NASA/TM-2007-215083

Model Checking a Byzantine-Fault-Tolerant Self-Stabilizing Protocol for Distributed Clock Synchronization Systems

Mahyar R. Malekpour
Langley Research Center, Hampton, Virginia

November 2007

NASA/TM-2007-215083

Model Checking 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

November 2007

Acknowledgments

This effort was conducted under the Integrated Vehicle Health Management project of NASA's Aviation Safety program. The author would like to thank the following for their reviews, helpful comments, consultations and support: Ricky Butler, Victor Carreno, Eric Cooper, Jeff Maddalon, Ben DiVito, Paul Miner, Cesar Munoz, Radu Siminiceanu, Kristin Rozier, 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)
7115 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

This report presents the mechanical verification of a simplified model of a rapid Byzantine-fault-tolerant self-stabilizing protocol for distributed clock synchronization systems. This protocol does not rely on any assumptions about the initial state of the system. This protocol tolerates bursts of transient failures, and deterministically converges within a time bound that is a linear function of the self-stabilization period. A simplified model of the protocol is verified using the Symbolic Model Verifier (SMV) [SMV]. The system under study consists of 4 nodes, where at most one of the nodes is assumed to be Byzantine faulty. The model checking effort is focused on verifying correctness of the simplified model of the protocol in the presence of a permanent Byzantine fault as well as confirmation of claims of determinism and linear convergence with respect to the self-stabilization period. Although model checking results of the simplified model of the protocol confirm the theoretical predictions, these results do not necessarily confirm that the protocol solves the general case of this problem. Modeling challenges of the protocol and the system are addressed. A number of abstractions are utilized in order to reduce the state space. Also, additional innovative state space reduction techniques are introduced that can be used in future verification efforts applied to this and other protocols.

Table of Contents

1. Introduction

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. 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]. There are many algorithms that address permanent faults [Srikanth 1987], 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 1987, Welch 1988] or existence of a common pulse at the nodes, e.g. the first protocol in [Dolev 2004]. There are many clock synchronization algorithms that are based on randomization and, therefore, are non-deterministic, e.g. the second protocol in [Dolev 2004].

Solving these special cases is insufficient to claim that an algorithm is self-stabilizing. 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 an efficient convergence time for the proposed self-stabilizing protocol. Typically, verification of a protocol is conducted by the composition of a paper-and-pencil proof. Verification of such proofs is another challenge associated with self-stabilization, especially as the complexity of the protocol increases. Such proofs are error prone. One recent work in this area is the algorithm developed by Daliot et al [Daliot 2003] called the Byzantine self-stabilization pulse synchronization (BSS-Pulse-Synch) protocol. A flaw in BSS-Pulse-Synch protocol was found and documented in a report by Malekpour et al. [Malekpour 2006A]. Such flaws are harder to pinpoint in the proof argument than finding a counterexample via simulation or model checking.

Another technique sometimes used to verify the correctness of a design is based on extensive simulation but it too can miss significant errors when the number of possible states is very large. Simulation of specific scenarios requires proper set up of the system for each case. As the number of cases to be examined increases, this process becomes impractical.

Model checking is a method for mechanically verifying finite-state concurrent systems. Specifications about the system are expressed as temporal logic formulas, and efficient symbolic algorithms are used to traverse the model defined by the system and check if the specification holds or not. The verification procedure is an exhaustive search of the state space of the design. As a result, model checking is a viable means for mechanically verifying the claims of a distributed clock synchronization protocol. Model checking also provides insight into the behavior of the system even if it cannot fully explore the entire state space. Therefore, model checking is a practical alternative for accessing correctness of a protocol and proving correctness of a protocol instance.

This report presents model checking efforts in support of the claims of a rapid Byzantine-fault-tolerant self-stabilizing protocol for distributed clock synchronization systems [Malekpour

2006B, 2006C]. In particular, this effort encompasses the verification of correctness of a simplified model of the protocol by confirming that a candidate system self-stabilizes from any state and tolerates bursts of transient failures in the presence of permanent Byzantine faulty nodes. A **permanent Byzantine** faulty node is a node with arbitrarily malicious behavior. This effort, furthermore, includes the verification of claims of determinism and linear convergence of the simplified model of the protocol with respect to the self-stabilization period and in the presence of *permanent Byzantine* faulty nodes. Although model checking results of the simplified model of the protocol are promising, these results do not necessarily imply that the protocol solves the general case of this problem.

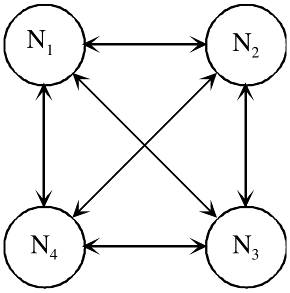


Figure 1. A 4-node system.

As shown in Figure 1, the system under study consists of 4 nodes, where 3 of the nodes are assumed to be good and one of the nodes is Byzantine faulty. Toward this objective, a number of abstractions and reduction techniques are devised to reduce the state space. Also, in order to further reduce the state space to a more manageable size, system parameters are reduced to their minimal values. The amount of memory needed for the construction of the Binary Decision Diagram (BDD) readily reaches the 4GB available after construction of the state space. Therefore, model checking of larger and more complex systems poses a greater challenge.

The following sections describe the model checking efforts in detail. The report begins with a description of the protocol followed by a brief history of the model checking effort. Modeling specifications and abstractions used in describing a basic case of this protocol are described in the following section. The underlying topology and network models are defined, followed by the SMV models of the individual parts. The propositions are then enumerated. A summary of the model checking results is presented. Additional reduction techniques are also introduced, followed by the concluding remarks.

2. The Protocol

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.

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. 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 the state of the system in order to maintain and guarantee its stabilized status, and a periodic 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 parts 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. 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 time difference between 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, $\mathit{\Delta}_{AA}$. To guarantee that a message from a good node is received by all other good nodes before a subsequent message is transmitted, $\mathit{\Delta}_{AA}$ is constrained such that $\mathit{\Delta}_{AA} \geq (D + d)$. Unless stated otherwise, all time dependent parameters of this protocol are measured locally and expressed as functions of $\mathit{\Delta}_{AA}$.

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, G, T_A and T_R, which are integer numbers, other parameters are real numbers. In particular, $\mathit{\Delta}_{AA}$ 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.

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

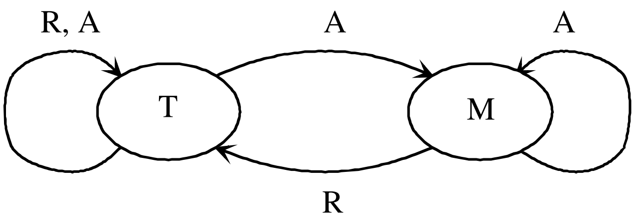


Figure 2. The node state machine.

2.1. Transitory Conditions

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*.

2.2. Message Validity

Starting from the last transmission of the *Resync* message consecutive *Affirm* messages are transmitted at Δ_{AA} intervals, where $\Delta_{AA} \geq (D + d)$. In [Malekpour 2006B, 2006C] $\Delta_{RR,min}$ is defined to be $\Delta_{RR,min} = 2F\Delta_{AA} + 1$ clock ticks. 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. A message shall remain valid for the duration of one Δ_{AA}.
- An *Affirm* message from a given source is ***early*** if it arrives earlier than (Δ_{AA} - d) after previous *valid* message (*Resync* or *Affirm*) from the same source.
- A *Resync* message from a given source is ***early*** if it arrives earlier than $\Delta_{RR,min}$ after previous *valid Resync* message from the same source.
- 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*.

2.3. System Assumptions

1. The cause of transient faults has dissipated.
2. All good nodes actively participate in the self-stabilization process and correctly 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.)

2.4. 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 an 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 determines if at least T_R other nodes have transitioned out of the *Maintain* state, where $T_R = F + 1$. 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 determines 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* 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 a 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 a 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 response to such a timeout, the node broadcasts an *Affirm* message.

2.5. 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, where $t_0 = 0$ when the *system assumptions* are satisfied, and for all $t > t_0$ the system operates within the *system assumptions*. Let

- C be the bound on the maximum convergence time,
- $\Delta_{Local\_Timer}(t)$, for real time t, the maximum difference of values of the local timers of any two good nodes N_i and N_j, where N_i, $N_j \in K_G$, and K_G is the set of all good nodes, and
- $\Delta_{Precision}$, also referred to as self-stabilization precision, the guaranteed upper bound on the maximum separation between the local timers of any two good nodes N_i and N_j in the presence of a maximum of F faulty nodes, where N_i, $N_j \in K_G$.

A good node N_i resets its variable $Local\_Timer_i$ periodically but at different points in time than other good nodes. The difference of local timers of all good nodes at time t, $\Delta_{Local\_Timer}(t)$, is determined by the following equation while recognizing the variations in the values of the $Local\_Timer_i$ across all good nodes.

$$\Delta_{Local\_Timer}(t) = min\ ((Local\_Timer_{max}(t) - Local\_Timer_{min}(t)),$$
$$(Local\_Timer_{max}(t - \lceil \Delta_{Precision} \rceil) - Local\_Timer_{min}(t - \lceil \Delta_{Precision} \rceil))),$$

where,

$$Local\_Timer_{min}(x) = min\ (\{Local\_Timer_i(x) \mid N_i \in K_G\}),$$
$$Local\_Timer_{max}(x) = max\ (\{Local\_Timer_i(x) \mid N_i \in K_G\}), \text{ and}$$

There exist C and $\Delta_{Precision}$:

Convergence: $\Delta_{Local\_Timer}(C) \le \Delta_{Precision}$

Closure: $\forall t,\ t \ge C,\ \Delta_{Local\_Timer}(t) \le \Delta_{Precision}$

The values of C, $\Delta_{Precision}$, and the maximum value for $Local\_Timer_i$, $Local\_Timer\_Max$, are determined to be:

$$C = (2P_T + P_M)\ \Delta_{AA},$$
$$\Delta_{Precision} = (3F - 1)\ \Delta_{AA} - D + \Delta_{Drift},$$
$$Local\_Timer\_Max = P_T + P_M,$$

and the amount of drift from the initial precision is given by

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

Note that since $Local\_Timer\_Max > P_T /2$ and since the *Local_Timer* is reset after reaching *Local_Timer_Max* (worst case wraparound), a trivial solution is not possible.

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

The presented protocol is described in Figure 3 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
```

Figure 3. The self-stabilization protocol.

2.7. 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.

3. Mechanical Verification

Several approaches were explored toward the mechanical verification of the correctness of the initial design of this protocol. This effort started, chronologically, by simulation of the known cases and grew into model checking of all scenarios using various model-checking tools. Initially, verification of a self-stabilizing protocol for a 4-node system seemed deceptively trivial, but in time its complexity became clearer.

The initial model of the 4-node system required more memory for the construction of the state space than the available 2GB of memory. As a result, many abstractions were made and a number of reduction techniques were devised to circumvent the state space explosion problem. Some of the techniques used are explained in the following sections.

3.1. Simulation

The first mechanical verification was accomplished using a VHSIC Hardware Description Language (VHDL)[1] implementation that verified the proper operations of the protocol for specific cases. The VHDL tools run on a PC with 1GB of memory under the Windows 2000 operating system. The VHDL environment is primarily suited for simulation of specific scenarios where examination of the known cases requires proper set up of the system for each case, separately. The simulation effort provided the sanity checks needed to embark into more complex model checking efforts. Nevertheless, within the simulation environment, proper operation of the protocol under fault-free conditions were examined and verified. Proper operation of the protocol in the presence of faults and for the known scenarios were also examined and verified. As the number of cases to be examined increased, this process became impractical. As a result, and in an effort to examine all possible scenarios, this approach was abandoned in favor of symbolic model checkers.

3.2. SMV

The Symbolic Model Verifier (SMV) was used in the second attempt at modeling of this protocol on a PC with 2GB of memory running Linux. SMV allows the designers to formally verify temporal logic properties of finite state systems. Developers use SMV to verify the design for all possible input sequences, instead of a chosen selection of sequences as in simulation.

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

SMV's language description and modeling capability provide relatively easy translation from VHDL. SMV also provides the desired capability to introduce randomness into the initial values of the variables. Despite many abstractions employed, the model's large state space was beyond SMV's capability for the available platform. In fact, the amount of memory needed for the construction of the Binary Decision Diagram (BDD), approximately 10^{44} initial states, readily exceeded the 2GB available on the PC after a few steps. To further reduce the state space, only a subset of critical scenarios was selected. Although this subset was much larger than the number of simulation cases, it still lacked the full coverage needed to rule out unforeseen scenarios.

Clearly, more memory and computing power were needed. A new PC with 4GB of memory running Linux was purchased. Once again, the amount of memory needed by SMV readily exceeded the 4GB available memory.

3.3. SMART

The next modeling effort of this protocol was in Stochastic Model checking Analyzer for Reliability and Timing (SMART) [Ciardo 2003] on a PC with 4GB of memory running Linux. SMART is a software package that integrates various high-level logical and stochastic modeling formalisms (e.g., stochastic Petri nets) in a single modeling study. For model checking, SMART uses Multi Decision Diagram (MDD) to store large sets of states, a Kronecker matrix encoding of transition relation between state, and the saturation algorithm for state space construction [Siminiceanu 2004]. This symbolic approach can manage the memory consumption problem in a more efficient manner. Unlike SMV, SMART lacks an intuitive interface, thus, using it requires greater level of expertise. Unfortunately, due to the complexity of the protocol, the analysis of the model in SMART also exceeded the 4GB available memory and could not fully examine all possible cases in a reasonable amount of time. Nevertheless, using SMART, more scenarios were examined than with SMV and the protocol was demonstrated to be self-stabilizing as expected. Many attempts were made to get around the limitations, but at the end this effort was also abandoned.

3.4. SMV Revisited

The intuitive solution to this problem is to provide more memory. There is a hardware limitation on the amount of memory that can be added to a given system. Furthermore, although additional memory would ease the state space construction, it may not eliminate the problem.

Another solution, if there is one, is to redesign the protocol. What is presented in [Malekpour 2006B and 2006C] and model checked here is the redesigned version of the protocol. The amount of memory needed to fully model check the general case of this protocol far exceeds the available 4GB of memory. Nevertheless, the protocol can now be exhaustively model checked for a 4-node system.

4. Modeling Simplifications and Abstractions

The local measures within each node are used to keep track of timing of the self-stabilization events. Although the derived parameters are defined with respect to the real time, ultimately, in implementations they have to be translated into discrete values. Discretization of the derived parameters is performed using the ceiling operation. In this protocol, all local variables and watchdog timers are discretized and represented by integer values. These local variables are, therefore, measured with respect to the local clock.

The state space for modeling of the general case of this protocol far exceeds the available 4GB memory. Thus, in a bottom-up approach, a basic case is modeled such that the number of parameters needed are minimal and the range of each parameter is at its minimum. A distributed system tolerating as many as F Byzantine faults requires a network size of more than $3F$ nodes [Lamport 1982, Lamport 1985] to maintain synchrony. In other words, to guarantee the closure property a minimum of $3F+1$ nodes are needed. Therefore, the **basic case** is defined as the minimum number of nodes that can self-stabilize in the presence of at least one Byzantine faulty node and with all other parameters at their minimum. Thus, for the *basic case*, the number of nodes in the system $K = 4$, the upper bound on the number of faulty nodes $F = 1$, and the minimum number of good nodes, G, is determined to be $G = K - F = 3$ nodes.

Other aspects of the *basic case* are topological issues. The logical topology is a fully connected graph of a 4-node system, where each node is directly connected to another node via a dedicated bi-directional channel. As shown in Figure 4, each node and the source of a message is distinctly identifiable by other nodes. The physical topology can be either a fully connected graph, similar to the logical topology, or equivalently, a graph where a message from a source is broadcast to all other nodes at the same time. For the *basic case*, broadcast is modeled using a single variable.

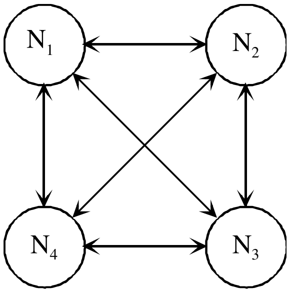


Figure 4. A 4-node system.

Recall that all parameters are defined as integers. The event response delay, D, and the network imprecision, d, are chosen to be at their minimum values of 1 and 0 clock ticks, respectively. As a result, Δ_{AA} is at its minimum of one clock tick. This simplification, consequently, implies that the logical timers of the good nodes are in phase with each other. Note that this simplification does not imply that the nodes are synchronized with each other. To further minimize the state space, the clock drift rate, ρ, is chosen to be zero. This simplification guarantees that the nodes' *State_Timer* will remain in phase with each other. Model checking of the system with $\Delta_{AA} > 1$ where the logical timers of the good nodes are in phase with respect to each other, is equivalent to model checking for $\Delta_{AA} = 1$ and the *basic case*. However, model checking of the system with $\Delta_{AA} > 1$, where the logical timers of the good nodes are out-of-phase with respect to each other, poses a greater challenge.

We recognize that the choice of the value for network imprecision, $d = 0$, is a nonrealistic assumption. Nonetheless, these simplifications are necessary in order to reduce the state space to a manageable size. Furthermore, we believe that the *basic case* specifies the set of necessary conditions that all candidate solutions to this problem should satisfy. As an example, the flaw in [Daliot 2003] was discovered as a direct result of applying that protocol to the *basic case* as documented in [Malekpour 2006A]. We also acknowledge that satisfying the *basic case* does not necessarily imply that the candidate solution solves the general case of this problem.

In order to expedite the self-stabilization process, in general, and in order to minimize the state space for model checking purposes, in particular, the convergence time has to be minimized. It was argued in [Malekpour 2006B] that $P_{T,min} = 10$ and $P_M \geq P_T$. Although the maximum duration of the *Restore* state, P_T, can be any value larger than the required minimum, P_T is chosen to be $P_{T,min}$. In order to minimize the state space, P_M is chosen to be equal to P_T. Therefore, synchronization period, P, for the *basic case* is chosen to be $P = P_M = P_T = 10$. For the *basic case*, the parameters d and ρ are chosen to be zeros. In other words, there are no variations in the communication delay and the nodes do not drift with respect to each other. Model checking of the system with larger values for P_M and P_T is equivalent to model checking for $P = P_M = P_T = 10$.

A system clock, *SCLK*, is introduced to keep track of passage of time from the global perspective. The *SCLK* is managed at the system level and is incremented per SMV cycle. Each node has a logical clock, *Local_Timer*, that locally keeps track of time. This logical clock is used to measure the convergence time, C, as well as the self-stabilization precision, $\Delta_{Precision}$, across good nodes (i.e. external view of the system). Since for the *basic case* the logical timers (*State_Timer* and *Local_Timer*) of the good nodes are in phase with each other and since $\Delta_{AA} = 1$ and $\rho = 0$, a single *SCLK* suffices to drive timers of all nodes. The use of a single *SCLK* also eliminates redundancies at the node level for replicating behavior of local oscillators and, thus, reduces the state space substantially. The *SCLK*, therefore, binds the whole system together, providing a means for advancing the *State_Timer* and *Local_Timer* at the node and an external view of the system at any time. Although the use of a single clock does not imply synchrony at the nodes, it does imply that the nodes are in phase with each other at the *State_Timer* and *Local_Timer* levels. However, due to the inherent randomness of the operation of the model checkers, the order of execution of the nodes is not predetermined. Since there is no control over the order of transmission of messages and the start of execution of the nodes at each model checker cycle, the nodes potentially broadcast and receive messages out of order of issuance.

5. Modeling the System

To accommodate for proper timing of operations of the system, variables are needed to keep track of passage of time in each monitor and node. Introduction of such variables exponentially increases the state space beyond the 4GB available memory. For the general case of modeling this protocol, a *Transmit_Timer* is needed at every node to regulate proper timing of outgoing messages. A *Receive_Timer* is needed at each monitor to keep track of proper timing of incoming messages from its corresponding source [Malekpour 2006B]. As Δ_{AA} increases linearly, the state space associated with *Transmit_Timer* and *Receive_Timer* increases exponentially.

There are two different ways of modeling this protocol, either all operations are done sequentially in one big module, or the operations are partitioned between the node and its monitors. In a sequential model, all activities take place within the same scope and during one clock tick. Such a model is not readily scalable. A modular model is readily scalable, but requires coordinated interactions between the node and its monitors. Either the monitors have to inform the node of the changes in their current status or the node has to poll the status of the monitors to stay current with the changes in the system. In turn, the monitors have to be informed by the node to take certain actions at the appropriate time. Since the node and its monitors operate with respect to a local clock, there will be a delay in a monitor's response to the node's commands. The interactions between the node and its monitors can be coordinated either based on time or by passing a control token in a master-target fashion.

In this SMV model, a modular approach is employed where the interactions between a node and its monitors are coordinated based on time. Also, to minimize the state space both positive and negative edges of the *SCLK* are used. In particular, the nodes operate at the positive edge of the *SCLK* while the monitors operate at the negative edge of the *SCLK*. For $\Delta_{AA} = 1$, operating at the positive edge of the *SCLK*, the nodes are guaranteed not to violate the minimum transmission time requirement for their consecutive output messages. Therefore, for the *basic case* there is no need for the *Transmit_Timer* variable and, consequently, no need for the *Receive_Timer* variable. Thus, further reduction in memory and computation requirements is achieved. Since $\Delta_{AA} = D = 1$ and $\Delta_{Drift} = 0$,

$$\Delta_{Precision} = (3F - 1)\ \Delta_{AA} - D + \Delta_{Drift} = 2\Delta_{AA} - D + 0 = \Delta_{AA}, \text{ and}$$
$$\lceil \Delta_{Precision} \rceil = \Delta_{AA} = 1.$$

Since $\Delta_{AA} = 1$ and $P_T = P_M = P = 10$,

$$C = (2P_T + P_M)\ \Delta_{AA} = 3P = 30\Delta_{AA} = 30.$$

6. Models and Data Structures

In this section, the system components are modeled and subsequently their data structures are defined.

6.1. Modeling Faulty Nodes

The fault tolerant requirement of $K \geq 3F+1$ implies that the system of 4 nodes can tolerate up to one Byzantine faulty node. Therefore, the system is devised to consist of 3 good nodes and one faulty node. In Figure 5 the faulty node, N_4, is shown in gray.

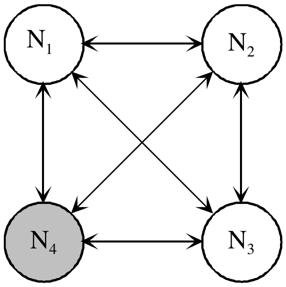
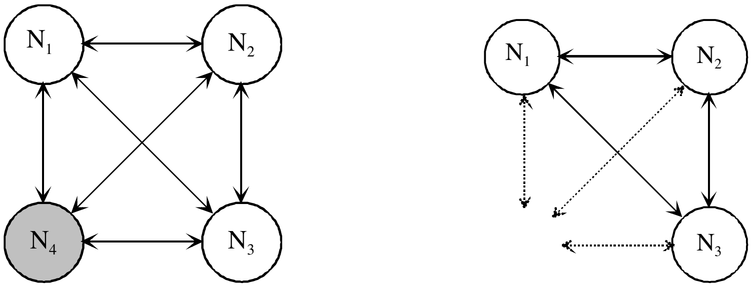


Figure 5. A 4-node system with a faulty node.

To properly portray the behavior of the faulty node, Figure 5 needs to be redrawn. Figure 6 portrays a symmetric faulty node and a crash-silent node that is a special case of a symmetric faulty node where every good node, N_1 through N_3, have the same view of the faulty node, N_4.



Symmetric faulty

Crash-silent

Figure 6. A 4-node system with a symmetric faulty node.

Modeling of an asymmetric (Byzantine) faulty node is more complex than the symmetric faulty node. The malicious nature of the Byzantine faulty node is such that as if each good node is affected independently by the Byzantine faulty node. Such behavior of the Byzantine faulty node is depicted in Figure 7 by replicating the effects of the Byzantine faulty node, N_4, for each good node N_1 through N_3. Furthermore, the Byzantine faulty behavior modeled here is a node with arbitrarily malicious behavior. Defined earlier as *permanent Byzantine* faulty, the Byzantine faulty node is allowed to influence other nodes at every clock tick and at all time.

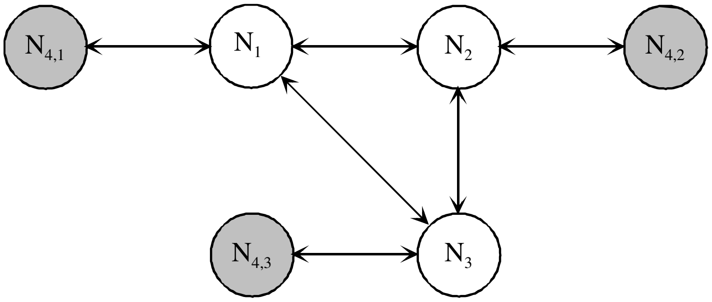


Figure 7. A 4-node system with an asymmetric (Byzantine) faulty node.

Since the behavior of a faulty node is not the same as a good node, modeling of a faulty node requires rethinking. Proper modeling of faulty nodes can potentially result in considerable state space reduction. It particular, a Byzantine faulty node may transmit any one of the three possible messages, namely, *NONE*, *Resync*, or *Affirm* at any time. Additionally, unlike the good nodes, local state of a faulty node does not play a role in the operation of this protocol. Therefore, the faulty node is modeled as a special node only capable of randomly producing any one of the three messages at any clock tick and without any internal state. Consequently, the faulty node's data structure has only one parameter, *Message_Out*. The range of values that this element can hold is enumerated as follows.

Message_Out = {*NONE, Resync, Affirm*}

6.2. Modeling Monitors

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 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 8, where M_j is the monitor for the corresponding node N_j.

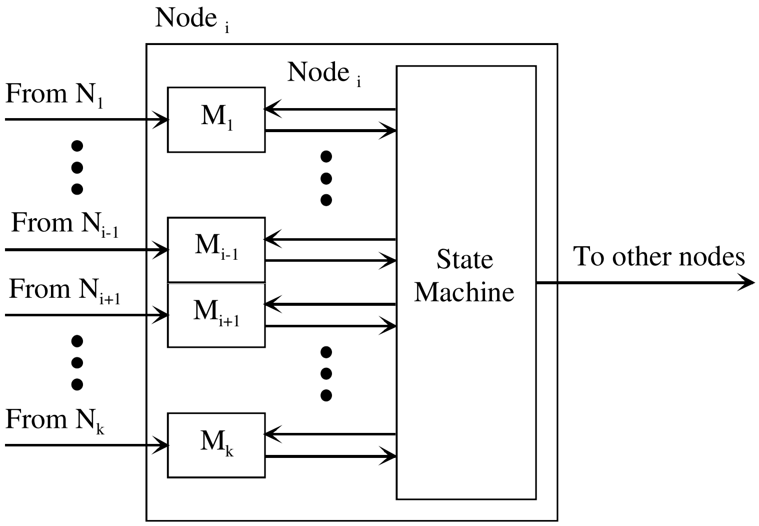


Figure 8. Interaction of the node's state machine and its monitors.

A monitor keeps track of 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 received from that node. A monitor also keeps track of the state of the source node by keeping track of received *Resync* messages, separately. The monitor's data structure consists of *Last_Message*, *Receive_Timer*, *Message_Valid*, *Delta_RR_Timer*, and *Received_Resync*. The *Last_Message* element represents the last *valid* message received from the corresponding source node. The *Receive_Timer* element represents the time interval between arrival of the last two messages from the corresponding source node. As discussed in the previous section, there is no need to model this element for the *basic case*. The *Message_Valid* element indicates whether or not the last message received was *valid.* The *Delta_RR_Timer* element represents the duration of time between any two consecutive *valid Resync* messages from the corresponding source. The *Received_Resync* element indicates whether the last *valid* message received was a *Resync* message. The range of values that these elements can hold is enumerated as follows.

Last_Message	= {*Resync, Affirm*}
Receive_Timer	= $\{0 .. (\Delta_{AA}+1)\}$
Message_Valid	= {0, 1}
Delta_RR_Timer	= $\{0 .. (P_T + P_M)\}$
Received_Resync	= {0, 1}

6.3. Modeling Good Nodes

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 8. The good node's data structure consists of *State*, *Accept_Events*, *State_Timer*, *Local_Timer*, *Transmit_Timer*, and *Message_Out*. The *State* element represents the current state of the node. The *Accept_Events* element is the count of *accept events* since the node entered the *Restore* state. The *State_Timer* element represents the duration of current state of the node. The *Local_Timer* element represents the duration of time since the node has been synchronized with other good nodes. The *Transmit_Timer* element represents the passage of time since the transmission of the last message by the node. As discussed in the previous section, there is no need to model this element for the *basic case*. The *Message_Out* element represents the out going message of the node. The range of values that these elements can hold is enumerated as follows.

State = {*Restore, Maintain*}
Accept_Events = {0 .. (F+1)}
State_Timer = {0 .. P_M}
Local_Timer = {0 .. ($P_T + P_M$)}
Transmit_Timer = {0 .. (Δ_{AA}+1)}
Message_Out = {*NONE, Resync, Affirm*}

6.4. Modeling Communication Channels

The communication channel's data structure consists of *Message_In*, *Comm_Delay*, and *Message_Out*. The *Message_In* element represents the message deposited by the transmitting node. The *Comm_Delay* represents the amount of delay associated with the channel. The *Message_Out* element represents the delayed message being delivered to the destination nodes. The range of values that these elements can hold is enumerated as follows.

Message_In = {*NONE, Resync, Affirm*}
Comm_Delay = {1 .. Δ_{AA}}
Message_Out = {*NONE, Resync, Affirm*}

Since for the *basic case* Δ_{AA} is one clock tick, a deposited message on a communication channel is available to the destination nodes at the next clock tick. Therefore, a channel of depth one suffices. Also since a message is broadcast to other nodes, a single variable suffices to represent the communication channel from a node to all other nodes. Therefore, in order to reduce the state space, the communication channel is modeled implicitly and as part of the node's out going message instead of introducing a new SMV module for the channels.

7. Propositions

Computational tree logic (CTL), a temporal logic, is used to express properties of a system in this context. CTL uses atomic propositions as its building blocks to make statements about the states of a system. CTL then combines these propositions into formulas using logical and temporal operators with quantification over runs. The CTL operators have the following format.

	Q	**T**	
there exists an execution	*E*	*X*	next
for all executions	*A*	*F*	finally (eventually)
		G	globally
		U	until

In this section the claims of convergence and closure properties as well as the claims of maximum convergence time and determinism of the protocol for the *basic case* are examined. Although in the description of the protocol these properties are stated separately, nevertheless, they are examined via one *CTL* proposition. Validation of this general *CTL* proposition requires examination of a number of underlying propositions. In particular, since $\Delta_{Local\_Timer}(t)$ is defined in terms of the *Local_Timer* of the good nodes and the *Local_Timer* is defined in terms of the *State_Timer*, examination of the properties that described proper behavior of the *State_Timer* take precedence. As a result, in this section, the four underlying propositions are examined followed by the general proposition that validates the convergence and closure properties of the protocol as well as the claims of maximum convergence time and determinism.

The following properties are described with respect to only one good node, namely *Good_Node_1*. Since all good nodes are identical, due to the symmetry, the result of the propositions equally similarly applies to other good nodes.

Proposition 1: This property specifies whether or not the *State_Timer* of a good node takes on a given value in its range infinitely often, for instance, its maximum value of *P*. The expected result for this proposition is a true value.

```
AF (Good_Node_1.State_Timer = P)
```

Examining the negation of this property is expected to produce a false value. This proposition verifies that the *State_Timer* of a good node cannot never reach a given value.

```
EG !(Good_Node_1.State_Timer = P)
```

Similar properties apply to the *Local_Timer*, but within its expected range.

Proposition 2: This property specifies whether or not the *State_Timer* of a good node takes on all values in its range infinitely often. In other words, it verifies that the model does not deadlock. Furthermore, the value of the *State_Timer* of a good node at the next clock tick is different from its current value and is its expected next value in the sequence of 0 to *P*. The expected result for this proposition is a true value.

```
AG (((SCLK = 1) & (Good_Node_1.State_Timer = i)) ->
   AX ((SCLK=0) & ((Good_Node_1.State_Timer= i) | (Good_Node_1.State_Timer = i+1)))) &
AG (((SCLK = 1) & (Good_Node_1.State_Timer = P)) ->
   AX ((SCLK = 0) &  (Good_Node_1.State_Timer = 0)))

For all i = 0 .. (P-1)
```

Examining the negation of this property is expected to produce a false value. This proposition verifies that the next value of the *State_Timer* of a good node cannot be the same as its current value. In other words, its value always advances within the expected range.

```
EG (((SCLK = 1) & (Good_Node_1.State_Timer = i)) ->
   EX ((SCLK = 0) & (Good_Node_1.State_Timer = i))) |

For all i = 0 .. (P-1)
```

Similar properties apply to the *Local_Timer*, but within its expected range.

Proposition 3: This property specifies whether or not time advances and the amount of time elapsed, *Elapsed_Time*, has advanced beyond the predicted convergence time, *Convergence_Time*. The expected result for this proposition is a true value.

```
Elapsed_Time := (Global_Clock >= Convergence_Time) ;

AF (Elapsed_Time)
```

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

```
init (Global_Clock) := 0 ;
next (Global_Clock) :=
   case
      (SCLK = 1) & (Global_Clock < Convergence_Time) : Global_Clock + 1 ;
      1 : Global_Clock ;
   esac ;

Elapsed_Time := (Global_Clock >= Convergence_Time) ;
```

Proposition 4: Similar to Proposition 2, this property specifies whether or not the *State_Timer* of a good node takes on all values in its range infinitely often but beyond the convergence time, i.e. after *Elapsed_Time* has become true. The expected result for this proposition is a true value. Examining the negation of this property is expected to produce a false value. Similar properties apply to the *Local_Timer*, but within its expected range.

```
AF (Elapsed_Time) &
AG (((SCLK = 1) & (Elapsed_Time) & (Good_Node_1.State_Timer = i)) ->
   AX ((SCLK=0) & ((Good_Node_1.State_Timer= i) | (Good_Node_1.State_Timer = i+1)))) &
AG (((SCLK = 1) & (Elapsed_Time) & (Good_Node_1.State_Timer = j)) ->
   AX ((SCLK = 0) &  (Good_Node_1.State_Timer = j+1))) &
AG (((SCLK = 1) & (Elapsed_Time) & (Good_Node_1.State_Timer = P)) ->
   AX ((SCLK = 0) &  (Good_Node_1.State_Timer = 0)))

For all i = 0 .. 4
For all j = 5 .. (P-1)
```

Proposition 5: The convergence and closure properties are described in Section 2.5. This proposition encompasses the criteria for the convergence and the closure properties as well as the claims of maximum convergence time and determinism. This proposition specifies whether or not the system will converge to the predicted precision after the elapse of convergence time, *Elapsed_Time*, and whether or not it will remain within that precision thereafter. The expected result for this property is a true value.

```
AF (Elapsed_Time) &                                 -- Determinism Property
AG (Elapsed_Time -> All_Within_Precision) &         -- Convergence Property
AG ((Elapsed_Time & All_Within_Precision) ->
      AX (Elapsed_Time & All_Within_Precision))     -- Closure Property
```

The proper value of the *All_Within_Precision* is determined by measuring the difference of maximum and minimum values of the *Local_Timers* of all good nodes for the current *SCLK* tick and in conjunction with the result from the previous *SCLK* tick. The expected difference of *Local_Timers* is the predicted precision bound.

The negation of the above proposition is listed below and the expected result is a false value. This property specifies that after the elapse of convergence time, *Elapsed_Time*, whether or not the system will not converge or if it converges, whether or not it drifts apart beyond the expected precision bound.

```
AF (Elapsed_Time) &
AG (Elapsed_Time -> All_Within_Precision) &
AG ((Elapsed_Time & All_Within_Precision) -> EX (! All_Within_Precision))
```

8. Results

This SMV model checking effort was performed on a PC with 4GB of memory running Linux. SMV was able to examine all possible scenarios and the *basic case* of the protocol was model checked. The model checking results are listed in the following tables. The negation of a property is denoted by using the unary operator '!'.

The Byzantine faulty behavior modeled here is a node with arbitrarily malicious behavior. The Byzantine faulty node is allowed to influence other nodes at every clock tick and at all time as depicted in Figure 7. Regardless of the nature of the faulty node, no assumptions are made about the initial internal status of the nodes, the monitors, and the system. For instance, a node can wake up in the *Maintain* state and transmit a *Resync*, message. Although such behavior from a good node is not exhibited during normal operation, nevertheless, it is allowed for the random start up. Such a model is for the weakest assumptions about the behavior of the faulty nodes, the internal state of data structures of the nodes, the monitors, and the system as a whole, and thus produces the strongest results.

Table 1. Results in the presence of a Byzantine faulty node.

Proposition	Result	Time (sec)	Mem (GB)
1	T	1311	1.2
1!	F	1318	1.2
2	T	0.2	0.012
2!	F	8866	1.2
3	T	0.04	-
4	T	19	0.056
4!	F	4702	1.2
5	T	2313	2
5!	F	3413	2.1

Table 1 lists the results of model checking of the *basic case* for the stated propositions 1 through 5, where the duration of the *Maintain* and *Restore* states, P_M and P_T, are chosen to be $P_M = P_T = Period = 10$ and the maximum convergence time, *Convergence_Time*, is 30. As shown in Table 1, the maximum memory usage is about 2GB after applying the state space reduction techniques. The amount of memory used and processing time needed depend on the BDD construction and the nature of the query. Although verification of the stated propositions suffices to validate the claims of correctness and determinism of the protocol and in the presence of a Byzantine fault, the propositions are further examined for other, and hence less severe, types of faults. For the following scenarios, the values for the *Period* and *Convergence_Time* are the same as for Table 1.

8.1. Symmetric Fault

In this case, all good nodes receive identical messages from a single faulty node as depicted in Figure 6. The faulty node still behaves randomly, but its effect at the receiving nodes is identical. As shown in Table 2, the maximum available memory is used to model check this case. Due to the BDD construction, the memory usage is far more than the Byzantine faulty case.

Table 2. Results in the presence of a symmetric faulty node.

Proposition	Result	Time (sec)	Mem (GB)
1	T	2573	2.0
2	T	0.2	0.012
3	T	0.04	-
4	T	62	0.160
5	T	3975	3.5*

* Of 4GB available memory, maximum memory utilized by SMV is approximately 3.5GB.

8.2. Crash-Silent Fault, a.k.a. Stuck-at *NONE* Message

This case is a special case of the symmetric faulty node where the faulty node is not transmitting any messages. This case is modeled such that the associated message from the faulty node to all good nodes is a *NONE* message signifying lack of transmission by the faulty node. This case is depicted in Figure 6.

Table 3. Results in the presence of a symmetric faulty node.

Proposition	Result	Time (sec)	Mem (GB)
1	T	28	0.045
2	T	0.15	-
3	T	0.04	-
4	T	6	0.015
5	T	365	0.34

8.3. Stuck-at *Resync* Message

This case is another special case of the symmetric faulty node where all good nodes receive identical messages from a single faulty node. The faulty node transmits the same message to all good nodes all the same time.

Table 4. Results in the presence of a symmetric faulty node.

Proposition	Result	Time (sec)	Mem (GB)
1	T	81	0.25
2	T	0.15	-
3	T	0.04	-
4	T	7	0.025
5	T	605	0.61

8.4. Stuck-at *Affirm* Message

This case is another special case of the symmetric faulty node where all good nodes receive identical messages from a single faulty node. The faulty node transmits the same message to all good nodes all the same time.

Table 5. Results in the presence of a symmetric faulty node.

Proposition	Result	Time (sec)	Mem (GB)
1	T	19	0.033
2	T	0.15	-
3	T	0.04	-
4	T	5	0.017
5	T	276	0.3

9. Additional Reduction Techniques

New state space reduction techniques are presented here that can be used in mechanical verification of other protocols. Although these techniques were not used in the model checking efforts reported here, they are intended to be used in the future efforts. The underlying assumption for these state space reduction techniques is that a message from a good node will eventually (see requirements for message validity for this protocol) be accepted as *valid*. Since this assumption is true for the good nodes and once true they do not violate the message timing requirements, the associated monitors for the corresponding good nodes can be simplified so that they do not have to examine proper timing of message arrival.

In the SMV model reported here, the faulty node is modeled as a special node only capable of randomly producing any one of the three messages at any time. Per protocol requirements, a good node must keep track of the incoming messages from all other nodes. Therefore, *K*-1 monitors at each good node are needed to accommodate this requirement. Hereafter, such straightforward model of a faulty node is referred to as **explicit fault model** and the associated monitors as **explicit fault monitors**.

Recall that the *Accept()* function uses the threshold value $T_A = G - 1 = 2F$ where potentially up to F of these messages are from as many faulty nodes. Looking from a different perspective, at least F of these messages have to be from as many good nodes. Similarly, the *Retry()* function uses the threshold value $T_R = F + 1$ and potentially up to F of these messages are from as many faulty nodes. In other words, at least one of these messages have to be from a good node. Now, let's assume that a good node receives messages only from the other good nodes. In this case, for the *Accept()* function, unless the node receives at least F messages, no matter how many messages (up to F) from the faulty nodes are assumed to be present, the *Accept()* function will not return a true value. Similarly, for the *Retry()* function, unless the node receives at least one message, no matter how many messages (up to F) from the faulty nodes are assumed to be present, the *Retry()* function will not return a true value.

After receiving at least F messages from as many good nodes for the *Accept()* function and at least one message from a good node for the *Retry()* function, the behavior of the faulty nodes can either strengthen a good node's current status or cause the good node to lose synchronization with other nodes. Therefore, only at such moments does the behavior of the faulty nodes impact the operations of the good nodes and, thus, the behavior of the faulty nodes can be inferred as needed at the good nodes. Exploiting this concept reveals that the faulty nodes, the associated *explicit fault monitors* for the corresponding faulty nodes, and the corresponding communication channels are no longer needed. Hereafter, such an indirect model of a faulty node is referred to as an **implicit fault model**. This concept is depicted in Figure 9 where the good nodes are denoted by N_1 .. N_{i-1}, N_{i+1} .. N_{K-F} and their associated explicit monitors are denoted by M_1 .. M_{i-1}, M_{i+1} .. M_{K-F} and the monitors M_{K-F+1} .. M_K represent the *implicit fault models*.

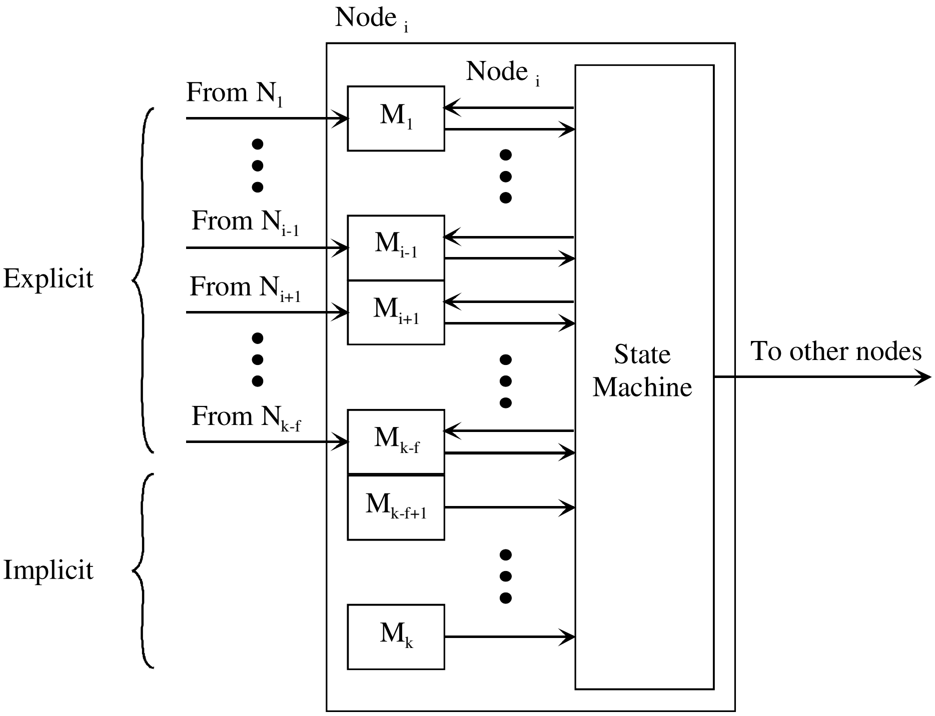


Figure 9. Implicit fault model.

In the *implicit fault model* approach a good node receives messages only from other good nodes and after accumulating enough messages (F for the T_A and one for T_R), the good node's subsequent behavior will be determined by randomly introducing up to F messages for the faulty nodes. Therefore, in this approach, behaviors of faulty nodes are imitated at the good node and when appropriate. Thus, the *implicit fault model* substantially improves the model checking performance. In particular, if a node's behavior will not be influenced by the behavior of the faulty nodes for a duration of time, the model checking time can advance to the end of that time interval. This performance increase is more noticeable in protocols that do not require periodic transmissions of messages. Also, by eliminating the *explicit fault monitors* and the associated channels, the *implicit fault model* results in substantial reduction in the state space.

The *implicit fault model* can be used directly in protocols that do not require keeping track of a history of a node's behavior. Otherwise, an additional measure is required to compensate for the removal of the *explicit fault monitors*. In particular, for the protocol presented in this report, elimination of an *explicit fault monitor* can be compensated by the introduction of a new **implicit fault monitor** at the node. Such a monitor has to guarantee proper timing of any two consecutive actions associated with their corresponding messages.

Alternatively, the faulty node can be modeled as a special node that is still capable of randomly producing any one of the three messages but its outgoing messages are regulated such that the message validity requirements of the protocol are not violated. Such a well-behaved model of a faulty node is referred to as a **semi-explicit fault model**. In this approach, the nodes are modeled explicitly with K-1 explicit monitors but they assume that all incoming messages meet their protocol requirements and, therefore, are *valid*. Therefore, the model of the monitors can be simplified.

The *explicit fault model* is simpler to model, easier to scale to a larger system, but requires more memory than the *implicit fault model*. Modeling of the *implicit fault model* requires more care, but the improved performance and the reduction gained in the state space far outweigh its added complexity. Because of its simplicity and direct approach and avoiding any assumptions regarding message validity, the *explicit fault model* was used in this verification effort. The *semi-explicit fault model* and *implicit fault model* will be used in future work.

10. 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[2] project [Torres 2005A, 2005B] at NASA-LaRC are a few

[2] Scalable Processor-Independent Design for Enhanced Reliability (SPIDER).

examples. These are some of the many areas of distributed systems that can use self-stabilization in order to design more robust distributed systems.

11. Summary and Future Work

In this report a SMV model of a simplified model of a rapid Byzantine-fault-tolerant self-stabilizing protocol for distributed clock synchronization systems is presented. The simplified model of the protocol is model checked using SMV where the entire state space is examined and proven to self-stabilize in the presence of one *permanent Byzantine* faulty node. Furthermore, the simplified model of the protocol is proven to deterministically converge with a linear convergence time with respect to the self-stabilization period as predicted. This protocol does not rely on any assumptions about the initial state of the system and no assumptions are made about the internal status of the nodes, the monitors, and the system as a whole, thus making the weakest assumptions and, therefore, producing the strongest results. The Byzantine faulty behavior modeled here is a node with arbitrarily malicious behavior. The Byzantine faulty node is allowed to influence other nodes at every clock tick and at all time. The only constraint is that the interactions are restricted to defined interfaces.

In this report, modeling challenges are addressed and abstraction techniques are illustrated. A number of innovative state space reduction techniques, in particular the *implicit fault model* of the faulty nodes and their corresponding monitors, are introduced that can be used in a verification process of other protocols. In addition, the *basic case* is introduced that specifies the set of necessary conditions that all candidate solutions to this problem should satisfy. The flaw in [Daliot 2003] was discovered as a direct result of applying that protocol to the *basic case* [Malekpour 2006A]. Although model checking results of the *basic case* of the protocol are promising, these results are not sufficient to confirm that the protocol solves the general case of this problem.

Having mechanically verified a simplified model of the protocol, new hypothesis and conjectures are now practical for examination. The current modeling approach is a very powerful tool for asking “What if?" questions that are difficult to answer either by manual analysis or by testing real hardware.

In our ongoing efforts toward the verification of this protocol for the general case, the SMV model of the simplified version of this protocol has been redesigned and restructured. Also, the protocol has been redesigned and further simplified. As a result, the current model requires less memory, making exploration of more complex and larger configurations easier. Consequently, instances of the protocol representing the out-of-phase scenario where $D > 1$ and $d = 0$, and hence, $\Delta_{AA} > 1$, have been explored. Thus far, the analyses indicate that the protocol solves the out-of-phase scenario. Instances of the protocol representing a more complex system where $D \geq 1$ and $0 \leq d \leq 1$ have also been examined. Thus far, the analyses indicate that the protocol is applicable to realizable systems and practical applications. In addition, some instances of the protocol representing larger systems, where $F > 1$, have also been studied. Thus far, the analyses indicate that the protocol does not solve the general case of this problem where $F > 1$. A detailed explanation of the analyses is beyond the scope of this report. Nevertheless, so far this model checking effort proved that, at a minimum, a deterministic solution for specific cases of this problem exists. We expect that this protocol serves as the starting point toward finding a comprehensive solution for the general case. In-depth analyses of the simplified version of this protocol for more complex and larger systems will be the subject of a subsequent report. This analysis will include pitfalls, relevant counterexamples, an argument toward impossibility results, as well as scenarios where this protocol can be used as a basis for larger systems and, thus, for realizable systems and practical applications.

References:

[Ciardo 2003] Ciardo, G.; Siminiceanu, R.: Structural Symbolic CTL Model Checking, CAV 2003, Boulder, Colorado, LNCS 2725, pp.40-53, July 2003.

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

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

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

[Lamport 1982] Lamport, L.; Shostak, R.; Pease, M.: 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 2006A] Malekpour, M.R.; Siminiceanu, R.: Comments on the “Byzantine Self-Stabilizing Pulse Synchronization” Protocol: Counterexamples. NASA/TM-2006-213951, pp. 12, February 2006.

[Malekpour 2006B] Malekpour, M.R.: A Byzantine-Fault Tolerant Self-Stabilizing Protocol for Distributed Clock Synchronization Systems. NASA/TM-2006-214322, pp. 37, August 2006.

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

[Siminiceanu 2004] Siminiceanu, R.: Structural Model Checking, Ph.D. Thesis, College of William and Mary, 2004.

[SMV] http://www-2.cs.cmu.edu/~modelcheck/smv.html

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

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

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

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

Appendix A. Symbols

The symbols used in the protocol are described in detail in [Malekpour 2006B] and are listed here for reference.

Symbols	Descriptions
ρ	bounded drift rate with respect to real time
d	network imprecision
D	event-response delay
F	maximum number of faulty nodes
G	minimum number of 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 duration while in the *Restore* state
P_T	duration while in the *Restore* state
P_M	duration while in the *Maintain* state
$P_{Effective}$	the effective self-stabilization period
Δ_{AA}	time difference between the last consecutive *Affirm* messages
Δ_{RR}	time difference between the last consecutive *Resync* messages
C	convergence time
$\Delta_{Local\_Timer}(t)$	maximum time difference of *Local_Timers* of all good nodes at real time *t*
$\Delta_{Precision}$	self-stabilization precision
Δ_{Drift}	maximum deviation from the initial synchrony
N_i	the ith node
M_i	the ith monitor of a node

REPORT DOCUMENTATION PAGE	*Form Approved* *OMB No. 0704-0188*

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
01- 11 - 2007	Technical Memorandum	

4. TITLE AND SUBTITLE
Model Checking 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
645846.02.07.07.06

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
NASA Langley Research Center
Hampton, VA 23681-2199

8. PERFORMING ORGANIZATION REPORT NUMBER
L-19408

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-2007-215083

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

This report presents the mechanical verification of a simplified model of a rapid Byzantine-fault-tolerant self-stabilizing protocol for distributed clock synchronization systems. This protocol does not rely on any assumptions about the initial state of the system. This protocol tolerates bursts of transient failures, and deterministically converges within a time bound that is a linear function of the self-stabilization period. A simplified model of the protocol is verified using the Symbolic Model Verifier (SMV) [SMV]. The system under study consists of 4 nodes, where at most one of the nodes is assumed to be Byzantine faulty. The model checking effort is focused on verifying correctness of the simplified model of the protocol in the presence of a permanent Byzantine fault as well as confirmation of claims of determinism and linear convergence with respect to the self-stabilization period. Although model checking results of the simplified model of the protocol confirm the theoretical predictions, these results do not necessarily confirm that the protocol solves the general case of this problem. Modeling challenges of the protocol and the system are addressed. A number of abstractions are utilized in order to reduce the state space. Also, additional innovative state space reduction techniques are introduced that can be used in future verification efforts applied to this and other protocols.

15. SUBJECT TERMS
Byzantine-Fault-Tolerant; Clock Synchronization; Model Checking; Self-Stabilization

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	36	(301) 621-0390