"Slow is Fast" for Wireless Sensor Networks in the Presence of Message Losses*

Mahesh Arumugam[1], Murat Demirbas[2], and Sandeep Kulkarni[3]

[1] maarumug@cisco.com, Cisco Systems Inc., San Jose, CA, 95134
[2] demirbas@cse.buffalo.edu, SUNY Buffalo, Buffalo, NY, 14260
[3] sandeep@cse.msu.edu, Michigan State University, East Lansing, MI, 48824

Abstract. Transformations from shared memory model to wireless sensor networks (WSNs) quickly become inefficient in the presence of prevalent message losses in WSNs, and this prohibits their wider adoption. To address this problem, we propose a variation of the shared memory model, the SF shared memory model, where the actions of each node are partitioned into slow actions and fast actions. The traditional shared memory model consists only of fast actions and a lost message can disable the nodes from execution. Slow actions, on the other hand, enable the nodes to use slightly stale state from other nodes, so a message loss does not prevent the nodes from execution. We quantify over the advantages of using slow actions under environments with varying message loss probabilities, and find that a slow action has asymptotically better chance of getting executed than a fast action when the message loss probability increases. We also present guidelines for helping the protocol designer identify which actions can be marked as slow so as to enable the transformed program to be more loosely-coupled, and tolerate communication problems (latency, loss) better.

1 Introduction

Several computation models have been proposed for distributed computing, including shared memory model, read/write model, and message passing model. These models differ with respect to the level of abstraction they provide. Low level models such as the message passing model permits one to write programs that are closer to the actual system implementation and, hence, the programs can potentially be implemented more efficiently. However, since such programs need to analyze low level communication issues such as channel contention, message delays, etc, they are difficult to design and prove. Using a high level abstraction enables the designers to ignore low-level details of process communication and facilitates the design and verification of the protocols. For example, shared memory model, which allows a node to simultaneously read all its neighbors and update its own state, has been used extensively in the distributed systems literature. The drawback of using a high level abstraction model is that the system

* This work was partially sponsored by NSF CNS 0914913 and Air Force Contract FA9550-10-1-0178

implementation requires more effort. While transformations from shared memory model to read/write model or message passing model have been considered in the literature [5, 9, 10], the efficiency of the transformed program suffers.

Wireless sensor networks (WSNs) warrant a new computation model due to their wireless broadcast communication mode, not captured in any of the above-mentioned models. Write-all-with-collision (WAC) model has been proposed in [6] to capture the important features of wireless broadcast communication for WSNs. In this model, in one step, a node can write its own state and communicate it to its neighbors. Due to the nature of shared medium, if one node is being updated by two (or more) of its neighbors simultaneously then the update fails (message collisions leads to message loss). While WAC model enables us to analyze the energy efficiency and message cost of protocols in WSNs more easily, it is not easy to design and prove protocols in WAC model compared to a higher level model such as the shared memory model.

Transformations from shared memory model to WAC model exist [6,8], however, these transformations have practical problems prohibiting their wider adoption. Although the shared memory model and the WAC model are similar in spirit (in that the former allows a node to read all its neighbors whereas the latter allows the node to write to all its neighbors), direct transformation becomes inefficient when message losses are considered. In [6], a CSMA based transformation, Cached Sensornet Transform (CST), from shared memory model to WAC model has been presented. In CST, a single message loss may violate the correctness of the resultant concrete program in the WAC model. The proof in [6] shows that if the abstract program was designed to be self-stabilizing and no other message loss occurs for a sufficiently long period, the concrete program will stabilize and start making progress. Thus, given the message loss rates at WSNs, this transformation incurs heavy correctness and performance loss at the WAC level. In [8], a transformation from read/write model to WAC model has been presented, and as we show in Section 3, it also applies for transformation from shared memory model to WAC model. This transformation employs a TDMA schedule to reduce message losses in the WAC model, however, due to interference, fading, or sleeping nodes, message losses are still likely in the real deployment. Message losses do not violate safety in this transformation, but they reduce the performance because the loss of a broadcast from a node prevents the evaluation of the actions at other nodes that depended on that information.

Contributions of the paper. To address the performance problems of transformations to WAC model, we propose a variation of the shared memory model, the SF shared memory model. In the SF shared memory model, a node is allowed to read the state of its neighbors and write its own state. However, actions of each node are partitioned into 'slow' actions and 'fast' actions. If a node j determines that a fast action is enabled then j must execute the action immediately before j's neighbors change their state. Otherwise, j must verify whether that action is still enabled the next time it evaluates its guards. On the other hand, if j determines that a slow action is enabled then j can execute the action at any point later as long as j does not execute any other action in between. Note that neighbors of j could change their state in the meanwhile.

We show that the use of SF shared memory model improves the performance of the transformed program under environments with message loss. The traditional shared memory model consists only of fast actions, and a lost message can disable the nodes from execution. Slow actions, on the other hand, enable the nodes to use slightly stale state from other nodes, so a message loss does not prevent the node from execution. We show that a slow action has asymptotically better chance of getting executed than a fast action when the message loss probability increases. By the same token, SF model also allows us to deal with another aspect of WSNs where nodes sleep periodically to save energy.

We present guidelines for the protocol designer to identify slow and fast actions. The designer can mark an action as slow only if 1) guard is stable, 2) guard depends only on local variables (this covers a rich set of programs), or 3) guard is a "locally stable" predicate. These conditions are increasingly more general; stable predicate implies locally stable predicates, but not vice versa. Local stable predicate with respect to j can change after j executes (then other neighbors can execute as the local stable contract is over at that time).

We also introduce slow-motion execution of fast actions. Under continuous message losses, fast actions may never get to execute. This results in bad performance and also violates strong fairness. In order to ensure strong fairness and to achieve graceful degradation of performance under environments with high message loss, we use slow-motion execution. Slow motion execution deliberately slows down the actions that the fast-action depends on, so that the fast action can execute as a pseudo-slow action. The fast action does not need to use the latest state, but it can use a recent consistent state.

Last but not least, our work draws lessons for protocol designers working at the shared memory model level. In order to preserve performance during the transformation, the designers should try to write actions as slow actions. This enables the concrete system to be more loosely-coupled, and tolerate communication problems (latency, loss) better.

Organization of the paper. First, in Section 2, we introduce the structure of programs and the computational models considered in this paper. In Section 3, we present the transformation from shared memory model to WAC model. Then, in Section 4, we introduce the notion of slow and fast actions. Subsequently, in Section 5, we provide an illustrative example. And, in Section 6, we analyze the effect of slow and fast actions. In Section 7, we present an approach for slow-motion execution of fast actions. In Section 8, we discuss some of the questions raised by this work, and finally, in Section 9, we make concluding remarks.

2 Preliminaries

A program is specified in terms of its processes. Each process consists of a set of variables and a set of guarded commands that update a subset of those variables [4]. Each guarded command (respectively, action) is of the form

$$\mathit{guard} \quad \longrightarrow \quad \mathit{statement},$$

where *guard* is a predicate over program variables and *statement* updates the program variables. An action $g \longrightarrow st$ is enabled when g evaluates to true and to execute that action st is executed. A computation consists of a sequence $s_0, s_1, \ldots,$ where s_{l+1} is obtained from s_l $(0 \leq l)$ by executing one or more actions in the program.

Observe that a process can read variables of other processes while evaluating guards of its actions. The copies of these variables can be used in updating the process variables. Hence, we allow declaration of constants in the guard of an action. Intuitively, these constants *save* the value of the variable of the other process so that it can be used in the execution of the statement. As an illustration, consider a program where there are two processes j and k with variables $x.j$ and $x.k$ respectively. Hence, an action where j copies the value of $x.k$ when $x.j$ is less than $x.k$ is specified as follows:

```
Let $y = x.k$
$x.j < y \longrightarrow x.j = y$
```

Note that in a distributed program, for several reasons, it is necessary that a process can only read the variables of a small subset of processes called the neighborhood. More precisely, the neighborhood of process j consists of all the processes whose variables can be read by j.

A computation model limits the variables that an action can read and write. We now describe shared memory model and WAC model.

Shared memory model. In shared memory model, in one atomic step, a process can read its state as well as the state of all its neighbors and write its own state. However, it cannot write the state of other processes.

Write all with collision (WAC) model. In WAC model, each process (or *node*) consists of write actions (to be precise, write-all actions). In one atomic action, a process can update its own state and the state of all its neighbors. However, if two or more processes simultaneously try to update the state of another process, say l, then the state of l remains unchanged. Thus, this model captures the broadcast nature of shared medium.

3 Basic Shared Memory Model to WAC Model

In this section, we present an algorithm (adapted from [8]) for transforming programs written in shared memory model into programs in WAC model. First, note that in WAC model, there is no equivalent of read action. Hence, an action by which node j reads the state of k in shared memory model needs to be modeled in WAC model by requiring k to write its state at j. When k executes this write action, no other neighbor of j can execute simultaneously. Otherwise, due to collision, j remains unchanged. To deal with collisions, TDMA (e.g., [2, 3, 7]) is used to schedule execution of actions. Figure 1 outlines the transformation from shared memory model to WAC model. This algorithm assumes that message losses (other than collisions) do not occur.

```
Input: Program $p$ in shared memory model
begin
Step 1: Slot computation
    compute TDMA schedule using a slot assignment algorithm (e.g., [2, 3, 7])
Step 2: Maintain state of neighbors
    for each variable $v.k$ at $k$, node $j$ ($k \in N.j$, where $N.j$ denotes neighbors
    of $j$) maintains a copy $copy_j.v.k$ that captures the value of $v.k$
Step 3: Transformation
    if slot $s$ is assigned to node $j$
        for each action $g_i \longrightarrow st_i$ in $j$
            evaluate $g_i$
                if $g_i$ mentions variable $v.k, k \neq j$
                    use the $copy_j.v.k$ to evaluate $g_i$
                end-if
        end-for
        if some guard $g_i$ is enabled
            execute $st_i$
        else
            skip;
        end-if
        for all neighbors $k$ in $N.j$
            for each variable $v.j$ in $j$, $copy_k.v.j = v.j$
        end-for
    end-if
end
```

Fig. 1. TDMA based transformation algorithm

In the algorithm, each node maintains a copy of all (public) variables of its neighbors. And, each node evaluates its guards and executes an enabled action in the slots assigned to that node. Suppose slot s is assigned to node j. In slot s, node j first evaluates its guards. If a guard, say g, includes variable $v.k$ of neighbor k, j uses $copy_j.v.k$ to evaluate g. And, if there are some guards that are enabled then j executes one of the enabled actions. Subsequently, j writes its state at all its neighbors. Since j updates its neighbors only in its TDMA slots, collisions do not occur during the write operation.

In this algorithm, under the assumption of no message loss, whenever a node writes its state at its neighbors, it has an immediate effect. Thus, whenever a node is about to execute its action, it has *fresh* information about the state of all its neighbors. Hence, if node j executes an action based on the copy of the state of its neighbors then it is utilizing the most recent state. Moreover, the algorithm in Figure 1 utilizes TDMA and, hence, when node j is executing its action, none of its neighbors are executing. It follows that even if multiple nodes execute their shared memory actions at the same time, their effect can be serialized. Thus, the execution of one or more nodes in a given time instance is equivalent to a serial execution of one or more shared memory actions.

Theorem 1. *Let p be the given program in shared memory model. And, let p' be the corresponding program in WAC model transformed using the algorithm in Figure 1. For every computation of p' in WAC model there is an equivalent computation of p in shared memory model.* □

4 Slow and Fast Actions

According to Section 3, when a process executes its shared memory actions, it utilizes the copy of the neighbors' state. However, when message losses occur, it is possible that the information j has is stale. In this section, we discuss how a node can determine whether it is safe to execute its action.

For the following discussion, let $g \longrightarrow st$ be a shared memory action A at node j. To execute A, j needs to read the state of some of its neighbors to evaluate g and then execute st if g evaluates to true. Let N denote the set of neighbors whose values need to be read to evaluate g. In the context of WSNs, j obtains its neighbors' values by allowing the neighbors to write the state of j. In addition to the algorithm in Figure 1, we require the update to be associated with a timestamp which can be implemented easily and efficiently [4]. Next, we focus on how j can determine whether g evaluates to true.

4.1 When do we evaluate the guard?

The first approach to evaluate g is to ensure that the knowledge j has about the state of nodes in N is up-to-date. Let Cur denote the current time and let t_k denote the time when k notified j of the state of k. The information j has about nodes in N is latest iff for every node k in N, k was not assigned any TDMA timeslot between (t_k, Cur).

Definition 1 (Latest). *We say that j has the latest information with respect to action* A *iff $latest(j, \mathrm{A})$ is true, where*

$$\begin{aligned} latest(j,\mathrm{A}) = (\forall k : k \in \mathrm{N} :\ & k \textit{ updated the state of } j \textit{ at time } t_k \textit{ and} \\ & k \textit{ does not have a TDMA slot in the interval } (t_k, \mathrm{Cur}), \\ & \textit{where } \mathrm{Cur} \textit{ denotes the current time.)} \end{aligned}$$

Clearly, if $latest(j, \mathrm{A})$ is true and g evaluates to true then g is true in the current global state, and, j can execute action A. Of course, if action A depends upon several neighbors then in the presence of message loss or sleeping nodes,

[4] In the context of TDMA and the algorithm in Figure 1, the timestamp information can be relative. Based on the results in Section 6, it would suffice if only 2-4 bits are maintained for this information.

it is difficult for j to ensure that g holds true in the current state. For this reason, we change the algorithm in Figure 1 as follows: Instead of maintaining just one copy for its neighbors, j maintains several copies with different time values (i.e., snapshots). Additionally, whenever a node updates its neighbors, instead of just including the current time, it includes an interval (t_1, t_2) during which this value remains unchanged. Based on these, we define the notion that j has a consistent information about its neighbors although the information may not be most recent.

Definition 2 (Consistent). *We say that j has consistent information as far as action* A *is concerned iff $consistent(j, t, \mathrm{A})$ is true, where*

$$consistent(j,t,\mathrm{A}) = (\forall k : k \in \mathrm{N} : k \textit{ updated the state of } j \textit{ at time } t_k \textit{ and } k \textit{ does not have a TDMA slot in the interval } (t_k, t))$$

Observe that if $consistent(j, t, \mathrm{A})$ is true and g evaluates to true based on most up-to-date information at time t then this implies that it is safe to execute A at time t. After j executes it can discard the old snapshots, and start collecting new snapshots. As we show in Section 6, at most 3 or 4 snapshots is enough for finding a consistent cut, so the memory overhead is low.

Even though satisfying $latest(j, \mathrm{A})$ may be difficult due to message losses and/or sleeping nodes, satisfying $consistent(j, t, \mathrm{A})$ is easier (cf. Section 6). If j misses an update from its neighbor, say k, in one timeslot then j may be able to obtain it in the next timeslot. Moreover, if state of k had not changed in the interim, j will be able to detect if a guard involving variables of k evaluates to true. Furthermore, if action A involves several neighbors of j then it is straightforward to observe that the probability that $consistent(j, t, \mathrm{A})$ is true for some t is significantly higher than the probability that $latest(j, \mathrm{A})$ is true.

The notion of consistency can be effectively used in conjunction with sleeping nodes. If node k is expected to sleep during an interval (t_1, t_2), it can include this information when it updates the state of j. This will guarantee j that state of k will remain unchanged during the interval (t_1, t_2) thereby making it more feasible to ensure that it can find a consistent state with respect to its neighbors.

4.2 When do we execute the action?

The problem with the notion of consistency is that even though the guard of an action evaluated to true at some point in the past, it may no longer be true. Towards this end, we introduce the notion of a slow action and the notion of a fast action. (We call the resulting model as SF shared memory model.)

Definition 3 (slow action). *Let* A *be an action of j of the form $g \longrightarrow st$. We say that* A *is a slow action iff the following constraint is true:*

$$\begin{aligned}&(g \textit{ evaluates true at time } t) \wedge \\ &(j \textit{ does not execute any action between interval } [t, t']) \\ &\qquad\qquad \Rightarrow \quad (g \textit{ evaluates true at time } t')\end{aligned}$$

Rule 1: Rule for execution of a slow action. Let A be a slow action of node j. Node j can execute A provided there exists t such that $consistent(j, t, \mathrm{A})$ is true and j has not executed any action in the interval [t, Cur) where Cur denotes the current time.

Definition 4 (fast action). *Let* A *be an action of* j *of the form* $g \longrightarrow st$. *We say that* A *is a fast action iff it is not a slow action.*

Rule 2: Rule for execution of a fast action. Let A be a fast action of node j. Node j can execute A provided $latest(j, \mathrm{A})$ is true.

If the algorithm in Figure 1 is modified based on the above two rules, i.e., slow actions can be executed when their guard evaluates to true at some time in the past and fast actions are executed only if their guard evaluates to true in the current state, then we can prove the following theorems:

Theorem 2. *Let j and k be two neighboring nodes with actions* A_1 *and* A_2 *respectively. If both* A_1 *and* A_2 *are slow actions then their execution by Rule 1 is serializable.* □

Theorem 3. *Let j and k be two neighboring nodes with actions* A_1 *and* A_2 *respectively. If both* A_1 *and* A_2 *are fast actions then their execution by Rule 2 is serializable.* □

Theorem 4. *Let j and k be two neighboring nodes with actions* A_1 *and* A_2 *respectively. Let* A_1 *be a slow action and let* A_2 *be a fast action. Then, their execution according to Rules 1 and 2 is serializable.* □

5 An Illustrative Example

In this section, we use the tree program from [1] to illustrate the notion of slow and fast actions. In this tree program (cf. Figure 2), each node j maintains three variables: $P.j$, that denotes the parent of node j, $root.j$ that denotes the node that j believes to be the root, and $color.j$ that is either green (i.e., the tree is not broken) or red (i.e., the tree is broken). Each node j also maintains an auxiliary variable $up.j$ that denotes whether j is *up* or whether j has failed.

The protocol consists of five actions. The first three are program actions whereas the last two are environment actions that cause a node to fail and recover respectively. The first action allows a node to detect that the tree that it is part of may be broken. In particular, if j finds that its parent has failed then it sets its color to red. This action also fires if there is a parent and the parent is colored red. Observe that with the execution of this action, if a node is red then it will eventually cause its descendents to be red. The second action allows a red node to separate from the current tree and form a tree by itself provided it has no children. The third action allows one node to join the tree of another node. In particular, if j observes that its neighbor k has a higher root value and both j and k are colored green then j can change its tree by changing $P.j$ to k and $root.j$ to $root.k$. The fourth action is a fault action that causes a node to fail (i.e., $up.j = false$). Due to the execution of this action, the first action will be enabled at the children. And, finally, the last action allows a node to recover. When a node recovers, it sets its color to red.

```
$AC1:$  $color.j = green \wedge$
        $(\neg up.(P.j) \vee color.(P.j) = red)$   $\longrightarrow color.j = red$
$AC2:$  $color.j = red \wedge$
        $(\forall k : k \in Nbr.j : P.k \neq j)$   $\longrightarrow color.j, P.j, root.j = green, j, j$
$AC3:$  Let $x = root.k$
        $root.j < x \wedge color.j = green \wedge$
        $color.k = green$                          $\longrightarrow P.j, root.j = k, x$
$AC4:$  $up.j$                                     $\longrightarrow up.j = false$
$AC5:$  $\neg up.j$                                $\longrightarrow up.j, color.j = true, red$
```

Fig. 2. Coloring tree program

We can make the following observations about this program.

Theorem 5. *$AC1$ and $AC2$ are slow actions.*

Proof. If a node detects that its parent has failed or its parent is red then this condition is stable until that node (child) separates from the tree by executing action $AC2$. Hence, $AC1$ is a slow action. Likewise, if a node is red and has no children then it cannot acquire new children based on the guard of $AC3$. □

Theorem 6. *$AC3$ is a fast action.*

Proof. After j evaluates its own guard for $AC3$, it is possible that the guard becomes false subsequently if k changes its color by executing $AC1$ or if k changes its root by executing $AC3$. Hence, $AC3$ is a fast action. □

6 Effect of Slow versus Fast Actions During Execution

In this section, we evaluate the execution conditions of slow and fast actions in the presence of message loss. To execute a fast action, each node j needs to evaluate whether it has obtained the latest information about the state of its neighbors. If $latest(j, \mathrm{A})$ evaluates to true for some action then j can evaluate the guard of that action and execute the corresponding statement. If $latest(j, \mathrm{A})$ is false for all actions then j must execute a 'skip' operation and see if it can obtain the latest information in the next TDMA round. For the execution of a slow action, j proceeds in a similar fashion. However, if j obtains consistent information about its neighbors that is not necessarily from the latest TDMA round, j can execute its action.

Next, we evaluate the probability that j can obtain the necessary consistent and latest state information. Let p be the probability of a message loss and let

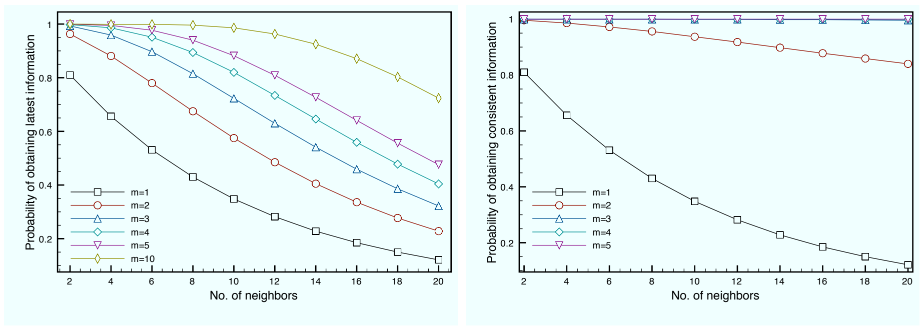
N denote the number of neighbors whose status needs to be known to evaluate the guard of the action. If j cannot successfully obtain consistent and/or latest state information in one TDMA round then it tries to do that in the next round. Hence, we let m denote the number of TDMA rounds that j tries to obtain the consistent and/or latest information. Assuming that states of the neighbors do not change during these m rounds, we calculate the probability that j can obtain the required consistent and/or latest state information [5].

Probability of obtaining latest information. To obtain the latest information in one TDMA round, j needs to successfully receive message from each of its neighbors. Probability of successfully receiving message from one neighbor is $(1-p)$. Hence, the probability of obtaining latest information in one round is $(1-p)^N$. And, the probability of not obtaining the latest information in one round is $1-(1-p)^N$. Therefore, probability of not obtaining the latest information in any one of m rounds is $(1-(1-p)^N)^m$. Thus, the probability that j can obtain the latest information in at least one m rounds is $(1-(1-(1-p)^N)^m)$.

Probability of obtaining consistent information. To obtain consistent information in the earliest of m rounds, j needs to obtain information from each of its neighbors in some round. (Observe that since the nodes include the intervals where their value is unchanged, receiving a message from each node at some round is enough for identifying the first round as the consistent cut.) The probability that j does not receive message from one of its neighbors in either of m rounds is p^m. Hence, probability of successfully receiving message from one neighbor is $(1-p^m)$. Therefore, probability of successfully receiving message from every neighbor is $((1-p^m))^N$. Furthermore, there is an additional conditional probability where j fails to get consistent information in the first (earliest) round but obtains it in the next round. We account for this in our calculations and graphs, but omit the full formula here for the sake of brevity.

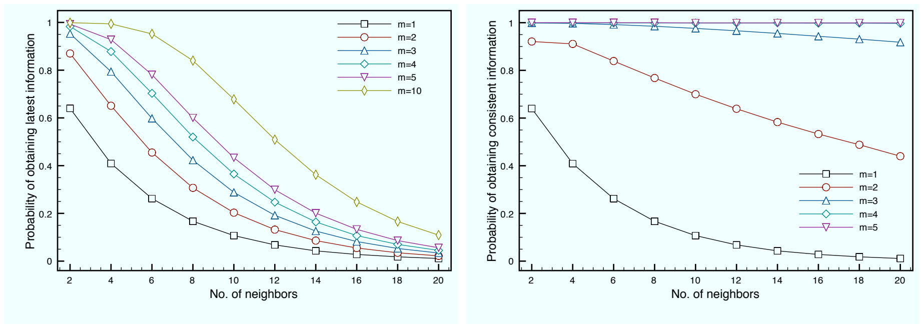
Figures 3-5 show the probabilities for $p = 10\%$, $p = 20\%$, and $p = 30\%$ respectively. First, we note that the probability of obtaining latest information decreases as N increases for different values of m. A given node has to receive updates from all its neighbors in order to obtain the latest information. Hence, as N increases, latest probability decreases. Moreover, in a high message loss environment (e.g., $p = 20\%$ and $p = 30\%$), latest probabilities decrease significantly as N increases. For small neighborhoods, the probability of getting latest information improves as m increases. This suggests that if the neighbors remain silent for some rounds then the probability of obtaining latest information improves. On the other hand, although the probability of obtaining consistent information decreases as N increases, for $m \geq 3$, it remains close to 1. By choosing $m = 3$, the probability of finding a consistent cut is virtually certain at $p = 10\%$.

[5] We can relax this assumption by requiring the nodes to include their old values in previous rounds with their broadcast. These values are then used for finding a consistent cut in the past. Our results show that it suffices for the node to include values from the last 3 rounds for most cases. Observe that this method does not help "latest" because learning an older snapshot does not allow executing a fast action.



(a) Latest probabilities (b) Consistent probabilities

Fig. 3. Latest and consistent probabilities for $p = 10\%$



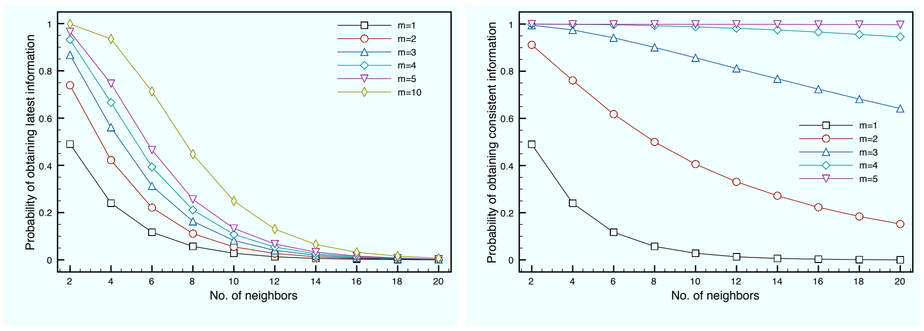
(a) Latest probabilities (b) Consistent probabilities

Fig. 4. Latest and consistent probabilities for $p = 20\%$

Thus, the probability of obtaining the consistent information is significantly higher than that of latest information. This suggests that it is better to utilize protocols that have slow actions verses protocols that have fast actions. In particular, it is better if actions that depend on the value of several neighbors are slow actions. On the other hand, if protocols must have fast actions, then it is better if they rely on a small number (preferably 1) of neighbors.

7 Pseudo-slow Action

The results in Section 6 show that if actions of a program are slow then their execution is expected to be more successful. Thus, the natural question is what happens if all program actions were fast? Can we allow such a program to utilize an old consistent state to evaluate its guard. We show that for a subset of the original actions, this is feasible if we analyze the original shared memory program to identify *dependent* actions.



(a) Latest probabilities (b) Consistent probabilities

Fig. 5. Latest and consistent probabilities for $p = 30\%$

We illustrate our approach in the context of the tree example in Section 5. For the sake of discussion, let us assume that all actions are fast actions; this is reasonable since it adds more restrictions on how each action can be executed. Furthermore, let us consider the case that we want j to be able to execute $AC3$ by utilizing a consistent state although not necessarily the latest state. Recall that action $AC3$ causes j to join the tree of k. If j is using a consistent state that is not necessarily the latest state, it is possible that k has changed its state in the interim. Observe that if k had increased the value of $root.k$ by executing $AC3$ then it is still safe for j to execute action $AC3$. However, if k executes $AC1$ and changes its color to red, subsequently observes that it has no children and executes $AC2$ then it may not be safe for j to join the tree of k. Thus, if we want to allow j to execute $AC3$ using a consistent state that is not necessarily latest then k must be prevented from executing either $AC1$ or $AC2$. Again, for the sake of discussion, let us assume that we want to restrict k from executing $AC2$. Hence, in this case, we will say that pseudo-slow execution of $AC3.j$ is dependent upon slowing down $AC2.k$.

In this approach, we allow j to utilize consistent snapshots for up to x previous TDMA rounds (i.e., j can execute $AC3.j$ if it obtains a consistent state that is no more than x rounds before the current time and evaluates that the guard of $AC3$ is true). However, in this case, if k ever wants to execute action $AC2$ then it must stay silent for at least $x+1$ TDMA rounds before executing action $AC2$. Note that this will essentially disable execution of action $AC3.j$ (i.e., at the end of $x+1$ silent rounds k knows that j cannot simultaneously execute $AC3.j$ and interfere with the execution of $AC2.k$) [6].

[6] We can relax this $x+1$ silent rounds requirement. For this, we modify the algorithm in Figure 1 slightly where a node, say j, not only notifies its neighbors about its own state but also includes a timestamp information about messages received from its neighbors. With this change, k can either execute its action $AC2$ if it stops transmitting for $x+1$ rounds or if it checks that j is aware of its color being red and, hence, will not execute action $AC3.j$.

We generalize this approach in terms of the following 4-step algorithm.

Step 1: Identify pseudo-slow actions. First, the designer needs to identify the set of actions, $\mathbb{A}$, that are fast actions but it is desired that they can execute as slow actions, where a node can utilize consistent (but not necessarily the latest) information about the state of neighbors. The choice of $\mathbb{A}$ is application dependent, i.e., it is based on the designers' belief/observation that quick execution of these actions is likely to help execution of the program. We denote the actions in $\mathbb{A}$ as a set of pseudo-slow actions since they are not slow actions but behave similar to the slow actions.

Step 2: Identify dependent actions. Let $\mathcal{A}_j$ be one of the pseudo-slow actions in $\mathbb{A}$ that is to be executed by node j. Let $\mathcal{A}_j$ be of the form $g \longrightarrow st$. Since $\mathcal{A}_j$ is a fast action, this implies that if the guard of $\mathcal{A}_j$ is true in some state then it can become false by execution of actions of one or more neighbors of j. Hence, the goal of this step is to identify the set of actions, say $\mathfrak{A}$, such that if (1) g evaluates to true in some state, (2) no action from $\mathfrak{A}$ is executed, and (3) no action of j is executed then it is still acceptable to execute the statement st in the given shared memory program. The value obtained for $\mathfrak{A}$ is called the dependent actions of $\mathcal{A}_j$.

In this step, for each action in $\mathbb{A}$, we identify the corresponding set of dependent actions. The dependent actions for $\mathbb{A}$ is obtained by taking the union of these dependent actions. Step 2 is successful if $\mathbb{A}$ and its dependent actions are disjoint. If there is an overlap between these two sets then the set of pseudo-slow actions needs to be revised until this condition is met.

Step 3: Choosing the delay value. The next step is to identify how much old information can be used in evaluating the guard of an action. Essentially, this corresponds to the choice of x in the above example. We denote this as the delay value of the corresponding action. The delay value x chosen for efficient implementation of pseudo-slow actions is also user dependent. The value will generally depend upon the number of neighbors involved in the execution of the pseudo-slow action. Based on the analysis from Section 6, we expect that a value of 3-4 is expected to be sufficient for this purpose.

Step 4: Revising the transformation algorithm. The last step of the algorithm is to utilize $\mathbb{A}$ identified in Step 1, the corresponding dependent actions identified in Step 2 and the delay value identified in Step 3 to revise the transformation algorithm. In particular, we allow a pseudo slow action at j to execute if (1) j obtains consistent state information about its neighbors, (2) j does not have more recent information about its neighbors than the one it uses, and (3) no more than x TDMA rounds have passed since obtaining the consistent state.

Additionally, a dependent action at j can execute if j does not transmit its own state for at least $x + 1$ rounds. (It is also possible for j to execute a dependent action earlier based on the knowledge j got about the state of its neighbors. However, for reasons of space, we omit the details.)

8 Discussion

What is specific to write-all in our transformation algorithm? Why is this transformation not applicable for message passing?
Write-all-with-collision (i.e., wireless broadcast) model helps a lot for our transformation, but is not strictly necessary. Our transformation is also applicable for message-passing, if on execution of an action at k at its TDMA slot, its state is made available to all of its neighbors before the next slot starts. It may not be easy and inexpensive to guarantee this condition for message passing, whereas for write-all with TDMA this condition is easily and inexpensively satisfied.
Can we relax the TDMA communication assumption?
The definitions of "latest" and "consistent" depend on the assumption that "k does not have another (missed) TDMA slot until the cut". This is used for ensuring that k does not execute any action in that interval, so k's new state is not out of sync with the cached state in the cut. Without using TDMA, the same condition can be achieved by using an alternative mechanism to communicate that k will not update its state for a certain duration. For example k can include a promise in its message that it will not update its state for some interval (e.g., until its next scheduled update, or until its sleep period is over).

Given that our transformation can tolerate message losses in the concrete model, dropping the TDMA mechanism would not hurt the performance of the transformed program significantly. The round concept could be used without the TDMA slots, and the nodes would utilize CSMA to broadcast their messages.
What are the rules of thumb for marking actions as slow?
As mentioned in the Introduction, an action be marked slow only if 1) guard is a stable predicate, 2) guard depends only on local variables, or 3) guard is a "locally stable" predicate. While the first two conditions are easy to detect, the locally stable condition requires reasoning about the program execution. We expect the protocol designer to understand his program.

A big problem is marking a fast action as slow, as this would violate correctness! It is better to err on the side of safety and mark the action as fast if there is some doubt about it being a slow action. Marking a slow action as fast does not violate correctness, but would just reduce the performance.
Do we need to use slow-motion execution for every program?
If the designer can mark all program actions as slow, there is obviously no need for slow-motion execution as there is no fast action remaining. Even when there are some fast actions remaining, if most of the actions are slow actions and message loss rates are not very high, these fast actions may not reduce the performance of the program significantly. However, if message loss rates increase further, it could be more beneficial to switch to slow-motion execution than to suffer from message losses voiding the latest cut and blocking the fast actions.

9 Conclusion

We have presented an extension to the shared memory model, by introducing the concept of slow action. A slow action is one such that once it is enabled at a node j, it can be executed at any later point at j provided that j does not execute another action in between. Slow actions mean that the process can tolerate slightly stale state from other processes, which enables the concrete system to be more loosely-coupled, and tolerate communication problems better. We quantified the improvements possible by using a slow action, and gave practical rules that help a programmer mark his program actions as slow and fast. For reducing the performance penalty of fast actions under heavy message loss environments, we also introduced the notion of slow-motion execution for fast actions.

Our work enables a good performance for transformed programs in realistic WSN environments with message loss. In future work, we will investigate adaptive switching to slow-motion execution to curb the performance penalty that message losses incur on fast actions. To this end, we will determine the break-even point for switching to the slow-motion execution mode, and middleware for switching to and back from the slow-motion mode seamlessly.

References

1. Arora, A.: Efficient reconfiguration of trees: A case study in the methodical design of nonmasking fault-tolerance. Science of Computer Programming (1996)
2. Arumugam, M.: A distributed and deterministic TDMA algorithm for write-all-with-collision model. In Proceedings of the 10th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS) LNCS:5340 (November 2008)
3. Arumugam, M., Kulkarni, S.S.: Self-stabilizing deterministic time division multiple access for sensor networks. AIAA Journal of Aerospace Computing, Information, and Communication (JACIC) 3, 403–419 (August 2006)
4. Dijkstra, E.W.: Self-stabilizing systems in spite of distributed control. Communications of the ACM 17(11) (1974)
5. Dolev, S., Israeli, A., Moran, S.: Self-stabilization of dynamic systems assuming only read/write atomicity. Distributed Computing 7, 3–16 (1993)
6. Herman, T.: Models of self-stabilization and sensor networks. In Proceedings of the Fifth International Workshop on Distributed Computing (IWDC), Springer LNCS:2918, 205–214 (2003)
7. Kulkarni, S.S., Arumugam, M.: SS-TDMA: A self-stabilizing mac for sensor networks. In: Sensor Network Operations. Wiley-IEEE Press (2006)
8. Kulkarni, S.S., Arumugam, M.: Transformations for write-all-with-collision model. Computer Communications 29(2), 183–199 (January 2006)
9. Mizuno, M., Nesterenko, M.: A transformation of self-stabilizing serial model programs for asynchronous parallel computing environments. Information Processing Letters 66(6), 285–290 (1998)
10. Nesterenko, M., Arora, A.: Stabilization-preserving atomicity refinement. Journal of Parallel and Distributed Computing 62(5), 766–791 (2002)