Self-Stabilizing Distributed Formation of Minimal k-Dominating Sets in Mobile Ad Hoc Networks

Li-Hsing Yen and Zong-Long Chen
Dept. of Computer Science and Information Engineering
National University of Kaohsiung, Kaohsiung, Taiwan 811, R.O.C.

***Abstract*—Dominating set in a mobile ad-hoc network (MANET) is a collection of devices acting as servers that store, forward, or backup data for other devices not in the set. To fulfill the service requirement, every device is either a dominator or adjacent to some dominator. Devices of the latter case are dominatees. To provide a more robust service, we can extend the definition of dominating set to k-dominating set, where each dominatee must be adjacent to at least k dominators (k is a constant). This paper proposes a self-stabilizing protocol that identifies a k-dominating set in a MANET. The identified set is guaranteed minimal in the sense that it contains no proper subset that is also a k-dominating set. We prove correctness and analyze stability property of this protocol. Simulation results indicate that the proposed protocol finds k-dominating sets of smaller size when compared with existing approaches.**

***Keywords*-self-stabilization; dominating set; distributed algorithms; MANET**

I. Introduction

In a mobile ad-hoc network (MANET), we can designate some devices as servers that provide a certain type of service to other nearby devices. Possible service types include message queuing, message forwarding, and data backlog. An important issue in this environment is to find a collection of devices that has the smallest size without degrading the service level. This issue relates to the classical dominating set problem, where dominators are servers in our environment. Dominating sets also serve other purposes. For example, dominating sets can be used for an energy-saving node scheduling in MANETs and wireless sensor networks [1], [2]. Connected dominating sets can also serve as backbone nodes in MANETs [3].

We can model a MANET as a connected, undirected graph $G = (V, E)$, where V are devices while E are communication links between devices. If S is a subset of V such that every node in $V - S$ is adjacent to some node in S, then S is a dominating set. Nodes in S are dominators and all nodes in $V - S$ are dominatees. If S contains no proper subset that is also a dominating set, then S is a *minimal* dominating set.

When the service level demanded by a dominatee exceeds that offered by a single dominator, we may need to aggregate service from several dominators to fulfill the need of the dominatee. Providing several dominators to a dominatee can also help load sharing and fault tolerance. Therefore, to provide a better quality of service, dominating sets can be extended to k-dominating sets. Given a k-dominating set S, every node not in S is adjacent to at least k nodes in S. Here k is a positive integer.

Self-stabilizing distributed algorithms guarantee that regardless of whether the initial state of a distributed system is legitimate or not, the whole system eventually enters a legitimate state and remains in the set of legitimate states [4]. Self-stabilization tolerates transient faults by always coming back to correct states after transient faults. In MANETs, devices dynamically joins or leaves the network due to mobility or changes of power status (e.g., entering or leaving doze mode). Furthermore, transmission may not necessarily succeed due to collisions or interference. Such dynamic participations and transmission failures place a design challenge to the correctness of conventional protocols and algorithms. On the other hand, transient faults caused by dynamic participations or transmission failures are no problems in self-stabilizing algorithms because these algorithms are designed to handle such faults. This is why we study self-stabilizing algorithms in MANETs.

Several self-stabilizing algorithms have been proposed to identify 1-dominating sets in distributed systems [5], [6], [7], [8], [9]. Huang et al. [10], [11] developed self-stabilizing algorithms for minimal 2-dominating sets. Kamei and Kakugawa [12] proposed a self-stabilizing approximation algorithm to find minimum k-domination. Recently, we proposed a game-theoretic self-stabilizing approach to multi-dominating set in a distributed system [13]. All the above-mentioned algorithms are expressed in guarded commands [14], which help formal correctness verification of the design. However, guarded commands are grounded in the shared-variable execution environment, which allows local variables of a node to be directly read by neighboring nodes. This is certainly not possible in MANETs. In this paper, we discuss challenges concerning the transformation of an algorithm in guarded commands to a protocol running in MANETs. We also propose a self-stabilizing protocol that is based on our previous work in [13]. Simulation results indicate that the proposed approach find smaller 1-dominating sets, 2-dominating sets, and k-dominating sets when compared with existing approaches.

The remainder of this paper is organized as follows: A brief background is described in Section II. Section III elaborates on our approach. In Section IV, simulation results are discussed and compared among subject schemes. Lastly Section V concludes this paper.

II. Preliminaries

Self-stabilization for a system can be defined with respect to a predicate over all states of the system [15]. The predicate under consideration specifies all correct or *legitimate* states of the system. In case of our minimal k-dominating set problem, the predicate identifies a system state legitimate if and only if all nodes claiming themselves dominators in this state indeed constitute a minimal k-dominating set. A distributed algorithm is self-stabilizing with respect to the predicate if the following two conditions hold:

- *Convergence*. Starting from arbitrary state (possibly illegitimate), the algorithm eventually reaches a legitimate state.
- *Closure*. Any state following a legitimate state is also legitimate.

Most existing self-stabilizing algorithms are expressed in the form of guarded commands [14]. A guarded command specifies one rule that consists of a condition part (a Boolean expression) followed by an action part (statements). A guarded command is enabled if its condition part is evaluated true. A process[1] executes the action part of a command only when that command is enabled. Processes update their local states in action parts. The execution of the action part is assumed atomic, i.e., not interleaved with the execution of any other guarded command. When a process have more than one enabled commands, only one of them can be executed at a time. Which command is executed in this case is nondeterministic.

Self-stabilizing algorithms usually assume some type of execution models that can be characterized by the presence of a particular scheduler or daemon. In a *central daemon* execution model, only one process can execute at a time. With a *synchronous daemon*, all processes are scheduled to execute in parallel, which is a perfect match for a synchronous distributed algorithm. A *distributed daemon* subsumes the aforementioned models in the sense that any non-empty subset of processes can execute in parallel.

In [13], we have presented a distributed algorithm that runs under a central daemon to identify a minimal k-dominating set. This algorithm consists of six guarded commands (R1 - R6) for each process p_i, as shown below

R1 $|N_i \cap \{p_j | x(j) = \textit{true}\}| < k \wedge x(i) \neq \textit{true} \wedge$
$g(i) \neq \text{UNDER} \rightarrow g(i) := \text{UNDER}$

R2 $|N_i \cap \{p_j | x(j) = \textit{true}\}| = k \wedge x(i) \neq \textit{true} \wedge$
$g(i) \neq \text{EQUAL} \rightarrow g(i) := \text{EQUAL}$

R3 $(|N_i \cap \{p_j | x(j) = \textit{true}\}| > k \vee x(i) = \textit{true}) \wedge$
$g(i) \neq \text{OVER} \rightarrow g(i) := \text{OVER}$

R4 $|N_i| < k \wedge x(i) \neq \textit{true} \rightarrow x(i) := \textit{true}$

R5 $|N_i| \geq k \wedge \exists p_j \in N_i : g(j) = \text{UNDER} \wedge x(i) \neq \textit{true}$
$\rightarrow x(i) := \textit{true}$

R6 $|N_i| \geq k \wedge \forall p_j \in N_i : g(j) = \text{OVER} \wedge x(i) \neq \textit{false} \rightarrow$
$x(i) := \textit{false}$

The condition part and the associated action part of each command are separated by '$\rightarrow$'. N_i denotes the set of p_i's neighbors. Each process p_i uses a local Boolean variable $x(i)$ to denote whether p_i chooses to be in the dominating set. It uses another variable $g(i)$ to indicate p_i's current need for dominators. While $g(i) = \text{UNDER}$ and $g(i) = \text{EQUAL}$ indicate p_i (a dominatee) has a less and an exact number of adjacent dominators than needed, respectively, $g(i) = \text{OVER}$ means either p_i has more adjacent dominators than needed or p_i itself is a dominator (so it needs no dominator at all.) Each process p_i decides $x(i)$ based on the $g(j)$ values of all its neighbor $p_j \in N_i$, and updates $g(i)$ according to $x(i)$ and the values of $x(j)$'s for which $p_j \in N_i$.

III. The Proposed Approach

A. Transformation of Guarded Commands

Guarded commands are grounded in the shared-variable execution environment and are characterized by two properties. First, a process can write and can only write its own local variables. Second, a process can read and can only read its own local variables plus local variables of its neighboring processes. This execution environment is not readily available in MANETs.

The first challenge in transforming guarded commands into a protocol for MANETs is that local variables of a node in MANETs cannot be directly read by their neighbors. Nodes therefore should propagate actively any change of local variables to their neighbors, and neighbors can only have cached versions of these variables. We realize the propagation by multicast. We use $x_i(i)$ (resp. $g_i(i)$) to denote $x(i)$ (resp. $g(i)$) owned by p_i. Variables $x_j(i)$'s and $g_j(i)$'s, where $j \neq i$, are values of $x_i(i)$ and $g(i)$, respectively, cached by node $p_j \in N_i$. For all $p_j \notin N_i$, $x_j(i)$'s and $g_j(i)$'s are undefined.

The transformation uses a back-off timer T_i in each p_i. When p_i receives a message and detects the need to update $x_i(i)$ or $g_i(i)$, it loads a random value to T_i. Only when T_i expires can p_i update these variables. The purpose of this timer is to reduce the frequency of updates. Only one update is needed even if p_i receives more messages before T_i expires. In the extreme case, the scheduled update on $x_i(i)$ or $g_i(i)$ may no longer be needed because the precondition for the update is invalidated by the reception of subsequent messages during T_i's active period.

Each guarded command in the algorithm updates only one local variable (either $x_i(i)$ or $g_i(i)$). Since a central daemon

[1]We use *process* and *node* interchangeably in this paper.

C1: $\delta(p_i) < k \wedge x_i(i) \neq \mathit{true} \wedge g_i(i) \neq \text{UNDER}$
C2: $\delta(p_i) = k \wedge x_i(i) \neq \mathit{true} \wedge g_i(i) \neq \text{EQUAL}$
C3: $(\delta(p_i) > k \vee x_i(i) = \mathit{true}) \wedge g_i(i) \neq \text{OVER}$
C4: $|N_i| < k \wedge x_i(i) \neq \mathit{true}$
C5: $|N_i| \geq k \wedge \exists p_j \in N_i : g_i(j) = \text{UNDER} \wedge x_i(i) \neq \mathit{true}$
C6: $|N_i| \geq k \wedge \forall p_j \in N_i : g_i(j) = \text{OVER} \wedge x_i(i) \neq \mathit{false}$
where $\delta(p_i) = |N_i \cap \{p_j | x_i(j) = \mathit{true}\}|$

Figure 1. Six conditions used by Algorithm 1

schedules only one rule at a time, the values of $x(i)$'s and $g(i)$'s are not simultaneously and consistently updated. For example, it is possible that $x_i(i) = \mathit{true}$ and $k = 1$ but $g_i(i) = \text{UNDER}$ for some p_i. We prevent such inconsistency by rechecking and updating $g_i(i)$ following the update of $x_i(i)$.

Algorithm 1 shows the resulting transformation as three event-driven procedures for each process p_i. Executions of these procedures are mutually exclusive. Six conditions (C1 to C6) used by the algorithm are shown in Fig. 1. These conditions correspond to the condition parts of R1 to R6. Fig. 2 shows a typical run of the algorithm in a four-node system.

Algorithm 1 Event-driven procedures for each process p_i

```
On initialization                                        ▷ Procedure 1
    if any of C1 to C6 is true then
        load $T_i$ with a random value
    end if
end

On receiving a multicast $\{x_j(j), g_j(j)\}$ from $p_j$   ▷ Procedure 2
    $x_i(j) \leftarrow x_j(j)$
    $g_i(j) \leftarrow g_j(j)$
    if any of C1 to C6 is true then
        load $T_i$ with a random value if $T_i$ is not running
    else
        stop and reset $T_i$
    end if
end

When $T_i$ expires                                        ▷ Procedure 3
    if C4 or C5 is true then
        $x_i(i) \leftarrow true$
    else if C6 is true then
        $x_i(i) \leftarrow false$
    end if
    if C1 is true then
        $g_i(i) \leftarrow$ UNDER
    else if C2 is true then
        $g_i(i) \leftarrow$ EQUAL
    else if C3 is true then
        $g_i(i) \leftarrow$ OVER
    end if
    multicast $\{x_i(i), g_i(i)\}$ to neighbors
end
```

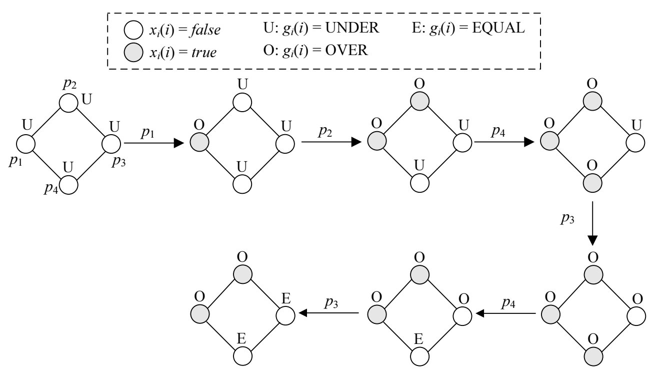

Figure 2. A typical execution run.

B. Correctness Proof

We shall now prove the correctness of Algorithm 1, i.e., it does identify a minimal k-dominating set when the system enters a stable state. We first introduce the following definitions and notations. For each process p_i, L_i is an assignment of one value to each of p_i's local variable. For each communication channel $c_{i,j}$ by which process p_i sends message to p_j, $M_{i,j}$ is the channel state of $c_{i,j}$, which contains all messages currently in-transit in $c_{i,j}$. Let τ_i denote the status of T_i such that $\tau_i = \mathit{true}$ if T_i is active (running) and $\tau_i = \mathit{false}$ otherwise. A *global state* (or state for short) is defined by $s = \{L_i, \tau_i, M_{i,j}\}_{i=1}^{n}$. This definition is specifically for networking environment. A global state for algorithms running in shared memory model comprises only $\{L_i\}$ and need not include the contents of local timers and channel states.

Stable states in our design correspond to *quiescent* global states, i.e., states where no further state transition is possible. In the distributed algorithm expressed in guarded commands, a state is stable only if no command is enabled in any process. This condition corresponds to a state in our transformation where none of $\{\text{C1}, \ldots, \text{C6}\}$ holds in any node. However, as the transformation additionally involve timers and message transmissions, stable states in our transformation also demands no active timers and in-transit messages.

Definition 1: A state $s = (\{L_i\}, \{\tau_i\}, \{M_{i,j}\})$ is stable if the following conditions all hold.

- C1 to C6 are all false in every p_i.
- $\tau_i = \mathit{false}$ for all p_i.
- $M_{i,j} = \emptyset$ for all $c_{i,j}$.

Let $A(s, i)$ denote the last procedure executed by p_i before reaching state s. The values of $A(s, i)$ are defined to be *init*, *recv*, and *Tout* when the last executed procedures are Procedure 1, 2, and 3, respectively.

Variable $g_i(i)$ is said to be *correct* if its value matches those shown in Tables I. Observe that $g_i(i)$ is correct right after the execution of Procedure 3. The following property is also easy to see.

Table I
CORRECT VALUES OF $g_i(i)$.

$\delta(p_i)$	$x_i(i)$ *false*	*true*
$< k$	UNDER	OVER
$= k$	EQUAL	OVER
$> k$	OVER	OVER

Property 1: $g_i(i)$ is correct if and only if C1, C2, and C3 are false in p_i.

The following lemmas are need for our main result.

Lemma 1: If state s is stable, $g_i(i)$ is correct for all p_i in s.

Proof: Suppose, by way of contradiction, that $g_i(i)$ is not correct for some p_i in s. By Property 1, this implies that C1, C2, or C3 holds. Now consider the value of $A(s,i)$. $A(s,i) \neq$ *Tout* because Procedure 3 guarantees the correctness of $g_i(i)$. If $A(s,i)$ were *init* or *recv*, then C1, C2, or C3 would be detected true and T_i would be activated during the execution of $A(s,i)$. Consequently, $\tau_i =$ *true* in s, which contradicts with the assumption that s is a stable state. We thus have the proof. ■

Lemma 2: If $|N_i| < k$ for some node p_i, $x_i(i)$ must be *true* in any stable state.

Proof: $|N_i| < k$ implies the truth of C4, which causes the activation of T_i initially through Procedure 1. The expiration of T_i and the subsequent execution of Procedure 3 will then set $x_i(i)$ to *true*. The only way to change $x_i(i)$ back to *false* demands the truth of C6, which is impossible because $|N_i| < k$. Therefore, $x_i(i)$ must be *true* in any stable state. ■

Theorem 1: In stable states, $D = \{p_i | x_i(i) = \textit{true}\}$ is a minimal k-dominating set.

Proof: We first prove that D is a k-dominating set. By way of contradiction, assume that D is not a k-dominating set in some stable state s. This means that there exists at least one node p_j such that $x_j(j) = \textit{false} \wedge \delta(p_j) < k$ in s. By Lemma 1, $g_j(j)$ must be correct in s, which means

$$g_j(j) = \text{UNDER}. \tag{1}$$

By Lemma 2, $x_j(j) =$ *false* implies that $|N_j| \geq k$. It follows that there exists some $p_i \in N_j$ such that

$$x_i(i) = \textit{false}. \tag{2}$$

By Lemma 2 again, (2) implies that

$$|N_i| \geq k. \tag{3}$$

(1) to (3) together imply that C5 must be true for p_i in s. However, this contradicts with the assumption that s is stable. Therefore, D must be a k-dominating set.

We then prove that D is also minimal. If D were not minimal, then there would exist at least one node $p_i \in D$ such that $D \setminus \{p_i\}$ is still a k-dominating set, which implies that $\delta(p_i) \geq k$ and $\forall p_j \in N_i : \delta(p_j) > k$. The former condition implies that $|N_i| \geq k$. The latter condition implies that $\forall p_j \in N_i : g_j(j) =$ OVER since all $g_j(j)$'s are correct by Lemma 1. Therefore, C6 holds for p_i in stable states, which contradicts with the assumption that s is stable. ■

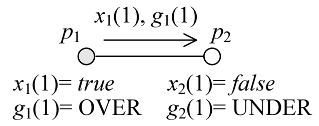

Figure 3. An incoherent state where a message from p_1 to p_2 is in transit.

C. Stability Issues

Three factors, namely, incoherent states, simultaneous moves, and indirect information, affect stability property of Algorithm 1 in MANETs. A state is *coherent* if $\forall p_i : \forall p_j \in N_i : x_i(j) = x_j(j) \wedge g_i(j) = g_j(j)$. Intuitively, all nodes in a coherent state have up-to-date information about neighboring node's variables. A coherent state becomes *incoherent* after some node p_i has updated its variables but at least one of its neighbors has yet been informed of the update. The only reason of such incoherence is due to in-transit message as stated below:

$$x_i(j) \neq x_j(j) \vee g_i(j) \neq g_j(j) \Longrightarrow M_{j,i} \neq \emptyset.$$

If nodes are allowed to make changes to their local variables (i.e., execute Procedure 3) in incoherent states, then the system may not enter a stable state. Consider the example shown in Fig. 3, where p_1 has changed $x_1(1)$ to *true* and $g_1(1)$ to OVER, but the multicast message to p_2 is currently in transit. This is an incoherent state since $x_2(1) \neq x_1(1)$ and $g_2(1) \neq g_1(1)$. If p_2 executes Procedure 3 at this moment, it will change $x_2(2)$ to *true* and $g_2(2)$ to OVER, causing considerable subsequent state transitions. If p_2 executes Procedure 3 after receiving the message and updating $x_2(1)$ and $g_2(1)$ (i.e., p_2 makes its decision in a coherent state), then $x_2(2)$ will be *false* and $g_2(2)$ will be EQUAL, and no further state transition is possible.

If message delays are small enough to guarantee that $M_{i,j} = \emptyset$ whenever node p_j makes a change to $x_j(j)$ or $g_j(j)$ (i.e., executes Procedure 3), then all decisions are made in coherent states.

Even nodes make decisions all in coherent states, instability of the algorithm may still occur if *simultaneous moves* of nodes are permitted. The guarded-command version precludes simultaneous moves by assuming a central daemon, which allows only one process to execute at a time. It is possible to realize or emulate a central daemon in MANETs. We may use a distributed mutual exclusion algorithm to ensure that modifications to local variables are mutually exclusive between neighboring nodes. Identifiers of nodes can be used to break ties when two or more neighboring nodes attempt modifications.

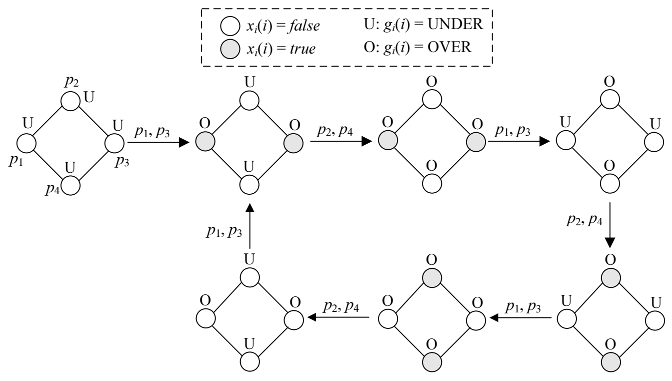

Figure 4. A state transition loop that comprises only coherent states without simultaneous moves.

Unfortunately, decisions made in coherent states without simultaneous moves still do not guarantee stability. In fact, although stable states imply coherent states, the converse does not necessarily hold because any condition in $\{\mathrm{C1}, \dots, \mathrm{C6}\}$ can be true in a coherent state. An infinite sequence of state transitions that comprises only coherent states without simultaneous moves is possible. Fig. 4 shows an example that corresponds to the four-node system shown in Fig. 2.

The root of this problem comes from *indirect information*. The main idea behind the algorithm design is that p_i's decision of being a dominator or not mainly depends on $\sigma(p_j)$ of all its neighbors $p_j \in N_i$ (except those p_j for which $|N_j| < k$), where $\sigma(p_j) = \{p_k | p_k \in N_j \wedge x_k(k) = \mathit{true}\}$. p_i learns of $\sigma(p_j)$ indirectly by $g_i(j)$. It is ensured in coherent states that $g_i(j) = g_j(j)$, but $g_j(j)$ does not reflect $\sigma(p_j)$ when some $p_k \in N_j$ has changed $x_k(k)$ but p_j has not yet updated $g_j(j)$.

To solve this problem, we can let each p_i evaluate $\sigma(p_j)$ directly from $x_k(k)$ for all $p_k \in N_j$. To this end, each node p_k is required to multicast $x_k(k)$ with a transmission range that is twice of that p_k uses to communicate with its neighbors. This approach also eliminates the need for all $g_i(j)$'s.

IV. Simulation Results

We conducted simulations to study the performance of the proposed approach and compare it with those of existing methods. In our simulations, 50 to 100 wireless nodes were randomly deployed in a 1000×1000 m^2 area. Each node has a default transmission range of 200 m. Two wireless nodes are neighbors only when they are within the transmission range of each other. Message delays are sufficiently small to disable decision makings in incoherent states. We are primarily concerned with the sizes of the dominating sets identified by these methods.

We considered several self-stabilizing distributed algorithms that find minimal dominating sets. These algorithms

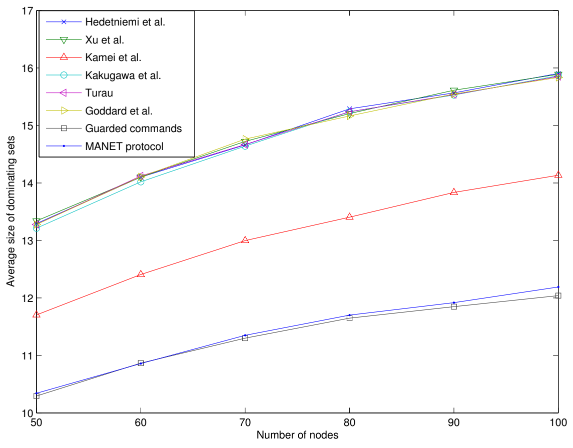

Figure 5. Average sizes of dominating sets ($k = 1$).

were respectively proposed by by Hedetniemi et al. [5], Xu et al. [6], Kakugawa et al. [7], Turau [8], and Goddard et al. [9]. We also tested the k-dominating set algorithm proposed by Kamei et al. [12] by setting k to 1. The guarded-command algorithm and the transformed MANET protocol were also tested. Fig. 5 shows average sizes of dominating sets generated by each method when the number of nodes was varied from 50 to 100. Each average was obtained over 1000 runs.

All classical self-stabilizing single-domination algorithms [5], [6], [7], [8], [9] performed nearly the same. These algorithms all generated more dominators than the k-dominating set algorithm proposed by Kamei et al. [12]. Kamei's algorithm is next to both the guarded-command version and the transformed protocol. However, since different algorithms might run under different types of daemons, the results here were not obtained on a fair basis and should not be overstretched.

For 2-dominating sets, we compared the proposed approaches with two algorithms proposed by Huang et al. [10], [11]. The algorithm proposed by Kamei et al. [12] was also considered. Fig. 6 shows the results for 2-dominating sets. The results indicate that Huang's algorithm for central daemon [11] found more dominators than his algorithm for distributed daemon [10]. Kamei's algorithm performed better than Huang's algorithms but generally worse than the two proposed approaches.

Figure 7 compares Kamei's algorithm and the two proposed approaches with the number of nodes fixed to 100 and k varied from 1 to 5. The result still shows the outperformance of the proposed approaches over Kamei's algorithm.

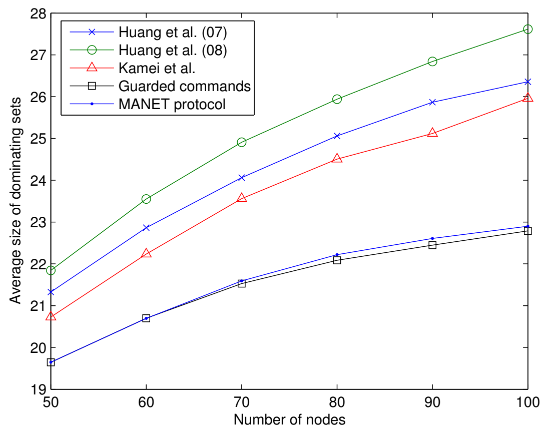

Figure 6. Average sizes of 2-dominating sets.

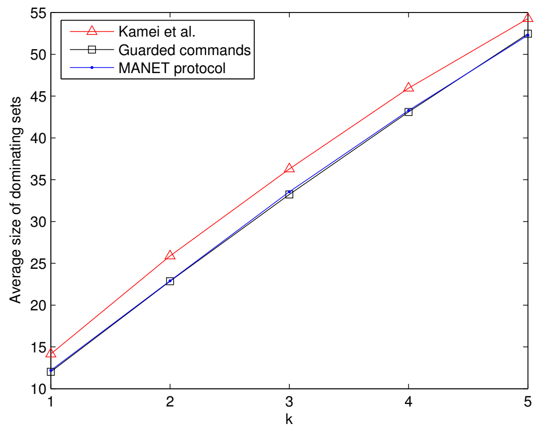

Figure 7. Comparison between Kamei's algorithm and the proposed approaches in term of average sizes of k-dominating sets.

V. Conclusions

We have transformed a distributed algorithm expressed in guarded commands that identifies a minimal k-dominating set in a distributed system into a protocol for MANETs. We have proved that this protocol identifies minimal k-dominating sets in stable states. Three factors that affect the stability of the protocol, namely, incoherent states, simultaneous moves, and indirect information, have been discussed. Simulation results indicate that the transformed protocol performs nearly the same as the distributed algorithm expressed in guarded commands, and both perform better than conventional self-stabilizing algorithms in terms of average size of dominating sets.

References

[1] T. Moscibroda and R. Wattenhofer, "Maximizing the lifetime of dominating sets," in *Proc. 19th IEEE Int'l Parallel and Distributed Processing Symp.*, Apr. 2005.

[2] B. Pazand and A. Datta, "Minimum dominating sets for solving the coverage problem in wireless sensor networks," in *Lecture Notes in Computer Science 4239*, H. Youn, M. Kim, and H. Morikawa, Eds. Springer-Verlag, 2006, pp. 454–466.

[3] B. Liang and Z. Haas, "Virtual backbone generation and maintenance in ad hoc network mobility management," in *Proc. IEEE INFOCOM*, Mar. 2000, pp. 1293–1302.

[4] E. W. Dijkstra, "Self-stabilizing systems in spite of distributed control," *Comm. ACM*, vol. 17, no. 11, pp. 643–644, Nov. 1974.

[5] S. M. Hedetniemi, S. Hedetniemi, D. P. Jacobs, and P. K. Srimani, "Self-stabilizing algorithms for minimal dominating sets and maximal independent sets," *Computers & Mathematics with Applications*, vol. 46, no. 5-6, pp. 805–811, Sep. 2003.

[6] Z. Xu, S. T. Hedetniemi, W. Goddard, and P. K. Srimani, "A synchronous self-stabilizing minimal domination protocol in an arbitrary network graph," in *Lecture Notes in Computer Science 2918*. Springer-Verlag, 2003, pp. 26–32.

[7] H. Kakugawa and T. Masuzawa, "A self-stabilizing minimal dominating set algorithm with safe convergence," in *Int'l Parallel and Distributed Processing Symposium*, Apr. 2006.

[8] V. Turau, "Linear self-stabilizing algorithms for the independent and dominating set problems using an unfair distributed scheduler," *Inform. Process. Lett.*, vol. 103, no. 3, pp. 88–93, 2007.

[9] W. Goddard, S. T. Hededtniemi, D. P. Jacobs, P. K. Srimani, and Z. Xu, "Self-stabilizing graph protocols," *Inform. Process. Lett.*, vol. 18, no. 1, pp. 189–199, 2008.

[10] T. C. Huang, J. C. Lin, C. Y. Chen, and C. P. Wang, "A self-stabilizing algorithm for finding a minimal 2-dominating set assuming the distributed demon model," *Computers & Mathematics with Applications*, vol. 54, no. 3, pp. 350–356, 2007.

[11] T. C. Huang, C. Y. Chen, and C. P. Wang, "A linear-time self-stabilizing algorithm for the minimal 2-dominating set problem in general networks," *Journal of Information Science and Engineering*, vol. 24, no. 1, pp. 175–187, 2008.

[12] S. Kamei and H. Kakugawa, "A self-stabilizing approximation algorithm for the distributed minimum k-domination," *IEICE Trans. on Fundamentals of Electronics, Communications and Computer Sciences*, no. 5, pp. 1109–1116, 2005.

[13] L.-H. Yen and Z.-L. Chen, "Game-theoretic approach to self-stabilizing distributed formation of minimal multi-dominating sets," *IEEE Trans. Parallel Distrib. Syst.*, to appear.

[14] E. W. Dijkstra, "Guarded commands, nondeterminacy, and formal derivation of programs," *Comm. ACM*, vol. 18, no. 8, pp. 453–457, Aug. 1975.

[15] A. D. Kshemkalyani and M. Singhal, *Distributed Computing: Principles, Algorithms, and Systems*. Cambridge, UK: Cambridge University Press, 2008, p. 634.