A Perturbation-Proof Self-stabilizing Algorithm for Constructing Virtual Backbones in Wireless Ad-Hoc Networks

Amirreza Ramtin(✉), Vesal Hakami, and Mehdi Dehghan

Department of Information Technology, Amirkabir University of Technology, Tehran, Iran
a_ramtin@aut.ac.ir

Abstract. Self-stabilization is a key property of fault-tolerant distributed computing systems. A self-stabilizing algorithm ensures that the system eventually converges to a legitimate configuration from arbitrary initializations without any external intervention, and it remains in that legitimate configuration as long as no transient fault occurs. In this paper, the problem of virtual backbone construction in wireless ad-hoc networks is first translated into its graph-theoretic counterpart, i.e., approximate minimum connected dominating set construction. We then propose a self-stabilizing algorithm with time complexity O(n). Our algorithm features a perturbation-proof property in the sense that the steady state of the system gives rise to a Nash equilibrium, effectively discouraging the selfish nodes from perturbing the legitimate configuration by changing their valid states. Other advantages of this algorithm include increasing accessibility, reducing the number of update messages during convergence, and stabilizing with minimum changes in the topological structure. Proofs are given for the self-stabilization and perturbation-proofness of the proposed algorithm. The simulation results show that our algorithm outperforms comparable schemes in terms of stabilization time and number of state transitions.

Keywords: Self-stabilization · Wireless ad-hoc network · Virtual backbone · Selfishness · Perturbation · Nash equilibrium

1 Introduction

It is a well-established fact that the fundamental source of energy consumption in wireless ad hoc networks (WANETs) is the exchange of packets between nodes. Hence, when it comes to the design of routing mechanisms for WANETs, a key measure of efficiency is low communication overhead. A communication-efficient structure for supporting routing and multicast in WANETs is the virtual backbone architecture [1]. The virtual backbone approach to routing consists of two phases: (a) creation and update of a virtual backbone substrate, (b) finding and updating the paths. In this paper, we focus on the first phase. With regards to virtual backbone creation and maintenance, the two foremost desirable properties are stability and self-configuration without external intervention, given the dynamic and unstable topology of WANETs

© Springer International Publishing Switzerland 2014
A. Movaghar et al. (eds.): CNDS 2013, CCIS 428, pp. 66–76, 2014.
DOI: 10.1007/978-3-319-10903-9_6

and the multi-hop nature of communications [4]. A promising approach to realize stability and self-configuration is to rely on the notion of self-stabilization in distributed fault-tolerance [8]. A self-stabilizing algorithm guarantees that the system eventually converges to the desirable state regardless of its initial configuration, and that it remains in the desirable configuration as long as no transient fault occurs. Hence, designing virtual backbones with the self-stabilization property has the advantages of automatic structuring, and robustness against: transient faults, node failures, changes in their internal states, and occasional breakages in the communication structure of the system [8].

However, in the majority of self-stabilizing protocols for wireless ad-hoc networks, it is routinely assumed that the network nodes will cooperate with each other so that the overall stabilization of the system is guaranteed. This is while in most practical settings, the nodes neither belong to the same authority, nor do they operate under a single administration domain. Hence, it might be the case that the nodes pursue some private goals that may be in conflict with the system-wide objective. Consider, in particular, virtual backbone construction using a self-stabilizing algorithm. Obviously, once the protocol stabilizes, the nodes serving in the backbone have to sacrifice more processing and communication resources to the benefit of the entire network. Hence, each backbone node faces a dilemma as to whether maintain its serving role in the constructed backbone, or alternatively, perturb the system hoping that the algorithm would re-stabilize this time into a new configuration where the node is a backbone client rather than a server.

Motivated by the impact of node selfishness on protocol stabilization in ad-hoc networks, in this paper, we deal with perturbation-proneness in the context of virtual backbone construction in WANETs. The problem is first translated into minimum connected dominating set (MCDS) construction in the topological graph of the network. We then propose a self-stabilizing MCDS algorithm that prevents selfish nodes from post convergence perturbation of the system. A byproduct of our proposed scheme is faster recovery from all single-fault configurations with reduced message complexity, lower number of state transitions, and minimal topological re-structuring, which contributes to saving energy and increasing network life.

The rest of the paper is organized as follows: We briefly introduce the basic concepts and review the previous studies in Sect. 2. In Sect. 3, the proposed algorithm is discussed and proofs are given to establish its correctness. Section 4 deals with the numerical evaluation of the algorithm and comparisons are made to contrast its performance against prior art. The paper ends with conclusions.

2 Theoretical Background and Relevant Works

A system is self-stabilizing, if and only if, two conditions are satisfied starting with any arbitrarily initial state and with non-deterministically executing algorithm rules, as follows: (a) system converges to a legitimate global configuration (*convergence*) after finite moves, (b) and system remains in that legitimate configuration (*closure*) until no transient faults happens [2]. In graph theory, a connected dominated set (CDS) of graph G is a set D of nodes if two conditions are met: (a) D is a connected sub-graph from G.

(b) Any node of G is in D or adjacent to at least one node of D. A CDS of G is a MCDS, if it has the minimum members among all CDSs of G.

In the recent years, several self-stabilizing algorithms have been proposed for constructing CDS, but a majority of them has been designed based on central daemon (scheduler) which is practically impossible to implement in wireless ad-hoc network [3]. Furthermore, most of these works solely construct a CDS and their final product is not an approximation of MCDS [6]. Another drawback of all such algorithms is that they do not differentiate faults management with respect to their spread.

The self-stabilizing algorithm proposed in [7] which works under distributed scheduler with $O(n^2)$ time complexity is chosen among the relevant research works. We refer to this algorithm as MCDS*ss* in the rest of this paper. CDS Constructed by this algorithm is based on a sequential algorithm [7] that produces an $8opt + 1$ approximation of MCDS in graph.

It is not unlikely to consider a selfish self-stabilizing system under the assumption that nodes deviate from valid states and change their state to increase their utility by re-convergence to a different valid state. Nash equilibrium, a criterion for termination of a game, in self-stabilizing systems is discussed in [5] aiming at checking if it is possible to use it to prevent from selfish node perturbation in such systems. Fixed points in a game could be considered identical with final states in self-stabilizing systems. Distributed self-stabilizing systems are divided into four categories considering if fixed points are in Nash equilibrium or not. The main category is absolutely perturbation-proof. A system is absolutely perturbation-proof, if every system fixed point is in Nash equilibrium for any set of utility functions.

Nash equilibrium is a strategy per player in which none of the players intends to deviate from equilibrium strategies unilaterally i.e. it does not gain any profit by unilateral deviation from strategy and adopting another one under the condition that other players remain in their equilibrium states.

3 Proposed Algorithm

In this section, a high-level description of self-stabilizing and perturbation–proof virtual backbone construction algorithm based on MCDS abbreviated as MCDS*pp* will be explained.

3.1 Discussion on Design and Functionality of Algorithm

MCDS*pp* is designed based on marathe et al. algorithm [7]. This algorithm first constructs a breadth-first spanning (BFS) tree in network graph. Then, there is a repeated loop from root to the last depth of tree. In each loop, a maximal independent set (MIS) forms in the same depth nodes, not dominated by nodes of lower depths. Integration of all these sets produces a MIS in network graph. It is proven that this MIS is a WCDS. Finally, a fully connected set is established by adding connecting nodes to that set. Connecting nodes are the father of members of MIS in BFS tree. The final fully connected set is a CDS-tree. It is proven that a CDS-tree is an 8opt + 1 approximation of MCDS.

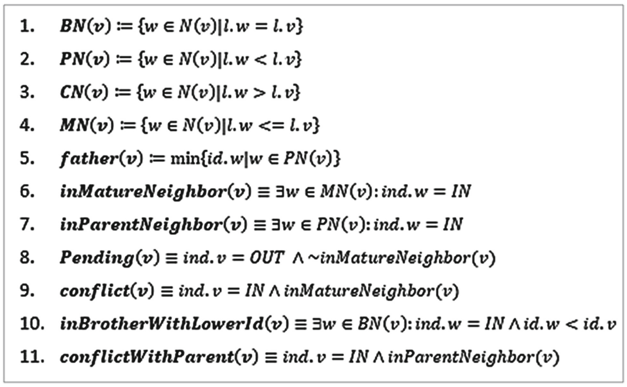
We assume that the tree T is formed in network graph through a self-stabilizing BFS tree algorithm. The distance of each node from root is determined by "l" variable. The validation of tree state and accuracy of "l" variable are the basic assumptions in MCDS*pp* algorithm implementation.

The state of each node is specified by two variables of *ind* and *dom* in MCDS configuration. These variables may hold two statuses: IN and OUT. Each node in legitimate configuration holds one of three states (IN, IN), (IN, OUT) and (OUT, OUT) based on the values of two variables (*dom*, *ind*). Set of nodes having state (IN, IN) are members of MIS. Union of nodes in state (IN, IN) and (IN, OUT) is a CDS. In legitimate configuration, state transition between four possible states is a transient fault.

In a legitimate configuration of self-stabilizing system, 1-fault situation is the occurrence a fault in a node like *v* that is generated by an undesirable change in its variables. It can be shown that two conditions apply (a) one of the rules is active in *v*. (b) it's possible to reach stability by execution of only one rule in *v*. The aim of designing the proposed algorithm is to detect and resolve 1-fault states by applying rules only in that faulty node and prevention from error propagation by unwanted execution of rules in neighboring nodes of *v*, *N*(*v*).

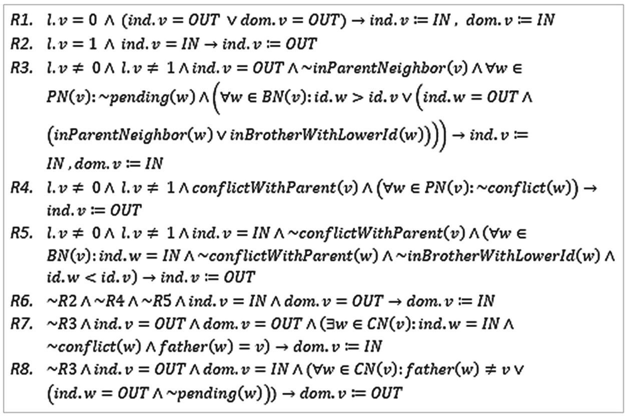
For simplification of understanding our proposed algorithm, we first define some terms that are preconditions of state transition operations in nodes and some sets for better readability of algorithm pseudo-code (Fig. 1).

Sets of PN, BN, MN, and CN refer to parent nodes (lower depth neighbors), brother nodes (same depth neighbors), mature nodes (union of PN and BN), and child nodes (higher depth neighbors) of a node in T tree respectively. Fifth term specifies father of a node. Father of a node is one of parent neighbors that has the lowest *id*. The 6th and 7th terms specify whether a mature or parent neighbor is member of MIS or not. The 8th term specifies if a node is pending. If neither a node nor its mature neighbors are members of MIS, that node is pending. The 9th term specifies if a node conflicts. If a node and at least one of its mature neighbors are members of MIS, conflict term will



```
1.  $BN(v) \coloneqq \{w \in N(v) | l.w = l.v\}$
2.  $PN(v) \coloneqq \{w \in N(v) | l.w < l.v\}$
3.  $CN(v) \coloneqq \{w \in N(v) | l.w > l.v\}$
4.  $MN(v) \coloneqq \{w \in N(v) | l.w <= l.v\}$
5.  $father(v) \coloneqq \min\{id.w | w \in PN(v)\}$
6.  $inMatureNeighbor(v) \equiv \exists w \in MN(v): ind.w = IN$
7.  $inParentNeighbor(v) \equiv \exists w \in PN(v): ind.w = IN$
8.  $Pending(v) \equiv ind.v = OUT \wedge \sim inMatureNeighbor(v)$
9.  $conflict(v) \equiv ind.v = IN \wedge inMatureNeighbor(v)$
10. $inBrotherWithLowerId(v) \equiv \exists w \in BN(v): ind.w = IN \wedge id.w < id.v$
11. $conflictWithParent(v) \equiv ind.v = IN \wedge inParentNeighbor(v)$
```

Fig. 1. Set and predicate definitions



```
R1. $l.v = 0 \wedge (ind.v = OUT \vee dom.v = OUT) \rightarrow ind.v := IN,\ dom.v := IN$
R2. $l.v = 1 \wedge ind.v = IN \rightarrow ind.v := OUT$
R3. $l.v \neq 0 \wedge l.v \neq 1 \wedge ind.v = OUT \wedge \sim inParentNeighbor(v) \wedge \forall w \in$
    $PN(v): \sim pending(w) \wedge \left(\forall w \in BN(v): id.w > id.v \vee \left(ind.w = OUT \wedge\right.\right.$
    $\left.\left.(inParentNeighbor(w) \vee inBrotherWithLowerId(w))\right)\right) \rightarrow ind.v :=$
    $IN, dom.v := IN$
R4. $l.v \neq 0 \wedge l.v \neq 1 \wedge conflictWithParent(v) \wedge (\forall w \in PN(v): \sim conflict(w)) \rightarrow$
    $ind.v := OUT$
R5. $l.v \neq 0 \wedge l.v \neq 1 \wedge ind.v = IN \wedge \sim conflictWithParent(v) \wedge (\forall w \in$
    $BN(v): ind.w = IN \wedge \sim conflictWithParent(w) \wedge \sim inBrotherWithLowerId(w) \wedge$
    $id.w < id.v) \rightarrow ind.v := OUT$
R6. $\sim R2 \wedge \sim R4 \wedge \sim R5 \wedge ind.v = IN \wedge dom.v = OUT \rightarrow dom.v := IN$
R7. $\sim R3 \wedge ind.v = OUT \wedge dom.v = OUT \wedge (\exists w \in CN(v): ind.w = IN \wedge$
    $\sim conflict(w) \wedge father(w) = v) \rightarrow dom.v := IN$
R8. $\sim R3 \wedge ind.v = OUT \wedge dom.v = IN \wedge (\forall w \in CN(v): father(w) \neq v \vee$
    $(ind.w = OUT \wedge \sim pending(w))) \rightarrow dom.v := OUT$
```

Fig. 2. Rules of MCDS*pp* algorithm

hold in that node. The 10th term is active in a node if at least one of its brother is a member of MIS and its *id* is lower than that node. Activity of 11th term indicates conflict between node and one of its parent neighbors.

Rules of MCDS*pp* are depicted in Fig. 2. The process of constructing MIS in T tree proceeds from root towards the last depth according to rules of 1–5. First rule determines root state. This node must become a member of MIS. Second rule determines the membership of root neighboring nodes (first depth). Rules 3, 4, and 5 specify remained nodes membership in MIS. One node becomes a member by performing rule 3 and cancels its membership by performing rule 4 or 5. While MIS forms, deeper nodes states has no effect on upper nodes states in T. The state of deeper nodes has no effect on the shallower ones. We give priority of MIS membership to the nodes that have lower *id* than their brothers in each same depth level of tree, in order to break symmetry of nodes. To detect 1-fault situations, each node needs to know its 2-hop neighbors membership, too. This information guarantees that if 1-fault occurs in a neighbor of parent or brother of the node, in a legitimate configuration, no rules will become active on the node. According to rule 6, members of MIS i.e. the nodes for which the *ind* is in IN state, join MCDS. Rule 7 or 8 checks membership or none-membership of remained nodes in CDS respectively. Nodes that are fathers of members of MIS, become members of MCDS by executing rule 7.

3.2 Proof of Correctness

Lemma 1. Assuming that spanning tree T is valid up to ith depth and MIS is constructed up to (i − 1)th depth by MCDS*pp* rules and exists in valid state, MIS is constructed after maximum of m rounds up to ith depth and exists in valid state. No node changes its membership of MIS as long as no transient fault happens.

Proof. The root becomes a member of MIS by executing rule 1 at the first round. It is obvious that this membership is permanent because rules 2–5 are not executed in the

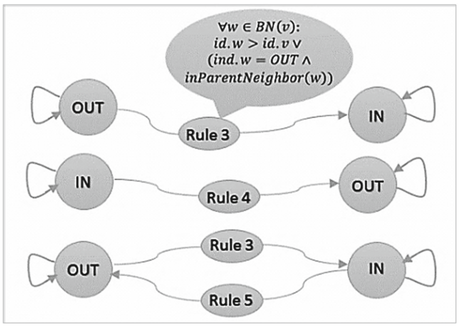

∀w ∈ BN(v): id.w > id.v ∨ (ind.w = OUT ∧ inParentNeighbor(w))
OUT
Rule 3
IN
IN
Rule 4
OUT
Rule 3
OUT
IN
Rule 5

Fig. 3. *ind* variable state transition diagram regards to validation of shallower levels of tree

root. Similarly, neighbors of root (l = 1) leave membership of MIS via rule 2 at the first round and this decision will be permanent. It is clear that the membership of deeper nodes has no effect on the membership of ith depth in MIS according to rules 1–5. The diagram of *ind* variable state transition of ith depth nodes (i > 1) is depicted in Fig. 3 assuming that MIS is already formed by rules 1–5. It can be shown in Fig. 3 that if a node passes path 1 or 2 it will be permanent i.e. those paths are traversed just once. The reason is that all predictions in paths 1 and 2 are related either to lower depth nodes for which the validation and stability are assumed, or to the base information like *id*. There is a predicate (inbrotherwithlowerid) in paths 3 and 4 which is also related to the state of the same depth nodes. At the first round, *ind* variable becomes equal to OUT by running rule 4 in all nodes that can traverse path 2. It is obvious that OUT state (non-membership in MIS) is permanent in these nodes. Following the first round, in the second round, all nodes which can traverse path 1 are activated and *ind* variable in those nodes become equal to IN. Consequently, no node traverses path 1 or 2 by traversing this round. After the second round, either *ind* variable value is permanently OUT in all nodes of ith depth or at least there is one node (v) that is in IN state, a permanent state, through path 1. In the third round, neighboring nodes with the same depth of node v which are in IN state switch to OUT state (rule 5). It can be shown that this state is permanent in those nodes and does not change in following rounds. In the next round, nodes that rule 5 is active in them travers path 3 and it is permanent. Then, rounds 3 and 4 will be repeated until there is still some nodes in which rules 3 or 5 are active. So, a number of rounds up to a maximum equal to the number of ith depth nodes are traversed until MIS is constructed at this depth.

Lemma 2. MIS structure in T is formed after n rounds. n is number of tree nodes.

Proof. We use induction to proof this lemma. In Lemma 1, it has been shown that root and second depth nodes of T enter to valid state of MIS just in one round (basis: statement holds for d = 1, 2). Using Lemma 1 inductive step will be proven for d > 1. Dou to Lemma 1 if MIS is formed up to ith depth, after m_i round, it is formed up to i + 1th depth. Therefore time complexity of MIS construction is $o\left(\sum_{i=2}^{D} m_i\right)$ which is equal to $o(n)$. D is depth of T.

Lemma 3 (convergence condition). In a graph with help of MCDS*pp* algorithm, MCDS is constructed after R_T + n + 1 rounds.

Proof. T in R_T and then MIS in n rounds are constructed. According to rules 6–8, members of MIS and connecting nodes join to MCDS and nodes that are member of MCDS and rule 8 is active in them exit from it just in one round. Because all terms of those three rules depend on *id* and *ind* variables, not *dom*, final states are permanent.

Lemma 4 (closure condition).

Proof. Correctness of this condition proves by contradiction. Suppose that closure condition does not hold, thus at least one rule is active in legitimate configuration, but referring to demonstrations in Lemma 1–3, final states are permanent and no rules will be executed in the legitimate configuration.

3.3 Proof of Absolutely Perturbation-Proof Feature

Lemma 5. Happening 1-fault in *ind* variable of a i^{th} depth node has no effect on the state of upper or lower depth nodes.

Proof. MIS had been formed before 1-fault incident in the legitimate configuration, therefore it causes one of pending or conflict predicates holds. State of parents effects on the preconditions of rules 3–5 in a node. To be sure that those rules do not activate by 1-fault incident in lower depth nodes, some terms are added to them in order to checking that pending or conflict predicates are active in parent neighbors or not. Similarly in rules 7–8, those predicates are checked for upper depth nodes, because state of children effects on preconditions of those rules. It is obvious that preconditions of rules 1, 2 and 6 have no connection to states of neighbors.

Lemma 6. Happening 1-fault in *ind* variable of a i^{th} depth node has no effect on the other i^{th} depth nodes.

Proof. It is obvious that *ind* variable change in a node has no effect on the *dom* variables of its brothers. Hence, we investigate effect of 1-fault in node v on *ind* variables of the i^{th} depth 1-hop neighbor z and 2-hop neighbor k.

If 1-fault (IN to OUT) happens in v, the only rule that might be active in z is rule 3. Note that state of z is OUT. If node z has a parent in IN state or its *id* is greater than v, rule 3 does not activate. Otherwise it is evident that in the valid states, rule 3 did not execute in z because of another brother like w that had a lower *id* than z and was in IN state. Because 1-fault happens in z, not w, rule 3 still do not activate in z. If state of k is IN, the only rule that might be active in that node is rule 5. However, in rule 5, even if term '$\sim$inbrotherwithlowerid' is active, term '*ind*.w = IN' must be active concurrently either, but in the previous paragraph we show that 1-hop brother of v remains in OUT state. If node k is in OUT state, rule 3 certainly cannot be active in it, because there is no preconditions in that rule that holds with occurrence of 1-fault.

Assuming that 1-fault (OUT to IN) happens in v, if state of z is OUT, it cannot activate any rule in z. If state of z is IN, the only rule that might be active is rule 5. Because valid state of v had been OUT, there were some preconditions of rule 3 that had not hold. It is not possible that node v can activate rule 5 in another node because of those preconditions. In legitimate configuration, MIS membership states in two-hop neighborhood of v (k z v) is one of these three cases: (010, 100, 000). In the first case, the only rule that might be active is rule 3, but term '*ind*.w = OUT' must hold if rule 3 is active. Therefore 1-fault cannot activate rule 3 in k, because state of z is still IN. In the second case, the only rule that can be active is rule 5, but as the term '*ind*.w = IN' exists in rule 5, it cannot activate, because z is in OUT state. In the last case, although it seems that rule 3 can activate in node k, but a brother or a parent in IN state has existed and they still do not allow rule 3 being active in node k.

Lemma 7. Happening 1-fault in *dom* variable of a i^{th} depth node has no effect on the states of neighbors.

Proof. Since in preconditions of rules 1–8 do not refer to *dom* variables of neighbors, So it is obvious that change of *dom* variable in a node has no effect on the others.

Theorem 1. If 1-fault occurs in a system based on MCDS*pp* algorithm, faulty node and only that node by executing just one rule enters to the valid state that it was in before.

Proof. Convergence feature of algorithm explains that system converges from an illegitimate configuration to a legitimate one. We also showed in Lemma 5–7 that occurrence of 1-fault has no effect on neighbors. Considering these two explanations, it is proven that with execution of self-stabilizing rules in the faulty node, system will return to legitimate configuration. Investigating algorithm rules shows that it will be done by just one rule execution.

Theorem 2. If self-stabilizing rules cause that after perturbation of any selfish node in a legitimate configuration, system returns to that legitimate configuration, that configuration is in Nash equilibrium.

Proof. According to definition of Nash equilibrium, a legitimate configuration of a self-stabilizing system is in Nash equilibrium, if no node can obtain profit by unilateral deviation from its state and going to a different state. Incentive of a node from perturbation in a self-stabilizing system is convergence to another legitimate configuration so that its utility in new configuration is more than pervious. In a legitimate configuration of a self-stabilizing system, perturbation of a node models with occurrence of 1-fault in that node. If self-stabilizing rules cause that after 1-fault or perturbation of a node, system again converges to the previous legitimate configuration, no nodes will have perturbation incentive and thus that configuration is in Nash equilibrium.

Theorem 3. A system based on MCDS*pp* algorithm is absolutely perturbation-proof.

Proof. According to Theorem 1, in a system based on MCDS*pp* algorithm, after 1-fault incident in legitimate configuration, system will return to that legitimate configuration again only by one move. In Theorem 2, we said that if self-stabilizing rules force system to return to the previous legitimate configuration after perturbation of a selfish node, that configuration is in Nash equilibrium. Therefore, stable states of a self-stabilizing system based on MCDS*pp* algorithm are in Nash equilibrium for any utility functions. It means that the MCDS*pp* algorithm is absolutely perturbation-proof.

4 Performance Analysis

In this section, we conduct a number of experiments to compare the performance of our virtual backbone construction algorithm MCDSpp with that of MCDS*ss* [7]. The comparisons are made in terms of the number of state transitions and stabilization time. We simulate the algorithms under two operational scenarios: arbitrary configuration (*ind* and *dom* variables are randomly equal to IN or OUT) and multiple fault configuration. All experiments are implemented with OMnet++ simulator under an unfair scheduler, and the reported data points are the average of 100 tests in each scenario. Each node periodically notifies its neighbors of its current state by broadcasting beacon packets. The MAC configuration adheres to IEEE 802.11 and the channel model is simple path loss.

In Figs. 4 and 5, the number of state transitions and stabilization time of MCDS*pp* and MCDS*ss* are reported, respectively. Average connectivity degree is 8 and we have varied the number of nodes. The number of state transitions is equal to the total number of *ind* and *dom* variable changes. From these two diagrams, it can be deduced that the performance superiority of MCDS*pp* over MCDS*ss* becomes even more apparent as the number of nodes increases.

In another scenario, performance of these two algorithm is compared by fault injections to the legitimate configuration. In this test (see Fig. 6), the number of fault injections is variable from 1 to 20. Topology is formed from 20 nodes with average connectivity degree 3. MCDS*pp* stabilizes from single faults by only one state transition (move). While MCDS*ss* needs on average 5 state transitions. With more fault injections, the performance gain of MCDS*pp* over MCDS*ss* decreases and reaches 1.5.

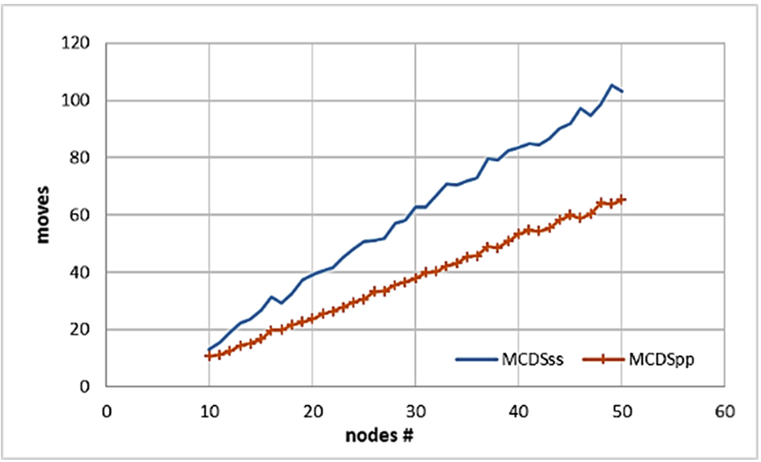


Fig. 4. The impact of the number of nodes on the number of state transitions.

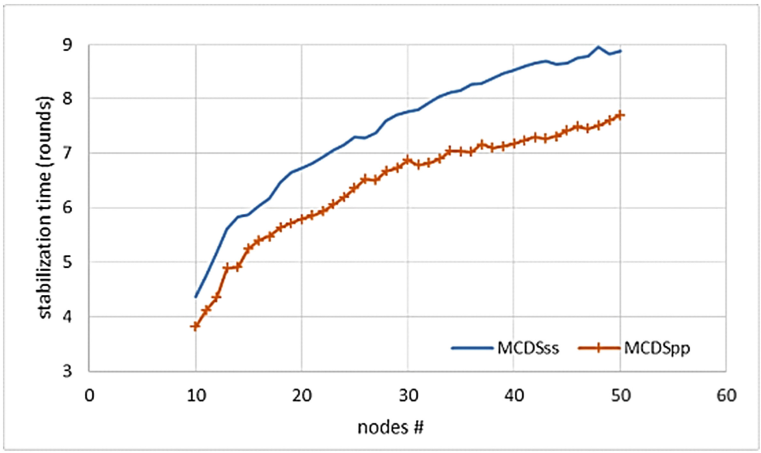


Fig. 5. The impact of the number of nodes on stabilization time.

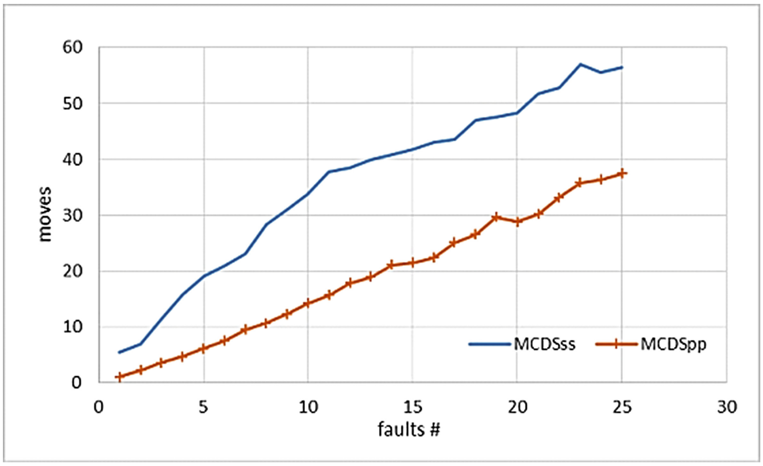


Fig. 6. The impact of the number of fault injections on the number of state transitions.

5 Conclusion

In this paper, a distributed virtual backbone construction algorithm has been proposed for wireless ad-hoc networks based on the notion of MCDS in graph theory. The proposed algorithm is self-stabilizing against transient faults and topology changes. We also proved that the algorithm's stable configuration gives rise to a Nash equilibrium, and thus, selfish nodes have no motivation to perturb the constructed backbone once the system converges. The other merit featured by our algorithm is fast convergence from single fault configurations. We plan to extend this algorithm to accommodate situations where nodes may also exhibit selfish behavior during convergence.

References

1. Das, B., Bharghavan, V.: Routing in ad-hoc networks using minimum connected dominating sets. In: ICC 97 Montreal, Towards the Knowledge Millennium. 1997 IEEE International Conference on Communications, pp. 376–380 (1997)
2. Dasgupta, A., Ghosh, S., Tixeuil, S.: Selfish stabilization. In: Datta, A.K., Gradinariu, M. (eds.) SSS 2006. LNCS, vol. 4280, pp. 231–243. Springer, Heidelberg (2006)
3. Dubhashi, D., et al.: Fast distributed algorithms for (weakly) connected dominating sets and linear-size skeletons. J. Comput. Syst. Sci. **71**(4), 467–479 (2005)
4. Gao, L., et al.: Virtual backbone routing structures in wireless ad-hoc networks. Global J. Comput. Sci. Technol. **10**(4), 21 (2010)
5. Gouda, M.G., Acharya, H.B.: Nash equilibria in stabilizing systems. In: Guerraoui, R., Petit, F. (eds.) SSS 2009. LNCS, vol. 5873, pp. 311–324. Springer, Heidelberg (2009)
6. Jain, A., Gupta, A.: A distributed self-stabilizing algorithm for finding a connected dominating set in a graph. In: 2005 Sixth International Conference on Parallel and Distributed Computing, Applications and Technologies, PDCAT 2005, pp. 615–619 (2005)
7. Kamei, S., Kakugawa, H.: A self-stabilizing distributed approximation algorithm for the minimum connected dominating set. In: 2007 IEEE International Conference on Parallel and Distributed Processing Symposium, IPDPS 2007, pp. 1–8 (2007)
8. Tixeuil, S.: Self-stabilizing algorithms. In: Atallah, M.J., Blanton, M. (eds.) Algorithms and Theory of Computation Handbook. Chapman & Hall/CRC, Boca Raton (2010)