properties that provide system view from the perspective of an external viewer, the congruence property provides a local view from the perspective of a node by providing necessary and sufficient conditions for the node to locally determine whether or not the system has converged. The congruence property is essential in the integration of this underlying self-stabilization protocol with higher level protocols in the system. The congruence property is described with respect to only one node, namely *Node_1*. Since all nodes are symmetric, the result of the proposition equally applies to other nodes.

$$AF\ (ElapsedTime) \wedge$$
$$AG\ ((ElapsedTime \wedge (Node\_1.LocalTimer = \lceil \pi \rceil)) \rightarrow AX\ (ElapsedTime \wedge AllWithinPrecision))$$

**Proposition *ProtocolLiveness*:** This property specifies the criteria for the liveness property of the protocol. The property examines whether or not a node takes on all discrete values within an expected range. Again, since all nodes are symmetric, this property is described with respect to only one node, namely *Node_1*.

$$AF\ (ElapsedTime) \wedge$$
$$AG\ (((ElapsedTime) \wedge (Node\_1.LocalTimer = i)) \rightarrow AX\ ((Node\_1.LocalTimer = i) \mid (Node\_1.LocalTimer = i+1))) \wedge$$
$$AG\ (((ElapsedTime) \wedge (Node\_1.LocalTimer = P_{LT})) \rightarrow AX\ (Node\_1.LocalTimer = 0))$$
$$For\ all\ i = 0\ ..\ (P_{ST} - \pi - \gamma)$$

The model checking results of the bounded model of the protocol have verified the correctness of the protocol for fully connected networks with $K \geq 2F_S + 1$ nodes, starting from an arbitrary state, and for the following scenarios. $F_S = 0, 1, 2, 3$, simultaneous symmetric faults, $0 \leq \rho << 1$, $D = 1$ and $d = 0$. $F_S = 2$ simultaneous symmetric faults, $0 \leq \rho << 1$, $D = 2, 3$, and $d = 0, 1$. In addition, the results have confirmed the claims of determinism and linear convergence.

## 5. CONCLUSIONS

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, a fault-tolerant system needs a clock synchronization algorithm that tolerates imprecise local clocks and faulty behavior by some processes. In this paper we presented a strategy for synchronizing a distributed system in the presence of various faults, including any number of arbitrary (Byzantine) faults. The main issue in solving the self-stabilization problem is a lack of a symmetric view in the system by the participating good nodes. Thus, the crux of our idea was to first convert any message to a symmetric message and then use a verified protocol, based on the message symmetry assumption, to solve the synchronization problem. We first enumerated several ways of achieving message symmetry across the system, and then presented a new protocol based on message symmetry assumption. We also presented a mechanical verification of the protocol for up to three simultaneous, symmetric faults. The model-checking effort was focused on verifying the correctness of a bounded model of the protocol as well as confirming claims of determinism and linear convergence with respect to the self-stabilization period. As a result, we believe that our solution solves the general case of this problem for fully connected graphs. We leave however the generalization of our solution to other topologies, including an arbitrary graph that meets the minimum requirements of number of nodes and connectivity, to future works.

The proposed self-stabilizing protocol is expected to have many practical applications as well as many theoretical implications. GPS (Global Positioning System) denied environment or where GPS is non-existent (e.g., Mars mission), 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, TDMA (time division multiple access), and banking and commerce are a few examples. These are some of the many areas of distributed systems that can use synchronization in order to design more robust distributed systems.

## REFERENCES


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

[2] 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.

[3] Torres-Pomales, W; Malekpour, M.R.; Miner, P.S.: *ROBUS-2: A fault-tolerant broadcast communication system*, NASA/TM-2005-213540, pp. 201, March 2005.

[4] 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.

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

[6] Srikanth, T.K.; Toueg, S.: *Optimal clock synchronization*, Journal of the ACM, 34(3), pp. 626–645, July 1987.
[7] 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.
[8] Davies, D.; Wakerly, J.F.: *Synchronization and matching in redundant systems*, IEEE Transactions on Computers, 27(6), pp. 531-539, June 1978.
[9] Butler, R.: *A primer on architectural level fault tolerance*, NASA/TM-2008-215108, February 2008.
[10] Lamport, L.; Shostak, R.; Pease, M.: *The Byzantine General Problem*, ACM Transactions on Programming Languages and Systems, 4(3), pp. 382-401, July 1982.
[11] 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.
[12] Malekpour, M.R.; Siminiceanu, R.: *Comments on the 'Byzantine Self-Stabilizing Pulse Synchronization' Protocol: Counterexamples*, NASA/TM-2006-213951, February 2006.
[13] 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.
[14] 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.
[15] Malekpour, M.R.: *A Self-Stabilizing Byzantine-Fault-Tolerant Clock Synchronization Protocol*, NASA/TM-2009-215758, June 2009.
[16] Malekpour, M.R.: Verification of a Byzantine-Fault-Tolerant Self-Stabilizing Protocol for Clock Synchronization, IEEE Aerospace Conference, March 2008.
[17] Pease, M.; Shostak, R.; and Lamport, l.: *Reaching agreement in the presence of faults*, Journal of the ACM, 27(2): 228-234, April 1980.
[18] Hoyme, K.; Driscoll, K.: *SAFEbusTM*, 11th AIAA/IEEE Digital Avionics Systems Conference, pages 68–73, Seattle, WA, October 1992.
[19] Aeronautical Radio, Inc., Annapolis, MD. *ARINC Specification 659: Backplane Data Bus*, December 1993. Prepared by the Airlines Electronic Engineering Committee.
[20] Kopetz, H.; Grünsteidl, G.: *TTP – a time-triggered protocol for fault-tolerant real-time systems*, Fault Tolerant Computing Symposium 23, pages 524–533, Toulouse, France, June 1993. IEEE Computer Society.
[21] Bauer, G.; Kopetz, H.; and Steiner, W.: *The central guardian approach to enforce fault isolation in a time-triggered system*, Proc. of 6th International Symposium on Autonomous Decentralized Systems (ISADS 2003), pp. 37–44, April 2003.
[22] Malekpour, M.R.: *Model Checking A Self-Stabilizing Synchronization Protocol For Arbitrary Digraphs*, The 31st Digital Avionics Systems Conference (DASC 2012), Williamsburg, Virginia, pp. 11, October 2012.
[23] http://www-2.cs.cmu.edu/~modelcheck/smv.html
[24] Steiner, W.; Rushby, J.; Sorea, M.; Pfeifer, H.: *Model Checking a Fault-Tolerant Startup Algorithm: From Design Exploration To Exhaustive Fault Simulation*, The International Conference on Dependable Systems and Networks (DSN'04), 2004.
[25] Steiner, W.; Dutertre, B.: *Automated Formal Verification of the TTEthernet Synchronization Quality*, 3rd NASA Formal Method Symposium, April 2011.
[26] Lönn, H.; and Pettersson, P.: *Formal verification of a TDMA protocol start-up mechanism*, In Pacific Rim International Symposium on Fault-Tolerant Systems, pages 235–242, Taipei, Taiwan, Dec. 1997. IEEE Computer Society.
[27] Clarke, E.M.; Emerson, E.A.: *Design and synthesis of synchronization skeletons using branching time temporal logic*, In Logic of Programs: Workshop, Yorktown Heights, NY, May 1981, LNCS 131. Springer, 1981.

## BIOGRAPHY

***Mahyar R. Malekpour*** *is a research engineer at NASA Langley Research Center, in Hampton, VA. His research interests include fault-tolerance, distributed clock synchronization, algorithm development, and model checking. He holds B.S. in computer engineering and an M.S. in electrical engineering from Old Dominion University.*

# APPENDIX A

The purpose of this example is to give the reader a quick review of and help in understanding the behavior of the protocol. The following is an example of a fully connected graph consisting of 5 nodes, where $F = 2$. Table A.1 shows an execution trace of the system and has eight columns; one for time reference, two for each good node listing values for the *StateTimer* and *LocalTimer*, and the last column is for network precision, $\pi$. Each row depicts activities of all good nodes at the corresponding time. Cell contents for the node columns consist of a number corresponding to the value of the *StateTimer* of the node in conjunction with an activity: 1) *Sync* if the node transmits the message, and 2) *Accept* if the node received $T_A$ messages. The received messages at a node are depicted in superscripts, one position for each corresponding node, where a '-' means no messages from that node and an 'x' means a *Sync* message was received.

This table depicts activities of the network during a *resynchronization process* when the network is in steady state. Even though the good nodes started the cycle in synchrony, they gradually drifted apart. The table shows a scenario where node 1 is the fastest and node 3 the slowest of the good nodes and by the end of the synchronization period they have drifted part by as much as 12 clock ticks from an external perspective. Since the faulty nodes can transmit messages at any time, their activities are not listed in the table. However, their messages are recorded at the receiving good nodes. For instance, at (t + 8) a message from node 5 (a faulty node) is received by nodes 1 and 2 and $d$ ticks later node 3 records receiving the same message. The columns representing *LocalTimer* values are shaded gray for visual purposes. The '$\pi$' column shows that although the instantaneous differences between the *LocalTimers* spike up to a value of 999 at ($t$ + 17), the precision $\pi$ as defined in Section 3.4 remains within the theoretical predicted value of 16.

System parameters:

$D = 3$ clock ticks, $d = 1$ clock tick → $\gamma = 4$ clock ticks

$K = 5$ nodes, $G = 3$ nodes, $F = 2$ nodes → $T_A = 3$ nodes

$P_{ST} = 1000$ clock ticks

$0 \le \rho \le 1$ → $0 \le \delta(P_{ST}) \le 5$ clock ticks

$\pi_{init} = d + \gamma + \delta(d + \gamma)$ → $\pi_{init} = 6$ clock ticks

$\pi = \pi_{init} + 2\delta(P_{ST}) \ge 0$ → $\pi = 16$ clock ticks

$r = \lceil \pi (1 + \rho) \rceil = 17$ clock ticks

$t_{rp} = \pi + 2\gamma + \pi_{init}$ → $t_{rp} = 30$ clock ticks

$P_{LT} \ge P_{ST} + t_{rp}$ → $P_{LT} = 1030$ clock ticks

*ResetLocalTimerAt* = $\pi_{init}$ → *ResetLocalTimerAt* = 6 clock ticks

$C = P_{LT} + ResetLocalTimerAt + 2\gamma$ → $C = 1044$ clock ticks

Table A.1. An execution trace of a network of 5 nodes.

| ***Time*** | $N_{1.StateTimer}$ | $N_{2.StateTimer}$ | $N_{3.StateTimer}$ | $N_{1.LocalTimer}$ | $N_{2.LocalTimer}$ | $N_{3.LocalTimer}$ | $\Delta_{Net}(C)$ |
|---|---|---|---|---|---|---|---|
| … | 6 | 6 | 5 | 0 | 0 | 999 | 12 |
| … | 7 | 7 | 6 | 1 | 1 | 0 | 1 |
| … | … | … | … | … | … | … | … |
| t + 0 | $1000^{-----}$, *Sync* | $998^{-----}$ | $988^{-----}$ | 994 | 992 | 982 | 12 |
| t + 1 | $1000^{-----}$ | $999^{-----}$ | $989^{-----}$ | 995 | 993 | 983 | 12 |
| t + 2 | $1000^{-----}$ | $1000^{-----}$, *Sync* | $990^{-----}$ | 996 | 994 | 984 | 12 |
| t + 3 | $1000^{-----}$ | $1000^{x----}$ | $991^{-----}$ | 997 | 995 | 985 | 12 |
| t + 4 | $1000^{x----}$, *Sync* | $1000^{x----}$ | $992^{x----}$ | 998 | 996 | 986 | 12 |
| t + 5 | $1000^{xx---}$ | $1000^{x----}$ | $993^{xx---}$ | 999 | 997 | 987 | 12 |
| t + 6 | $1000^{xx---}$ | $1000^{xx---}$, *Sync* | $994^{xx---}$ | 1000 | 998 | 988 | 12 |
| t + 7 | $1000^{xx---}$ | $1000^{xx---}$ | $995^{xx---}$ | 1001 | 999 | 989 | 12 |
| t + 8 | $0^{xx--x}$, *Accept* | $0^{xx--x}$, *Accept* | $996^{xx---}$ | 1002 | 1000 | 990 | 12 |
| t + 9 | $0^{xx--x}$, *Accept* | $0^{xx--x}$, *Accept* | $0^{xx--x}$, *Accept* | 1003 | 1001 | 991 | 12 |
| t + 10 | $0^{xx--x}$, *Accept* | $0^{xx--x}$, *Accept* | $0^{xx--x}$, *Accept* | 1004 | 1002 | 992 | 12 |
| t + 11 | $0^{xx-xx}$, *Accept* | $0^{xx--x}$, *Accept* | $0^{xx--x}$, *Accept* | 1005 | 1003 | 993 | 12 |
| t + 12 | $1^{-x-x-}$ | $1^{-x-x-}$ | $0^{-x-xx}$, *Accept* | 1006 | 1004 | 994 | 12 |
| t + 13 | $2^{---x-}$ | $2^{-x-x-}$ | $1^{---x-}$ | 1007 | 1005 | 995 | 12 |
| t + 14 | $3^{---x-}$ | $3^{---x-}$ | $2^{---x-}$ | 1008 | 1006 | 996 | 12 |
| t + 15 | $4^{-----}$ | $4^{---x-}$ | $3^{---x-}$ | 1009 | 1007 | 997 | 12 |
| t + 16 | $5^{-----}$ | $5^{-----}$ | $4^{-----}$ | 1010 | 1008 | 998 | 12 |
| t + 17 | $6^{-----}$ | $6^{-----}$ | $5^{-----}$ | 0 | 0 | 999 | 12 |
| t + 18 | $7^{-----}$ | $7^{-----}$ | $6^{-----}$ | 1 | 1 | 0 | 1 |
| t + 19 | $8^{-----}$ | $8^{-----}$ | $7^{-----}$ | 2 | 2 | 1 | 1 |
| … | … | … | … | … | … | … | 1 |