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

Mahyar R. Malekpour

NASA Langley Research Center, Hampton, VA 23681, USA
m.r.malekpour@larc.nasa.gov

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.

Keywords: Byzantine, fault tolerant, self-stabilization, clock synchronization, distributed, protocol, algorithm, model checking, formal proof, verification.

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 [1]. 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. [2] 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. [1, 3] were the first to present the problem and show that Byzantine agreement cannot be achieved for fewer than $3F$ +1 nodes. Dolev et al. [4] 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 [5]. 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 [6], 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 [6, 7] or existence of a common pulse at the nodes, e.g. the first protocol in [8]. There are many clock synchronization algorithms that are based on randomization and, therefore, are non-deterministic, e.g. the second protocol in [8]. Some clock synchronization algorithms have provisions for initialization and/or reintegration [7, 9]. 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 [10, 11]. The algorithm in [11] is called the Byzantine self-stabilization pulse synchronization (BSS-Pulse-Synch) protocol. A flaw in BSS-Pulse-Synch protocol was found and documented in [12]. The biologically inspired Pulse Synchronization protocol in [10] 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

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

protocol has been the subject of rigorous verification efforts that support the claim of correctness.

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 [13], 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 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.

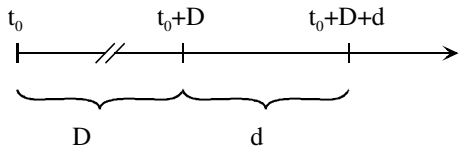

Fig. 1. Event-response delay, *D*, and network imprecision, *d*.

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. 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, where *Resync* messages are represented as *R* and *Affirm* messages are represented as *A*. 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*.

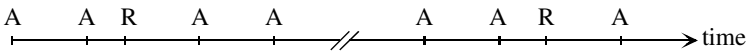

Fig. 2. Timing diagram of transmissions of a good node during the *steady state*.

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 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.

Three **fundamental parameters** characterize the self-stabilization protocol presented here, namely K, D, and d. The bound on 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 3.

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.

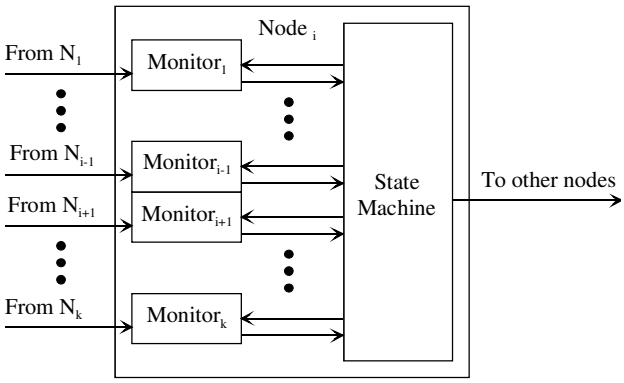

Fig. 3. The i[th] node, N_i, with its *monitors* and state machine.

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 4. The state machine describes the 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 3, 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.

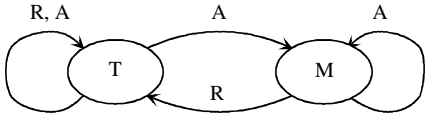

Fig. 4. The node state machine.

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 2*F 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}. Although a *Resync* message is transmitted immediately after the node realizes that it is no longer stabilized, an *Affirm* message is transmitted once every Δ_{AA}.

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 at least $\lceil \Delta_{Precision} \rceil$, 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. 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}. As described earlier, 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

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* (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 *DeltaAA_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. $N_i, N_j \in K_G, \Delta_{Local\_Timer}(C) \leq \Delta_{Precision}$.
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 to a given self-stabilization precision, $\Delta_{Precision}$, 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,

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

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

min() and max() are absolute differences,

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

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

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

and the amount of drift from the initial precision is given 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 5 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 *DeltaAA_Timer*,
Reset *Accept_Event_Counter*,
Stay in *Restore* state,

elsif *TimeOutAcceptEvent()* then
Transmit *Affirm* message,
Reset *DeltaAA_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 *DeltaAA_Timer*,
Reset *Accept_Event_Counter*,
Go to *Restore* state,

elsif *TimeOutAcceptEvent()* then
if *Accept()* then
Consume *valid* messages.,
if (*State_Timer* = $\lceil \Delta_{Precision} \rceil$)
Reset *Local_Timer*.,
Transmit *Affirm* message,
Reset *DeltaAA_Timer*,
Stay in *Maintain* state,

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

Fig. 5. The self-stabilization protocol.

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.

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 $\Delta_{Precision}$. A sketch of the proof of the protocol is presented here. Details of the proof are documented in [14].

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.

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

Theorem ResyncWithinP_T – *A good node remaining in the Restore state transmits a Resync message within at most P_T Δ_{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.*

Theorem RestoreToMaintain – *A good node in the Restore state will always transition to the Maintain state.*

From Theorem *RestoreToMaintain*, the maximum possible *transitory delay* for a node in the *Restore* state is $8F\Delta_{AA}$. 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 $P_{T,min} = (8F+2)\ \Delta_{AA}$. 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 $2P_T$.*

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 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 ≥ 3F + 1 nodes, where none of the good nodes are in the Maintain state and have not met the transitory conditions, will always converge.

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 determined to be $\Delta_{LMEM} = (3F - 1)\ \Delta_{AA} - D$. 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. Therefore, the guaranteed self-stabilization precision, $\Delta_{Precision}$, after elapsed time of $P_M\ \Delta_{AA}$ clock ticks, is bounded by, $\Delta_{Precision} = \Delta_{LMEM} + \Delta_{Drift}$, where the amount of drift from the initial precision is given 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.*

Theorem ConvergeAllMaintain – *A system of $K \geq 3F + 1$ nodes, where all good nodes are in the Maintain state, will always converge.*

Theorem ConvergeSomeMaintain – *A system of $K \geq 3F + 1$ nodes, where some of the good nodes are in the Maintain state will always converge.*

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 $\Delta_{Precision}$.*

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 $\Delta_{Precision}$.*

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

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. $\Delta_{Local\_Timer}(t) \leq \Delta_{Precision}$.*

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.

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

1. $N_i, N_j \in K_G, \Delta_{Local\_Timer}(C) \leq \Delta_{Precision}$.
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:

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.

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.

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.

6 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. 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.

7 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. 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. 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.

A sketch of proof of this protocol has been presented in this paper. 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.

References

1. L Lamport, R Shostak, and M Pease, *The Byzantine General Problem*, ACM Transactions on Programming Languages and Systems, 4(3), pp. 382-401, July 1982.
2. K Driscoll, B Hall, H Sivencronam, and P Zumsteg, *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
3. L Lamport and P M Melliar-Smith, *Synchronizing clocks in the presence of faults*, J. ACM, vol. 32, no. 1, pp. 52-78, 1985.
4. D Dolev, J Y Halpern, and R Strong, *On the Possibility and Impossibility of Achieving Clock Synchronization*, 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.)
5. B W Dijkstra, *Self stabilizing systems in spite of distributed control*, Commun. ACM 17,643-644m 1974.
6. T K Srikanth and S Toueg, *Optimal Clock Synchronization*, proceedings of the Fourth Annual ACM Symposium on Principles of Distributed Computing, 1985, pp. 71-86.
7. J L Welch and N Lynch, *A New Fault-Tolerant Algorithm for Clock Synchronization*, Information and Computation volume 77, no. 1, April 1988, pp.1-36.
8. S Dolev and J L Welch, *Self-Stabilizing Clock Synchronization in the Presence of Byzantine Faults*, Journal of the ACM, Vol.51, Np. 5, September 2004, pp. 780-799.
9. D. Dolev, J. Y. Halpern, B. Simons, and R. Strong, *Dynamic Fault-Tolerant Clock Synchronization*, J. ACM, Vol. 42, No.1, 1995.
10. A Daliot, D Dolev, and H Parnas, *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.
11. A Daliot, D Dolev, and H Parnas, *Linear Time Byzantine Self-Stabilizing Clock Synchronization*, Proceedings of 7th International Conference on Principles of Distributed Systems (OPODIS-2003), La Martinique, France, December 2003.
12. M R Malekpour and R Siminiceanu, *Comments on the "Byzantine Self-Stabilizing Pulse Synchronization" Protocol: Counterexamples*, NASA/TM-2006-213951, Feb 2006, pp. 7.
13. H Kopetz, *Real-Time Systems, Design Principles for Distributed Embedded Applications*, Kluwar Academic Publishers, ISBN 0-7923-9894-7, 1997.
14. M R Malekpour, *A Byzantine-Fault Tolerant Self-Stabilizing Protocol for Distributed Clock Synchronization Systems*, NASA/TM-2006-214322, August 2006, pp. 37.