The Truth System: Can a System of Lying Processes Stabilize?

Mohamed G. Gouda and Yan Li

Department of Computer Sciences
The University of Texas at Austin
1 University Station (C0500)
Austin, Texas 78712-0233
{gouda,yanli}@cs.utexas.edu

Abstract. We introduce a new abstract system, called the truth system. In the truth system, a process deduces a true value, with high probability, from an incoming stream of both true and false values, where the probability that a value in the incoming stream is true is at least 0.6. At each instant, the receiving process maintains at most one candidate of the true value, and eventually the process reaches the conclusion that its candidate value equals, with high probability, the true value. In this paper, we present three versions of the truth system, discuss their properties, and show how to choose their parameters so that their probability of error is small, i.e. about 10^{-6}. The third version, called the stable system, is the most valuable. We employ the stable system as a building block in a stabilizing unidirectional token ring of n processes. When n is small, i.e. about 100 or less, the stable system can be considered error-free and we argue that the resulting token ring is stabilizing with high probability. We simulate the token ring, when n is at most 100, and observe that the ring always stabilizes even though each process lies about its state 40% of the time.

Keywords: Distributed Systems, Network Protocols, Self-Stabilization.

1 Introduction

Faults, that are often assumed to plague the communications between different processes in a distributed system, can be distinguished into natural faults and malicious faults [7]. On one hand, *natural communication faults* are assumed to occur independently of the underlying computation of the distributed system, and are usually assumed to be random. On the other hand, *malicious communication faults* are assumed to occur in the worst possible times for the underlying computation of the distributed system, and are usually assumed to be deliberate and based on complete knowledge (by the adversary) of the underlying computation.

Distributed systems that tolerate natural communication faults are elegant, inexpensive, and practical to implement and use. However, such systems cannot

T. Masuzawa and S. Tixeuil (Eds.): SSS 2007, LNCS 4838, pp. 311–324, 2007.

tolerate malicious communication faults if they happen to occur. By contrast, distributed systems that tolerate malicious communication faults are complex, expensive, and sometimes impossible to design.

Well-known examples of natural communication faults and how to tolerate them are as follows.

- *Loss*: Some sent values from one process to another are lost. These faults can be tolerated by making the sending process send the same value repeatedly until the sending process receives an "acknowledgement" from the receiving process [9].
- *Delay*: Some sent values from one process to another are delayed for an unbounded time period before these values are received by their intended receivers. Sometimes, these faults cannot be tolerated as discussed in [6]. This realization has led researchers to adopt weaker model of faults, for example imperfect fault detectors, that can be tolerated [3] and [16].
- *Corruption*: Some sent values are corrupted randomly after they are sent by one process and before they are received by another process. These faults can be detected by adding a checksum to each sent value. In this case, any random corruption of a sent value and its checksum can be detected, with high probability, by the receiving process [14].
- *Topology Change*: The topology of the distributed system changes over time, for example due to the mobility of the processes within the system. Methods for tolerating these "faults" are discussed in [15].
- *Anonymity*: Each sent value does not include the identity of the sending process and is received by an arbitrary process in the distributed system. Methods for tolerating these "faults" are discussed in [1].
- *Modification*: A sent value from one process to another is *modified* before it is received as follows. The value is replaced by any wrong (possibly malicious) value in such a way that the receiving process cannot tell, by examining the received value, that the received value is in fact a false value different from the true value that was sent. For example, if a checksum is attached to the sent value, then a modification fault causes both the value and its checksum to be replaced as follows. The true value is replaced by any false (possibly malicious) value, and the checksum is replaced by the checksum of the false value and so the receiving process cannot tell that the true value and its checksum are replaced by a false value and its checksum. Methods for tolerating modification faults in the context of distributed voting are presented in [13] and [11].

Well-known examples of malicious communication faults and how to tolerate them are discussed in [10] and [12].

In this paper, we present a new method, which we call the truth system, for tolerating modification faults. The truth system is different from the distributed voting systems, discussed in [13] and [11], in several ways. Most notably, the two systems have different objectives. The objective of the truth system is to deduce, with a predefined high probability, the true value from an *unbounded* stream of both true and false values. In particular, the parameters of the truth system can

be set such that the probability of the deduced value being wrong is very small, say 10^{-6}. By contrast, the objective of a distributed voting system is to deduce the value that has the highest probability of being true from a *bounded* stream of both true and false values, but the probability of the deduced value being wrong can be as high as 0.5.

To save space, we omit the proofs of all theorems from this paper. An interested reader can get these proofs from the full version of the paper [8].

2 Three Versions of the Truth System

The goal of this paper is to introduce a new abstract system, called the truth system, discuss its properties, and show that this system can tolerate, with high probability, modification faults. In the truth system, a process deduces a true value from a mixed stream of both true and false values. The probability, that a value in the mixed stream is true (or false), is at least 0.6 (or at most 0.4, respectively). The process correctly deduces the true value with a high probability, that is around $(1\text{-}10^{-6})$.

It is important to explain why we chose the probability, of a value in the mixed stream being true, to be at least 0.6. First, if we chose this probability to be at most 0.5, then no system can deduce the true value with any probability greater than 0. Second, if we chose this probability to be higher than 0.5 but less than 0.6, then as shown at the end of Section 3 the truth system may need up to 1700 values in the input stream to deduce the true value. This is 10 times the number of values needed in the input stream, to deduce the true value, when the probability of a value in the mixed stream being true, is at least 0.6.

The truth system consists of two processes: source and monitor. Process source has an integer state, and process monitor attempts to correctly deduce the integer state of process source.

Periodically, process source sends an integer s to process monitor. With a probability of at least 0.6, the sent s is the true value of the state of process source, and with a probability of less than 0.4, the sent s is an arbitrary integer. Process monitor receives the s integers, one by one, and maintains at most one candidate for the state of process source. Eventually process monitor reaches the conclusion that its maintained candidate for the state of process source equals, with high probability, the true state of process source.

The state of process source can change over time, but we assume that this change occurs at a slow rate. This is because if the state of source is changed at a fast rate, the monitor process may never be able to catch up and deduce the state of source. As shown at the end of Section 3, to ensure that the change rate of the state of process source is slow, we assume that the state of source is not changed until process source has sent out this state 170 or more times.

Our presentation of the truth system in this paper consists of three steps. In each step, we present a version of the truth system and discuss its properties. We then point out some problem with this version and so clear the way for the

next version that is to be presented in the next step, and so on. The version presented in the third (and last) step has no problems, as far as we can tell.

In the first step, we present a version of the truth system where the monitor process terminates as soon as it concludes that its maintained candidate for the state of process source equals, with high probability, the true state of process source. The problem of this truth system is that the monitor process deduces exactly one true state of process source, even if the state of process source is changed many times afterwards. We refer to this truth system as the *one-shot system*.

In the second step, we modify the one-shot system to make the monitor process continue to operate indefinitely, even after it concludes that its maintained candidate for the state of process source equals, with high probability, the true state of process source. We refer to this truth system as the *continuing system*. The problem with the continuing system is that the conclusion reached by process monitor (that its candidate for the state of process source equals, with high probability, the true state of source) is not stable, but it can fluctuate wildly over time, even when the true state of process source remains fixed for a long time period.

In the third step, we modify the continuing system to ensure that, when the true state of process source remains fixed for a long time period, the conclusion reached by process monitor (that its candidate for the state of process source equals, with high probability, the true state of source) remains stable over time. We refer to this truth system as the *stable system*.

3 The One-Shot System

In this section, we present our first version of the truth system, called the one-shot system. In the one-shot system, as soon as process monitor concludes, that its maintained candidate for the state of process source equals, with high probability, the true state of process source, process monitor terminates.

Process source in the one-shot system is specified in Protocol 1. This process has one action that it executes over and over, since the guard of the action is **true**. During each execution of the action, process source sends an integer s to process monitor. With probability p/100, the sent s is the value of input state, and with probability (100-p)/100, the sent s can be any integer, where p is an input of process source.

Input p can be regarded as the *probability of process source telling the truth*, and input *state* can be regarded as the *true state of process source*. From the received s integers, process monitor is expected to deduce the true value of input *state* in process source. It is straightforward to show that if the value of p is in the range 0..50, then process monitor can never deduce the true value of *state*. At the end of Section 3, we argue that if the value of p is in the range 51..59, then process monitor may need to receive up to 1700 s integers (instead of 170) in order to deduce the true value of *state* with high probability. That is why we specified the value of p to be in the range pmin..99, where *pmin* is a constant whose value is in the range 60..99.

Protocol 1. process source

```
const    pmin :  60..99
input    p    :  pmin..99
         state :  integer  {state of source}
variable r    :  0..99     {random number}
         s    :  integer  {sent state}
begin
     true →
          r := random
          if r ≥ p then
             s := any          {assign malicious value}
          else
             s := state
          end
          send s to monitor
end
```

Both p and *state* are inputs to process source. Thus, their values can be changed over time by an outside agent. As shown at the end of Section 3, we assume that once the value of state is changed, the value of state remains fixed until process source executes its action 170 or more times.

Process monitor in the one-shot system is specified in Protocol 2. Process monitor has three variables: c, st and s. Variable c is a counter whose value is in the range $0..cmax$, where $cmax$ is a constant of process monitor. Variable st stores the latest candidate for the state of process source. Variable s stores the latest received integer from process source. The value of counter c indicates whether process monitor can conclude that the current value of st equals, with high probability, the value of state in process source. Process monitor reaches this conclusion when, and only when, the value of counter c is $cmax$.

Process monitor has only one action that is executed each time the process receives an integer s from process source. When an integer s is received, process monitor checks the value of its counter c. If $c = 0$, then variable st is assigned s and counter c is assigned 1. If $c > 0$ and st is different from the received s, then c is decreased by 1. If $c > 0$ and st equals the received s, then c is increased by 1 (provided that c does not exceed its maximum value $cmax$). Then process monitor compares the values of c with $cmax$. If $c = cmax$, then process monitor concludes that the current value of its variable st equals, with high probability, the value of *state* in process source.

To complete the specification of process monitor, we need now to compute the value of constant $cmax$ in monitor. The value of $cmax$ should be chosen such that the probability of error of the one-shot system is kept small, say around 10^{-6}.

The *probability of error*, denoted *p(error)*, of the one-shot system is the probability that starting from its initial global state where $c = 0$, the system reaches a global state where $c = cmax$ and $cs \neq state$.

Protocol 2. process monitor of the one-shot system

```
const    pmin : 60..99   {same as pmin in source}
         cmax : integer
variable c    : 0..cmax  {counter, init. 0}
         s    : integer  {received state}
         cs   : integer  {candidate state}
begin
    rcv s from source →
        if c = 0 then
           c := 1
           cs := s
        else if cs ≠ s then
           c := c - 1
        else
           c := min(c+1, cmax)
        end
        if c = cmax then
           {conclude: cs = state} terminate
        end
end
```

In Theorem 1 below, we give a formula that describes the relationship between *pmin*, *cmax*, and *p(error)* for the one-shot system. Our proof of this theorem is based on two simplifying assumptions. First, we assume that the state of process source does not change over time. This assumption is acceptable given our understanding that the change rate of the state of process source is slow anyway. Second, we assume that whenever process source sends an arbitrary integer s to process monitor, process source always sends the same integer that is different from the state of process source. This assumption represents the worst case scenario that assigns *p(error)* its highest value. We adopt these two assumptions in proving all the theorems in this paper. (Recall that the proofs of all the theorems are in [8].)

Theorem 1 (pmin, cmax, and p(error) for the one-shot system)

$$p(error) \;=\; \frac{(1-pmin)^{cmax}}{(1-pmin)^{cmax} \;+\; (pmin)^{cmax}} \;. \qquad \square$$

For many applications, it is reasonable to expect that *p(error)* should be around 10^{-6}. In this case, we can use the formula in Theorem 1 to produce the relationship between *pmin* and *cmax*, for the one-shot system, shown in Table 1.

An execution *step* of the one-shot system consists of two parts. First, process source executes its (sending) action, then process monitor executes its (receiving) action.

If *pmin* in this system is 0.6 and the value of variable *cs* in process monitor is the correct state of process source, then in each step of the system, counter c in process monitor is incremented by 1 with probability 0.6, and is decremented

Table 1.

pmin	*cmax*
0.6	34
0.7	16
0.8	10
0.9	6

by 1 with probability 0.4. In other words, each step of the system increments counter c by 0.2 on the average. Thus, the system needs to execute cmax/0.2 steps on the average before counter c reaches its maximum value *cmax* and the system terminates. Because *cmax* in this system is 34 from Table 1, the system needs to execute 170 steps before it terminates.

If we choose *pmin* in this system to be 0.51, and assume that *cmax* remains 34 (rather than being increased in value as it should), and follow the same analysis in the previous paragraph, we conclude that the system will execute on average (34/0.02)= 1700 steps before it terminates. In other words, choosing *pmin* to be in the range from 0.51 to 0.59 can lead to (sometimes substantial) increase in the number of steps to be executed. This should explain our choice of *pmin* to be at least 0.6.

The above analysis for computing the average number of steps that need to be executed by the one-shot system before it terminates is based on the assumption that the state of process source does not change during execution. This is an important assumption; for instance, if the state is changed at least once every 5 execution steps, the one-shot system may never terminate. This should explain our above requirement that the state of process source remains fixed for the duration of 170 steps.

4 The Continuing System

The problem of the one-shot system is that process monitor terminates as soon as it concludes that the value of its *cs* variable equals, with high probability, the value of input state in process source. Thus, monitor cannot observe any change in the state of process source. To remedy this problem, we modify process monitor such that the process continues to execute indefinitely. This modification is achieved by replacing statement **terminate** by a statement **skip** in the action of process monitor. We refer to the resulting system as the continuing system.

Because the continuing system is nonterminating, the initial state of the system, where $c = 0$, is irrelevant. Rather, we define the *probability of error p(error)* of the continuing system as the steady state probability that the system is in a global state where $c = cmax$ and $cs \neq state$. The following theorem gives a formula that describes the relationship between *pmin*, *cmax*, and *p(error)* for the continuing system.

Theorem 2 (pmin, cmax, and p(error) for the continuing system)

$$p(error) \;=\; \frac{(1-pmin)^{2\times cmax}}{(1-pmin)^{2\times cmax} \;+\; (pmin)^{2\times cmax}} \;. \qquad \square$$

Assuming that *p(error)* is around 10^{-6}, we can use the formula in Theorem 2 to produce the relationship between *pmin* and *cmax*, for the continuing system, shown in Table 2. Notice that the *cmax* values in the one-shot system, shown in Table 1, are twice the *cmax* values in the continuing system, shown in Table 2.

The continuing system has an interesting problem. Even if the state of process source remains fixed for a long time period T, the value of counter c in process monitor can fluctuate during period T between $c < cmax$ (when process monitor cannot conclude that $cs = state$) and $c = cmax$ (when process monitor can conclude that $cs = state$). This observation suggests the following definition.

The *probability of no conclusion*, denoted *p(no-conclusion)*, of the continuous system is the steady state probability that the system is in a global state where $c < cmax$. The following theorem gives a formula for computing *p(no-conclusion)* as a function of *pmin* and *cmax*.

Theorem 3 (p(no-conclusion) for the continuing system)

$$p(no\text{-}conclusion) \;=\; \frac{\sum_{i=1}^{2\times cmax-1} (\frac{1-pmin}{pmin})^i}{\sum_{j=0}^{2\times cmax} (\frac{1-pmin}{pmin})^j} \;. \qquad \square$$

Using the formula in Theorem 3 and the values of *pmin* and *cmax* in Table 2, we can compute the values of *p(no-conclusion)*, for the continuing system, as shown in Table 2.

From the first row in Table 2, if *pmin* and *cmax* for the continuing system are 0.6 and 17 respectively, then *p(no-conclusion)* for this system is 0.667. This means that even if the state of process source remains fixed for a long time period T, process monitor cannot conclude that $cs = state$ for 66.7% of the time during period T. Clearly, this is not acceptable and a modification of the continuing system is in order. In the next section, we describe how to modify the continuing system to remedy this problem. We refer to the modified system as the stable system.

Table 2.

pmin	*cmax*	*p(no-conclusion)*
0.6	17	0.667
0.7	8	0.429
0.8	5	0.250
0.9	3	0.111

5 The Stable System

The stable system is obtained from the continuing system, discussed in the previous section, by making the following two modifications to process monitor (in the continuing system). First, a new variable named *ss* is added to process monitor. Variable *ss* stores the latest stable estimate (by process monitor) of the state of process source. Second, the last **if**-statement in the action of process monitor is modified to become as shown in Figure 1. (The first **if**-statement in the action of process monitor remains unchanged.)

```
if c = cmax then
   {conclude: cs = state} ss := cs
end
```

Fig. 1.

The *probability of error p(error)* of the stable system is defined as the steady state probability that the system is in a global state where $ss \neq state$. The following theorem describes the relationship between *pmin*, *cmax*, and *p(error)* for the stable system.

Theorem 4 (pmin, cmax, and p(error) for the stable system). *Given that p(error) for the stable system is around* 10^{-6}*, we have the following relationship between pmin and cmax for the stable system.*

A *step* of the stable system consists of two parts. First, process source executes its action once. Second, process monitor executes its action once. The *convergence span* of the stable system is the average number of steps that needs to be executed by the stable system in order to change the global state of the system from one where $c = cmax$ and $ss \neq state$ to one where $c = cmax$ and $ss = state$. The following theorem gives an approximate formula for computing the convergence span of the stable system.

Theorem 5 (convergence span of the stable system).

$$convergence\ span\ \approx\ \frac{2 \times cmax}{2 \times pmin\ -\ 1}\ .$$

□

Table 3.

pmin	*cmax*
0.6	20
0.7	9
0.8	5
0.9	4

□

Table 4.

pmin	*cmax*	*convergence span*
0.6	20	200
0.7	9	45
0.8	5	17
0.9	4	10

Using the formula in this theorem and the values of *pmin* and *cmax* from Table 3, we compute the convergence span of the stable system as shown in Table 4.

6 A Stabilizing Token Ring

In this section, we discuss how to employ the stable system, presented in the previous section, as a building block in constructing a stabilizing unidirectional token ring of up to 100 processes, where each process can lie about its state at most 40% of the time. The use of the stable system as a building block in constructing a stabilizing token ring, (where each process can lie about its state at most 40% of the time) can be roughly viewed as an example of the cross-over composition proposed in [2].

We start our discussion by presenting a unidirectional token ring, in Protocol 3, where processes do not lie about their states. Note that this ring is similar to Dijkstra's token ring in [4] with two exceptions. First, the execution of this ring is synchronous, whereas the execution of Dijkstra's token ring is asynchronous. Second, this ring uses message passing primitives whereas Dijkstra's token ring uses shared memory primitives.

Protocol 3. process p[i : 0..n-1] in the original token ring

```
variable s  :  0..n-1 {sent/received state}
         ss :  0..n-1 {state}
begin
     true  →
          s := ss
          send s to p[i+1 mod n]

||   rcv s from p[i-1 mod n]  →
          if i > 0 then
             ss := s
          else if ss = s then
             ss := ss + 1 mod n
          end
end
```

In this ring, each process p[i] has two variables, *s* and *ss*, where variable *s* stores the latest state that p[i] has sent or received, and variable *ss* stores the state of p[i]. Each p[i] also has two actions: a sending action where p[i] sends its own state to p[i+1 mod n], and a receiving action where p[i] receives the state of p[i-1 mod n] then modifies its own state based on the received state.

A *global state* of this ring is defined by a value for each *ss* variable in the ring. (This means that the *s* variables are not considered part of the global state of the ring.)

A *transition* of this ring is a pair (S, S') of global states of the ring such that if the ring is in a global state S and a "step" is executed, then the ring becomes in a global state S'. Executing a step in the ring consists of two parts. First, each process in the ring executes its sending action, then each process in the ring executes its receiving action. Thus, each process in the ring ends up executing both its (sending and receiving) actions in a step.

A *computation* of this ring is an infinite sequence S.0, S.1, ... of global states of the ring such that each pair (S.i, S.(i+1)) of consecutive states in the sequence is a transition of the ring.

It is straightforward to show that each computation of this ring reaches a legitimate global state where the values of all the ss variables are equal after at most 2n transitions, and so this ring is stabilizing.

Clearly, stabilization of the ring in Protocol 3 depends heavily on the fact that the ring processes do not lie when they send their states to other processes. To allow the ring processes to lie about their states, 40% of the time, and still retain the stabilization of the ring, we employ the stable system, discussed in the previous section, in constructing the new ring. Specifically, each process p[i] in the ring is modified to act as a source when p[i] sends a state *s* to process p[i+1 mod n], and act as a monitor when p[i] receives a state *s* from process p[i-1 mod n]. The new ring is shown in Protocol 4.

A *global state* of the new ring is defined by a value for each *ss* variable in the ring. (This means that none of the other variables, namely *r*, *c*, *s*, and *cs*, is considered part of the global state of the new ring.)

A *transition* of the new ring is a pair (S, S') of the global states of the new ring such that if the ring is in a global state S and a "step" is executed, then the ring becomes in a global state S'. Executing a step in the new ring consists of two parts. First, each process in the ring executes its sending (or source) action. Second, each process in the ring executes its receiving (or monitor) action.

A *computation* of the new ring is an infinite sequence S.0, S.1, ... of global states of the ring such that each pair (S.i, S.(i+1)) of consecutive states in the sequence is a transition of the new ring.

The new ring can be viewed as consisting of n stable systems, and each ss variable in the ring can be viewed as belonging to the monitor of one of those stable systems. For the new ring to stabilize, if it is to stabilize, each *ss* variable needs to be assigned around 2×n new values. From Table 4 and given that pmin is 0.6, a stable system needs to execute on average 200 steps in order to assign a new value to its *ss* variable. Therefore, for the new ring to stabilize, if it is to

Protocol 4. process p[i : 0..n-1] in the new token ring

```
const    pmin :  60..99    {pmin = 60}
         cmax :  integer   {cmax = 20}
variable r    :  0..99     {random number}
         c    :  0..cmax   {counter, init. 0}
         s    :  0..n-1    {sent/received state}
         cs   :  0..n-1    {candidate state}
         ss   :  0..n-1    {stable state}
begin
     true →
          r := random
          if r ≥ pmin then
             s := any
          else
             s := ss
          end
          send s to p[i+1 mod n]

||   rcv s from p[i-1 mod n] →
          if c = 0 then
             c := 1
             cs := s
          else if cs ≠ s then
             c := c - 1
          else
             c := min(c+1, cmax)
          end
          if c = cmax then
             if i > 0 then
                ss := cs
             else if ss = cs then
                ss := ss + 1 mod n
             end
          end
end
```

stabilize, each of the n stable systems in the ring needs to execute around 400×n steps. By choosing n to be relatively small, say 100, each stable system in the ring needs to execute a small number of steps, around 40000 steps in order for the ring to stabilize. Because the probability of error of a stable system is very small, around 10^{-6}, it is reasonable to assume that whenever any *ss* variable is assigned a value, in the first 40000 transitions of a computation, it is assigned a correct value. We refer to this assumption as the *no-use-lying* assumption.

Now consider a computation S.0, ..., S.40000, ... of the new ring. Under the no-use-lying assumption, whenever an *ss* variable is assigned a value in the first 40000 transitions of this computation, it is assigned a correct value. Therefore,

the global state S.40000 in this computation is a legitimate state, with high probability. Therefore, the new ring is stabilizing, with high probability.

The probabilistic stabilization of the new ring depends heavily on the validity of the no-use-lying assumption. To check the validity of this assumption, we have run 100 simulations of the new ring. For each simulation, we chose n (the number of processes in the ring) to be 100, the initial global state of the ring to be random, and the wrong state that each process sends in place of its correct state to be 0. We observed that each simulation has stabilized to a legitimate global state, after no more than 32440 transitions. These simulation results justify our adoption of the no-use-lying assumption.

A probabilistic token ring is proposed in [5]. This ring is significantly different from our new token ring in Protocol 4 in the following sense. In the probabilistic ring in [5], when p[i] receives a (possibly wrong) value s from p[i-1 mod n], p[i] uses the received s to update its own state. By contrast, in our new ring in Protocol 4, when p[i] receives a (possibly wrong) value s from p[i-1 mod n], p[i] first uses its counter c to check whether s is correct with high probability, and only when p[i] is certain that s is correct with high probability, does p[i] use s to update its own state.

7 Concluding Remarks

The truth system is a building block that can be employed in a distributed system to ensure that the system performs its intended function, with high probability, even if up to 40% of the sent values by each process in the system are completely arbitrary. In this paper, we presented three versions of the truth system: the one-shot system, the continuing system, and the stable system. We also compared the properties of these three versions and concluded that the stable system is superior to the other two. Finally we showed how to employ the stable system in a unidirectional token ring so that the ring performs its intended function even if up to 40% of the values sent by each process in the ring are arbitrary.

This paper suggests a number of interesting problems that merit further research. First, are there interesting versions of the truth system other than those discussed in this paper? Second, are there algorithms that take a distributed system that performs a function f under the assumption of perfect communication and produce a distributed system that employs a version of the truth system as a building block and performs function f, with high probability, under the assumption that up to 40% of the values sent by each process in the system are arbitrary. Third, are there effective methods to compute the probability of error and the convergence span for a distributed system where a version of the truth system is employed as a building block?

Acknowledgments. The work of M. G. Gouda is supported in part by a grant 0520250 from the National Science Foundation. The authors are grateful to the referees who made useful suggestions to improve the presentation. We are especially grateful to the referee who remembered Gouda's early proposal to reject

any self-stabilization paper that uses Dijkstra's token ring as an example, and who accepted our paper nonetheless. We appreciate your tolerance!

References

1. Angluin, D., Aspnes, J., Eisenstat, D., Ruppert, E.: On the power of anonymous one-way communication. In: Anderson, J.H., Prencipe, G., Wattenhofer, R. (eds.) OPODIS 2005. LNCS, vol. 3974, pp. 396–411. Springer, Heidelberg (2006)
2. Beauquier, J., Gradinariu, M., Johnen, C.: Cross-over composition - enforcement of fairness under unfair adversary. In: Datta, A.K., Herman, T. (eds.) WSS 2001. LNCS, vol. 2194, pp. 19–34. Springer, Heidelberg (2001)
3. Chandra, T.D., Toueg, S.: Unreliable failure detectors for asynchronous systems. Journal of the ACM 43(2), 225–267 (1996)
4. Dijkstra, E.W.: Self-stabilization systems in spite of distributed control. In: CACM, pp. 643–644 (November 1974)
5. Dolev, S., Herman, T.: Dijkstra's self-stabilizing algorithm in unsupportive environments. In: Datta, A.K., Herman, T. (eds.) WSS 2001. LNCS, vol. 2194, pp. 67–81. Springer, Heidelberg (2001)
6. Fischer, M.J., Lynch, N.A., Paterson, M.S.: Impossibility of distributed consensus with one faulty process. Journal of the ACM 32(2), 374–382 (1984)
7. Fischer, M.J., Merritt, M.: Appraising two decades of distributed computing theory research. Distributed Computing 16(2-3) (2003)
8. Gouda, M.G., Li,Y.: The truth system: Can a system of lying processes stabilize? UTCS Technical Report TR-07-42, Department of Computer Sciences, The University of Texas at Austin, Austin, TX, 2007.
9. Herzberg, A., Kutten, S.: Early detection of message forwarding faults. SIAM Journal of Computing, August-October issue (2000)
10. Hoch, E.N., Dolev, D., Daliot, A.: Self-stabilizing byzantine digital clock synchronization. In: Datta, A.K., Gradinariu, M. (eds.) SSS 2006. LNCS, vol. 4280, pp. 350–362. Springer, Heidelberg (2006)
11. Kumar, A., Malik, K.: Voting mechanisms in distributed systems. IEEE Transactions on Reliability 40(5), 593–600 (1991)
12. Malekpour, M.R.: Byzanting-fault tolerant self-stabilizing protocol for distributed clock synchronization systems. In: Datta, A.K., Gradinariu, M. (eds.) SSS 2006. LNCS, vol. 4280, pp. 411–427. Springer, Heidelberg (2006)
13. Paquette, M., Pelc, A.: Optimal decision strategies in byzantine environments. Journal of Parallel and Distributed Computing 66(3), 419–427 (2006)
14. Peterson, W.W., Brown, D.T.: Cyclic codes for error detection. In: Proceedings of the IRE, vol. 49, pp. 228–235 (1961)
15. Walter, J.E., Welch, J.L., Vaidya, N.H.: A mutual exlcusion algorithm for ad hoc mobile networks. Wireless Networks (2001)
16. Yang, J., Gafni, E., Neiger, G.: Structured derivations of consensus algorithms for failure detectors. In: PODC, Puerto Vallarta, Mexico (1998)