NASA/TM-2006-214322

A Byzantine-Fault Tolerant Self-Stabilizing Protocol for Distributed Clock Synchronization Systems

Mahyar R. Malekpour
Langley Research Center, Hampton, Virginia

August 2006

The NASA STI Program Office . . . in Profile

Since its founding, NASA has been dedicated to the advancement of aeronautics and space science. The NASA Scientific and Technical Information (STI) Program Office plays a key part in helping NASA maintain this important role.

The NASA STI Program Office is operated by Langley Research Center, the lead center for NASA's scientific and technical information. The NASA STI Program Office provides access to the NASA STI Database, the largest collection of aeronautical and space science STI in the world. The Program Office is also NASA's institutional mechanism for disseminating the results of its research and development activities. These results are published by NASA in the NASA STI Report Series, which includes the following report types:

- TECHNICAL PUBLICATION. Reports of completed research or a major significant phase of research that present the results of NASA programs and include extensive data or theoretical analysis. Includes compilations of significant scientific and technical data and information deemed to be of continuing reference value. NASA counterpart of peer-reviewed formal professional papers, but having less stringent limitations on manuscript length and extent of graphic presentations.

- TECHNICAL MEMORANDUM. Scientific and technical findings that are preliminary or of specialized interest, e.g., quick release reports, working papers, and bibliographies that contain minimal annotation. Does not contain extensive analysis.

- CONTRACTOR REPORT. Scientific and technical findings by NASA-sponsored contractors and grantees.

- CONFERENCE PUBLICATION. Collected papers from scientific and technical conferences, symposia, seminars, or other meetings sponsored or co-sponsored by NASA.

- SPECIAL PUBLICATION. Scientific, technical, or historical information from NASA programs, projects, and missions, often concerned with subjects having substantial public interest.

- TECHNICAL TRANSLATION. English-language translations of foreign scientific and technical material pertinent to NASA's mission.

Specialized services that complement the STI Program Office's diverse offerings include creating custom thesauri, building customized databases, organizing and publishing research results ... even providing videos.

For more information about the NASA STI Program Office, see the following:

- Access the NASA STI Program Home Page at ***http://www.sti.nasa.gov***

- E-mail your question via the Internet to help@sti.nasa.gov

- Fax your question to the NASA STI Help Desk at (301) 621-0134

- Phone the NASA STI Help Desk at (301) 621-0390

- Write to:
  NASA STI Help Desk
  NASA Center for AeroSpace Information
  7121 Standard Drive
  Hanover, MD 21076-1320

NASA/TM-2006-214322

A Byzantine-Fault Tolerant Self-Stabilizing Protocol for Distributed Clock Synchronization Systems

Mahyar R. Malekpour
Langley Research Center, Hampton, Virginia

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23681-2199

August 2006

Acknowledgment

This work was supported by NASA's Vehicle Systems Program. The author would like to thank the following for their reviews, helpful comments, consultations and support: Ricky Butler, Victor Carreno, Ben DiVito, Paul Miner, Cesar Munoz, Radu Siminiceanu, and Wilfredo Torres-Pomales. The author would especially like to thank Celeste Belcastro without whose support this work would not have been possible.

Available from:

NASA Center for AeroSpace Information (CASI)
7121 Standard Drive
Hanover, MD 21076-1320
(301) 621-0390

National Technical Information Service (NTIS)
5285 Port Royal Road
Springfield, VA 22161-2171
(703) 605-6000

Abstract

Embedded distributed systems have become an integral part of safety-critical computing applications, necessitating system designs that incorporate fault tolerant clock synchronization in order to achieve ultra-reliable assurance levels. Many efficient clock synchronization protocols do not, however, address Byzantine failures, and most protocols that do tolerate Byzantine failures do not self-stabilize. Of the Byzantine self-stabilizing clock synchronization algorithms that exist in the literature, they are based on either unjustifiably strong assumptions about initial synchrony of the nodes or on the existence of a common pulse at the nodes. The Byzantine self-stabilizing clock synchronization protocol presented here does not rely on any assumptions about the initial state of the clocks. Furthermore, there is neither a central clock nor an externally generated pulse system. The proposed protocol converges deterministically, is scalable, and self-stabilizes in a short amount of time. The convergence time is linear with respect to the self-stabilization period. Proofs of the correctness of the protocol as well as the results of formal verification efforts are reported.

Table of Contents

1. Introduction

Synchronization and coordination algorithms are part of distributed computer systems. Clock synchronization algorithms are essential for managing the use of resources and controlling communication in a distributed system. Also, a fundamental criterion in the design of a robust distributed system is to provide the capability of tolerating and potentially recovering from failures that are not predictable in advance. Overcoming such failures is most suitably addressed by tolerating Byzantine faults [Lamport 1982]. A Byzantine-fault model encompasses all unexpected failures, including transient ones, within the limitations of the maximum number of faults at a given time. Driscoll et al. [Driscoll 2003] addressed the frequency of occurrences of Byzantine faults in practice and the necessity to tolerate Byzantine faults in ultra-reliable distributed systems. A distributed system tolerating as many as F Byzantine faults requires a network size of more than $3F$ nodes. Lamport et al. [Lamport 1982, Lamport 1985] were the first to present the problem and show that Byzantine agreement cannot be achieved for fewer than $3F$ +1 nodes. Dolev et al. [Dolev 1984] proved that at least $3F + 1$ nodes are necessary for clock synchronization in the presence of F Byzantine faults.

A distributed system is defined to be self-stabilizing if, from an arbitrary state and in the presence of bounded number of Byzantine faults, it is guaranteed to reach a legitimate state in a finite amount of time and remain in a legitimate state as long as the number of Byzantine faults are within a specific bound. A legitimate state is a state where all good clocks in the system are synchronized within a given precision bound.

Therefore, a self-stabilizing system is able to start in a random state and recover from transient failures after the faults dissipate. The concept of self-stabilizing distributed computation was first presented in a classic paper by Dijkstra [Dijkstra 1974]. In that paper, he speculated whether it would be possible for a set of machines to stabilize their collective behavior in spite of unknown initial conditions and distributed control. The idea was that the system should be able to converge to a legitimate state within a bounded amount of time, by itself, and without external intervention.

This paper addresses the problem of synchronizing clocks in a distributed system in the presence of Byzantine faults. There are many algorithms that address permanent faults [Srikanth 1985], where the issue of transient failures is either ignored or inadequately addressed. There are many efficient Byzantine clock synchronization algorithms that are based on assumptions on initial synchrony of the nodes [Srikanth 1985, Welch 1988] or existence of a common pulse at the nodes [Dolev 2004]. There are many clock synchronization algorithms that are based on randomization and, therefore, are non-deterministic [Dolev 2004]. Some clock synchronization algorithms have provisions for initialization and/or reintegration. However, solving these special cases is insufficient to make the algorithm self-stabilizing. A self-stabilizing algorithm encompasses these special scenarios without having to address them separately. The main challenges associated with self-stabilization are the complexity of the design and the proof of correctness of the protocol. Another difficulty is achieving efficient convergence time for the proposed self-stabilizing protocol.

Other recent developments in this area are the algorithms developed by Daliot et al [Daliot 2003A and 2003B]. The algorithm in [Daliot 2003B] is called the Byzantine self-stabilization pulse synchronization (BSS-Pulse-Synch) protocol. A flaw in BSS-Pulse-Synch protocol was found and documented in [Malekpour 2006]. The biologically inspired Pulse Synchronization protocol in [Daliot 2003A] has claims of self-stabilization, but no mechanized[1] proofs are provided.

In this paper a rapid Byzantine self-stabilizing clock synchronization protocol is presented that self-stabilizes from any state, tolerates bursts of transient failures, and deterministically converges within a linear convergence time with respect to the self-stabilization period. Upon self-stabilization, all good clocks proceed synchronously. This protocol has been the subject of rigorous verification efforts that support the claim of correctness.

The following sections describe the proposed protocol in detail. The report begins with the underlying topology and network model, followed by a description of the protocol. A proof of the protocol is presented in the following section. The protocol characteristics are then discussed. A summary of the simulation and model checking results is reported. Some of the potential applications are enumerated, followed by potential future work in this area.

2. Topology

The underlying topology considered here is a network of K nodes that communicate by exchanging messages through a set of communication channels. The communication channels are assumed to connect a set of source nodes to a set of destination nodes such that the source of a given message is distinctly identifiable from other sources of messages. This system of K nodes can tolerate a maximum of F Byzantine faulty nodes, where $K \geq 3F + 1$. Therefore, the minimum number of good nodes in the system, G, is given by $G = K\text{-}F$ and thus $G \geq (2F + 1)$ nodes. Let K_G represent the set of good nodes. The nodes communicate with each other by exchanging broadcast messages. Broadcast of a message to all other nodes is realized by transmitting the message to all other nodes at the same time. The source of a message is assumed to be uniquely identifiable. The communication network does not guarantee any order of arrival of a transmitted message at the receiving nodes. To paraphrase Kopetz [Kopetz 1997], a consistent delivery order of a set of messages does not necessarily reflect the temporal or causal order of the events.

Each node is driven by an independent local physical oscillator. The oscillators of good nodes have a known bounded drift rate, $1 >> \rho \geq 0$, with respect to real time. Each node has two logical time clocks, *Local_Timer* and *State_Timer*, which locally keep track of the passage of time as indicated by the physical oscillator. In the context of this report, all references to clock synchronization and self-stabilization of the system are with respect to the *State_Timer* and the *Local_Timer* of the nodes. There is neither a central clock nor an externally generated global

[1] A mechanized proof is a formal verification via either a theorem prover or model checker.

pulse. The communication channels and the nodes can behave arbitrarily, provided that eventually the system adheres to the system assumptions (see Section 3.5).

The latency of interdependent communications between the 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 1. In Figure 1, a message transmitted by node N_i at real time t_0 is expected to arrive at all destination nodes N_j, be processed, and subsequent messages generated by N_j within the time interval of $[t_0 + D, t_0 + D + d]$ for all $N_j \in K_G$. Communication between independently clocked nodes is inherently imprecise. The network imprecision, d, is the maximum time difference between all good receivers, N_j, of a message from N_i with respect to real time. The imprecision is due to the drift of the clocks with respect to real time, jitter, discretization error, and slight variations in the communication delay due to various causes such as temperature effects and differences in the lengths of the physical communication medium. These two parameters are assumed to be bounded such that $D \geq 1$ and $d \geq 0$ and both have values with units of real time nominal tick. For the remainder of this report, all references to time are with respect to the nominal tick and are simply referred to as clock ticks.

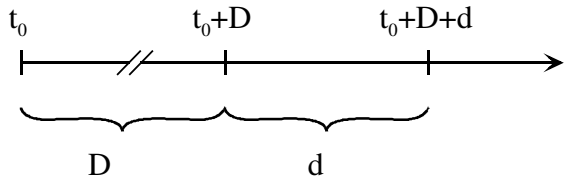

Figure 1. Event-response delay, D, and network imprecision, d.

3. Protocol Description

The self-stabilization problem has two facets. First, it is inherently **event-driven** and, second, it is **time-driven**. Most attempts at solving the self-stabilization problem have focused only on the event-driven aspect of this problem. Additionally, all efforts toward solving this problem must recognize that the system undergoes two distinct phases, un-stabilized and stabilized, and that once stabilized, the system state needs to be preserved. The protocol presented here properly merges the *time* and *event* driven aspects of this problem in order to self-stabilize the system in a gradual and yet timely manner. Furthermore, this protocol is based on the concept of a continual vigilance of state of the system in order to maintain and guarantee its stabilized status, and a continual reaffirmation of nodes by declaring their internal status. Finally, initialization and/or reintegration are not treated as special cases. These scenarios are regarded as inherent part of this self-stabilizing protocol.

The self-stabilization events are captured at a node via a selection function that is based on received valid messages from other nodes. When such an event occurs, it is said that a node has **accepted** or an **accept event** has occurred.

When the system is stabilized, it is said to be in the **steady state**.

In order to achieve self-stabilization, the nodes communicate by exchanging two self-stabilization messages labeled **Resync** and **Affirm**. The *Resync* message reflects the time-driven aspect of this self-stabilization protocol, while the *Affirm* message reflects the event-driven aspect of it. The *Resync* message is transmitted when a node realizes that the system is no longer stabilized or as a result of a resynchronization timeout. It indicates that the originator of the *Resync* message has to reset and try to reengage in the self-stabilization process with other nodes. The *Affirm* message is transmitted periodically and at specific intervals primarily in response to a legitimate self-stabilization *accept event* at the node. The *Affirm* message either indicates that the node is in the transition process to another state in its attempt toward synchronization, or reaffirms that the node will remain synchronized. The timing diagram of transmissions of a good node during the *steady state* is depicted in Figure 2. In the following figures, *Resync* messages are represented as *R* and *Affirm* messages are represented as *A*. The line segments indicate the time of the transmission of messages. As depicted, the expected sequence of messages transmitted by a good node is a *Resync* message followed by a number of *Affirm* messages, i.e. *RAAA … AAARAA*. The exact number of consecutive *Affirm* messages will be accounted for later in this report.

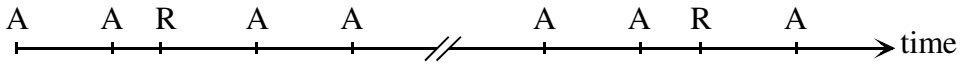

Figure 2. Timing diagram of transmissions of a good node during the *steady state*.

The time difference between the interdependent consecutive events is expressed in terms of the minimum event-response delay, D, and network imprecision, d. As a result, the approach presented here is expressed as a self-stabilization of the system as a function of the expected time separation between the consecutive *Affirm* messages, Δ_{AA}. To guarantee that a message from a good node is received by all other good nodes before a subsequent message is transmitted, Δ_{AA} is constrained such that $\Delta_{AA} \geq (D + d)$. Unless stated otherwise, all time dependent parameters of this protocol are measured locally and expressed as functions of Δ_{AA}.

In Figure 3, node N_i is shown to transmit two consecutive *Affirm* messages. In the *steady state*, N_i receives one *Affirm* message from every good node between any two consecutive *Affirm* messages it transmits. Since the messages may arrive at any time after the transmission of an *Affirm* message, the *accept event* can occur at any time prior to the transmission of the next *Affirm* message.

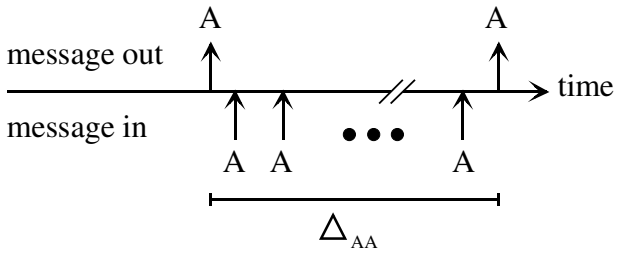

Figure 3. Typical activities of N_i between two *A* messages in a stabilized system.

Three **fundamental parameters** characterize the self-stabilization protocol presented here, namely K, D, and d. The number of faulty nodes, F, the number of good nodes, G, and the remaining parameters that are subsequently enumerated are **derived parameters** and are based on these three fundamental parameters. Furthermore, except for K, F, and G which are integer numbers, all other parameters are real numbers. In particular, Δ_{AA} is used as a threshold value for monitoring of proper timing of incoming and outgoing *Affirm* messages. The derived parameters $T_A = G - 1$ and $T_R = F + 1$ are used as thresholds in conjunction with the *Affirm* and *Resync* messages, respectively.

3.1. The Monitor

The transmitted messages to be delivered to the destination nodes are deposited on communication channels. To closely observe the behavior of other nodes, a node employs (K-1) *monitors,* one *monitor* for each source of incoming messages as shown in Figure 4. A node neither uses nor monitors its own messages. The distributed observation of other nodes localizes error detection of incoming messages to their corresponding *monitors*, and allows for modularization and distribution of the self-stabilization protocol process within a node. A *monitor* keeps track of the activities of its corresponding source node. A *monitor* detects proper sequence and timeliness of the received messages from its corresponding source node. A *monitor* reads, evaluates, time stamps, validates, and stores only the last message it receives from that node. Additionally, a *monitor* ascertains the health condition of its corresponding source node by keeping track of the current state of that node. As K increases so does the number of *monitors* instantiated in each node. Although similar modules have been used in engineering practice and, conceptually, by others in theoretical work, as far as the author is aware this is the first use of the *monitors* as an integral part of a self-stabilization protocol.

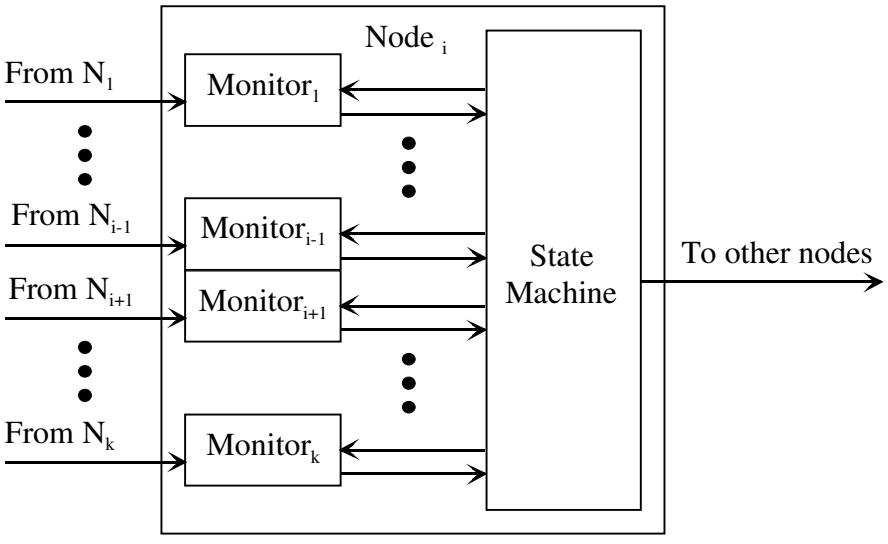

Figure 4. The i[th] node, N_i, with its *monitors* and state machine.

3.2. The State Machine

The assessment results of the monitored nodes are utilized by the node in the self-stabilization process. The node consists of a state machine and a set of (K-1) *monitors*. The state machine has two states, **Restore** state (T) and **Maintain** state (M), that reflect the current state of the node in the system as shown in Figure 5. The state machine describes the collective behavior of the node, N_i, utilizing assessment results from its *monitors*, M_1 .. M_{i-1}, M_{i+1} .. M_K as shown in Figure 4, where M_j is the *monitor* for the corresponding node N_j. In addition to the behavior of its corresponding source node, a *monitor*'s internal status is influenced by the current state of the node's state machine. In a master-slave fashion, when the state machine transitions to another state it directs the *monitors* to update their internal status.

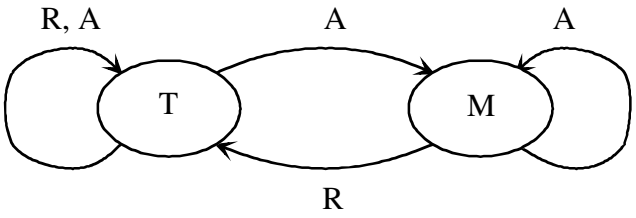

Figure 5. The node state machine.

The **transitory conditions** enable the node to migrate to the *Maintain* state and are defined as:

1. The node is in the *Restore* state,
2. At least $2F$ *accept events* in as many Δ_{AA} intervals have occurred after the node entered the *Restore* state,
3. No *valid Resync* messages are received for the last *accept event*.

The **transitory delay** is the length of time a node stays in the *Restore* state.

The minimum required duration for the *transitory delay* is $2F\Delta_{AA}$ after the node enters the *Restore* state. The maximum duration of the *transitory delay* is dependent on the number of additional *valid Resync* messages received. Validity of received messages is defined in Section 3.3. When the system is stabilized, the maximum delay is a result of receiving *valid Resync* messages from all faulty nodes. Since there are at most F faulty nodes present, during the *steady state* operation the duration of the *transitory delay* is bounded by $[2F\Delta_{AA}, 3F\Delta_{AA}]$.

A node in either of the *Restore* or *Maintain* state periodically transmits an *Affirm* message every Δ_{AA}. When in the *Restore* state, it either will meet the *transitory conditions* and transition to the *Maintain* state, or will remain in the *Restore* state for the duration of the self-stabilization period until it times out and transmits a *Resync* message. When in the *Maintain* state, a node either will remain in the *Maintain* state for the duration of the self-stabilization period until it times out, or will unexpectedly transition to the *Restore* state because T_R other nodes have transitioned out of the *Maintain* state. At the transition, the node transmits a *Resync* message.

The self-stabilization period is defined as the maximum time interval (during the *steady state*) that a good node engages in the self-stabilization process. In this protocol the self-

stabilization period depends on the current state of the node. Specifically, the self-stabilization period for the *Restore* state is represented by P_T and the self-stabilization period for the *Maintain* state is represented by P_M. P_T and P_M are expressed in terms of Δ_{AA}. The length of time a good node stays in the *Restore* state is denoted by L_T. During the *steady state* L_T is always less than P_T. The time a good node stays in the *Maintain* state is denoted by L_M. When the system is stabilized L_M is less than or equal to P_M. The effective self-stabilization period, $P_{Effective}$, is the time interval between the last two consecutive resets of the *Local_Timer* of a good node in a stabilized system, where $P_{Effective} = L_T + L_M < P_T + P_M$.

In Figure 6 the transitions of a node from the *Restore* state to the *Maintain* state (during the *steady state*) are depicted along a timeline of activities of the node. The line segments in Figure 6 indicate timing and order of the transmission of messages along the time axis. Two new parameters, Δ_{RA} and Δ_{AR}, are introduced in this figure in order to clarify other aspects of this protocol's behavior. These parameters are defined in terms of Δ_{AA}. Although a *Resync* message is transmitted immediately after the node realizes that it is no longer stabilized, i.e. $0 < \Delta_{AR} \leq \Delta_{AA}$, an *Affirm* message is transmitted once every Δ_{AA}, i.e. $\Delta_{RA} = \Delta_{AA}$.

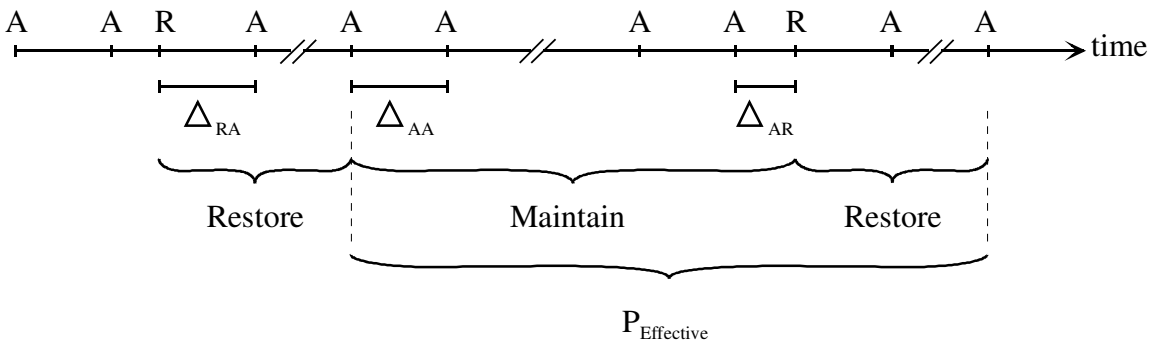

Figure 6. Timing diagram of activities of a good node during the *steady state*.

A node keeps track of time by incrementing a logical time clock, *State_Timer*, once every Δ_{AA}. After the *State_Timer* reaches P_T or P_M, depending on the current state of the node, the node experiences a timeout, transmits a new *Resync* message, resets the *State_Timer*, transitions to the *Restore* state, and attempts to resynchronize with other nodes. If the node was in the *Restore* state it remains in that state after the timeout. The current value of this timer reflects the duration of the current state of the node. It also provides insight in assessing the state of the system in the self-stabilization process.

In addition to the *State_Timer*, the node maintains the logical time clock *Local_Timer*. The *Local_Timer* is incremented once every Δ_{AA} and is reset only when the node has transitioned to the *Maintain* state and remained in that state for the duration of $\lceil \Delta_{Precision} \rceil$, where $\Delta_{Precision}$ is the maximum guaranteed self-stabilization precision. The *Local_Timer* is intended to be used by higher level protocols and is used in assessing the state of the system in the self-stabilization process.

The *monitor*'s status reflects its perception of its corresponding source node. In particular, a *monitor* keeps track of the incoming messages from its corresponding source and

ensures that only *valid* messages are stored. If the expected time of arrival of a message is violated or if the message arrives out of the expected sequence, then it is marked as invalid. Otherwise, it is marked as *valid* and stored for the host node's consumption. It is important to note that this protocol is expected to be used as the fundamental mechanism in bringing and maintaining a system within a known synchronization bound. This protocol neither maintains a history of past behavior of the nodes nor does it attempt to classify the nodes into good and faulty ones. All such determination about the health status of the nodes in the system is assumed to be done by higher level mechanisms.

3.3. Message Sequence

An **expected sequence** is defined as a stream of *Affirm* messages enclosed by two *Resync* messages where all received messages arrive within their expected arrival times. The time interval between the last two *Resync* messages is represented by Δ_{RR}.

The following are three sequences where '-' represents a missing message:

- *RAAA … AAAR* *expected sequence*, all *A* messages present
- *RA-A … A--R* unexpected message sequence, missing *A* messages
- *R--- … ---R* unexpected message sequence, no *A* messages present

When a node is in the *Restore* state, its output sequence of messages has one of two patterns. If the node does not transition to the *Maintain* state, it times out after P_T and its *expected sequence* of output messages will be *RAAA … AAAR,* consisting of P_T consecutive *A* messages. In this case, $\Delta_{RR} = P_T$. On the other hand, when the node synchronizes with other nodes, it transitions to the *Maintain* state before timing out, and its *expected sequence* of output messages will have at least $2F$ *Affirm* messages followed by those *Affirm* messages produced in the *Maintain* state. The shortest amount of time it takes a node to transition to the *Maintain* state is $2F\Delta_{AA}$. The shortest amount of time the node stays in the *Maintain* state is Δ_{AR}. Therefore, the time separation between any two consecutive *Resync* messages from a good node is given by $\Delta_{RR} \geq 2F\Delta_{AA} + \Delta_{AR}$. As a result, the shortest *expected sequence* consists of $2F$ *A* messages enclosed by two *R* messages with a duration of $\Delta_{RR,min} = 2F\Delta_{AA} + 1$ clock ticks.

When a node is in the *Maintain* state, it has two possible output sequences of messages. If it times out after P_M, its *expected sequence* of output messages will be *RAAA … AAAR* consisting of an *R* message, followed by *A* messages for when the node was in the *Restore* state, followed by at least P_M consecutive *A* messages for the duration of the *Maintain* state, followed by another *R* message. Therefore, $(P_T + P_M) > \Delta_{RR}$, in other words, $\Delta_{RR,max} = (P_T + P_M)$. On the other hand, when the node abruptly transitions out of the *Maintain* state, its output sequence of messages will consist of fewer *Affirm* messages. The sequence consists of an *R* message, followed by *A* messages for when the node was in the *Restore* state, followed by *A* messages for the duration of the *Maintain* state, followed by another *R* message.

As depicted in Figure 6, starting from the last transmission of the *Resync* message consecutive *Affirm* messages are transmitted at Δ_{AA} intervals. At the receiving nodes, the following definitions hold:

- A message (*Resync* or *Affirm*) from a given source is ***valid*** if it is the first message from that source.
- An *Affirm* message from a given source is ***early*** if it arrives earlier than (Δ_{AA} - d) of its previous *valid* message (*Resync* or *Affirm*).
- A *Resync* message from a given source is ***early*** if it arrives earlier than $\Delta_{RR,min}$ of its previous *valid Resync* message.
- An *Affirm* message from a given source is ***valid*** if it is not *early*.
- A *Resync* message from a given source is ***valid*** if it is not *early*.

The protocol works when the received messages do not violate their timing requirements. However, in addition to inspecting the timing requirements, examining the *expected sequence* of the received messages provides stronger error detection at the nodes.

3.4. Protocol Functions

The functions used in this protocol are described in this section.

Two functions, *InvalidAffirm()* and *InvalidResync()*, are used by the *monitors*. The *InvalidAffirm()* function determines whether or not a received *Affirm* message is *valid*. The *InvalidResync()* function determines if a received *Resync* message is *valid*. When either of these functions returns a true value, it is indicative of an unexpected behavior by the corresponding source node.

The *Accept()* function is used by the state machine of the node in conjunction with the threshold value $T_A = G - 1$. When at least T_A *valid* messages (*Resync* or *Affirm*) have been received, this function returns a true value indicating that an *accept event* has occurred and such event has also taken place in at least F other good nodes. When a node accepts, it consumes all *valid* messages used in the accept process by the corresponding function. Consumption of a message is the process by which a *monitor* is informed that its stored message, if it existed and was *valid*, has been utilized by the state machine.

The *Retry()* function is used by the state machine of the node with the threshold value $T_R = F + 1$. This function determines if at least T_R other nodes have transitioned out of the *Maintain* state. A node, via its *monitors*, keeps track of the current state of other nodes. When at least T_R *valid Resync* messages from as many nodes have been received, this function returns a true value indicating that at least one good node has transitioned to the *Restore* state. This function is used to transition from the *Maintain* state to the *Restore* state.

The *TransitoryConditionsMet()* function is used by the state machine of the node to determine proper timing of the transition from the *Restore* state to the *Maintain* state. This function keeps track of the *accept events*, by incrementing the *Accept_Event_Counter*, to determine if at least $2F$ *accept events* in as many Δ_{AA} intervals have occurred. It returns a true value when the *transitory conditions* (see Section 3.2) are met.

The *TimeOutRestore()* function uses P_T as a boundary value and asserts a timeout condition when the value of the *State_Timer* has reached P_T. Such timeout triggers the node to reengage in another round of self-stabilization process. This function is used when the node is in the *Restore* state.

The *TimeOutMaintain()* function uses P_M as a boundary value and asserts a timeout condition when the value of the *State_Timer* has reached P_M. Such timeout triggers the node to reengage in another round of synchronization. This function is used when the node is in the *Maintain* state.

In addition to the above functions, the state machine utilizes the *TimeOutAcceptEvent()* function. This function is used to regulate the transmission time of the next *Affirm* message. This function maintains a *DelatAA_Timer* by incrementing it once per local clock tick and once it reaches the transmission time of the next *Affirm* message, Δ_{AA}, it returns a true value. In the advent of such timeout, the node transmits an *Affirm* message.

3.5. System Assumptions

1. The source of the transient faults has dissipated.
2. All good nodes actively participate in the self-stabilization process and execute the protocol.
3. At most F of the nodes are faulty.
4. The source of a message is distinctly identifiable by the receivers from other sources of messages.
5. A message sent by a good node will be received and processed by all other good nodes within Δ_{AA}, where $\Delta_{AA} \geq (D + d)$.
6. The initial values of the state and all variables of a node can be set to any arbitrary value within their corresponding range. In an implementation, it is expected that some local capabilities exist to enforce type consistency of all variables.

3.6. The Self-Stabilizing Clock Synchronization Problem

To simplify the presentation of this protocol, it is assumed that all time references are with respect to a real time t_0 when the *system assumptions* are satisfied and the system operates within the *system assumptions*. Let

- C be the maximum convergence time,
- $\Delta_{Local\_Timer}(t)$, for real time t, the maximum time difference of the *Local_Timers* of any two good nodes N_i and N_j, and
- $\Delta_{Precision}$ the maximum guaranteed self-stabilization precision between the *Local_Timer*'s of any two good nodes N_i and N_j in the presence of a maximum of F faulty nodes, $\forall\ N_i, N_j \in K_G$.

Convergence: From any state, the system converges to a self-stabilized state after a finite amount of time.

1. $\forall\ N_i, N_j \in K_G$, $\Delta_{Local\_Timer}(C) \leq \Delta_{Precision}$.
2. $\forall\ N_i, N_j \in K_G$, at C, N_i perceives N_j as being in the *Maintain* state.

Closure: When all good nodes have converged such that $\Delta_{Local\_Timer}(C) \leq \Delta_{Precision}$ at time C, the system shall remain within the self-stabilization precision $\Delta_{Precision}$ for $t \geq C$, for real time t.

$$\forall N_i, N_j \in K_G,\ t \geq C,\ \Delta_{Local\_Timer}(t) \leq \Delta_{Precision},$$

where,

$$C = (2P_T + P_M)\ \Delta_{AA},$$

$$\begin{aligned}\Delta_{Local\_Timer}(t) = \min\ (&\max\ (Local\_Timer_i, Local\_Timer_j) - \\ &\min\ (Local\_Timer_i, Local\_Timer_j), \\ &\max\ (Local\_Timer_i - \lceil \Delta_{Precision} \rceil, Local\_Timer_j - \lceil \Delta_{Precision} \rceil) - \\ &\min\ (Local\_Timer_i - \lceil \Delta_{Precision} \rceil, Local\_Timer_j - \lceil \Delta_{Precision} \rceil)),\end{aligned}$$

$$\lceil \Delta_{Precision} \rceil = truncate\ (\Delta_{Precision} + 0.5),$$

and,

$(Local\_Timer - \lceil \Delta_{Precision} \rceil)$ is the $\lceil \Delta_{Precision} \rceil^{th}$ previous value of the *Local_Timer*

and,

$$\Delta_{Precision} = (3F - 1)\ \Delta_{AA} - D + \Delta_{Drift}$$

where, the amount of drift from the initial precision is give by

$$\Delta_{Drift} = ((1+\rho) - 1/(1+\rho))\ P_M\ \Delta_{AA}.$$

4. The Byzantine-Fault Tolerant Self-Stabilizing Protocol for Distributed Clock Synchronization Systems

The presented protocol is described in Figure 7 and consists of a state machine and a set of *monitors* which execute once every local oscillator tick.

```
Monitor:
case (incoming message from the
corresponding node)
{Resync:
    if InvalidResync() then
        Invalidate the message

    else
        Validate and store the message,
        Set state status of the source.
```

```
Affirm:
    if InvalidAffirm() then
        Invalidate the message
    else
        Validate and store the message.

Other:
    Do nothing.

} // case
```

```
Node:
case (state of the node)
{Restore:
    if TimeOutRestore() then
        Transmit Resync message,
        Reset State_Timer,
        Reset DelatAA_Timer,
        Reset Accept_Event_Counter,
        Stay in Restore state,

    elsif TimeOutAcceptEvent() then
        Transmit Affirm message,
        Reset DelatAA_Timer,
        if Accept() then
            Consume valid messages,
            Clear state status of the sources,
            Increment Accept_Event_Counter,
            if TransitoryConditionsMet() then
                Reset State_Timer,
                Go to Maintain state,
            else
                Stay in Restore state.
        else
            Stay in Restore state.,
    else
        Stay in Restore state.
```

```
Maintain:
    if TimeOutMaintain() or Retry() then
        Transmit Resync message,
        Reset State_Timer,
        Reset DelatAA_Timer,
        Reset Accept_Event_Counter,
        Go to Restore state,

    elsif TimeOutAcceptEvent() then
        if Accept() then
            Consume valid messages.,
        if (State_Timer = ⌈Δ_Precision⌉)
            Reset Local_Timer.,
        Transmit Affirm message, †
        Reset DelatAA_Timer,
        Stay in Maintain state,

    else
        Stay in Maintain state.
} // case
```

Figure 7. The self-stabilization protocol.

4.1. Semantics of the pseudo-code

- Indentation is used to show a block of sequential statements.
- ',' is used to separate sequential statements.
- '.' is used to end a statement.
- '.,' is used to mark the end of a statement and at the same time to separate it from other sequential statements.

† In a variation of this protocol and in conjunction with a higher level mechanism, a good node stops transmitting *Affirm* messages after it is determined by the higher level mechanism that the system has stabilized. It follows from Theorem *StopContinuousTransmit* that such variation preserves the self-stabilization properties. Nevertheless, such optimization in the number of exchanged self-stabilization messages is at a cost of delaying error detection, introducing jitters in the system, and prolonging the self-stabilization process.

5. Proof of the Protocol

The approach for the proof is to show that a system of $K \geq 3F + 1$ nodes converges from any condition to a state where all good nodes are in the *Maintain* state. This system is then shown to remain within the timing bounds of the self-stabilization precision of $\Delta_{Precision}$. The Lemmas and Theorems are presented in this section.

Since the oscillator drift rate, ρ, does not play a significant role in the convergence process, it is omitted from the expressions regarding parameters, constants, equations, and the proofs. However, ρ does affect the closure property and is included in expressions regarding $\Delta_{Precision}$. Omission of ρ does not change the behavior of the protocol or the validity of the proofs.

Assumptions: All good nodes are active and the system operates within the *system assumptions*. In this proof, unless otherwise stated in the Lemmas and Theorems, no other assumptions are made about the system. Also, throughout the proofs, unless stated otherwise, all references to the *Resync* and *Affirm* messages are with respect to *valid* messages.

A node behaves **properly** if it executes the protocol.

Lemma TransmitEveryΔ_{AA} – *A good node in either Restore* or *Maintain state transmits at least one message (Resync or Affirm) every Δ_{AA} interval.*
Proof – It follows from the protocol that the *DelatAA_Timer* is reset after transmission of a self-stabilization message (*Resync* or *Affirm*). It is expressed in function *TimeOutAcceptEvent()* that the node transmits an *Affirm* message every Δ_{AA} interval. Additionally, if after transmitting an *Affirm* message and within the next Δ_{AA} interval, a node times out to engage in another round of self-stabilization process, it will also transmit a *Resync* message within that Δ_{AA} interval. ♦

Theorem ResyncWithinP_T – *A good node remaining in the Restore state transmits a Resync message within at most $P_T \Delta_{AA}$ clock ticks.*
Proof – It is expressed in function *TimeOutRestore()* that if a node remains in the *Restore* state and does not transition to the *Maintain* state, it will time out within $P_T \Delta_{AA}$ clock ticks, transmit a *Resync* message, and stay in the *Restore* state. ♦

Theorem RestoreWithinP_M –*A good node in the Maintain state transitions to the Restore state within at most $P_M \Delta_{AA}$ clock ticks.*
Proof – It follows from the protocol that a node in the *Maintain* state will transition from the *Maintain* state to the *Restore* state either because of a resynchronization timeout, as expressed in function *TimeOutMaintain()*, or when the system becomes unstabilized, as expressed in function *Retry()*. Upon transitioning to the *Restore* state, the node transmits a *Resync* message. Since the longest such time interval is due to the timeout, the node transmits a *Resync* message in at most $P_M \Delta_{AA}$ clock ticks. ♦

Lemma DeltaRRmin – *The shortest time interval between any two consecutive Resync messages from a good node is $2F\Delta_{AA} + 1$ clock ticks.*
Proof – From the definition of the *transitory conditions* in Section 3.2, the minimum required duration for the *transitory delay* is $2F\Delta_{AA}$ after the node entered the *Restore* state. The shortest amount of time the node stays in the *Maintain* state is Δ_{AR}, as shown in Figure 6. Therefore, the time separation between any two consecutive *Resync* messages from a good node is given by $\Delta_{RR} \geq 2F\Delta_{AA} + \Delta_{AR}$. As a result, $\Delta_{RR,min} = 2F\Delta_{AA} + 1$ clock ticks. ♦

Theorem RestoreToMaintain – *A good node in the Restore state will always transition to the Maintain state.*
Proof – Let us consider the worst case scenario where a node wakes up with its internal variables randomly set except that its state is set to the *Restore* state. A sequence of activities of the good node N_5, for a system of $K = 7$, $F = 2$ and $G = 5$ nodes, is depicted in Figure 8. The activities of the good node are partitioned in different zones along the time axis. The following symbols are used in this figure.

X	= don't care
A	= an *Affirm* message transmitted
R_{gi}	= a *Resync* message received from the i^{th} good node
R_{fj}	= a *Resync* message received from the j^{th} faulty node

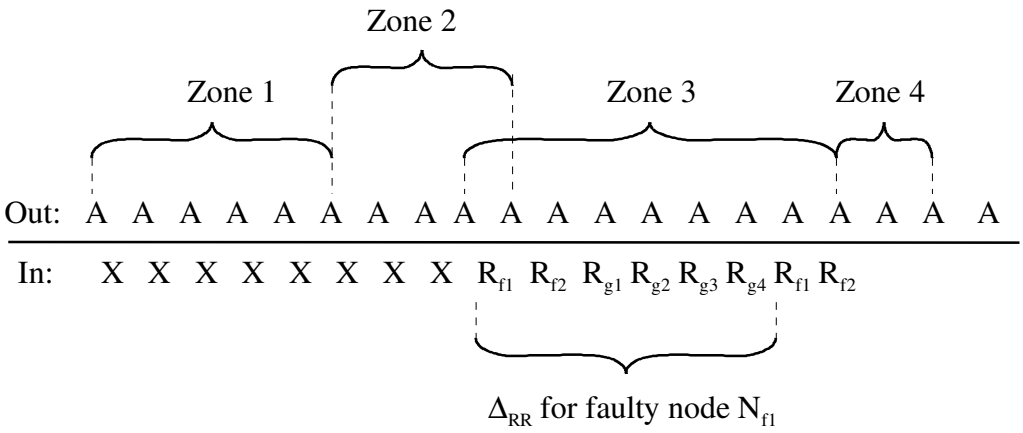

Figure 8. Worst case sequence of activities of a good node after random start up for $F = 2$.

Since receiving a *Resync* message can force the node to remain in a state of transition, only the *Resync* messages are shown in this figure. Also, since receiving one *Resync* message during the current Δ_{AA} interval can prevent the node from transitioning to the *Maintain* state, the sequence of activities are shown for the worst case scenario where only one *Resync* message is received within the time interval of any two consecutive transmissions of *Affirm* messages, i.e. at every Δ_{AA}.

Zone1: If a good node N_i perceives that it has received a *Resync* message from another good node N_j, it follows from Lemma *DeltaRRmin* that for the duration of $\Delta_{RR,min}$ a *Resync* message from N_j will be rejected. Therefore, for the worst case scenario, let us assume that a good node does not receive enough *valid* messages and *accept events* will not take place for the first $(2F+1)\Delta_{AA} > \Delta_{RR,min}$ clock ticks. However, it follows from Lemma *TransmitEvery*Δ_{AA} that all good nodes transmit a message every Δ_{AA} interval. Therefore, by Lemma *DeltaRRmin*, after $\Delta_{RR,min}$ a good node will receive at least T_A messages for all subsequent Δ_{AA} intervals and consequently *accept events* will take place during those intervals.

Zone 2: It follows from the protocol that a node has to wait for the minimum *transitory delay* of $2F\Delta_{AA}$ before transitioning to the *Maintain* state. To prevent the node from transitioning to the *Maintain* state, the minimum *transitory delay* should not be met. Therefore, a *Resync* message has to be received at the last Δ_{AA} interval. As a result, duration of this zone will be $(2F - 1)\Delta_{AA}$ intervals.

Zone 3: To prolong the duration of the *Restore* state for this node, N_5, the faulty nodes transmit *Resync* messages interleaved with the *Resync* messages from the other good nodes, N_1 through N_4, such that these messages are perceived *valid* by N_5. From Lemma *DeltaRRmin*, *Resync* messages have to be at least $\Delta_{RR,min}$ apart in order to be considered *valid*. Since there are F faulty nodes, the remaining $F + 1$ intervals between the consecutive *Resync* messages of a faulty node must be filled by *Resync* messages from other good nodes. As expressed in function *Retry()*, it takes T_R *valid Resync* messages for a good node to transition from the *Maintain* state to the *Restore* state.

Since $T_R = F + 1$, after transmission of T_R *Resync* messages from as many good nodes, N_1 through N_3, all other good nodes that remained in the *Maintain* state, e.g. N_4, will transition to the *Restore* state. Therefore, at this point, all good nodes will have transitioned to the *Restore* state. Also, none of the good nodes that had transitioned to the *Restore* state can meet the *transitory conditions* and transition back to the *Maintain* state in the mean time.

The longest sequence results when F *Resync* messages from as many faulty nodes are followed by $2F$ *Resync* messages from as many good nodes, N_1 through N_4, followed by F additional *Resync* messages from as many faulty nodes as depicted in Figure 8. Therefore, the maximum duration of this zone will be $F + 2F + F = 4F$ consecutive Δ_{AA} intervals.

Following the time line of activities in the figure, the node has been in the *Restore* state for the maximum possible *transitory delay* of $(2F+1)+ (2F - 1)+ (F+2F+F) = 8F$.

Zone 4: At this point, no other *Resync* messages are expected to arrive from other good nodes and from Lemma *DeltaRRmin* any additional *Resync* messages from the faulty nodes will be considered as invalid. Therefore,

$$State\_Timer(t) = State\_Timer(t_0) + 8F \leq P_T$$

where,

$$State\_Timer(t_0) = 0,$$

or,

$$0 < State\_Timer(t_0) \leq P_T.$$

The subsequent behavior of the node is, therefore, dependent on the initial value of its *State_Timer*, i.e. $State\_Timer(t_0)$. There are two possible initial scenarios for this node's *State_Timer*, either the *State_Timer* is reset to zero or it holds a non-zero value within its range, i.e. its initial value is less than or equal to P_T. If the *State_Timer* is initially reset, unless this node times out, it will have to transition out of the *Restore* state at the next Δ_{AA}. So, assuming P_T is large enough so that the node does not time out, it has to transition to the *Maintain* state at the next Δ_{AA} as shown in Figure 8.

If the *State_Timer* is initially non-zero and the current value of the *State_Timer* is less than P_T, the node does not time out and transitions to the *Maintain* state at the next Δ_{AA}, as shown in Figure 8. Otherwise, the current value of the *State_Timer* is P_T (hence the worst case scenario), and this node times out at the next Δ_{AA}, transmits a *Resync* message and remains in the *Restore* state as shown in Figure 9. The sequence of input message in Figure 9 reveals a potential circular pattern in the behavior of the node. However, unlike the initial scenario for this case where *State_Timer* was not reset to zero, the *State_Timer* is now reset to zero.

Out:	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	A	R	A	A	A	A	A
In:		X	X	X	X	X	X	X	R_{f1}	R_{f2}	R_{g1}	R_{g2}	R_{g3}	R_{g4}	R_{f1}	R_{f2}	X	X	X	X	R_{f1}	

Figure 9. Worst case sequence of activities of a good node at random start up for $F = 2$.

Using a similar argument as for the first case where *State_Timer(t_0)* = 0, this node will transition to the *Maintain* state within the next P_T. Therefore, a node will always transition from the *Restore* state to the *Maintain* state. ♦

From Theorem *RestoreToMaintain*, the maximum possible *transitory delay* for a node in the *Restore* state is $8F$. However, in order to allow the node to transition to the *Maintain* state at the next Δ_{AA}, it has to be prevented from timing out. Therefore, the required minimum period, $P_{T,min}$ is constrained to be $P_{T,min} = 8F+2$.

Although P_T can be any value larger than $P_{T,min}$, it follows from Theorem *RestoreToMaintain* that it cannot exceed that minimum value. Also, in order to expedite the self-stabilization process, the convergence time has to be minimized. Thus, P_T is constrained to $P_{T,min}$. The self-stabilization period for the *Maintain* state, P_M, is typically much larger than P_T. Thus, P_M is constrained to be $P_M \geq P_T$.

Corollary RestoreToMaintainWithin2P$_T$ – *A good node in the Restore state will always transition to the Maintain state within $2P_T$.*
Proof – From the proof of Theorem *RestoreToMaintain*, a node in the *Restore* state will either transition to the *Maintain* state within the first P_T, or it will time out and remain in that state. For the later case, it also follows that the node will transition to the *Maintain* state within the next P_T, therefore, the node will transition to the *Maintain* state within $2P_T$. ♦

All good nodes validate an *Affirm* message from a good node if the minimum arrival time requirement for that message is not violated. By Lemma *DeltaRRmin*, consecutive *Resync* messages from a good node are always more than $\Delta_{RR,min}$ apart. Therefore, after a random start-up, it takes more than $\Delta_{RR,min}$ clock ticks for *Resync* messages from a good node to be accepted by all other good nodes. If a node is in the *Restore* state, from Theorem *ResyncWithinP_T*, it will either time out and transmit a *Resync* message within P_T or from Theorem *RestoreToMaintain* and Corollary *RestoreToMaintainWithin2P_T*, it will transition to the *Maintain* state within $2P_T$. Therefore, for the proof of this protocol, and for the following lemmas and theorems, the state of the system is considered after $2P_T\ \Delta_{AA}$ clock ticks from a random start. At this point, the system is in one of the following three states and all *Resync* messages from the good nodes are at least $\Delta_{RR,min}$ apart and all *Affirm* messages from the good nodes meet their timing requirements at the receiving good nodes.

1 **None** of the good nodes are in the *Maintain* state
2 **All** good nodes are in the *Maintain* state
3 **Some** of the good nodes are in the *Maintain* state

Theorem ConvergeNoneMaintain – *A system of $K \geq 3F + 1$ nodes, where none of the good nodes are in the Maintain state and have not met the transitory conditions, will always converge.*
Proof – Since none of the good nodes are in the *Maintain* state, they are in the *Restore* state either because they have just transitioned there or are forced to remain there due to receiving *Resync* messages either from other good nodes or from the faulty nodes. Since these nodes accept each other's messages, they will receive at least T_A *valid* messages (*Affirm* or

Resync) from each other every Δ_{AA}, and will accept and transmit *Affirm* messages at every Δ_{AA} interval.

The **Earliest** a good node transitions to the *Maintain* state (*EM*) is after it has remained in the *Restore* state for the minimum duration of the *transitory delay* plus at least two *accept events* after the last good node transitioned to the *Restore* state. The earliest the first of the two *accept events* happens is D ticks after the transmission of the last *Resync* message. Therefore the *EM* happens at $D + \Delta_{AA}$. The **Latest** a good node transitions to the *Maintain* state (*LM*) is after remaining in the *Restore* state for the maximum duration of the *transitory delay*, i.e. after receiving *Resync* messages from all faulty nodes. In this case, the *LM* happens at the last good node transmitting the *Resync* message, i.e. at $(2F+F)\Delta_{AA} = 3F\Delta_{AA}$ since its transition to the *Restore* state. So, the time difference between the *LM* and *EM* nodes is given by

$\Delta_{LMEM} = 3F\Delta_{AA} - (\Delta_{AA} + D) = (3F - 1)\ \Delta_{AA} - D.$ ♦

The **self-stabilization precision**, $\Delta_{Precision}$, is the maximum time difference between the *Local_Timer*'s of any two good nodes when the system is stabilized. It is, therefore, the guaranteed precision of the protocol. From Theorem *ConvergeNoneMaintain*, the initial precision after the resynchronization is the maximum value of Δ_{LMEM}.

After the initial synchrony and due to the drift rate of the oscillators, *Local_Timers* of the good nodes will deviate from the initial precision. This phenomenon is depicted in Figure 10.

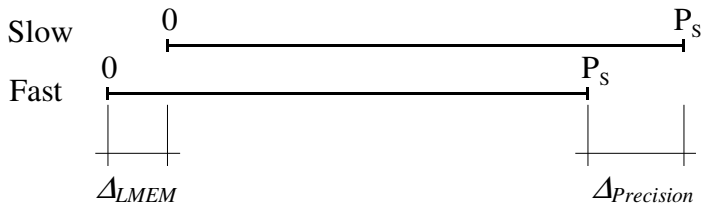

Figure 10. The self-stabilization precision.

Therefore, the guaranteed self-stabilization precision, $\Delta_{Precision}$, after elapsed time of P_M Δ_{AA} clock ticks, is bounded by,

$\Delta_{Precision} = \Delta_{LMEM} + \Delta_{Drift}$

where, the amount of drift from the initial precision is give by

$\Delta_{Drift} = ((1+\rho) - 1/(1+\rho))\ P_M\ \Delta_{AA}.$

The factors $(1+\rho)$ and $1/(1+\rho)$ are, respectively, associated with the slowest and fastest nodes in the system. Therefore,

$\Delta_{Precision} = (3F - 1)\ \Delta_{AA} - D + \Delta_{Drift}.$

Corollary MutuallyStabilized – *All good nodes mutually perceive each other as being in the Maintain state.*
Proof – It follows from Theorem *ConvergeNoneMaintain* that upon convergence and as the good nodes transition to the *Maintain* state, they mutually perceive each other to be in the *Maintain* state. ♦

Theorem ConvergeAllMaintain – *A system of $K \geq 3F + 1$ nodes, where all good nodes are in the Maintain state, will always converge.*
Proof – Since no assumptions are made about the relative timing of the good nodes, $\Delta_{Local\_Timer}(t) > \Delta_{Precision}$ is possible. In this case, all good nodes believe to be synchronized even though the system is not.

It follows from the protocol, Theorem $ResyncWithinP_T$ and Theorem *RestoreWithinPM*, that all good nodes will eventually time out, transition to the *Restore* state, and transmit *Resync* messages. The first (T_R - *1*) good nodes that transition to the *Restore* state may transition back to the *Maintain* state before all other good nodes transition to the *Restore* state. A good node in the *Maintain* state keeps track of other nodes that have transitioned to the *Restore* state. Therefore, after the T_R^{th} good node transitions to the *Restore* state, the remaining good nodes, a total of F nodes, will receive T_R *Resync* messages from as many good nodes, will transition to the *Restore* state and transmit *Resync* messages within the next Δ_{AA}. Any of the first (T_R - *1*) good nodes that had transitioned to the *Restore* state and then back to the *Maintain* state, will now receive (T_R + 1) *Resync* messages within $2\Delta_{AA}$ from as many good nodes, will transition to the *Restore* state, and transmit *Resync* messages within the next Δ_{AA}. At this point all good nodes in the system are in the *Restore* state, are within $2\Delta_{AA}$ of each other, and none of them has met the *transitory conditions*. It follows from Theorem *ConvergeNoneMaintain* that such a system always converges. ♦

Theorem ConvergeSomeMaintain – *A system of $K \geq 3F + 1$ nodes, where some of the good nodes are in the Maintain state will always converge.*
Proof – The good nodes that are in the *Restore* state are there either because they have just transitioned there or are forced to remain there due to receiving *Resync* messages either from other good nodes or from the faulty nodes. Furthermore, their transitions to the *Restore* state are recorded by the good nodes that are in the *Maintain* state. It follows from Lemma $MaintainWithinP_T$ that unless these unstabilized nodes time out within P_T, they'll transition to the *Maintain* state. It follows from the protocol and Theorem *RestoreWithinPM* that all good nodes that are in the *Maintain* state will eventually time out, transition to the *Restore* state, and transmit *Resync* messages.

There are two possible scenarios for the system. Since the transitions to the *Restore* state are recorded by other good nodes in the *Maintain* state, as soon as T_R good nodes have transitioned to the *Restore* state, in a similar argument as in Theorem *ConvergeAllMaintain*, the remaining good nodes will also transition to the *Restore* state. At this point, the system consists of all good nodes in the *Restore* state where none of them has met the *transitory conditions*. It follows from Theorem *ConvergeNoneMaintain* that such a system always converges.

The second possibility is that if the good nodes in the *Restore* state transition back to the *Maintain* state, the system consists of all good nodes in the *Maintain* state and it follows from Theorems *ConvergeNoneMaintain* and *ConvergeAllMaintain* that such a system always converges. ♦

Lemma PrecisionLargerThanTD – *The self-stabilization precision, $\Delta_{Precision}$, is greater than the minimum transitory delay (TD_{min}) for $F > 2$.*

Proof – In other words, $\Delta_{Precision} - TD_{min} \geq 0$.

$\Delta_{Precision} = (3F - 1)\ \Delta_{AA} - D + \Delta_{Drift}$

$\Delta_{Precision} - TD_{min} = (3F - 1)\ \Delta_{AA} - D + \Delta_{Drift} - 2F\Delta_{AA} = (F - 1)\ \Delta_{AA} - D + \Delta_{Drift}$.

For $F > 2$,

$(F - 1)\ \Delta_{AA} - D + \Delta_{Drift} > 0$. ♦

Theorem ClosureAllMaintain – *A system of $K \geq 3F + 1$ nodes, where all good nodes have converged such that all good nodes are mutually stabilized with each other (in other words, all good nodes are in the Maintain state where $\Delta_{Local\_Timer}(t) \leq \Delta_{Precision}$), shall remain within the self-stabilization precision $\Delta_{Precision}$.*

Proof – Since all good nodes are in the *Maintain* state, it follows from Theorem *RestoreWithinPM* that they will transition to the *Restore* state within P_M. As they transmit *Resync* messages, their transitions to the *Restore* state are recorded by other good nodes that are in the *Maintain* state. Since the system is stabilized, the good nodes will transition to the *Restore* state within $\Delta_{Precision}$ of each other. However, since from Lemma *PrecisionLargerThanTD*, and for $F > 2$, the $\Delta_{Precision}$ is greater than the minimum *transitory delay*, some good nodes can potentially transition to the *Restore* state and then to the *Maintain* state before all good nodes transition to the *Restore* state. The proof, therefore, proceeds in the following two parts:

Similar to the proof of Theorem *ConvergeAllMaintain*, the first $(T_R - 1)$ good nodes that transition to the *Restore* state may transition to the *Maintain* state before all other good nodes transition to the *Restore* state. Therefore, after the T_R^{th} good node transitions to the *Restore* state, the remaining good nodes, a total of F nodes, will have received T_R *Resync* messages from as many good nodes, will transition to the *Restore* state and transmit *Resync* messages within the next Δ_{AA}. Any of the first $(T_R - 1)$ good nodes that had transitioned to the *Restore* state and then to the *Maintain* state, will now receive $(T_R + 1)$ *Resync* messages within $2\Delta_{AA}$ from as many good nodes, will transition to the *Restore* state, and transmit *Resync* messages within the next Δ_{AA}. At this point all good nodes in the system are in the *Restore* state, are within $2\Delta_{AA}$ of each other, and none of them has met the *transitory conditions*. It follows from Theorem *ConvergeNoneMaintain* that such a system always converges to within $\Delta_{Precision}$.

On the other hand, if after transitioning to the *Restore* state, none of the good nodes transition to the *Maintain* state until all good nodes transition to the *Restore* state, the system consists of all good nodes in the *Restore* state where none of them has met the *transitory conditions*. It follows from Theorem *ConvergeNoneMaintain* that such a system always converges to within $\Delta_{Precision}$. ♦

Corollary StateTimerLessThanPrecision – *In a stabilized system and during the re-stabilization process, the maximum value of the State_Timer is always less than the self-stabilization precision* $\Delta_{Precision}$.

Proof – From the protocol, the *State_Timer* is reset when the node transitions to either the *Restore* state or the *Maintain* state. It follows from the first part of proof of Theorem *ClosureAllMaintain* that some good nodes that transition to the *Restore* state may transition back to the *Maintain* state before others. The value of the *State_Timer* of such nodes does not exceed $\Delta_{Precision}$. In other words, for these good nodes,

(*State_Timer*) $\Delta_{AA} = \Delta_{Precision} - (2F\Delta_{AA}) + (D + d)$.

Since $\Delta_{AA} \geq D + d$,

(*State_Timer*) $\Delta_{AA} \leq \Delta_{Precision} - 2F\Delta_{AA} + \Delta_{AA}$,

(*State_Timer*) $\Delta_{AA} \leq \Delta_{Precision} - F\Delta_{AA} < \Delta_{Precision}$. ♦

Therefore, the *Local_Timer* can be reset at any point where *State_Timer* is greater than or equal to the precision. In order to expedite the self-stabilization process, *Local_Timer* is reset when *State_Timer* reaches the next integer value greater than $\Delta_{Precision}$, i.e. $\lceil \Delta_{Precision} \rceil$ = *truncate* ($\Delta_{Precision}$ + 0.5). Alternatively, if the amount of drift is such that $\Delta_{Drift} < (\Delta_{AA} + D)$ the *Local_Timer* can be reset when *State_Timer* reaches $3F$.

$\Delta_{Precision} < 3F$

$(3F - 1)\Delta_{AA} - D + \Delta_{Drift} < 3F$

$-\Delta_{AA} - D + \Delta_{Drift} < 0$

$\Delta_{Drift} < \Delta_{AA} + D$

Corollary SteadyStateConvergeTime – *In a stabilized system, the maximum convergence time is less than* $6F\Delta_{AA}$.

Proof – It follows from the first part of Theorem *ClosureAllMaintain* that the time interval from when the first good node transitions to the *Restore* state until all good nodes transition to the *Maintain* state is given by the following equations.

$\Delta_{Precision}$ + Latest to *Maintain* state (*LM*),

From Theorem *ConvergeNoneMaintain*, $LM = (3F - 1)\Delta_{AA} - D$, therefore,

$((3F - 1)\Delta_{AA} - D + \Delta_{Drift}) + (3F - 1)\Delta_{AA} - D$

Since $\Delta_{AA} \geq D + d$,

$(6F - 2)\Delta_{AA} - 2D + \Delta_{Drift} \geq (6F - 4)\Delta_{AA} + \Delta_{Drift}$

but since typically $\Delta_{AA} > \Delta_{Drift}$,

$(6F - 4)\Delta_{AA} + \Delta_{Drift} < 6F < P_T$. ♦

Theorem LocalTimerWithinPrecision – *The difference of Local_Timers of all good nodes in a stabilized system of* $K \geq 3F + 1$ *nodes will always be within the self-stabilization precision, i.e.* $\Delta_{Local\_Timer}(t) \leq \Delta_{Precision}$.

Proof – Since the *Local_Timer* is reset when *State_Timer* = $\lceil \Delta_{Precision} \rceil$, it follows from the Corollaries *StateTimerLessThanPrecision* that, during the re-stabilization process, the *State_Timer* never reaches this value and thus the *Local_Timer* will not be reset during this process. On the other hand, it follows from Theorem *ClosureAllMaintain* that the good nodes will remain within $\Delta_{Precision}$ of each other, thus, $\Delta_{Local\_Timer}(t) \leq \Delta_{Precision}$. ♦

Theorem StabilizeFromAnyState – *A system of $K \geq 3F + 1$ nodes self-stabilizes from any random state after a finite amount of time.*
Proof – The proof of this theorem consists of proving the convergence and closure properties as defined in the Self-Stabilizing Clock Synchronization Problem section.

Assumptions: All good nodes are active and the system operates within the *system assumptions*.

Convergence – *From any state, the system converges to a self-stabilized state after a finite amount of time.*

1. $\forall N_i, N_j \in K_G, \Delta_{Local\_Timer}(C) \leq \Delta_{Precision}$.
2. $\forall N_i, N_j \in K_G$, *at C,* N_i *perceives* N_j *as being in the Maintain state.*

Proof – The proof is done in the following four parts. The approach for the proof is depicted in Figure 11. The system is shown to converge from any state and upon convergence maintain the closure property. The figure is partitioned via a dashed line into two regions. The left region depicts the state of the system in the convergence process. The right region depicts the system operating in the *steady state* and maintaining the self-stabilization precision.

In this figure, the states *All*, *None*, and *Some* represent one of three possible states of the system after $2P_T\ \Delta_{AA}$ clock ticks from a random start. The propositions labeled as theorems indicate that a transition from one state to another eventually takes place.

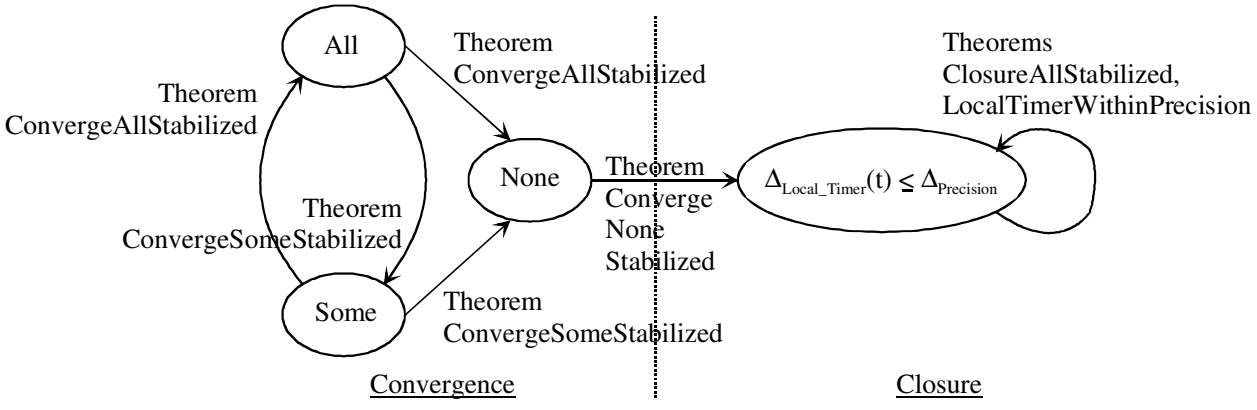

Figure 11. Approach for proof of convergence.

Convergence – *None of the good nodes are in the Maintain state***.**
Proof – It follows from Theorems *ConvergeNoneMaintain* and *ClosureAllMaintain* that such system always self-stabilizes.

Convergence – *All good nodes are in the Maintain state.*
Proof – It follows from Theorems *ConvergeNoneMaintain*, *ConvergeAllMaintain* and *ClosureAllMaintain* that such system always self-stabilizes.

Convergence – *Some of the good nodes are in the Maintain state.*
Proof – It follows from Theorems *ConvergeNoneMaintain*, *ConvergeAllMaintain*, *ConvergeSomeMaintain*, and *ClosureAllMaintain* that such system always self-stabilizes.

Mutually Stabilized – $\forall N_i, N_j \in K_G$, *at C,* N_i *perceives* N_j *as being in the Maintain state.*
Proof – It follows from Corollary *MutuallyStabilized* that all good nodes mutually perceive each other to be in the *Maintain* state.

Closure – *When all good nodes have converged such that* $\Delta_{Local\_Timer}(C) \leq \Delta_{Precision}$*, at time C, the system shall remain within the self-stabilization precision* $\Delta_{Precision}$ *for* $t \geq C$*, for real time t.*
$\forall N_i, N_j \in K_G, t \geq C, \Delta_{Local\_Timer}(t) \leq \Delta_{Precision}$.
Proof – It follows from Theorems *ClosureAllMaintain* and *LocalTimerWithinPrecision* that such system always remains stabilized and $\Delta_{Local\_Timer}(t) \leq \Delta_{Precision}$ for $t \geq C$. ♦

This protocol neither maintains a history of past behavior of the nodes nor does it attempt to classify the nodes into good and faulty ones. Since this protocol self-stabilizes from any state, initialization and/or reintegration are not treated as special cases. Therefore, a reintegrating node will always be admitted to participate in the self-stabilization process as soon as it becomes active. Continual transmission of the *Affirm* messages by the good nodes expedites the reintegration process.

Lemma ResyncWithinP_TPlusP_M – *A good node transmits a Resync message within at most* $(P_T + P_M)\ \Delta_{AA}$ *clock ticks.*
Proof – From Theorem *ResyncWithin*P_T, a node in the *Restore* state will time out within $P_T\ \Delta_{AA}$ clock ticks. So, if a node transitions from the *Restore* state to the *Maintain* state before it times out, it had remained in the *Restore* state for at most (P_T - 1). From Theorem *RestoreWithinPM*, the node will time out within P_M. Therefore, within at most $(P_T + P_M)\ \Delta_{AA}$ clock ticks a node transmits a *Resync* message. ♦

Theorem ConvergeTime – *A system of* $K \geq 3F + 1$ *nodes converges from any random state to a self-stabilized state within* $C = (2P_T + P_M)\ \Delta_{AA}$ *clock ticks.*
Proof – It follows from Lemma *ResyncWithin*P_T*Plus*P_M that a good node transmits a *Resync* message within at most $(P_T + P_M)\ \Delta_{AA}$ clock ticks. It follows from Theorems *ConvergeNoneMaintain*, *ConvergeAllMaintain*, *ConvergeSomeMaintain*, *ClosureAllMaintain*, *LocalTimerWithinPrecision*, and *StabilizeFromAnyState* that the system always converges. It also follows from these Theorem and Theorem *RestoreToMaintain* and Corollary *SteadyStateConvergeTime* that the system converges and all good nodes will transition to the *Maintain* state within the next $P_T\ \Delta_{AA}$ clock ticks. Therefore, the system convergence within $(P_T + P_M + P_T)\ \Delta_{AA}$. Thus, $C = (2P_T + P_M)\ \Delta_{AA}$. ♦

If $P_M = P_T$, then $C = 3P_M$, but since typically $P_M >> P_T$, therefore, C can be approximated to $C \cong P_M$. Therefore, the convergence time of this protocol is a linear function of the P_M.

Theorem StopContinuousTransmit –*A stabilized system of $K \geq 3F + 1$ nodes does not have to transmit Affirm messages continuously.*

Proof – When the system is stabilized, all good nodes are within $\Delta_{Precision}$ of each other. It follows from Corollary *MutuallyStabilized* that all good nodes mutually perceive each other to be in the *Maintain* state. Also, it follows from the protocol that the good nodes reset their *Local_timers* after $\lceil \Delta_{Precision} \rceil$ clock ticks of transitioning to the *Maintain* state. Since the good nodes will not engage in another round of self-stabilization process until they time out, therefore, stopping transmission of *Affirm* messages at this point will not affect the self-stabilization status of the system for the remainder of the current self-stabilization period. ♦

6. Overhead of the Protocol

Since only two self-stabilization messages, namely *Resync* and *Affirm* messages, are required for the proper operation of this protocol, a single bit suffices to represent both messages. Therefore, for a data message w bits wide, the self-stabilization overhead will be $1/w$ per transmission.

The continual aspect of the proposed protocol requires reaffirmation of self-stabilization status of good nodes by periodic transmission of *Affirm* messages at Δ_{AA} intervals. As a result, the maximum number of self-stabilization messages transmitted within any time interval is deterministic and is a function of that time interval. In particular, a good node transmits at most $P_{Effective} / \Delta_{AA}$ self-stabilization messages during a period of $P_{Effective}$,

where,

$P_{Effective}$ = time difference between any two consecutive resets of the *Local_Timer*
$P_{Effective} \leq P_M + 6F$.

Therefore,

Number of messages sent by a node = $P_{Effective} / \Delta_{AA}$

and

Total number of messages sent by K nodes = $K\, P_{Effective} / \Delta_{AA}$.

7. Achieving Tighter Precision

Since the self-stabilization messages are communicated at Δ_{AA} intervals, if Δ_{AA}, and hence $\Delta_{Precision}$, are larger than the desired precision, the system is said to be **Coarsely Synchronized**. Otherwise, the system is said to be **Finely Synchronized**. If the granularity provided by the self-

stabilization precision is coarser than desired, a higher synchronization precision can be achieved in a two step process. First, a system from any initial state has to be *Coarsely Synchronized* and guaranteed that the system remains *Coarsely Synchronized* and operates within a known precision, $\Delta_{Precision}$. The second step, in conjunction with the *Coarse Synchronization* protocol, is to utilize a proven protocol that is based on the initial synchrony assumptions to achieve optimum precision of the synchronized system as depicted in Figure 12.

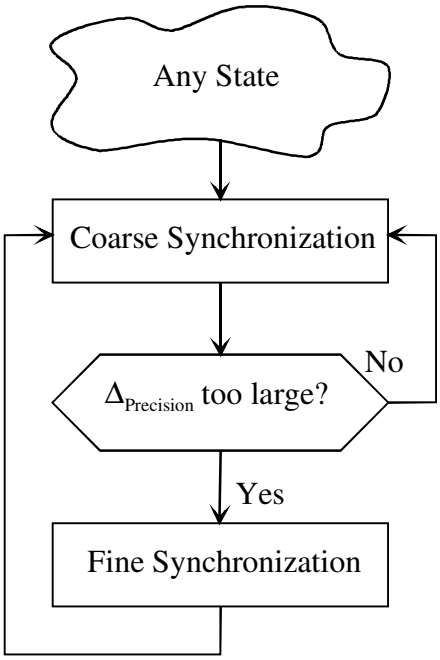

Figure 12. The interplay of *Coarse* and *Fine* level protocols.

As depicted in Figure 12, the *Coarse Synchronization* protocol initiates the start of the *Fine Synchronization* protocol if a tighter precision of the system is desired. The *Coarse* protocol maintains self-stabilization of the system while the *Fine Synchronization* protocol increases the precision of the system.

8. Simulations and Model Checking

Several approaches were taken toward verification of this protocol. The first is a VHSIC Hardware Description Language (VHDL)[2] simulation model that confirms the proper operations of the protocol for specific cases. The VHDL environment is primarily for simulation of specific scenarios where examination of the known cases requires proper set up of the system for each case separately. As the number of cases to be examined increases, this process becomes impractical. Therefore, symbolic model checkers are used which can examine all possible

[2] Very High Speed Integrated Circuit (VHSIC) Hardware Description Language.

scenarios. The Symbolic Model Verifier (SMV)[3] was used for the second modeling of this protocol. It was executed on a PC with 4GB of memory running Linux.

The topology considered is a system of 4 nodes, as shown in Figure 13, such that all nodes can directly communicate with all other nodes, where $K = 4$, $G = 3$ and $F = 1$. With $D = 1$ and $d = 0$, and $\Delta_{AA} = D+d = 1$, the number of states needed to represent all possible combinations of initial states for the entire 4-node system is 4.26×10^{46} states. However, with proper abstractions and employing a number of reduction techniques the state-space is reduced to 5.13×10^{24} states. SMV is able to handle all possible scenarios and the protocol is exhaustively model checked.

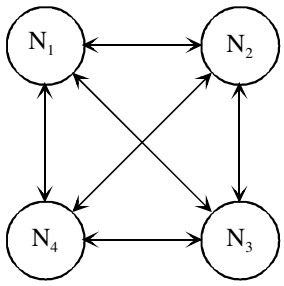

Figure 13. A 4-node fully-connected graph.

This verification effort was conducted to mechanically verify the claims of the protocol. Verification of self-stabilizing a system of 4 nodes in the presence of a Byzantine fault may deceptively seem trivial, but to the best of the author's knowledge, no other self-stabilization protocols has ever been mechanically verified to accomplish this goal. The amount of memory needed for the construction of the Binary Decision Diagram (BDD) readily reached the 4GB available on the PC after construction of the state-space. Therefore, model checking of larger and more complex systems poses a greater challenge. A detailed description of the model-checking efforts for this 4-node system will be the subject of subsequent reports.

9. Applications

The proposed self-stabilizing protocol is expected to have many practical applications as well as many theoretical implications. Embedded systems, distributed process control, synchronization, inherent fault tolerance which also includes Byzantine agreement, computer networks, the Internet, Internet applications, security, safety, automotive, aircraft, wired and wireless telecommunications, graph theoretic problems, leader election, time division multiple access (TDMA), and the SPIDER[4] project [Torres 2005] at NASA-LaRC are a few examples. These are some of the many areas of distributed systems that can use self-stabilization in order to design more robust distributed systems.

[3] http://www-2.cs.cmu.edu/~modelcheck/smv.html
[4] Scalable Processor-Independent Design for Enhanced Reliability (SPIDER).

10. Conclusions

In this paper, a rapid Byzantine self-stabilizing clock synchronization protocol is presented that self-stabilizes from any state. It tolerates bursts of transient failures, and deterministically converges with a linear convergence time with respect to the self-stabilization period. Upon self-stabilization, all good clocks proceed synchronously. This protocol has been the subject of a rigorous verification effort. A 4-node system consisting of 3 good nodes and one Byzantine faulty node has been proven correct using model checking.

The proposed protocol explores the *timing* and *event* driven facets of the self-stabilization problem. The protocol employs *monitors* to closely observe the activities of the nodes in the system. All timing measures of variables are based on the node's local clock and thus no central clock or externally generated pulse is used.

The proposed protocol is scalable with respect to the *fundamental parameters*, K, D, and d. The self-stabilization precision $\Delta_{Precision}$, $\Delta_{Local\_Timer}(t)$, and self-stabilization periods P_T and P_M are functions of K, D and d. The convergence time is a linear function of P_T and P_M and deterministic. As K increases so does the number of *monitors* instantiated in each node. Also, as K increases so does the number of communication channels in a system of fully connected communication network. Therefore, although there is no theoretical upper bound on the maximum values for the *fundamental parameters*, implementation of this protocol may introduce some practical limitations on the maximum value of these parameters and the choice of topology.

A proof of this protocol has been presented in this report. The VHDL simulation and SMV model-checking efforts that verified the correctness of this self-stabilizing protocol are reported. This protocol is expected to be used as the fundamental mechanism in bringing and maintaining a system within bounded synchrony.

Integration of a higher level mechanism with this protocol needs to be further studied. Furthermore, if a higher level secondary protocol is non-self-stabilizing, it is conjectured that it can be made self-stabilizing when used in conjunction with the protocol presented here.

We have started formalizing the integration process of other protocols with this protocol in order to achieve tighter synchronization. We are also planning to implement this protocol in hardware and characterize it in a representative adverse environment.

Symbols

ρ	bounded drift rate with respect to real time
d	network imprecision
D	event-response delay
F	sum of all faulty nodes
G	sum of all good nodes
K	sum of all nodes
K_G	set of all good nodes
Resync	self-stabilization message
Affirm	self-stabilization message
R	abbreviation for *Resync* message
A	abbreviation for *Affirm* message
T_A	threshold for *Accept()* function
T_R	threshold for *Retry()* function
Restore	self-stabilization state
Maintain	self-stabilization state
T	abbreviation for *Restore* state
M	abbreviation for *Maintain* state
$P_{T,min}$	minimum period while in the *Restore* state
P_T	period while in the *Restore* state
P_M	period while in the *Maintain* state
Δ_{AA}	time difference between the last consecutive *Affirm* messages
Δ_{RR}	time difference between the last consecutive *Resync* messages
C	maximum convergence time
$\Delta_{Local\_Timer}(t)$	maximum time difference of *Local_Timers* of any two good nodes at real time t
$\Delta_{Precision}$	maximum self-stabilization precision
Δ_{Drift}	maximum deviation from the initial synchrony
N_i	the i^{th} node
M_i	the i^{th} *monitor* of a node

References

[Daliot 2003A] Daliot, Ariel; Dolev, Danny; Parnas, Hanna: Self-Stabilizing Pulse Synchronization Inspired by Biological Pacemaker Networks, Proceedings of the Sixth Symposium on Self- Stabilizing Systems, DSN SSS '03, San Francisco, June 2003.

[Daliot 2003B] Daliot, Ariel; Dolev, Danny; Parnas, Hanna: Linear Time Byzantine Self-Stabilizing Clock Synchronization, Proceedings of 7th International Conference on Principles of Distributed Systems (OPODIS-2003), La Martinique, France, December 2003.

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

[Dolev 1984] Dolev, Danny; Halpern, J.Y.; Strong, R.: On the Possibility and Impossibility of Achieving Clock Synchronization. In proceedings of the 16th Annual ACM STOC (Washington D.C., Apr.). ACM, New York, 1984, pp. 504-511. (Also appear in J. Comput. Syst. Sci.)

[Dolev 2004] Dolev, Sholmi; Welch, Jennifer L.: Self-Stabilizing Clock Synchronization in the Presence of Byzantine Faults. Journal of the ACM, Vol.51, Np. 5, September 2004, pp. 780-799.

[Driscoll 2003] Driscoll, Kevin; Hall, Brendan; Sivencronam, Hakan; Zumsteg, Phil: Byzantine Fault Tolerance, from Theory to Reality: Computer Safety, Reliability, and Security, Publisher: Springer-Verlag Heidelberg, ISBN: 3-540-20126-2, Volume 2788 / 2003, October 2003, pp. 235 - 248

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

[Lamport 1982] Lamport, Leslie; Shostak, Robert.; Pease, Marshall: The Byzantine General Problem, ACM Transactions on Programming Languages and Systems, 4(3), pp. 382-401, July 1982.

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

[Malekpour 2006] Malekpour, M. R.; Siminiceanu, R.: Comments on the "Byzantine Self-Stabilizing Pulse Synchronization" Protocol: Counterexamples. NASA/TM-2006-213951, February 2006, pp. 7.

[Srikanth 1985] Srikanth, T. K.; Toueg, S.: Optimal Clock Synchronization. Proceedings of the Fourth Annual ACM Symposium on Principles of Distributed Computing, 1985, pp. 71-86.

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

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

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
01- 08 - 2006	Technical Memorandum	

4. TITLE AND SUBTITLE

A Byzantine-Fault Tolerant Self-Stabilizing Protocol for Distributed Clock Synchronization Systems

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Malekpour, Mahyar R.

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

457280.02.07.07

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

NASA Langley Research Center
Hampton, VA 23681-2199

8. PERFORMING ORGANIZATION REPORT NUMBER

L-19262

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

National Aeronautics and Space Administration
Washington, DC 20546-0001

10. SPONSOR/MONITOR'S ACRONYM(S)

NASA

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

NASA/TM-2006-214322

12. DISTRIBUTION/AVAILABILITY STATEMENT

Unclassified - Unlimited
Subject Category 62
Availability: NASA CASI (301) 621-0390

13. SUPPLEMENTARY NOTES

An electronic version can be found at http://ntrs.nasa.gov

14. ABSTRACT

Embedded distributed systems have become an integral part of safety-critical computing applications, necessitating system designs that incorporate fault tolerant clock synchronization in order to achieve ultra-reliable assurance levels. Many efficient clock synchronization protocols do not, however, address Byzantine failures, and most protocols that do tolerate Byzantine failures do not self-stabilize. Of the Byzantine self-stabilizing clock synchronization algorithms that exist in the literature, they are based on either unjustifiably strong assumptions about initial synchrony of the nodes or on the existence of a common pulse at the nodes. The Byzantine self-stabilizing clock synchronization protocol presented here does not rely on any assumptions about the initial state of the clocks. Furthermore, there is neither a central clock nor an externally generated pulse system. The proposed protocol converges deterministically, is scalable, and self-stabilizes in a short amount of time. The convergence time is linear with respect to the self-stabilization period. Proofs of the correctness of the protocol as well as the results of formal verification efforts are reported.

15. SUBJECT TERMS

Byzantine; Fault tolerant; Self-stabilization; Clock synchronization; Distributed; Protocol; Algorithm; Model checking; Formal proof; Verification

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			STI Help Desk (email: help@sti.nasa.gov)
					19b. TELEPHONE NUMBER *(Include area code)*
U	U	U	UU	37	(301) 621-0390