Proceedings of the 8th IPDPS Workshop Formal Methods for Parallel Programming: Theory and Applications, Nice, France, April 22-26, 2003

A Self-Stabilizing Distributed Algorithm for Minimal Total Domination in an Arbitrary System Graph *

Wayne Goddard, Stephen T. Hedetniemi David P. Jacobs and Pradip K Srimani
Department of Computer Science
Clemson University
Clemson, SC 29634-0974

Abstract

In a graph $G = (V, E)$, a set $S \subseteq V$ is said to be total dominating *if every $v \in V$ is adjacent to some member of S. When the graph represents a communication network, a total dominating set corresponds to a collection of servers having a certain desirable backup property, namely, that every server is adjacent to some other server.* Self-stabilization*, introduced by Dijkstra [1, 2], is the most inclusive approach to fault tolerance in distributed systems [3, 4]. We propose a new self-stabilizing distributed algorithm for finding a minimal total dominating set in an arbitrary graph. We also show how the basic ideas behind the proposed protocol can be generalized to solve other related problems.*

1 Introduction

In a distributed system, each node has a set of local variables whose contents specify the local state of the node. The state of the entire system, called its *global state*, is the union of the local states of all the nodes. Each node has only a partial view of the global state, and this depends on the connectivity of the system and the propagation delay of different messages. Yet, the objective in a distributed system is to arrive at a desirable global final state, or legitimate state. One goal of a distributed system is to function correctly, i.e., the global state should remain legitimate in the presence of faults (transient). Often, malfunctions or perturbations bring the system to some illegitimate state, and it is desirable that the system be automatically brought back to a legitimate state. *Self-stabilization*, introduced by Dijkstra [1, 2], is the most inclusive approach to fault tolerance in distributed systems [3, 4]; it brings the system back to a legitimate state, starting from any illegitimate state (caused by any transient fault), without any intervention by an external agent. In a self-stabilizing algorithm, each node maintains its local variables, and can make decisions based on the knowledge of its neighbors' states.

In a self-stabilizing algorithm, a node changes its local state by making a *move* (a change of local state). The algorithm is a set of rules of the form "**if** $p(i)$ **then** M", where $p(i)$ is a predicate and M is a move. A node i becomes *privileged* if $p(i)$ is true. When a node becomes privileged, it may execute the corresponding move. We assume a serial model in which no two nodes move simultaneously. A central daemon selects, among all privileged nodes, the next node to move. If two or more nodes are privileged, one cannot predict which node will move next. Multiple protocols exist [5, 6, 7] that provide such a scheduler. Our algorithms can easily be combined with any of these protocols to work under different schedulers as well.

A distributed system can be modeled with an undirected graph $G = (V, E)$, where V is a set of n nodes and E is a set of m edges. If $i \in V$, then $N(i)$, its *open neighborhood*, denotes the set of nodes to which i is adjacent, and $N[i] = N(i) \cup \{i\}$ denotes its *closed neighborhood*. Every node $j \in N(i)$ is called a *neighbor* of node i. Throughout this paper we assume G is connected and $n > 1$.

Recall that $S \subseteq V$ is a *dominating* set [8, 9] if $N(i) \cap S \neq \emptyset$ for every $i \in V - S$. In the Turing machine model, the problem of finding a dominating set of *minimum* size is NP-hard [10], but finding minimal dominating sets is straightforward and can be done in linear-time. In the self-stabilizing model, several linear-time and polynomial-time algorithms for finding *minimal* dominating sets appear in [11, 12]. A set $S \subseteq V$ is a *total dominating* set if $N(i) \cap S \neq \emptyset$ for every $i \in V$. If a dominating set in a communication network represents a set of nodes necessary to provide an acceptable level of service, then a total dominating set represents a similar set of servers with the added capability that each server is adjacent to at least

*This work has been supported by NSF grant # ANI-0073409 and NSF grant # ANI-0218495

one other server. In this way, each server has a backup resource. Should its capability as a server be compromised, it can obtain backup from another server with a minimum delay. Thus total dominating sets are more fault tolerant than dominating sets. It has been shown [10] that the problem of computing a total dominating set of minimum size is NP-hard; no self-stabizing algorithm exists to compute either the minimal total dominating sets or a minimal dominating set of minimum cardinality.

In this paper we are interested in minimal total dominating sets. We present a detreministic self-stabilizing algorithm for finding such sets. It is to be noted that the proposed algorithm uses distinct node IDs. We also show how the proposed algorithm also generalizes to other related problems.

2 Self-Stabilizing Total Domination Algorithm

Our algorithm requires that every node have a unique ID. We will sometimes use i interchangeably to denote a node, and the node's ID. We assume there is a total ordering on the IDs.

In our algorithm, each node i has two variables: a pointer $p(i)$ (which may be null) and a booean flag $x(i)$. If $p(i) = j$ then we say that i *points to* j. At any given time, we will denote with D the current set of nodes i with $x(i) =$ true .

Definition 1 *For a node i, we define $m(i)$ as its neighbor having the smallest ID.*

Definition 2 *We define the pointer expression $q(i)$ as follows:*

$$q(i) = \begin{cases} m(i) & \text{if } N(i) \cap D = \emptyset \\ j & \text{if } N(i) \cap D = \{j\} \\ null & \text{if } |N(i) \cap D| \geq 2. \end{cases}$$

Note that the value $q(i)$ can be computed by i (i.e., it uses only local information).

Definition 3 *We define the boolean condition $y(i)$ to be* true *if and only if some neighbor of i points to it.*

The algorithm consists of one rule shown in Figure 1. Thus, a node i is privileged if $x(i) \neq y(i)$ or $p(i) \neq q(i)$. If it executes, then it sets $x(i) = y(i)$ and $p(i) = q(i)$.

Lemma 1 *If Algorithm 1 stabilizes, then D is a minimal total dominating set.*

Proof: First, we claim that D is a total dominating set. For suppose, by contradiction, that some node i is not totally dominated (that is, has no neighbor in D). Then $N(i) \cap D = \emptyset$. Since the system is stable, $p(i) = q(i) = m(i)$, and $m(i) \notin D$. But this implies $y(m(i)) = true$ and $x(m(i)) = false$, and so node $m(i)$ is privileged, a contradiction. Thus D is a total dominating set.

if $(x(i) \neq y(i))$ or $(p(i) \neq q(i))$
then set $x(i) = y(i)$ and $p(i) = q(i)$

Figure 1. Algorithm 1: Minimal Total Dominating Set()

Next, we claim that D is minimal. For suppose there is some $j \in D$ for which $D - \{j\}$ is a total dominating set. Since $j \in D$, or $x(j) =$ true, there is some vertex $i \in N(j)$ for which $p(i) = j$. But since $p(i) = q(i)$, node j must be a unique neighbor of i with membership in D. Thus the removal of j will leave i undominated. □

We say that node i *invites* node j if, at some time *t*, node i has no neighbor in D and then executes the rule, causing $p(i) = m(i) = j$. For a node to join D, it must either be pointed to from an initial erroneous state or be invited.

We now show our algorithm stabilizes. Observe that if D remains the same, then every node can execute at most once (to correct its pointer). So it suffices to show that D changes at most a finite number of times.

Definition 4 *We say a move is an* in-move *if it causes $x(i)$ to become true, thereby causing a node i to enter D.*

Lemma 2 *Let i be a node and suppose that between time t and t', there is no in-move by any node $k > i$. Then during this time interval node i can make at most two in-moves.*

Proof: The first in-move made by i may have been because a neighboring node happened to initially point to i. The second in-move made by i must be by invitation. So suppose i is invited by node j. Then i is the smallest node in j's neighborhood, since $m(j) = i$, and at the time of invitation, all other nodes in j's neighborhood are out of D. By our assumption, their membership status does not change, so j remains pointing to i throughout, and so i remains in D for the remaining duration of the time interval. □

Theorem 1 *Algorithm 1 always stabilizes, and finds a minimal total dominating set.*

Proof: It suffices to show that every node makes only a finite number of in-moves. By Lemma 2, node n, which has largest ID, makes at most two in-moves. During each of the three time intervals, when node n is not making an in-move, using Lemma 2 again, node $n - 1$ makes at most two in-moves. It is easy to show this argument can be repeated, showing that each node can make only finitely many in-moves during the intervals in which larger nodes are inactive. □

if $(x(i) \neq y(i))$ or $(\mathcal{P}(i) \neq Q(i))$
then set $x(i) = y(i)$ and $\mathcal{P}(i) = Q(i)$

Figure 2. Algorithm 2: Minimal Extended Dominating Set()

3 Minimal Extended Domination

We now show how the basic ideas of the previous section can be generalized to obtain algorithms for other domination problems. A dominating set is a set in which, for all i,

$$|N[i] \cap D| \geq 1,$$

and a total dominating set satisfies

$$|N(i) \cap D| \geq 1.$$

Assume now that for each node $i \in V$, the set $\mathcal{N}(i)$ represents some fixed subset of its closed neighborhood $N[i]$. Assume further that each node has a target integer $t(i) \leq |\mathcal{N}(i)|$, indicating how many elements of $\mathcal{N}(i)$ are required to dominate i. Note that in the case of total domination $\mathcal{N}(i)$ is precisely $N(i)$ and $t(i)$ is uniformly one. Given these assumptions we seek a minimal set D in which, for all i,

$$|\mathcal{N}(i) \cap D| \geq t(i). \quad (1)$$

Now, for the algorithm, each node has a set of pointers, denoted $\mathcal{P}(i)$, whose cardinality is bounded by $t(i)$; we allow $\mathcal{P}(i)$ to contain i. Each node also has a boolean flag $x(i)$. As before, $x(i)$ should be true if and only if some node points to i, and also as before, D will denote the set of nodes with true flags at any point in time.

At a given time, assume $|D \cap \mathcal{N}(i)| = k \leq t(i)$. Then since $t(i) \leq |\mathcal{N}(i)|$, there are at least $t(i) - k$ members in $\mathcal{N}(i) - D$. Let M_i denote the unique set of those $t(i) - k$ nodes in $\mathcal{N}(i) - D$ having smallest ID's. Note this set depends on D.

We define a set of pointers $Q(i)$ as follows.

$$Q(i) = \begin{cases} (D \cap \mathcal{N}(i)) \cup M_i & \text{if } |\mathcal{N}(i) \cap D| = k \leq t(i) \\ \emptyset & \text{if } |\mathcal{N}(i) \cap D| > t(i). \end{cases}$$

As before, we define the boolean condition $y(i)$ to be *true* if and only if some neighbor of i points to it. The algorithm consists of one rule shown in Algorithm 2. Thus, a node i is privileged if $x(i) \neq y(i)$ or $\mathcal{P}(i) \neq Q(i)$. If it executes, then it sets $x(i) = y(i)$ and $\mathcal{P}(i) = Q(i)$. It is easy to see that Algorithm 2 reduces to Algorithm 1 when $\mathcal{N}(i) = N(i)$ and $t(i) = 1$ for all i.

Lemma 3 *If Algorithm 2 stabilizes then D is a minimal set satisfying (1).*

Proof: We claim that D satisfies (1). By contradiction suppose that for some i, $|D \cap \mathcal{N}(i)| < t(i)$. Then $M_i \neq \emptyset$, and so there is some neighbor $j \in \mathcal{P}(i)$, $j \notin D$. But $y(j)$ is true and $x(j)$ is false, a contradiction. We now claim D is minimal as well. For every node $j \in D$, there is some node i that points to it. Since $\mathcal{P}(i) = Q(i)$, and since $\mathcal{P}(i) \neq \emptyset$, we must have $|\mathcal{N}(i) \cap D| = k \leq t(i)$. Thus, the removal of j from D will leave $|D \cap \mathcal{N}(i)| < t(i)$. □

Again, we use the terminology that node i *invites* node j (with $j = i$ allowed) if at some time $|D \cap \mathcal{N}(i)| = k < t(i)$, $j \in M_i$, i executes a move. For a node to join D, it must be pointed to from an initial state or be invited.

Theorem 2 *Algorithm 2 always stabilizes, and finds a minimal extended dominating set.*

Proof: In light of Lemma 3 we need only show stabilization. As before, observe that if D remains the same, then every node can make at most one move (to correct its pointers). So it suffices to show that D changes at most a finite number of times. In particular, it suffices to show that if during the time interval from t to t', $x(k)$ remains unchanged for all nodes $k > i$, then during this interval node i can make at most two in-moves. If i is never invited during this interval, then once i leaves D, it cannot rejoin. So suppose that during this interval i is invited by node j, allowing i to make an in-move. Once i enters D it must remain there if j continues pointing at it. And this is ensured, provided $|D \cap \mathcal{N}(j)| \leq t(j)$ throughout. Suppose at the time of invitation, $|D \cap \mathcal{N}(j)| = k$. Nodes having ID's larger than i do not move during this period, but the smaller nodes can. At the time of invitation, i is among the $t(j) - k$ smallest nodes in $\mathcal{N}(j) - D$. Even if all nodes smaller than i were to enter D, we would still have $|D \cap \mathcal{N}(j)| \leq t(j)$. It follows that j will remain pointing to i throughout, and i will remain in D. Hence, $x(i)$ can make at most two in-moves during this interval. □

4 Conclusion

We have propsed a self-stabilizing distributed algorithm to maintain a minimal total domination set in a distributed system graph; we have also shown how the ideas behind the algorithm are powerful enough to design self stabilizing algorithm for more complicated minimal extended dominating sets in a graph. We briefly discuss how the ideas can be further generalized.

In *signed domination*, we require that the members of D be in the majority of every closed neighborhood. An assignment $f : V \to \{-1, 1\}$ is a signed dominating function if, for every $i \in V$, the sum of the values in $N[i]$ is positive.

Equivalently, f is signed dominating if a strict majority of the values in every closed neighborhood are positive. The function f is minimal if the function f' obtained by reducing the value at any positive node, is never signed dominating. It is easy to see that minimal signed dominating functions correspond to certain minimal extended dominating sets. In particular, $f \rightarrow \{-1, 1\}$ is a minimal signed dominating function if and only if the set $D = \{i | f(i) = 1\}$ is a minimal extended dominating set in which for all i, $\mathcal{N}(i) = N[i]$ and $t(i) = \left\lfloor \frac{|N[i]|}{2} \right\rfloor + 1$.

One may extend these ideas even further to *weighted domination*. Here each node i has an allowable range of values $\{0, 1, \ldots, b(i)\}$ (in the previous section $b(i)$ was uniformly 1) and is assigned a weight $w(i)$. Each node also has a target $t(i)$ for the sum $\sum_{j \in \mathcal{N}(i)} w(j)$. We want a minimal assignment of values that satisfy the constraints. A primitive way to achieve this is to provide each node with $b(i)$ flags each with separate ID. It is more efficient though to provide each node with a counter $X(i)$ limited to the range and an array of weights $P(i)$ that counts how many times the node points to each neighbor. We omit the details.

This extension handles other forms of graph domination such as weak, strong and optional domination [8, 9]. It can also be altered to allow a node to have weights in a range $\{-b'(i), \cdots, b(i)\}$ and so handle minus domination.

References

[1] E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644, November 1974.

[2] E. W. Dijkstra. A belated proof of self-stabilization. *Distributed Computing*, 1(1):5–6, 1986.

[3] G. Tel. *Introduction to Distributed Algorithms*. Cambridge University Press, 1994.

[4] H. Attiya and J. Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics*. McGrah Hill, 1998.

[5] G Antonoiu and PK Srimani. Mutual exclusion between neighboring nodes in an arbitrary system graph tree that stabilizes using read/write atomicity. In *Euro-Par'99 Parallel Processing, Proceedings LNCS:1685*, pages 823–830, 1999.

[6] J Beauquier, AK Datta, M Gradinariu, and F Magniette. Self-stabilizing local mutual exclusion and daemon refinement. In *DISC00 Distributed Computing 14th International Symposium, Springer LNCS:1914*, pages 223–237, 2000.

[7] M Nesterenko and A Arora. Stabilization-preserving atomicity refinement. In *DISC99 Distributed Computing 13th International Symposium, Springer LNCS:1693*, pages 254–268, 1999.

[8] T.W. Haynes, S.T. Hedetniemi, and P.J. Slater. *Fundamentals of Domination in Graphs*. Marcel Dekker, 1998.

[9] T.W. Haynes, S.T. Hedetniemi, and P.J. Slater, editors. *Domination in Graphs: Advanced Topics*. Marcel Dekker, 1998.

[10] M. R. Garey and M. R. Johnson. *Computers and Intractability*. Freeman, New York, 1979.

[11] S Ghosh, A Gupta, and MH Karaata SV Pemmaraju. Self-stabilizing dynamic programming algorithms on trees. In *Proceedings of the Second Workshop on Self-Stabilizing Systems*, pages 11.1–11.15, 1995.

[12] S. M. Hedetniemi, S. T. Hedetniemi, D. P. Jacobs, and P. K. Srimani. Self-stabilizing algorithms for minimal dominating sets. To appear in *Computer Mathematics & Applications*, 2002.